Artificial Intelligence

Approaches to AI

Turing Test and Rational Agent Approaches:

Turing Test:

The **Turing Test**, proposed by the brilliant computer scientist **Alan Turing** in 1950, serves as a litmus test for evaluating a machine's ability to exhibit human-like intelligence. Here's how it works:

1. **The Setup:**
    - Imagine a game with three players: two humans (A and B) and one computer ©.
    - An **interrogator** (also a human) is isolated from the other two players.
    - The interrogator's task is to discern which player is human and which is the computer by asking questions to both.
2. **The Goal:**
    - The computer (player A) strives to be indistinguishable from a human.
    - If the interrogator cannot consistently differentiate between the human and the machine based on their responses, the computer is deemed to have passed the Turing Test.
3. **The Conversation:**
    - The entire interaction occurs via a text-only channel (such as a computer keyboard and screen).
    - Sample exchange:
        - **Interrogator ©:** "Are you a computer?"
        - **Computer (A):** "No."
        - **C:** "Multiply 158745887 by 56755647."
        - **A:** (After a pause) "An incorrect answer!"
        - **C:** "Add 5478012 and 4563145."
        - **A:** (Another pause) "10041157."
4. **The Verdict:**
    - If the interrogator cannot consistently distinguish between the human and computer responses, the machine passes the test.
    - In other words, a computer is considered intelligent if its conversation cannot be easily distinguished from a human's.
5. **Critiques:**
    - While the Turing Test has been a benchmark for AI research, critics argue that it focuses too heavily on language and neglects other facets of intelligence (such as perception, problem-solving, and decision-making).

Rational Agent Approaches:

- A **rational agent** is an intelligent agent that makes decisions based on logical reasoning and optimizes its behavior to achieve specific goals.

- Such agents perceive their environment through various sensors or inputs and act in ways that maximize their expected utility.
- Rationality involves making choices that lead to desirable outcomes, given the available information.

State space representation:

State space representation is a fundamental concept in **Artificial Intelligence (AI)**.

Let's delve into it:

1. **Definition**:
    - A **state space** is a mathematical framework used to represent a problem by defining **all possible states** in which the problem can exist.
    - In search algorithms, we utilize the state space to represent the following:
        - **Initial state**: The starting point of the problem.
        - **Goal state**: The desired outcome or solution.
        - **Current state**: The state we are currently exploring during the search process.
    - Each state within the state space is described using a set of variables.
2. **Purpose and Importance**:
    - State space representation is crucial for problem-solving in AI.
    - It allows us to explore all possible states systematically to find a solution.
    - Applications of state space search range from game-playing algorithms to natural language processing.
3. **Features of State Space Search**:
    - **Exhaustiveness**: State space search explores **all possible states** of a problem.
    - **Completeness**: If a solution exists, state space search will find it.
    - **Optimality**: Searching through the state space yields an **optimal solution**.
    - **Uninformed and Informed Search**:
        - **Uninformed search** provides no additional information about the problem.
        - **Informed search** (e.g., using heuristics) guides the search process.
4. **Common State Space Search Algorithms**:
    - **A* algorithm**: A well-known informed search algorithm.
    - Other commonly used algorithms include:
        - **Breadth-First Search (BFS)**: Explores neighbors before deeper levels.
        - **Depth-First Search (DFS)**: Explores as far as possible along a branch before backtracking.
        - **Hill climbing**: Iteratively improves the solution.
        - **Simulated annealing**: Used for optimization problems.
        - **Genetic algorithms**: Inspired by natural selection.
5. **Steps in State Space Search**:
    - Initialize the current state to the initial state.
    - Check if the current state is the goal state. If so, terminate and return the result.
    - Generate the set of possible successor states from the current state.
    - For each successor state:
        - Check if it has been visited. If not, add it to the queue of states to explore.

- Set the next state in the queue as the current state and repeat the process.
- If all possible states are explored without finding the goal state, return with no solution.

Heuristic Search Techniques

A heuristic is a technique that is used to solve a problem faster than the classic methods. These techniques are used to find the approximate solution of a problem when classical methods do not. Heuristics are said to be the problem-solving techniques that result in practical and quick solutions.

Heuristics are strategies that are derived from past experience with similar problems. Heuristics use practical methods and shortcuts used to produce the solutions that may or may not be optimal, but those solutions are sufficient in a given limited timeframe.

We can perform the Heuristic techniques into two categories:

Direct Heuristic Search techniques in AI

It includes Blind Search, Uninformed Search, and Blind control strategy. These search techniques are not always possible as they require much memory and time. These techniques search the complete space for a solution and use the arbitrary ordering of operations.

The examples of Direct Heuristic search techniques include Breadth-First Search (BFS) and Depth First Search (DFS).

Weak Heuristic Search techniques in AI

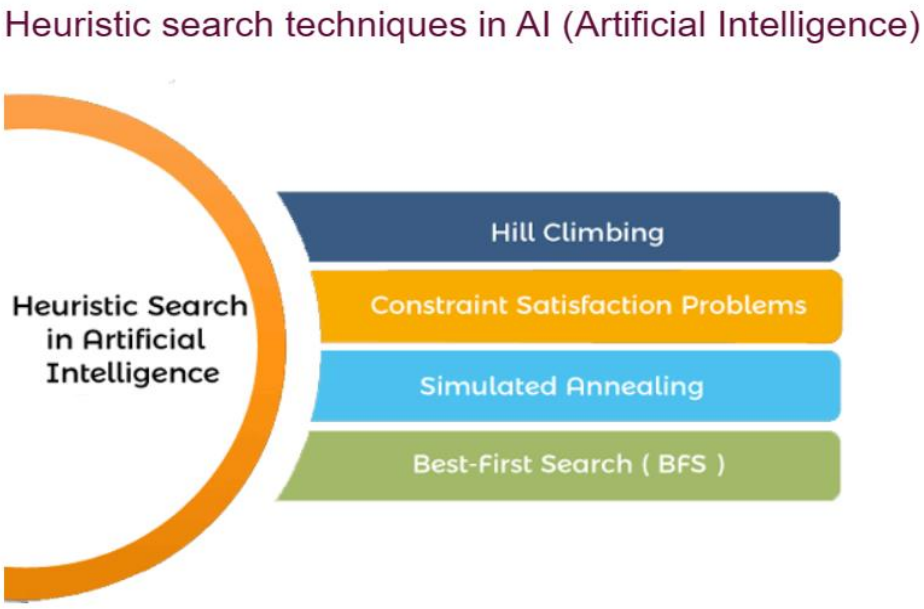
It includes Informed Search, Heuristic Search, and Heuristic control strategy. These techniques are helpful when they are applied properly to the right types of tasks. They usually require domain-specific information.

The examples of Weak Heuristic search techniques include Best First Search (BFS) and A*.

Some of the techniques listed below:

- Bidirectional Search
- A* search
- Simulated Annealing

- Hill Climbing
- Best First search
- Beam search



Game Playing in Artificial Intelligence

Game Playing is an important domain of artificial intelligence. Games don't require much knowledge; the only knowledge we need to provide is the rules, legal moves and the conditions of winning or losing the game. Both players try to win the game. So, both of them try to make the best move possible at each turn. Searching techniques like BFS(Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time. So, we need another search procedures that improve –

- **Generate procedure** so that only good moves are generated.
- **Test procedure** so that the best move can be explored first.

Game playing is a popular application of artificial intelligence that involves the development of computer programs to play games, such as chess, checkers, or Go. The goal of game playing in artificial intelligence is to develop algorithms that can learn how to play games and make decisions that will lead to winning outcomes.

1. One of the earliest examples of successful game playing AI is the chess program Deep Blue, developed by IBM, which defeated the world champion Garry Kasparov in 1997. Since then, AI has been applied to a wide range of games, including two-player games, multiplayer games, and video games.

There are two main approaches to game playing in AI, rule-based systems and machine learning-based systems.

1. **Rule-based systems** use a set of fixed rules to play the game.
2. **Machine learning-based systems** use algorithms to learn from experience and make decisions based on that experience.

In recent years, machine learning-based systems have become increasingly popular, as they are able to learn from experience and improve over time, making them well-suited for complex games such as Go. For example, AlphaGo, developed by DeepMind, was the first machine learning-based system to defeat a world champion in the game of Go.

Game playing in AI is an active area of research and has many practical applications, including game development, education, and military training. By simulating game playing scenarios, AI algorithms can be used to develop more effective decision-making systems for real-world applications.

The most common search technique in game playing is **Minimax search procedure**. It is depth-first depth-limited search procedure. It is used for games like chess and tic-tac-toe.

Mini-Max Algorithm in Artificial Intelligence

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Pseudo-code for MinMax Algorithm:

```
1. function minimax(node, depth, maximizingPlayer) is
2. if depth ==0 or node is a terminal node then
3. return static evaluation of node
4.
5. if MaximizingPlayer then      // for Maximizer Player
6. maxEva= -infinity
7.  for each child of node do
8.  eva= minimax(child, depth-1, false)
9. maxEva= max(maxEva,eva)       //gives Maximum of the values
10. return maxEva
11.
12. else                         // for Minimizer player
13.  minEva= +infinity
14.  for each child of node do
15.  eva= minimax(child, depth-1, true)
16.  minEva= min(minEva, eva)       //gives minimum of the values
17.  return minEva
```

Current TimeÂ 0:00
/
DurationÂ 18:10
Â

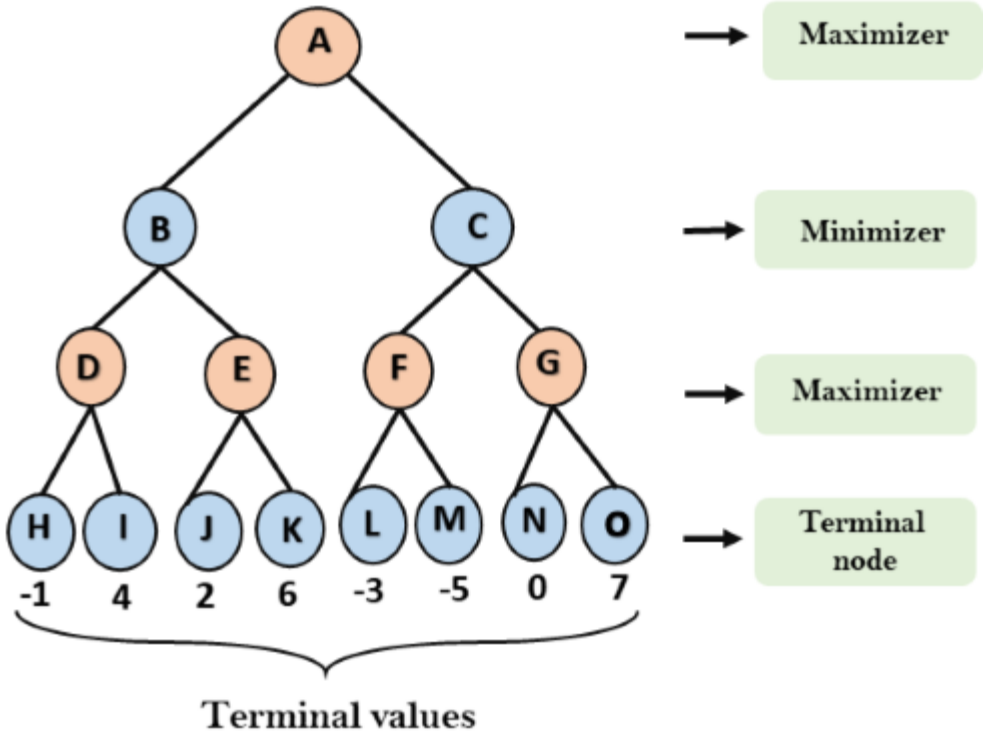
Initial call:

Minimax(node, 3, true)

Working of Min-Max Algorithm:

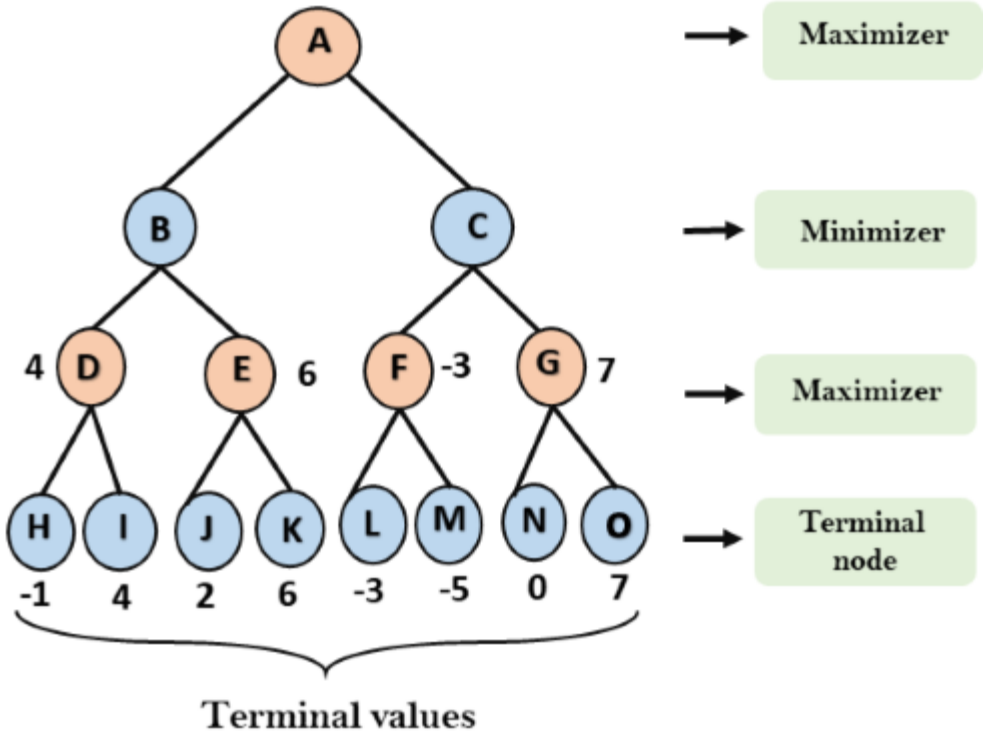
- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



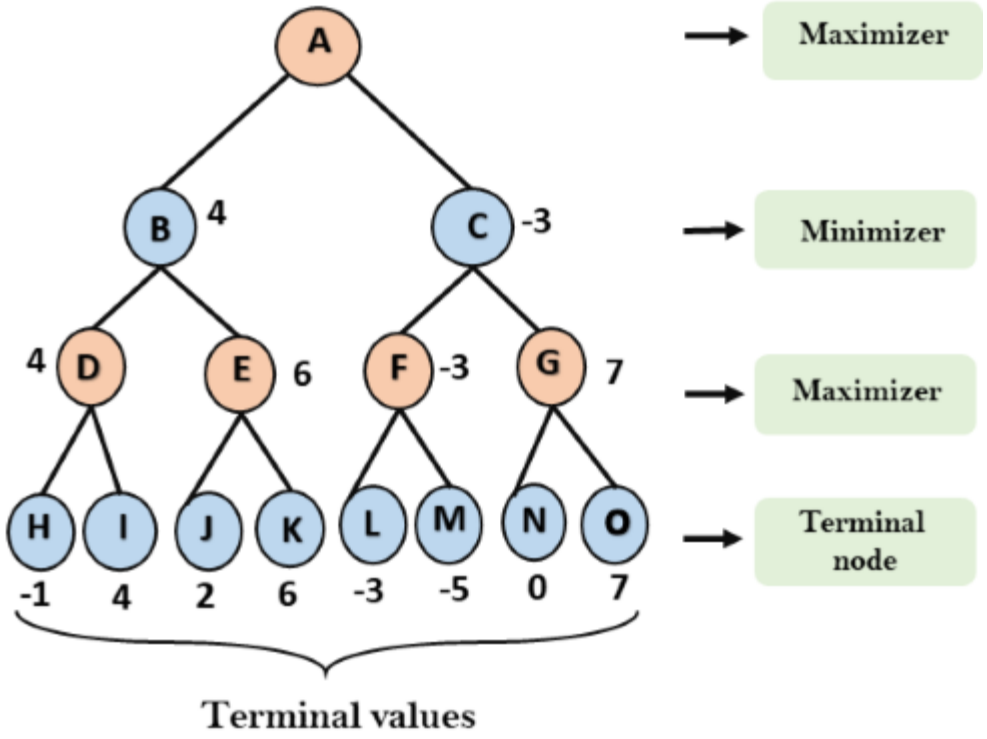
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



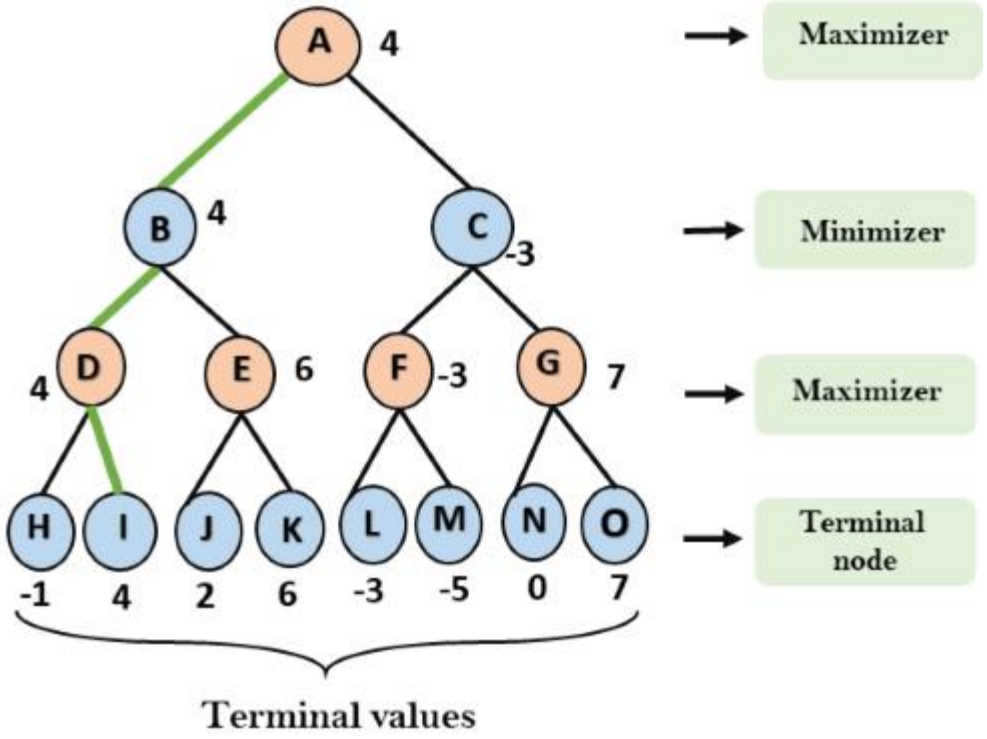
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3)= 4



That was the complete workflow of the minimax two player game.

Properties of Mini-Max algorithm:

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Connect 4, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.[1]

History

Allen Newell and Herbert A. Simon who used what John McCarthy calls an "approximation"[2] in 1958 wrote that alpha–beta "appears to have been reinvented a number of times".[3] Arthur Samuel had an early version for a checkers simulation. Richards, Timothy Hart, Michael Levin and/or Daniel Edwards also invented alpha–beta independently in the United States.[4] McCarthy proposed similar ideas during the Dartmouth workshop in 1956 and suggested it to a group of his students including Alan Kotok at MIT in 1961.[5] Alexander Brudno independently conceived the alpha–beta algorithm, publishing his results in 1963.[6] Donald Knuth and Ronald W. Moore refined the algorithm in 1975.[7][8] Judea Pearl proved its optimality in terms of the expected running time for trees with randomly assigned leaf values in two papers.[9][10] The optimality of the randomized version of alpha–beta was shown by Michael Saks and Avi Wigderson in 1986.[11]

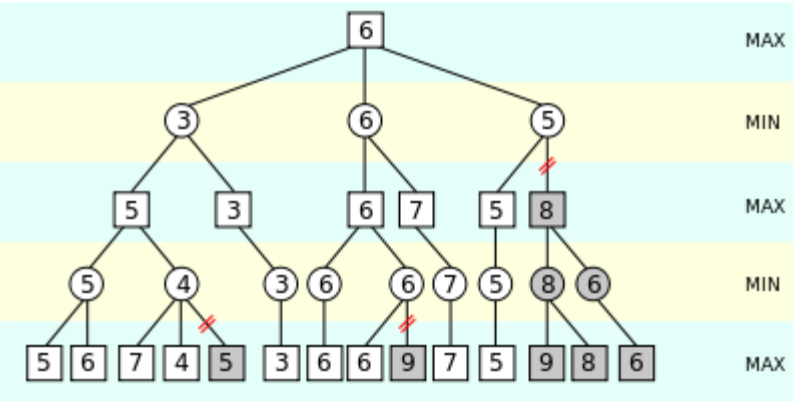
Core idea

A game tree can represent many two-player zero-sum games, such as chess, checkers, and reversi. Each node in the tree represents a possible situation in the game. Each terminal node (outcome) of a branch is assigned a numeric score that determines the value of the outcome to the player with the next move.[12]

The algorithm maintains two values, alpha and beta, which respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta < alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

To illustrate this with a real-life example, suppose somebody is playing chess, and it is their turn. Move "A" will improve the player's position. The player continues to look for moves to make sure a better one hasn't been missed. Move "B" is also a good move, but the player then realizes that it will allow the opponent to force checkmate in two moves. Thus, other outcomes from playing move B no longer need to be considered since the opponent can force a win. The maximum score that the opponent could force after move "B" is negative infinity: a loss for the player. This is less than the minimum position that was previously found; move "A" does not result in a forced loss in two moves.

Improvements over naive minimax

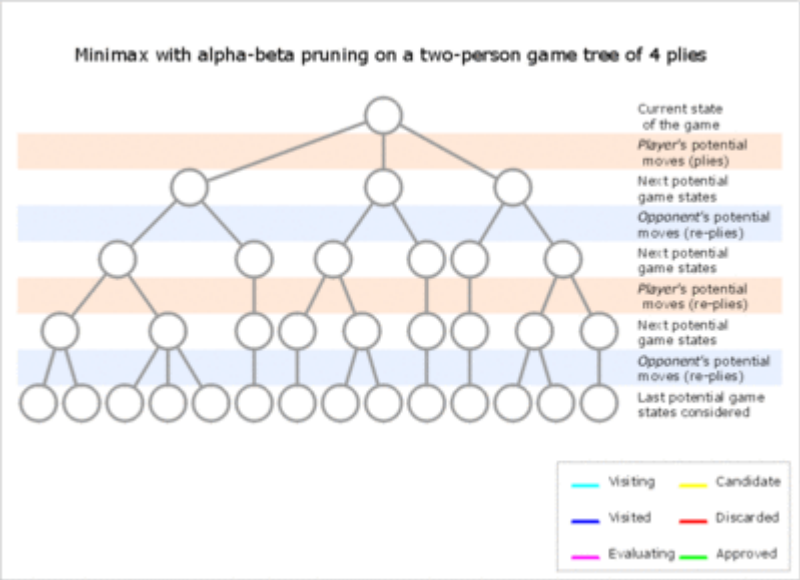


An illustration of alpha–beta pruning. The grayed-out subtrees don't need to be explored (when moves are evaluated from left to right), since it is known that the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The max and min levels represent the turn of the player and the adversary, respectively.

The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated.[12] This way, the search time can be limited to the 'more promising' subtree, and a deeper search can be performed in the same time. Like its predecessor, it belongs to the branch and bound class of algorithms. The optimization reduces the effective depth to slightly more than half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best choice for side on move ordered first at each node).

With an (average or constant) branching factor of b, and a search depth of d plies, the maximum number of leaf node positions evaluated (when the move ordering is pessimal) is $O(b \times b \times \ldots \times b) = O(b^d)$ – the same as a simple minimax search. If the move ordering for the search is optimal (meaning the best moves are always searched first), the number of leaf node positions evaluated is about $O(b \times 1 \times b \times 1 \times \ldots \times b)$ for odd depth and $O(b \times 1 \times b \times 1 \times \ldots \times 1)$ for even depth, or $O(b^{d/2}) = O(\sqrt{b^d})$. In the latter case, where the ply of a search is even, the effective branching factor is reduced to its square root, or, equivalently, the search can go twice as deep with the same amount of computation.[13] The explanation of $b \times 1 \times b \times 1 \times \ldots$ is that all the first player's moves must be studied to find the best one, but for each, only the second player's best move is needed to refute all but the first (and best) first player move—alpha–beta ensures no other second player moves need be considered. When nodes are considered in a random order (i.e., the algorithm randomizes), asymptotically, the expected number of nodes evaluated in uniform trees with binary leaf-values is $\Theta(((b-1+\sqrt{b^2+14b+1})/4)^d)$.[11] For the same trees, when the values are assigned to the leaf values independently of each other and say zero and one are both equally probable, the expected number of nodes evaluated is $\Theta((b/2)^d)$, which is much smaller than the work done by the randomized algorithm, mentioned above, and is again optimal for such random trees.[9] When the leaf values are chosen independently of each other but from the [0,1]interval uniformly at random, the expected number of nodes evaluated increases $\Theta(b^{d/\log(d)})$to in the d--> ∞ imit,[10] which is again optimal for these kind random trees. Note that the actual work for "small" values of d is better approximated using $0.925d^{0.747}$.[10][9]

A chess program that searches four plies with an average of 36 branches per node evaluates more than one million terminal nodes. An optimal alpha-beta prune would eliminate all but about 2,000 terminal nodes, a reduction of 99.8%.[12]



An animated pedagogical example that attempts to be human-friendly by substituting initial infinite (or arbitrarily large) values for emptiness and by avoiding using the negamax coding simplifications.

Normally during alpha–beta, the subtrees are temporarily dominated by either a first player advantage (when many first player moves are good, and at each search depth the first move checked by the first player is adequate, but all second player responses are required to try to find a refutation), or vice versa. This advantage can switch sides many times during the search if the move ordering is incorrect, each time leading to inefficiency. As the number of positions searched decreases exponentially each move nearer the current position, it is worth spending considerable effort on sorting early moves. An improved sort at any depth will exponentially reduce the total number of positions searched, but sorting all positions at depths near the root node is relatively cheap as there are so few of them. In practice, the move ordering is often determined by the results of earlier, smaller searches, such as through iterative deepening.

Additionally, this algorithm can be trivially modified to return an entire principal variation in addition to the score. Some more aggressive algorithms such as MTD(f) do not easily permit such a modification.

Pseudocode

The pseudo-code for depth limited minimax with alpha–beta pruning is as follows:[13]

Implementations of alpha–beta pruning can often be delineated by whether they are "fail-soft," or "fail-hard". With fail-soft alpha–beta, the alphabeta function may return values (v) that exceed ($v < \alpha$ or $v > \beta$) the α and β bounds set by its function call arguments. In comparison, fail-hard alpha–beta limits its function return value into the inclusive range of α and β. The main difference between fail-soft and fail-hard implementations is whether α and

β are updated before or after the cutoff check. If they are updated before the check, then they can exceed initial bounds and the algorithm is fail-soft.

The following pseudo-code illustrates the fail-hard variation.[1]

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth == 0 or node is terminal then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            if value > β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            if value < α then
                break (* α cutoff *)
            β := min(β, value)
        return value
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

The following pseudocode illustrates fail-soft alpha-beta.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if value ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if value ≤ α then
                break (* α cutoff *)
        return value
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

Heuristic improvements

Further improvement can be achieved without sacrificing accuracy by using ordering heuristics to search earlier parts of the tree that are likely to force alpha–beta cutoffs. For example, in chess, moves that capture pieces may be examined before moves that do not, and moves that have scored highly in earlier passes through the game-tree analysis may be

evaluated before others. Another common, and very cheap, heuristic is the killer heuristic, where the last move that caused a beta-cutoff at the same tree level in the tree search is always examined first. This idea can also be generalized into a set of refutation tables.

Alpha–beta search can be made even faster by considering only a narrow search window (generally determined by guesswork based on experience). This is known as *aspiration search*. In the extreme case, the search is performed with alpha and beta equal; a technique known as *zero-window search*, *null-window search*, or *scout search*. This is particularly useful for win/loss searches near the end of a game where the extra depth gained from the narrow window and a simple win/loss evaluation function may lead to a conclusive result. If an aspiration search fails, it is straightforward to detect whether it failed *high* (high edge of window was too low) or *low* (lower edge of window was too high). This gives information about what window values might be useful in a re-search of the position.

Over time, other improvements have been suggested, and indeed the Falphabeta (fail-soft alpha–beta) idea of John Fishburn is nearly universal and is already incorporated above in a slightly modified form. Fishburn also suggested a combination of the killer heuristic and zero-window search under the name Lalphabeta ("last move with minimal window alpha–beta search").

Knowledge Representation

Knowledge Representation in AI describes the representation of knowledge. Basically, it is a study of how the **beliefs, intentions**, and **judgments** of an **intelligent agent** can be expressed suitably for automated reasoning. One of the primary purposes of Knowledge Representation includes modeling intelligent behavior for an agent.

Knowledge Representation and Reasoning (**KR, KRR**) represents information from the real world for a computer to understand and then utilize this knowledge to solve **complex real-life problems** like communicating with human beings in natural language. Knowledge representation in AI is not just about storing data in a database, it allows a machine to learn from that knowledge and behave intelligently like a human being.

The different kinds of knowledge that need to be represented in AI include:

- **Object:** All the facts about objects in our world domain. E.g., Guitars contains strings, trumpets are brass instruments.
- **Events:** Events are the actions which occur in our world.
- **Performance:** It describe behavior which involves knowledge about how to do things.
- **Meta-knowledge:** It is knowledge about what we know.
- **Facts:** Facts are the truths about the real world and what we represent.
- **Knowledge-Base:** The central component of the knowledge-based agents is the knowledge base. It is represented as KB. The Knowledgebase is a group of the Sentences (Here, sentences are used as a technical term and not identical with the English language).

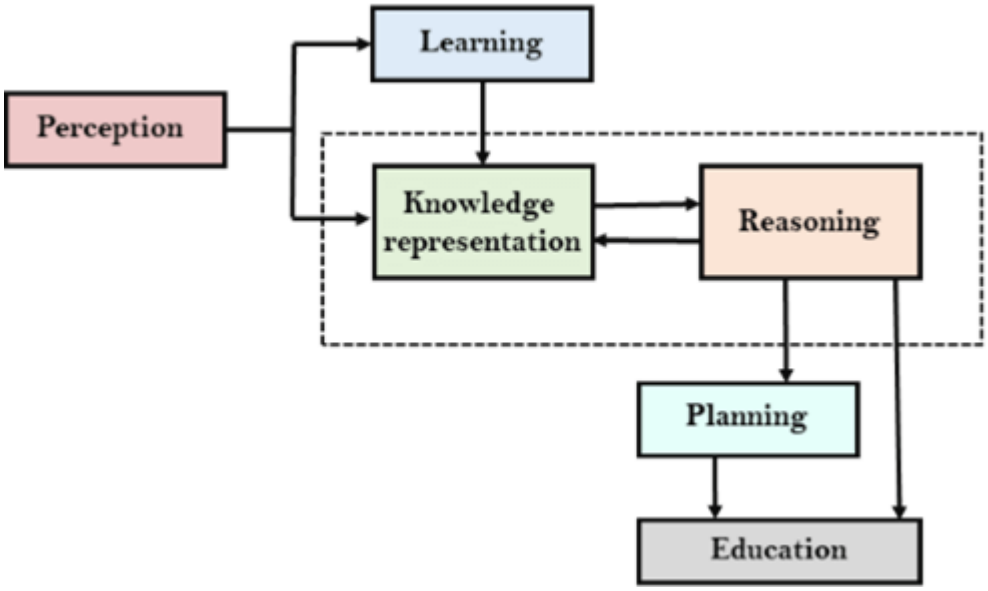
Different Types of Knowledge

There are 5 types of Knowledge such as:

AI knowledge cycle:

An Artificial intelligence system has the following components for displaying intelligent behavior:

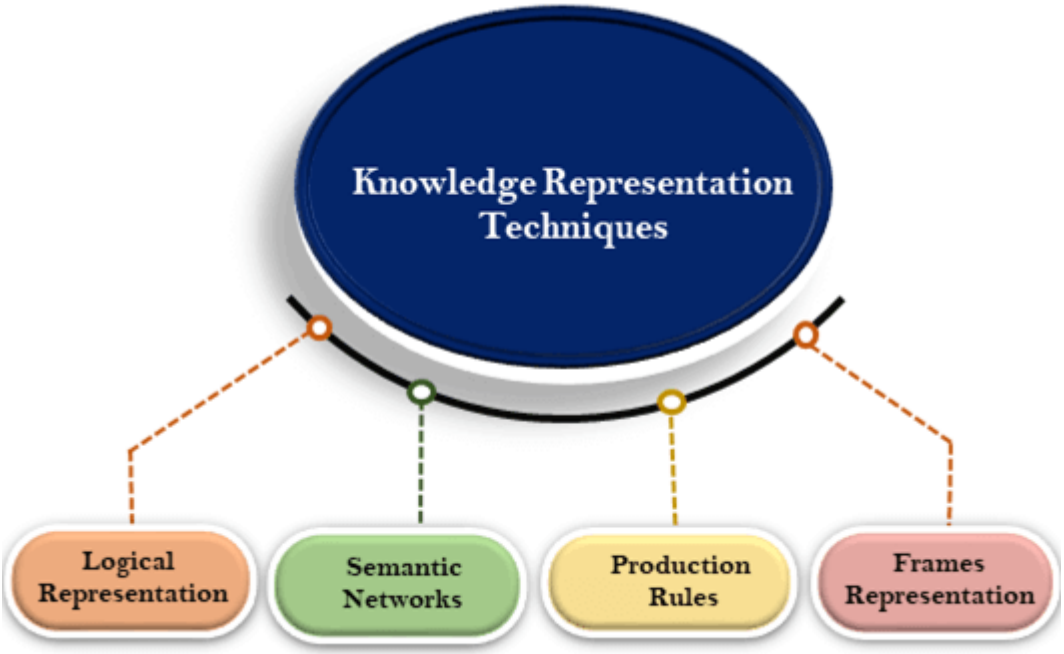
- Perception
- Learning
- Knowledge Representation and Reasoning
- Planning
- Execution



Techniques of knowledge representation

There are mainly four ways of knowledge representation which are given as follows:

1. Logical Representation
2. Semantic Network Representation
3. Frame Representation
4. Production Rules



1. Logical Representation

Logical representation is a language with some concrete rules which deals with propositions and has no ambiguity in representation. Logical representation means drawing a conclusion based on various conditions. This representation lays down some important communication rules. It consists of precisely defined syntax and semantics which supports the sound inference. Each sentence can be translated into logics using syntax and semantics.

Syntax:

- Syntaxes are the rules which decide how we can construct legal sentences in the logic.
- It determines which symbol we can use in knowledge representation.
- How to write those symbols.

Semantics:

- Semantics are the rules by which we can interpret the sentence in the logic.
- Semantic also involves assigning a meaning to each sentence.

Logical representation can be categorised into mainly two logics:

a. Propositional Logics
b. Predicate logics

Advantages of logical representation:

1. Logical representation enables us to do logical reasoning.
2. Logical representation is the basis for the programming languages.

Disadvantages of logical Representation:

1. Logical representations have some restrictions and are challenging to work with.
2. Logical representation technique may not be very natural, and inference may not be so efficient.

2. Semantic Network Representation

Semantic networks are alternative of predicate logic for knowledge representation. In Semantic networks, we can represent our knowledge in the form of graphical networks. This network consists of nodes representing objects and arcs which describe the relationship between those objects. Semantic networks can categorize the object in different forms and can also link those objects. Semantic networks are easy to understand and can be easily extended.

This representation consist of mainly two types of relations:

a. IS-A relation (Inheritance)
b. Kind-of-relation

Example: Following are some statements which we need to represent in the form of nodes and arcs.

Statements:

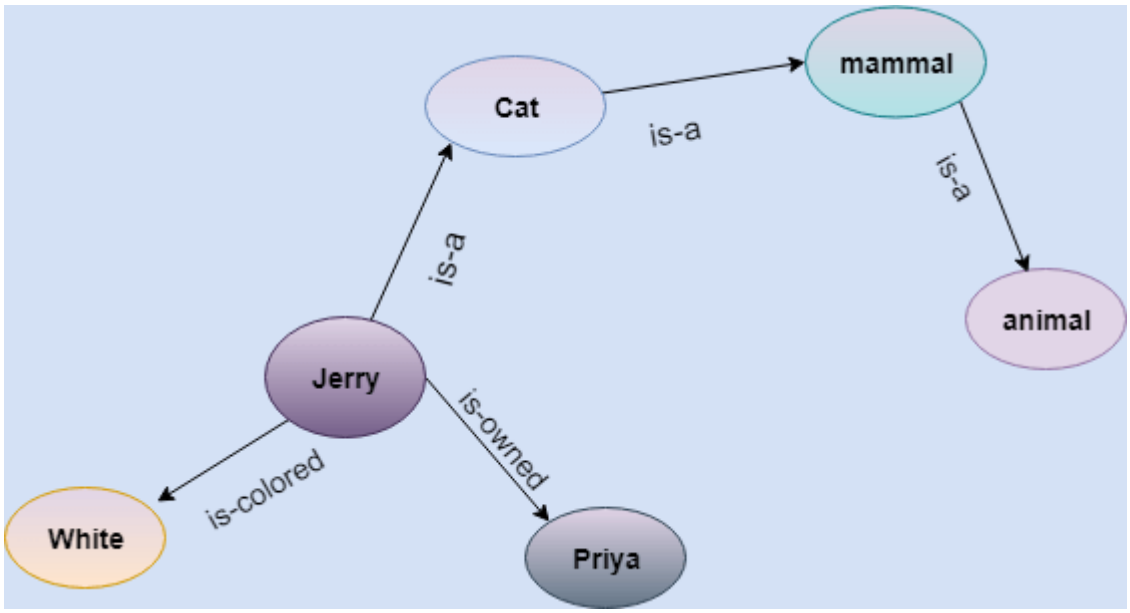
a. Jerry is a cat.

b. Jerry is a mammal

c. Jerry is owned by Priya.

d. Jerry is brown colored.

e. All Mammals are animal.



In the above diagram, we have represented the different type of knowledge in the form of nodes and arcs. Each object is connected with another object by some relation.

Drawbacks in Semantic representation:

1. Semantic networks take more computational time at runtime as we need to traverse the complete network tree to answer some questions. It might be possible in the worst case scenario that after traversing the entire tree, we find that the solution does not exist in this network.
2. Semantic networks try to model human-like memory (Which has 1015 neurons and links) to store the information, but in practice, it is not possible to build such a vast semantic network.

3. These types of representations are inadequate as they do not have any equivalent quantifier, e.g., for all, for some, none, etc.
4. Semantic networks do not have any standard definition for the link names.
5. These networks are not intelligent and depend on the creator of the system.

Advantages of Semantic network:

1. Semantic networks are a natural representation of knowledge.
2. Semantic networks convey meaning in a transparent manner.
3. These networks are simple and easily understandable.

3. Frame Representation

A frame is a record like structure which consists of a collection of attributes and its values to describe an entity in the world. Frames are the AI data structure which divides knowledge into substructures by representing stereotypes situations. It consists of a collection of slots and slot values. These slots may be of any type and sizes. Slots have names and values which are called facets.

Facets: The various aspects of a slot is known as **Facets**. Facets are features of frames which enable us to put constraints on the frames. Example: IF-NEEDED facts are called when data of any particular slot is needed. A frame may consist of any number of slots, and a slot may include any number of facets and facets may have any number of values. A frame is also known as **slot-filter knowledge representation** in artificial intelligence.

Frames are derived from semantic networks and later evolved into our modern-day classes and objects. A single frame is not much useful. Frames system consists of a collection of frames which are connected. In the frame, knowledge about an object or event can be stored together in the knowledge base. The frame is a type of technology which is widely used in various applications including Natural language processing and machine visions.

Example: 1

Let's take an example of a frame for a book

Slots	Filters
Title	Artificial Intelligence
Genre	Computer Science
Author	Peter Norvig
Edition	Third Edition
Year	1996
Page	1152

Example 2:

Let's suppose we are taking an entity, Peter. Peter is an engineer as a profession, and his age is 25, he lives in city London, and the country is England. So following is the frame representation for this:

Slots	Filter
Name	Peter
Profession	Doctor
Age	25

Marital status	Single
Weight	78

Advantages of frame representation:

1. The frame knowledge representation makes the programming easier by grouping the related data.
2. The frame representation is comparably flexible and used by many applications in AI.
3. It is very easy to add slots for new attribute and relations.
4. It is easy to include default data and to search for missing values.
5. Frame representation is easy to understand and visualize.

Disadvantages of frame representation:

1. In frame system inference mechanism is not be easily processed.
2. Inference mechanism cannot be smoothly proceeded by frame representation.
3. Frame representation has a much generalized approach.

4. Production Rules

Production rules system consist of (**condition, action**) pairs which mean, "If condition then action". It has mainly three parts:

- The set of production rules
- Working Memory
- The recognize-act-cycle

In production rules agent checks for the condition and if the condition exists then production rule fires and corresponding action is carried out. The condition part of the rule determines which rule may be applied to a problem. And the action part carries out the associated problem-solving steps. This complete process is called a recognize-act cycle.

The working memory contains the description of the current state of problems-solving and rule can write knowledge to the working memory. This knowledge match and may fire other rules.

If there is a new situation (state) generates, then multiple production rules will be fired together, this is called conflict set. In this situation, the agent needs to select a rule from these sets, and it is called a conflict resolution.

Example:

- **IF (at bus stop AND bus arrives) THEN action (get into the bus)**
- **IF (on the bus AND paid AND empty seat) THEN action (sit down).**
- **IF (on bus AND unpaid) THEN action (pay charges).**
- **IF (bus arrives at destination) THEN action (get down from the bus).**

Advantages of Production rule:

1. The production rules are expressed in natural language.
2. The production rules are highly modular, so we can easily remove, add or modify an individual rule.

Disadvantages of Production rule:

1. Production rule system does not exhibit any learning capabilities, as it does not store the result of the problem for the future uses.
2. During the execution of the program, many rules may be active hence rule-based production systems are inefficient.

Propositional logic in Artificial intelligence

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Following are some basic facts about propositional logic:

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- A proposition formula which is always false is called **Contradiction**.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

Negation: A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.

Conjunction: A sentence which has ∧ connective such as, **P ∧ Q** is called a conjunction.
Example: Rohan is intelligent and hardworking. It can be written as,
P=Rohan is intelligent,
Q= Rohan is hardworking. → P∧ Q.

Disjunction: A sentence which has ∨ connective, such as **P ∨ Q**. is called disjunction, where P and Q are the propositions.
Example: "Ritika is a doctor or Engineer",
Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as **P ∨ Q**.

Implication: A sentence such as P → Q, is called an implication.
Implications are also known as if-then rules. It can be represented as

If it is raining, then the street is wet.

Let P= It is raining, and Q= Street is wet, so it is represented as P → Q

Biconditional: A sentence such as **P⇔ Q is a Biconditional sentence, example If I am breathing, then I am alive**

P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

Rules of Inference in Artificial intelligence

Inference:

In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, **so generating the conclusions from evidence and facts is termed as Inference**.

Types of Inference rules:

1. Modus Ponens:

The Modus Ponens rule is one of the most important rules of inference, and it states that if P and P → Q is true, then we can infer that Q will be true. It can be represented as:

Notation for Modus ponens: $\frac{P \rightarrow Q, \quad P}{\therefore Q}$

Example:

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q

Statement-2: "I am sleepy" ==> P

Conclusion: "I go to bed." ==> Q.

Hence, we can say that, if P→ Q is true and P is true then Q will be true.

2. Modus Tollens:

The Modus Tollens rule state that if P→ Q is true and **¬ Q is true, then ¬ P** will also true. It can be represented as:

Notation for Modus Tollens: $\frac{P \rightarrow Q, \quad \sim Q}{\sim P}$

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q

Statement-2: "I do not go to the bed."==> ~Q

Statement-3: Which infers that "**I am not sleepy**" => ~P

3. Hypothetical Syllogism:

The Hypothetical Syllogism rule state that if P→R is true whenever P→Q is true, and Q→R is true. It can be represented as the following notation:

Example:

Statement-1: If you have my home key then you can unlock my home. **P→Q**
Statement-2: If you can unlock my home then you can take my money. **Q→R**
Conclusion: If you have my home key then you can take my money. **P→R**

4. Disjunctive Syllogism:

The Disjunctive syllogism rule state that if P∨Q is true, and ¬P is true, then Q will be true. It can be represented as:

Notation of Disjunctive syllogism: $\frac{P \vee Q,\ \neg P}{Q}$

Example:

Statement-1: Today is Sunday or Monday. ==>P∨Q

Statement-2: Today is not Sunday. ==> ¬P

Conclusion: Today is Monday. ==> Q

5. Addition:

The Addition rule is one the common inference rule, and it states that If P is true, then P∨Q will be true.

Notation of Addition: $\frac{P}{P \vee Q}$

Statement: I have a vanilla ice-cream. ==> P

Statement-2: I have Chocolate ice-cream.

Conclusion: I have vanilla or chocolate ice-cream. ==> (P∨Q)

6. Simplification:

The simplification rule state that if **P∧ Q** is true, then **Q or P** will also be true. It can be represented as:

Notation of Simplification rule: $\frac{P \wedge Q}{Q}$ Or $\frac{P \wedge Q}{P}$

Conceptual Dependency

Conceptual dependency theory is a model of natural language understanding used in artificial intelligence systems.

Conceptual Dependency theory is based on the use of limited number of primitive concepts and rules of formation to represent any natural language statement. Sentences are represented as a series of diagrams depicting actions using both abstract and real physical situations.

•The agent and the objects are represented.

•The actions are built up from a set of primitive acts which can be modified by tense.

•Building blocks – entities, actions, conceptual cases, conceptual dependencies and conceptual tenses.

Roger Schank developed the model to represent knowledge for natural language input into computers. Partly influenced by the work of Sydney Lamb, his goal was to make the meaning independent of the words used in the input, i.e. two sentences identical in meaning, would have a single representation. The system was also intended to draw logical inferences.

The model uses the following basic representational tokens:[3]

- *real world objects*, each with some *attributes*.
- *real world actions*, each with attributes
- *times*
- *locations*

A set of *conceptual transitions* then act on this representation, e.g. an ATRANS is used to represent a transfer such as "give" or "take" while a PTRANS is used to act on locations such as "move" or "go". An MTRANS represents mental acts such as "tell", etc.

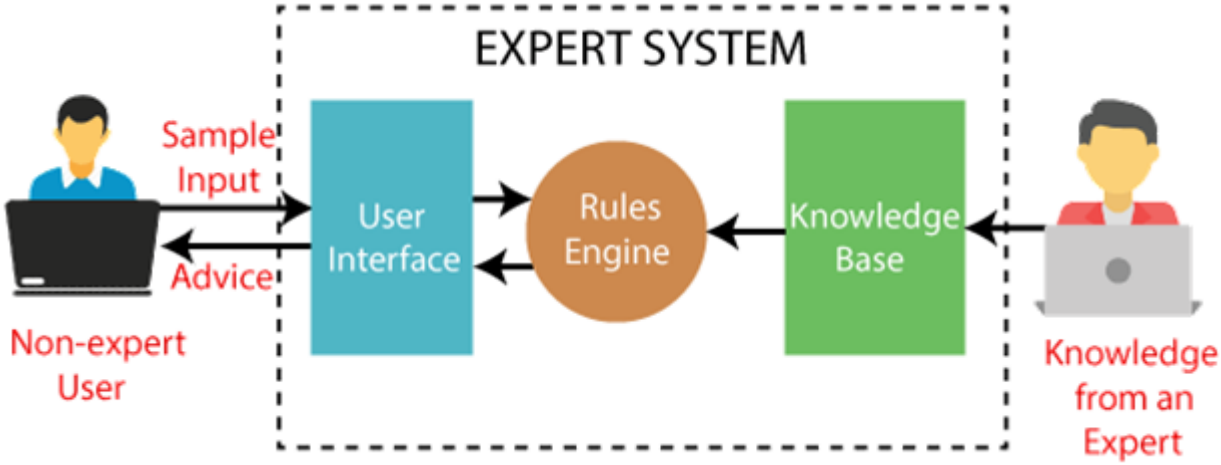
Expert System

An expert system is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert. It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries.

The expert system is a part of AI, and the first ES was developed in the year 1970, which was the first successful approach of artificial intelligence. It solves the most complex issue as an expert by extracting the knowledge stored in its knowledge base. The system helps in decision making for complex problems using **both facts and heuristics like a human expert**. It is called so because it contains the expert knowledge of a specific domain and can solve any complex problem of that particular domain. These systems are designed for a specific domain, such as **medicine, science,** etc.

The performance of an expert system is based on the expert's knowledge stored in its knowledge base. The more knowledge stored in the KB, the more that system improves its performance. One of the common examples of an ES is a suggestion of spelling errors while typing in the Google search box.

Below is the block diagram that represents the working of an expert system:



Components of Expert System

An expert system mainly consists of three components:

- **User Interface**
- **Inference Engine**
- **Knowledge Base**

1. User Interface

With the help of a user interface, the expert system interacts with the user, takes queries as an input in a readable format, and passes it to the inference engine. After getting the response from the inference engine, it displays the output to the user. In other words, **it is an interface that helps a non-expert user to communicate with the expert system to find a solution**.

2. Inference Engine(Rules of Engine)

- The inference engine is known as the brain of the expert system as it is the main processing unit of the system. It applies inference rules to the knowledge base to derive a conclusion or deduce new information. It helps in deriving an error-free solution of queries asked by the user.
- With the help of an inference engine, the system extracts the knowledge from the knowledge base.
- There are two types of inference engine:
- **Deterministic Inference engine:** The conclusions drawn from this type of inference engine are assumed to be true. It is based on **facts** and **rules**.
- **Probabilistic Inference engine:** This type of inference engine contains uncertainty in conclusions, and based on the probability.

Inference engine uses the below modes to derive the solutions:

- **Forward Chaining:** It starts from the known facts and rules, and applies the inference rules to add their conclusion to the known facts.
- **Backward Chaining:** It is a backward reasoning method that starts from the goal and works backward to prove the known facts.

3. Knowledge Base

- The knowledgebase is a type of storage that stores knowledge acquired from the different experts of the particular domain. It is considered as big storage of knowledge. The more the knowledge base, the more precise will be the Expert System.
- It is similar to a database that contains information and rules of a particular domain or subject.
- One can also view the knowledge base as collections of objects and their attributes. Such as a Lion is an object and its attributes are it is a mammal, it is not a domestic animal, etc.

Components of Knowledge Base

- **Factual Knowledge:** The knowledge which is based on facts and accepted by knowledge engineers comes under factual knowledge.
- **Heuristic Knowledge:** This knowledge is based on practice, the ability to guess, evaluation, and experiences.

Participants in the development of Expert System

There are three primary participants in the building of Expert System:

1. **Expert:** The success of an ES much depends on the knowledge provided by human experts. These experts are those persons who are specialized in that specific domain.
2. **Knowledge Engineer:** Knowledge engineer is the person who gathers the knowledge from the domain experts and then codifies that knowledge to the system according to the formalism.
3. **End-User:** This is a particular person or a group of people who may not be experts, and working on the expert system needs the solution or advice for his queries, which are complex.

Handling uncertainty in knowledge is a crucial aspect of learning and decision-making. As humans, we encounter situations where we lack complete information or face ambiguity. Here are some strategies to navigate uncertainty:

1. **Acknowledge the Gap**: Recognize when you don't have all the facts or when there's uncertainty. It's okay not to know everything.

2. **Seek Additional Information**: Gather more data or research to fill the gaps. Consult reliable sources, experts, or conduct experiments if possible.
3. **Quantify Uncertainty**: Use probabilities or confidence intervals to express uncertainty. For instance, saying “there’s a 70% chance of rain” acknowledges uncertainty.
4. **Bayesian Thinking**: Update your beliefs based on new evidence. Bayesian probability allows you to adjust your confidence levels as you acquire more information.
5. **Scenario Planning**: Consider multiple possible outcomes and plan accordingly. Prepare for different scenarios rather than relying on a single prediction.
6. **Risk Management**: Assess potential risks and benefits. Make decisions based on the balance between uncertainty and potential consequences.
7. **Learn from Mistakes**: Embrace uncertainty as an opportunity for growth. Learn from failures and adapt your approach.

Remember that uncertainty is a natural part of life, and our ability to handle it effectively contributes to our resilience and decision-making skills.

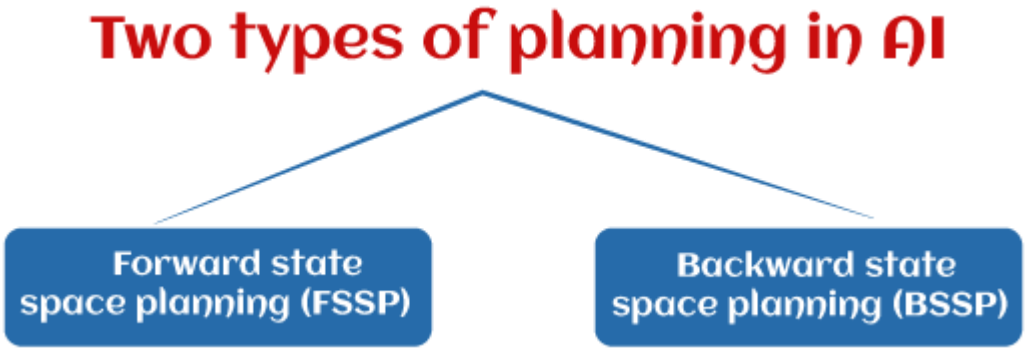
Planning in Artificial Intelligence

Artificial intelligence is an important technology in the future. Whether it is intelligent robots, self-driving cars, or smart cities, they will all use different aspects of artificial intelligence!!! But Planning is very important to make any such AI project.

Even Planning is an important part of Artificial Intelligence which deals with the tasks and domains of a particular problem. Planning is considered the logical side of acting.

Everything we humans do is with a definite goal in mind, and all our actions are oriented towards achieving our goal. Similarly, Planning is also done for Artificial Intelligence.

We have **Forward State Space Planning (FSSP)** and **Backward State Space Planning (BSSP)** at the basic level.



1. Forward State Space Planning (FSSP)

FSSP behaves in the same way as forwarding state-space search. It says that given an initial state S in any domain, we perform some necessary actions and obtain a new state S' (which also contains some new terms), called a progression. It continues until we reach the target position. Action should be taken in this matter.

2. Backward State Space Planning (BSSP)

BSSP behaves similarly to backward state-space search. In this, we move from the target state g to the sub-goal g, tracing the previous action to achieve that goal. This process is called regression (going back to the previous goal or sub-goal). These sub-goals should also be checked for consistency. The action should be relevant in this case.

The key components of a planning system include **mission, objectives, policies, procedures, budget, programme, and strategies**. Other components of a planning system include:

- Initial state: The starting conditions or the current state of the world or system.
- Goal: The desired state or the outcome that the planning system aims to achieve.
- Actions: The set of actions that can be taken to achieve the goal.
- Transition model: The model that describes how the world or system changes as a result of taking an action.
- Search algorithm: The algorithm that searches for a plan that achieves the goal.
- Plan representation: The representation of the plan that the planning system generates.
- Plan execution and monitoring: The execution and monitoring of the plan that the planning system generates.

linear and **non-linear planning** in the context of **Artificial Intelligence (AI)**.

Linear Planning

Linear planning, also known as **Goal Stack Planning**, involves a problem-solving approach where the problem solver maintains a single stack containing operators and the goal state. Here's how it works:

1. The problem solver constructs a sequence of operators to achieve the first goal.
2. As the plan progresses, additional operators are added to the stack to achieve subsequent goals.
3. The resulting plan is a linear sequence of complete sub-plans.

In summary, linear planning follows a straightforward, step-by-step approach where each action builds upon the previous ones to reach the ultimate goal.

Non-Linear Planning

Now, let's explore **non-linear planning**:

1. **Non-linear planning** breaks down the problem into sub-problems that are interdependent. Unlike linear planning, where actions

follow a strict sequence, non-linear planning allows for partial ordering of actions.
2. In a non-linear plan, operators are not arranged in a specific order. Instead, they form a partial sequence.
3. These sub-problems interact with each other, and their execution order is not fixed.

Hierarchical Planning

Another interesting concept related to planning is **hierarchical planning**:

- In complex problems, reaching the goal state from the initial state can be challenging.
- Hierarchical planning involves eliminating some problem details until a solution is found. These details are not removed from the actual description of operators but are deferred.
- Once a high-level solution is obtained, the missing details are filled in.

In summary, non-linear planning provides flexibility, allowing AI systems to handle dynamic and uncertain environments effectively. It's like solving a puzzle where pieces fit together in a non-linear fashion, leading to a coherent solution.

Goal Stack Planning is an intriguing method in artificial intelligence where we work **backward** from the **goal state** to the **initial state**.

Blocks World Problem:

- Imagine a table with several blocks placed on it. Some blocks may be stacked on top of others, and we have a robot arm capable of picking up or putting down these blocks.
- The robot arm can move only one block at a time, and it must ensure that no other block is stacked on top of the one it intends to move.
- Our objective is to transform the configuration of blocks from the **initial state** to the **goal state**.

STRIPS

STRIPS (STanford Research Institute Problem Solver) is an automated planning technique that plays a crucial role in artificial intelligence (AI).

What is STRIPS?

- STRIPS is an acronym for **"STanford Research Institute Planning System."** Developed by the Stanford AI Lab in the early 1970s, it was initially designed for use with a robotic arm. However, its applications extend beyond robotics to various other planning problems.
- At its core, STRIPS aims to find a solution by executing a **domain** (which describes the world) and a **problem** (which defines the initial state and goal condition).

2. **How Does STRIPS Work?**
   - **Describing the World:**
     - In STRIPS, you start by describing the world using several components:
       - **Objects:** These represent entities in the game world (e.g., ogres, trolls, dragons, magical items).
       - **Actions:** Specify what can be done (e.g., picking up items, building weapons).
       - **Preconditions:** Conditions that must be met before an action can be executed.
       - **Effects:** Changes that occur after an action is performed.
     - This description sets the stage for the planning process.
   - **Creating a Problem Set:**
     - A problem consists of an **initial state** (where things begin) and a **goal condition** (what you want to achieve).
     - STRIPS then searches through all possible states, starting from the initial state, and executes various actions until it reaches the goal.
3. **Using PDDL: Planning Domain Definition Language**
   - **PDDL (Planning Domain Definition Language)** is a common language for writing STRIPS domain and problem sets.
   - It allows you to express most of the code using English words, making it readable and understandable.
   - Writing simple AI planning problems using PDDL is relatively straightforward.
4. **What Can STRIPS Solve?**
   - A wide range of problems can be tackled using STRIPS and PDDL:
     - **Stacking Blocks:** Imagine arranging blocks in a specific order.
     - **Rubik's Cube:** Solving the classic puzzle.
     - **Navigating a Robot:** In scenarios like Shakey's World.
     - **Starcraft Build Orders:** Planning optimal strategies.
     - And much more!

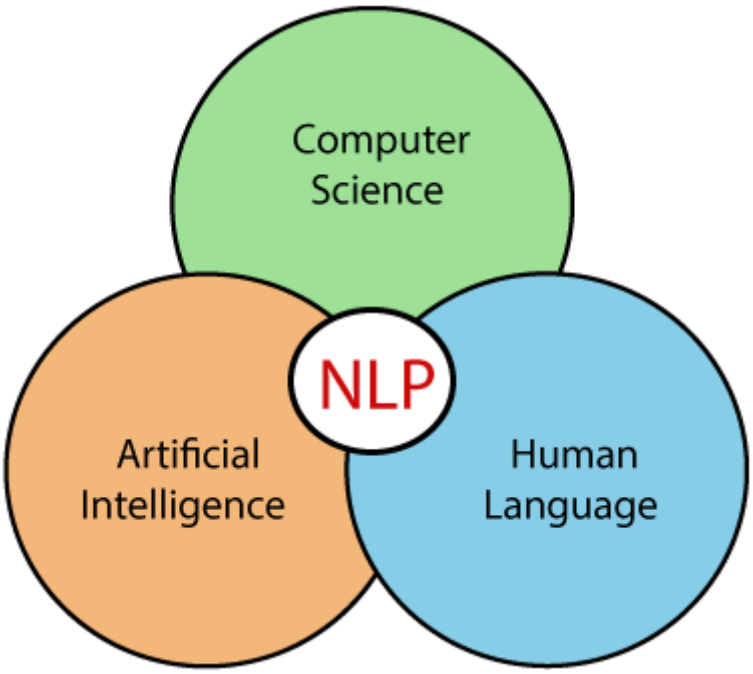
Partial-order planning is an approach to automated planning that maintains a **partial ordering** between actions and only commits to ordering between actions when forced to do so. In other words, it doesn't specify which action will come out first when two actions are processed. Let's delve into the details:

Partial-Order Planning (POP):

- POP, also known as **Partial-Order Causal Linking (POCL)**, follows a **least-commitment philosophy**. It postpones decisions about action orderings and parameter bindings until a decision becomes necessary.

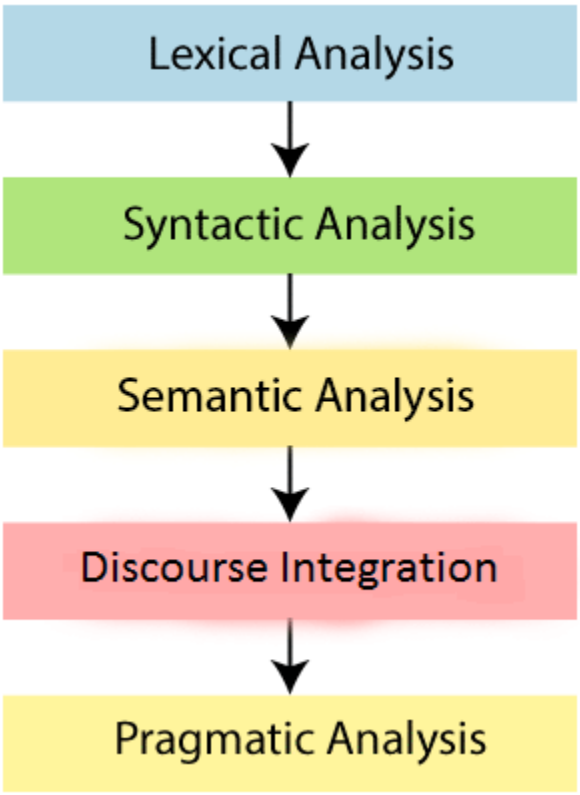
What is NLP?

NLP stands for **Natural Language Processing**, which is a part of **Computer Science, Human language,** and **Artificial Intelligence**. It is the technology that is used by machines to understand, analyse, manipulate, and interpret human's languages. It helps developers to organize knowledge for performing tasks such as **translation, automatic summarization, Named Entity Recognition (NER), speech recognition, relationship extraction,** and **topic segmentation**.



Phases of NLP

There are the following five phases of NLP:


Lexical Analysis
Syntactic Analysis
Semantic Analysis
Discourse Integration
Pragmatic Analysis

Parsing techniques in NLP :

- Top-Down and Bottom-Up parsing techniques
- Tokenization: the process of breaking text into individual words or phrases
- Part-of-speech tagging: the process of labelling each word in a sentence with its grammatical part of speech
- Stemming: a technique that comes from morphology and information retrieval which is used in natural language processing for pre-processing and efficiency purposes
- Text Segmentation
- Named Entity Recognition
- Relationship Extraction
- Sentiment Analysis

parsing techniques in the context of **compiler design**. Parsing is a crucial phase in the compilation process, where a token string (usually generated by lexical analysis) is transformed into an **Intermediate Representation (IR)** based on the given grammar. The parser, also known as the **Syntax Analyzer**, plays a pivotal role in this process.

Here are the primary types of parsers:

1. **Top-Down Parser**:
    - The top-down parser constructs the parse tree by **expanding non-terminals** using grammar productions. It starts from the **start symbol** and proceeds towards the terminals.
    - Two subtypes of top-down parsers are:
        - **Recursive Descent Parser**: Also known as the **Brute Force** or **Backtracking** parser, it generates the parse tree using brute force and backtracking.
        - **Non-Recursive Descent Parser (LL(1))**: This parser employs a **parsing table** to generate the parse tree without backtracking.
    - **Useful for**: Simple grammars and LL(1) languages.
2. **Bottom-Up Parser**:
    - The bottom-up parser constructs the parse tree by **compressing terminals**. It starts from the terminals and works its way up to the start symbol.
    - Two subtypes of bottom-up parsers are:
        - **LR Parser**: Generates the parse tree using unambiguous grammar. It has four variants: LR(0), SLR(1), LALR(1), and CLR(1).
        - **Operator Precedence Parser**: Constructs the parse tree based on operator grammars, where consecutive non-terminals do not appear without any terminal in between.
    - **Useful for**: Handling complex language constructs.

Semantic Analysis and **Pragmatics** in natural language.

1. **Semantic Analysis**:

- **Definition**: Semantic Analysis is a subfield of **Natural Language Processing (NLP)** that aims to understand the meaning of natural language. While humans often find understanding language straightforward, it's a complex task for machines due to the vast complexity and subjectivity inherent in human communication.
- **Focus**: Semantic Analysis captures the meaning of given text by considering context, logical sentence structuring, and grammar roles.
- **Parts**:
  - **Lexical Semantic Analysis**: This involves understanding the meaning of individual words in the text. It essentially fetches the dictionary meaning associated with each word.
  - **Compositional Semantics Analysis**: Knowing the meaning of individual words isn't enough to fully grasp the text's overall meaning. Compositional Semantics Analysis explores how combinations of words form the text's meaning. For instance, consider these two sentences:
    - Sentence 1: "Students love GeeksforGeeks."
    - Sentence 2: "GeeksforGeeks loves Students."
    - Although both sentences use the same root words ("student," "love," "GeeksforGeeks"), they convey entirely different meanings. Compositional Semantics Analysis helps us understand how word combinations contribute to overall meaning.

2. **Pragmatics**:
   - **Definition**: Pragmatics also deals with language meaning but takes a practical approach. It considers how language is used and how different usages create multiple meanings.
   - **Key Difference**:
     - **Semantics**: Focuses on words, phrases, and sentences in a literal sense.
     - **Pragmatics**: Goes beyond literal meaning by considering intended meaning and context.
   - **Context Dependency**: Pragmatics is context-dependent, whereas semantics is context-independent.

Multi-agent system

Defining Multi-Agent Systems

A multi-agent system is a collection of autonomous agents that interact with each other and their environment to achieve individual or collective goals. These agents can be physical entities, such as robots, or purely software-based, like computer programs.

- Multi-Agent Systems (MAS) consist of autonomous agents that interact with each other and their environment to achieve individual or collective goals.
- MAS are characterized by autonomy, interaction, goal orientation, and distributed control, distinguishing them from single-agent and distributed systems.
- The components of MAS include agents, the environment, organization, and interaction, each playing a crucial role in shaping the system's behaviour.
- Three key use cases of MAS include Swarm Robotics, Automated Bidding Systems in E-commerce, and Wildlife Tracking for Environmental Monitoring, each presenting unique challenges and solutions.
- Implementing MAS requires addressing the Agent Design Problem to equip agents with decision-making and adaptability tools, as well as the Society Design Problem to enable interaction and coordination among agents.
- Multi-Agent Systems offer a promising approach to solving complex problems and understanding collective behaviours and intelligence in an interconnected world.

Components of Multi-Agent Systems

There are 4 crucial components that make up the MAS: Agents, the environment, Organisation and interaction.

Types of Agents

Agents can be grouped into five classes based on their degree of perceived intelligence and capability:

- Simple Reflex Agents
- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents
- Learning Agent
- Multi-agent systems
- Hierarchical agents

Agents Vs Objects •

Agents natural extension\evolution of Objects but with some very fundamental differences

– Level of Autonomy

– Stronger design metaphor

– High Level Interactions

– Supporting Organizational structure

– Proactively

– Separate Thread of Execution

– Independent life span

Agents Vs Objects •

It is about adding new abstraction entities: – OOP = structured programming + objects that have persistent local states

– AOP = OOP + agents that have an independent execution thread + pro-activity + greater level of autonomy over itself •

An agent is able to act in a goal-directed fashion rather than just passively reacting to procedure calls – "An agent can say no!"

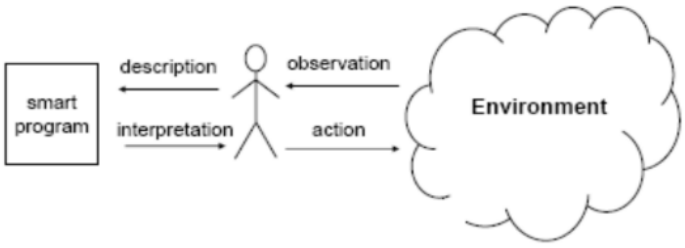
Agents and Expert Systems.

1. **Expert Systems (ES)**:
    - Expert systems are computer applications designed to tackle complex problems within specific domains, exhibiting intelligence comparable to extraordinary human expertise.
    - **Characteristics of Expert Systems**:
        - **High Performance**: They excel in solving intricate tasks.
        - **Understandable**: Their reasoning process is transparent and comprehensible.
        - **Reliable**: They provide consistent and dependable results.
        - **Highly Responsive**: They swiftly respond to queries and provide solutions.
    - **Capabilities of Expert Systems**:
        - Advising
        - Instructing and assisting in decision-making
        - Demonstrating solutions

        - Diagnosing issues
        - Explaining reasoning
        - Interpreting input
        - Predicting outcomes
        - Justifying conclusions
        - Suggesting alternative options
    - **Components of Expert Systems**:
        - **Knowledge Base**: Contains domain-specific, high-quality knowledge. It combines factual and heuristic knowledge.
            - **Factual Knowledge**: Accepted information in the task domain.
            - **Heuristic Knowledge**: Practical judgment, evaluation abilities, and guessing.
            - **Knowledge Representation**: Typically organized as IF-THEN-ELSE rules.
            - **Knowledge Acquisition**: Involves collecting accurate information from experts, scholars, and knowledge engineers.
        - **Inference Engine**: Manipulates knowledge from the knowledge base to arrive at solutions.
        - **User Interface**: Facilitates interaction with the expert system.

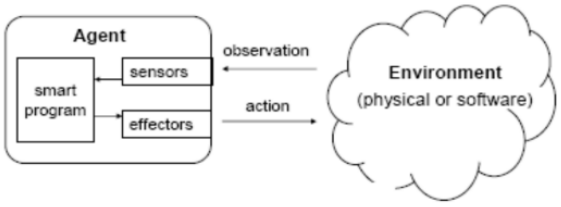
Intelligent Agents vs. Expert Systems

- In expert systems there is a human present between program environment



Intelligent Agents vs. Expert Systems

Agent reside in the Environment and interacts with it directly (no interface is required)



A multi-agent system (MAS) is a computerized system composed of multiple interacting intelligent agents. These agents can be software agents, robots, or even human agents. Let's delve into the key aspects of multi-agent systems:

1. Autonomy: Each agent in a MAS is at least partially independent and self-aware. They operate with a degree of autonomy, making decisions based on their own knowledge and goals.
2. Local Views: No agent has a complete global view of the system. Instead, each agent perceives a local view of its environment. The system's complexity often prevents any single agent from exploiting full global knowledge.
3. Decentralization: In a MAS, there is no designated controlling agent. Instead, the system operates in a decentralized manner, where agents interact and collaborate without a central authority. This decentralization can lead to emergent behaviours and robustness.

In summary, multi-agent systems allow for solving problems that are difficult or impossible for individual agents or monolithic systems to tackle. They find applications in diverse fields, including online trading, disaster response, target surveillance, and social structure modeling[1].

Semantic Web

Semantic Web is an extension to the World Wide Web. The purpose of the semantic web is to provide structure to the web and data in general. It emphasizes on representing a web of data instead of web of documents. It allows computers to intelligently search, combine and process the web content based on the meaning that the content has. Three main models of the semantic web are:

1. Building models
2. Computing with Knowledge
3. Exchanging Information

- **Building Models:**
  Model is a simplified version or description of certain aspects of the real-time entities. Model gathers information which is useful for the understanding of the particular domain.
- **Computing Knowledge:**
  Conclusions can be obtained from the knowledge present.
  Example: If two sentences are given as *'John is the son of Harry'* and another sentence given is- *'Hary's father is Joey'*, then the knowledge that can be computed from it is – *'John is the grandson of Joey'*
  Similarly, another example useful in the understanding of computing knowledge is-

'All A is B' and *'All B is C'*, then the conclusion that can be drawn from it is – *'All A are C'* respectively.

- **Exchanging Information:**
  It is an important aspect. Various communication protocols have been implemented for the exchange of information like the TCP/IP, HTML, WWW. Web Services have also been used for the exchange of the data.

Agent Communication:

- Agents need to communicate with each other.
- Communication can be:
  - **Point-to-Point**: Direct communication between two agents.
  - **Broadcast**: An agent sends information to a group of agents.
  - **Mediated**: Communication between two agents is facilitated by a third party.

OntoShare, a knowledge sharing system based on ontologies, offers a powerful approach to enhance collaboration and information management. Let's delve into the details:

1. **Semantic Web and Ontologies**:
   - The **Semantic Web** concept, championed by Tim Berners-Lee, aims to enrich information access by leveraging machine-processable metadata. Central to this vision are **ontologies**—formal conceptualizations of domains that facilitate knowledge sharing and re-use among agents, whether human or artificial[1].
   - **Ontologies** provide a consensual and structured representation of domain knowledge. They specify a hierarchy of concepts (ontological classes) to which users can assign information. In OntoShare, an ontology is used to create a shared understanding within a community[1].
2. **OntoShare: How It Works**:
   - **RDF (Resource Description Framework)** and **RDF(S)** are employed to define and populate the ontology in OntoShare.
   - Users in virtual communities contribute information, which is then associated with ontological concepts using **RDF annotations**.
   - The system semi-automatically builds an **RDF-annotated information resource** for the community, capturing evolving meanings and relationships between concepts over time[1].
3. **Advantages of OntoShare**:
   - **Knowledge Management**: OntoShare addresses the challenges posed by the exponential growth of information. It enables effective knowledge management by fostering

collaboration, understanding contextual knowledge, and representing organizational resources.
    - **Evolutionary Ontology**: OntoShare supports ontology evolution based on user interactions. As users share information and assign it to specific concepts, the ontology adapts dynamically.
4. **Future Directions**:
    - Ongoing research aims to refine OntoShare and evaluate its effectiveness.
    - The system's ability to capture evolving semantics and facilitate knowledge exchange holds promise for organizations and virtual communities.

Agent Development Tools:

a) BM-Aglet (Aglet Software Development Kit - ASDK):

- ASDK is an environment for developing mobile agents using Java.
- It's an open-source toolkit with the latest version being Aglet 2.5 alpha.

b) Voyager

Voyager [10, 1, 9] is an agent development tool developed by ObjectSpace, in mid1996. ObjectSpace has been taken over by Recursion Software Inc. since 2001 and it's now their commercial product.

c) JADE JADE (Java Agent DEvelopment Framework) [13, 3, 4, 6 , 5 , 17] is a software Framework fully implemented in Java language. It is developed by Tilab for the development of multi-agent applications based on peer-to-peer communication architecture.

d) Anchor Anchor [12] agent toolkit is developed by Lawerence Berkeley National Laboratory, U.S.A. it facilitates the transmission and secure management of mobile agents in a heterogeneous distributed environments. This toolkit is available in BSD style license.

e) Zeus Zeus [13, 7, 18] is an integrated environment for the rapid development of collaborative agent applications, developed by Advanced Applications & Technology Department of British Telecommunication labs

Fuzzy set:

The **'Fuzzy'** word means the things that are not clear or are vague. Sometimes, we cannot decide in real life that the given problem or statement is either true or false. At that time, this concept provides many values between the true and false and gives the flexibility to find the best solution to that problem.

Fuzzy set theory is a mathematical framework that extends classical set theory by allowing elements to have **degrees of membership** rather than a strict binary classification. Here are the key points:

1. **Classical Sets vs. Fuzzy Sets**:
    - In classical set theory, an element either **belongs** or **does not belong** to a set (bivalent condition).
    - In contrast, fuzzy set theory permits **gradual assessment** of membership. Elements can have partial membership based on a **membership function** that assigns a value between 0 and 1.
    - Fuzzy sets generalize classical sets, where the indicator functions of classical sets are special cases of the membership functions of fuzzy sets.
2. **Definition**:
    - A fuzzy set is defined by a **pair**: a **reference set** (universe of discourse) and a **membership function**.
    - The membership function assigns a value to each element in the reference set, representing the **grade of membership**.
    - For example, if we have a fuzzy set denoted as (A), the membership function (m(x)) describes how much element (x) belongs to (A).
3. **Membership Levels**:
    - An element can be:
        - **Not included** in the fuzzy set if (m(x) = 0) (no membership).
        - **Fully included** if (m(x) = 1) (full membership).
        - **Partially included** if (0 < m(x) < 1) (fuzzy membership).
4. **Applications**:
    - Fuzzy set theory is used in various domains where information is **incomplete or imprecise**, such as:
        - **Bioinformatics**
        - **Linguistics**
        - **Decision-making**
        - **Clustering**
5. **Example**:
    - Imagine a fuzzy set representing **“tall buildings.”** Instead of categorizing a building as tall or not tall, fuzzy set theory allows us to express the **degree of tallness** for each building.

Fuzzy sets provide a powerful way to handle imprecise information and vagueness, making them valuable in practical applications.

Notion of fuzziness:

The **notion of fuzziness** refers to the quality of being **unclear**, **vague**, or **imprecise**. In various contexts, we encounter situations where it's challenging to determine whether a state is **strictly true** or **completely false**. Fuzzy logic provides a valuable approach to reasoning in such scenarios, allowing us to consider **inaccuracies** and **uncertainties**.

Here are some key points about **fuzzy logic**:

1. **Definition**: Fuzzy logic is a form of **many-valued logic** where the truth values of variables can be any real number between **0 and 1**. Unlike traditional binary logic (true or false), fuzzy logic accommodates shades of gray in between.
2. **Applications**: Fuzzy logic finds applications in various fields, including:
    - **Control systems**: It's used to handle imprecise information in control processes.
    - **Image processing**: Fuzzy techniques enhance image analysis and feature extraction.
    - **Natural language processing**: Fuzzy logic aids in understanding and processing ambiguous language.
    - **Medical diagnosis**: It deals with uncertain medical data.
    - **Artificial intelligence**: Fuzzy systems model human reasoning.
3. **Membership Function**: The fundamental concept in fuzzy logic is the **membership function**. It maps an input value to a membership degree between 0 and 1, representing the degree of belonging to a certain set or category.
4. **Fuzzy Rules**: Fuzzy logic operates using **if-then rules** that express relationships between input and output variables in a fuzzy manner.
5. **Output**: The output of a fuzzy logic system is a **fuzzy set**, which provides membership degrees for each possible output value.

In summary, fuzzy logic allows for **partial truths** and is a mathematical method for handling vagueness and uncertainty in decision-making. It recognizes that the world isn't always black and white, but rather a spectrum of possibilities.

Fuzzification is the process of converting a crisp quantity into a fuzzy quantity. On the other hand, defuzzification is the process of translating a fuzzy quantity into a crisp quantity. Read this article to learn more about fuzzification and defuzzification and how they are different from each other.

What is Fuzzification?

Fuzzification may be defined as the process of transforming a crisp set to a fuzzy set or a fuzzy set to fuzzier set. Basically, this operation translates accurate crisp input values into linguistic variables. In a number of engineering applications, it is necessary to defuzzify the result or rather "fuzzy result" so that it must be converted to crisp result.

Fuzzification is done by recognizing various assumed crisp quantities as the non-deterministic and completely uncertain in nature. This uncertainty may be emerged

because of imprecision and uncertain that lead variables to be presented by a membership function because they can be fuzzy in nature.

Fuzzification translates the crisp input data into linguistic variables which are represented by fuzzy sets. After that, it applies the membership functions to measure and determine the degree of membership.

What is Defuzzification?

Defuzzification may be defined as the process of reducing a fuzzy set into a crisp set or to convert a fuzzy member into a crisp member. Mathematically, the process of Defuzzification is also called **"rounding it off"**. Defuzzification basically transforms an imprecise data into precise data. However, it is a relatively complex to implement defuzzification as compared to fuzzification.

Defuzzification is basically the reverse process of fuzzification because it converts the fuzzy data into crisp data. In some practical implementations, the defuzzification process is required for crisp control actions to operate the control.

Now, let us discuss the differences between fuzzification and defuzzification.

Difference between Fuzzification and Defuzzification

The following are the important difference between Fuzzification and Defuzzification −

Key	Fuzzification	Defuzzification
Definition	Fuzzification is the process of transforming a crisp set to a fuzzy set or a fuzzy set to fuzzier set.	Defuzzification is the process of reducing a fuzzy set into a crisp set or converting a fuzzy member into a crisp member.
Purpose	Fuzzification converts a precise data into imprecise data.	Defuzzification converts an imprecise data into precise data.
Example	Voltmeter.	Stepper motor, D/A converter.
Methods used	Inference, Rank ordering, Angular fuzzy sets, Neural network.	Maximum membership principle, Centroid method, Weighted average method, Center of sums.

Complexity	Fuzzification is easy.	Defuzzification is quite complex to implement.
Approach	Fuzzification uses if-then rules to fuzzify the crisp value.	Defuzzification uses center of gravity methods to get centroid of sets.

Conclusion

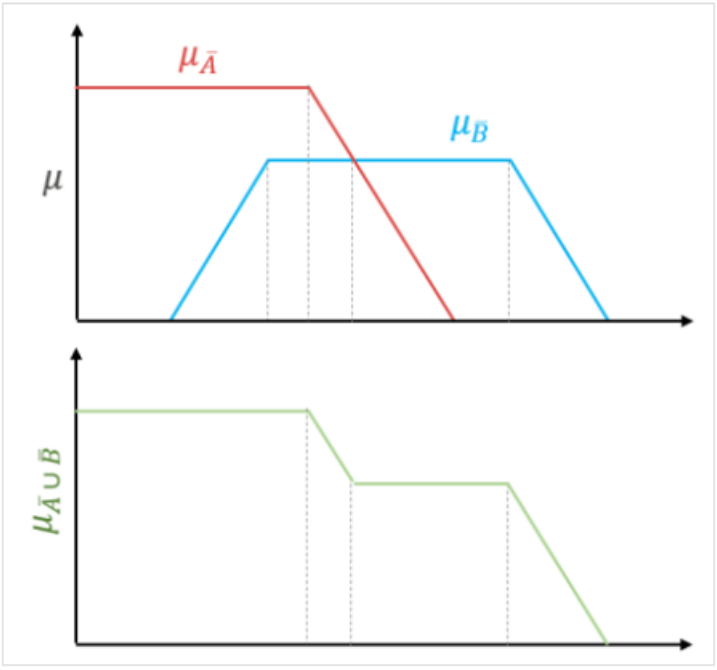
The most significant difference that you should note here is that fuzzification converts a precise data into imprecise data, while defuzzification converts an imprecise data into precise data.

Fuzzy operations – Explained with examples

Fuzzy operations are performed on fuzzy sets, whereas crisp operations are performed on crisp sets. Fuzzy operations are very useful in the design of a Fuzzy Logic Controller. It allows the manipulation of fuzzy variables by different means.

Union:

In the case of the union of crisp sets, we simply have to select repeated elements only once. In the case of fuzzy sets, when there are common elements in both fuzzy sets, we should select the element with the **maximum membership value**.
The **union** of two fuzzy sets A and B is a fuzzy set C, written as $C = A \cup B$
$C = A \cup B = \{(x, \mu_{A \cup B}(x)) \mid \forall x \in X\}$
$\mu C(x) = \mu_{A \cup B}(x) = \mu A(x) \vee \mu B(x)$
$= \max(\mu A(x), \mu B(x)), \forall x \in X$
Graphically, we can represent union operations as follows: Red and Blue membership functions represent the fuzzy value for elements in sets A and B, respectively. Wherever these fuzzy functions overlap, we have to consider the point with the maximum membership value.



Fuzzy union

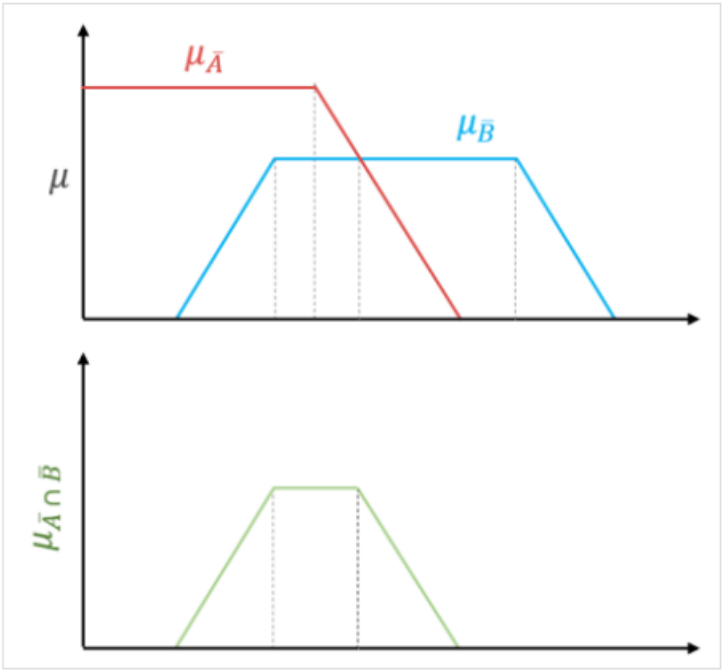
Example of Fuzzy Union:

$\underline{C} = \underline{A} \cup \underline{B} = \{(x, \mu_{\underline{A} \cup \underline{B}}(x)) \mid \forall x \in X\}$
$\underline{A} = \{ (x_1, 0.2), (x_2, 0.5), (x_3, 0.6), (x_4, 0.8), (x_5, 1.0) \}$
$\underline{B} = \{ (x_1, 0.8), (x_2, 0.6), (x_3, 0.4), (x_4, 0.2), (x_5, 0.1) \}$
$\mu_{\underline{A} \cup \underline{B}}(x_1) = \max(\mu_{\underline{A}}(x_1), \mu_{\underline{B}}(x_1)) = \max \{ 0.2, 0.8 \} = 0.8$
$\mu_{\underline{A} \cup \underline{B}}(x_2) = \max(\mu_{\underline{A}}(x_2), \mu_{\underline{B}}(x_2)) = \max \{ 0.5, 0.6 \} = 0.6$
$\mu_{\underline{A} \cup \underline{B}}(x_3) = \max(\mu_{\underline{A}}(x_3), \mu_{\underline{B}}(x_3)) = \max \{ 0.6, 0.4 \} = 0.6$
$\mu_{\underline{A} \cup \underline{B}}(x_4) = \max(\mu_{\underline{A}}(x_4), \mu_{\underline{B}}(x_4)) = \max \{ 0.8, 0.2 \} = 0.8$
$\mu_{\underline{A} \cup \underline{B}}(x_5) = \max(\mu_{\underline{A}}(x_5), \mu_{\underline{B}}(x_5)) = \max \{ 1.0, 0.1 \} = 1.0$
So, $\underline{A} \cup \underline{B} = \{ (x_1, 0.8), (x_2, 0.6), (x_3, 0.6), (x_4, 0.8), (x_5, 1.0) \}$

Intersection:

In the case of the intersection of crisp sets, we simply have to select common elements from both sets. In the case of fuzzy sets, when there are common elements in both fuzzy sets, we should select the element with **minimum membership value**.
The **intersection** of two fuzzy sets $\underline{A}$ and $\underline{B}$ is a fuzzy set $\underline{C}$, written as $\underline{C} = \underline{A} \cap \underline{B}$
$\underline{C} = \underline{A} \cap \underline{B} = \{(x, \mu_{\underline{A} \cap \underline{B}}(x)) \mid \forall x \in X\}$
$\mu_{\underline{C}}(x) = \mu_{\underline{A} \cap \underline{B}}(x) = \mu_{\underline{A}}(x) \wedge \mu_{\underline{B}}(x)$
$= \min(\mu_{\underline{A}}(x), \mu_{\underline{B}}(x)), \forall x \in X$

Graphically, we can represent the intersection operation as follows: Red and blue membership functions represent the fuzzy value for elements in sets A and B, respectively. Wherever these fuzzy functions overlap, we have to consider the point with the minimum membership value.



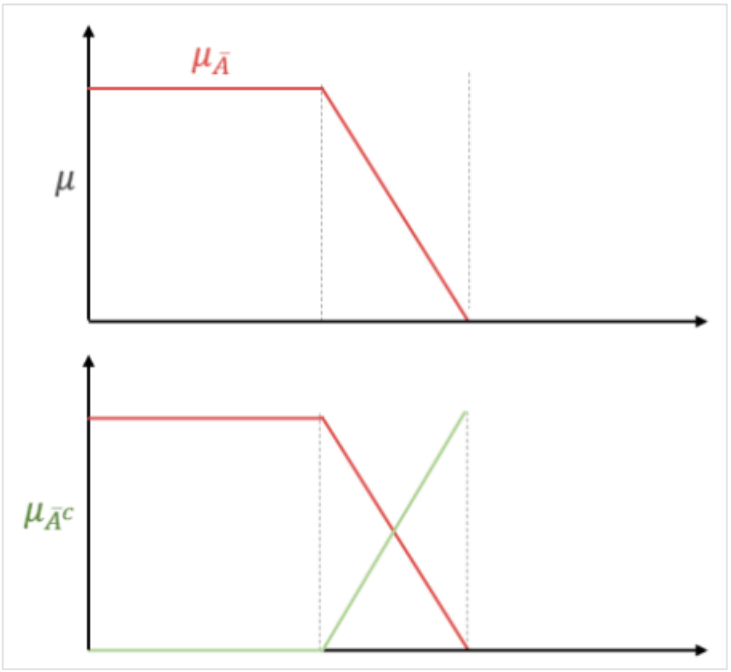
Fuzzy intersection

Example of Fuzzy Intersection:

$\underline{C} = \underline{A} \cap \underline{B} = \{(x, \mu_{\underline{A} \cap \underline{B}}(x)) \mid \forall x \in X\}$
$\underline{A} = \{ (x_1, 0.2), (x_2, 0.5), (x_3, 0.6), (x_4, 0.8), (x_5, 1.0) \}$
$\underline{B} = \{ (x_1, 0.8), (x_2, 0.6), (x_3, 0.4), (x_4, 0.2), (x_5, 0.1) \}$
$\mu_{\underline{A} \cap \underline{B}}(x_1) = \min(\mu_{\underline{A}}(x_1), \mu_{\underline{B}}(x_1)) = \min \{ 0.2, 0.8 \} = 0.2$
$\mu_{\underline{A} \cap \underline{B}}(x_2) = \min(\mu_{\underline{A}}(x_2), \mu_{\underline{B}}(x_2)) = \min \{ 0.5, 0.6 \} = 0.5$
$\mu_{\underline{A} \cap \underline{B}}(x_3) = \min(\mu_{\underline{A}}(x_3), \mu_{\underline{B}}(x_3)) = \min \{ 0.6, 0.4 \} = 0.4$
$\mu_{\underline{A} \cap \underline{B}}(x_4) = \min(\mu_{\underline{A}}(x_4), \mu_{\underline{B}}(x_4)) = \min \{ 0.8, 0.2 \} = 0.2$
$\mu_{\underline{A} \cap \underline{B}}(x_5) = \min(\mu_{\underline{A}}(x_5), \mu_{\underline{B}}(x_5)) = \min \{ 1.0, 0.1 \} = 0.1$
So, $\underline{A} \cap \underline{B} = \{ (x_1, 0.2), (x_2, 0.5), (x_3, 0.4), (x_4, 0.2), (x_5, 0.1) \}$

Complement:

Fuzzy complement is identical to crisp complement operation. The membership value of every element in the fuzzy set is complemented with respect to 1, i.e. it is subtracted from 1.
The **complement** of fuzzy set $\underline{A}$, denoted by $\underline{A}^C$, is defined as
$\underline{A}^C = \{(x, \mu_{\underline{A}^C}(x)) \mid \forall x \in X\}$
$\underline{A}^C(x) = 1 - \mu_{\underline{A}}(x)$



Fuzzy complement

Example of Fuzzy Complement:

$\underline{A}^C(x) = 1 - \mu_{\underline{A}}(x)$
$\underline{A} = \{ (x_1, 0.2), (x_2, 0.5), (x_3, 0.6), (x_4, 0.8), (x_5, 1.0) \}$
$\underline{A}^C = \{ (x_1, 0.8), (x_2, 0.5), (x_3, 0.4), (x_4, 0.2), (x_5, 0.0) \}$
$\underline{A} \cup \underline{A}^C = \{ (x_1, 0.8), (x_2, 0.5), (x_3, 0.6), (x_4, 0.8), (x_5, 1.0) \} \neq X$
$\underline{A} \cap \underline{A}^C = \{ (x_1, 0.2), (x_2, 0.5), (x_3, 0.4), (x_4, 0.2), (x_5, 0.0) \} \neq \Phi$

Unlike crisp sets, fuzzy sets do not hold the law of contradiction and the law of excluded middle.

	Crisp Set	Fuzzy Set
Law of contradiction	$A \cap A^c = \phi$	$\bar{A} \cap \bar{A}^c \neq \phi$
Law of excluded middle	$A \cup A^c = X$	$\bar{A} \cup \bar{A}^c \neq X$

Fuzzy Functions and Linguistic Variables:

fuzzy functions and **linguistic variables**.

1. **Linguistic Variables**:
    - In traditional mathematics, variables typically take numeric values. However, in **fuzzy logic**, we often encounter **linguistic variables** to express concepts more intuitively.
    - For instance, consider the variable **"Age"**. Instead of using precise numeric values, we can define linguistic terms like **"Child," "Young,"** and **"Old."**
    - The linguistic variable **"AGE"** can be represented as:
        - **AGE = {Child, Young, Old}**
    - Each linguistic term (e.g., "Child") has a **membership function** associated with a specific age range. These membership values help determine whether a person falls into the category of a child, young person, or elderly individual.
    - For example, if someone's age is 11, their membership values might be approximately:
        - **Child**: 0.75
        - **Young**: 0.2
        - **Old**: 0
    - The formal definition of a linguistic variable is:
        - **x, T(x), U, G, M**
            - **x**: Variable name (e.g., AGE)
            - **T(x)**: Set of linguistic terms (e.g., {Child, Young, Old})
            - **U**: Universe (the range of possible values)
            - **G**: Syntactical rules that modify the linguistic terms
            - **M**: Semantic rules associated with each linguistic term (giving meaning to the terms)

    !Representation of Linguistic Terms of Age
2. **Linguistic Hedges**:
    - **Linguistic hedges** allow us to modify linguistic variables, enhancing precision in communication.
    - For example, if we say **"John is Young"** with a value of 0.6, then **"very young"** can be deduced as having a value of 0.6 * 0.6 = 0.36.

Fuzzy relation

Cartesian product

Fuzzy relation defines the mapping of variables from one fuzzy set to another. Like crisp relation, we can also define the relation over fuzzy sets.
Let $\underline{A}$ be a fuzzy set on universe X and $\underline{B}$ be a fuzzy set on universe Y, then the Cartesian product between fuzzy sets $\underline{A}$ and $\underline{B}$ will result in a fuzzy relation $\underline{R}$ which is contained with the full Cartesian product space or it is a subset of the cartesian product of fuzzy subsets.
Formally, we can define fuzzy relation as,

$$\underline{R} = \underline{A} \times \underline{B}$$

and

$$\underline{R} \subset (X \times Y)$$

where the relation $\underline{R}$ has a membership function,

$$\mu_{\underline{R}}(x, y) = \mu_{\underline{A} \times \underline{B}}(x, y) = \min(\mu_{\underline{A}}(x), \mu_{\underline{B}}(y))$$

A binary fuzzy relation $\underline{R}(X, Y)$ is called a **bipartite graph** if $X \neq Y$.
A binary fuzzy relation $\underline{R}(X, Y)$ is called **directed graph** or **digraph** if $X = Y$. , which is denoted as $\underline{R}(X, X) = \underline{R}(X^2)$

Let $\underline{A} = \{a_1, a_2, \ldots, a_n\}$ and $\underline{B} = \{b_1, b_2, .., b_m\}$, then the fuzzy relation between $\underline{A}$ and $\underline{B}$ is described by the **fuzzy relation matrix** as,

$$\begin{bmatrix} \mu_{R(a_1,b_1)} & \mu_{R(a_1,b_2)} & \cdot & \cdot & \mu_{R(a_1,b_m)} \\ \mu_{R(a_2,b_1)} & \mu_{R(a_2,b_2)} & \cdot & \cdot & \mu_{R(a_2,b_m)} \\ \cdot & \cdot & & \cdot \ \cdot \ \cdot & \\ \cdot & \cdot & & \cdot \ \cdot \ \cdot & \\ \mu_{R(a_n,b_1)} & \mu_{R(a_n,b_2)} & \cdot & \cdot & \mu_{R(a_n,b_m)} \end{bmatrix}$$

Fuzzy relation matrix

We can also consider fuzzy relation as a mapping from the cartesian space (X, Y) to the interval [0, 1]. The strength of this mapping is represented by the membership function of the relation for every tuple $\mu_{\underline{R}(x,y)}$

Example:
Given $\underline{A} = \{ (a_1, 0.2), (a_2, 0.7), (a_3, 0.4) \}$ and $\underline{B} = \{ (b_1, 0.5), (b_2, 0.6)\}$, find the relation over $\underline{A} \times \underline{B}$

$$\bar{R} = \bar{A} \times \bar{B} = \begin{matrix} \\ a_1 \\ a_2 \\ a_3 \end{matrix} \begin{matrix} b_1 & b_2 \end{matrix} \begin{bmatrix} 0.2 & 0.2 \\ 0.5 & 0.6 \\ 0.4 & 0.4 \end{bmatrix}$$

Cartesian product

Fuzzy relation

Fuzzy relations are very important because they can describe interactions between variables.

Example: A simple example of a binary fuzzy relation on X = {1, 2, 3}, called ”approximately equal” can be defined as
R(1, 1) = R(2, 2) = R(3, 3) = 1
R(1, 2) = R(2, 1) = R(2, 3) = R(3, 2) = 0.8
R(1, 3) = R(3, 1) = 0.3
The membership function and relation matrix of R are given by

$$\bar{R}(x,y) = \begin{cases} 1, & if x = y \\ 0.7, & if |x-y| = 1 \\ 0.3, & if |x-y| = 2 \end{cases}$$

$$\bar{R} = \begin{matrix} \\ 1 \\ 2 \\ 3 \end{matrix} \begin{matrix} 1 & 2 & 3 \end{matrix} \begin{bmatrix} 1.0 & 0.7 & 0.3 \\ 0.7 & 1.0 & 0.7 \\ 0.3 & 0.7 & 1.0 \end{bmatrix}$$

Operations on fuzzy relation:

For our discussion, we will be using the following two relation matrices:

$$\bar{R} = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{bmatrix} 0.8 & 0.1 & 0.1 & 0.7 \\ 0.0 & 0.8 & 0.0 & 0.0 \\ 0.9 & 1.0 & 0.7 & 0.8 \end{bmatrix} \end{array}$$

$$\bar{S} = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{bmatrix} 0.4 & 0.0 & 0.9 & 0.6 \\ 0.9 & 0.4 & 0.5 & 0.7 \\ 0.3 & 0.0 & 0.8 & 0.5 \end{bmatrix} \end{array}$$

Union:

$\underline{R} \cup \underline{S} = \{ (a, b), \mu_{\underline{A} \cup \underline{B}}(a, b) \}$
$\mu_{\underline{R} \cup \underline{S}}(a, b) = \max(\mu_{\underline{R}}(a, b), \mu_{\underline{S}}(a, b))$
$\mu_{\underline{R} \cup \underline{S}}(x_1, y_1) = \max(\mu_{\underline{R}}(x_1, y_1), \mu_{\underline{S}}(x_1, y_1))$
$= \max(0.8, 0.4) = 0.8$

$\mu_{\underline{R} \cup \underline{S}}(x_1, y_2) = \max(\mu_{\underline{R}}(x_1, y_2), \mu_{\underline{S}}(x_1, y_2))$
$= \max(0.1, 0.0) = 0.1$

$\mu_{\underline{R} \cup \underline{S}}(x_1, y_3) = \max(\mu_{\underline{R}}(x_1, y_3), \mu_{\underline{S}}(x_1, y_3))$
$= \max(0.1, 0.9) = 0.9$

$\mu_{\underline{R} \cup \underline{S}}(x_1, y_4) = \max(\mu_{\underline{R}}(x_1, y_4), \mu_{\underline{S}}(x_1, y_4))$
$\max(0.7, 0.6) = 0.7$

.

.

.

$\mu_{\underline{R} \cup \underline{S}}(x_3, y_4) = \max(\mu_{\underline{R}}(x_3, y_4), \mu_{\underline{S}}(x_3, y_4))$
$= \max(0.8, 0.5) = 0.8$

Thus, the final matrix for union operation would be,

$$\bar{R} \cup \bar{S} = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{bmatrix} 0.8 & 0.1 & 0.9 & 0.7 \\ 0.9 & 0.8 & 0.5 & 0.7 \\ 0.9 & 1.0 & 0.8 & 0.8 \end{bmatrix} \end{array}$$

Union of fuzzy relations

Intersection:

$\underline{R} \cap \underline{S} = \{ (a, b), \mu_{\underline{A} \cap \underline{B}}(a, b) \}$
$\mu_{\underline{R} \cap \underline{S}}(a, b) = \min(\mu_{\underline{R}}(a, b), \mu_{\underline{S}}(a, b))$
$\mu_{\underline{R} \cap \underline{S}}(x_1, y_1) = \min(\mu_{\underline{R}}(x_1, y_1), \mu_{\underline{S}}(x_1, y_1))$
$= \min(0.8, 0.4) = 0.4$

$\mu_{\underline{R} \cap \underline{S}}(x_1, y_2) = \min(\mu_{\underline{R}}(x_1, y_2), \mu_{\underline{S}}(x_1, y_2))$
$= \max(0.1, 0.0) = 0.0$

$\mu_{\underline{R} \cap \underline{S}}(x_1, y_3) = \min(\mu_{\underline{R}}(x_1, y_3), \mu_{\underline{S}}(x_1, y_3))$
$= \max(0.1, 0.9) = 0.1$

$\mu_{\underline{R} \cap \underline{S}}(x_1, y_4) = \min(\mu_{\underline{R}}(x_1, y_4), \mu_{\underline{S}}(x_1, y_4))$
$\max(0.7, 0.6) = 0.6$

.

.

.

$\mu_{\underline{R} \cap \underline{S}}(x_3, y_4) = \min(\mu_{\underline{R}}(x_3, y_4), \mu_{\underline{S}}(x_3, y_4))$
$= \max(0.8, 0.5) = 0.5$

$$\bar{R} \cap \bar{S} = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{bmatrix} 0.4 & 0.0 & 0.1 & 0.6 \\ 0.0 & 0.4 & 0.0 & 0.0 \\ 0.3 & 0.0 & 0.7 & 0.5 \end{bmatrix} \end{array}$$

Intersection of relation

Complement:

$\underline{R}^c = \{ (a, b), \mu_{\underline{R}^c}(a, b) \}$
$\mu_{\underline{R}^c}(a, b) = 1 - \mu_{\underline{R}}(a, b)$
$\mu_{\underline{R}^c}(x_1, y_1) = 1 - \mu_{\underline{R}}(x_1, y_1) = 1 - 0.8 = 0.2$
$\mu_{\underline{R}^c}(x_1, y_2) = 1 - \mu_{\underline{R}}(x_1, y_2) = 1 - 0.1 = 0.9$
$\mu_{\underline{R}^c}(x_1, y_3) = 1 - \mu_{\underline{R}}(x_1, y_3) = 1 - 0.1 = 0.9$
.

.

$\mu_{\underline{R}^c}(x_3, y_4) = 1 - \mu_{\underline{R}}(x_3, y_4) = 1 - 0.8 = 0.2$
The complement of relation $\underline{R}$ would be,

$$\bar{R}^c = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{bmatrix} 0.2 & 0.9 & 0.9 & 0.3 \\ 1.0 & 0.2 & 1.0 & 1.0 \\ 0.1 & 0.0 & 0.3 & 0.2 \end{bmatrix} \end{array}$$

Complement of fuzzy relation

Projection:

The projection of $\underline{R}$ on X :
$\prod_x(x) = \sup(\underline{R}(x, y) \mid y \in Y)$
The projection of $\underline{R}$ on Y :
$\prod_Y(y) = \sup(\underline{R}(x, y) \mid x \in X)$
sup: Supremum of the set

$$\bar{R} = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \end{array} \begin{array}{c} \begin{array}{cccc} y_1 & y_2 & y_3 & y_4 \end{array} \\ \begin{bmatrix} 0.8 & 0.1 & 0.1 & 0.7 \\ 0.0 & 0.8 & 0.0 & 0.0 \\ 0.9 & 1.0 & 0.7 & 0.8 \end{bmatrix} \end{array}$$

The projection of $\underline{R}$ on X :

$\prod_X(x_1) = 0.8$

$\prod_X(x_2) = 0.8$

$\prod_X(x_3) = 1.0$

The projection of $\underline{R}$ on Y :

$\prod_Y(y_1) = 0.9$

$\prod_Y(y_2) = 1.0$

$\prod_Y(y_3) = 0.7$

$\prod_Y(y_4) = 0.8$

Fuzzy composition:

The **fuzzy composition** can be defined just as it is for crisp (binary) relations. Suppose R is a fuzzy relation on X × Y, S is a fuzzy relation on Y × Z, and T is a fuzzy relation on X × Z; then,

Fuzzy Max–Min composition is defined as:

$$\underline{T} = \underline{R} \circ \underline{S} = \mu_{\bar{T}}(x,z) = \bigvee_{y \in Y} (\mu_{\bar{R}}(x,y) \wedge \mu_{\bar{S}}(y,z))$$

$$= max_{y \in Y}\{min(\mu_{\bar{R}}(x,y), \mu_{\bar{S}}(y,z))\}$$

Fuzzy Max–Product composition is defined as:

$$\underline{T} = \underline{R} \circ \underline{S} = \mu_{\bar{T}}(x,z) = \bigvee_{y \in Y} (\mu_{\bar{R}}(x,y).\mu_{\bar{S}}(y,z))$$

$$= max_{y \in Y}\{(\mu_{\bar{R}}(x,y) \times \mu_{\bar{S}}(y,z))\}$$

Fuzzy rules:

Fuzzy rules are used within fuzzy logic systems to infer an output based on input variables. Modus ponens and modus tollens are the most important rules of inference. A modus ponens rule is in the form

Premise: *x is A*
Implication: ***IF*** *x is A* ***THEN*** *y is B*
Consequent: *y is B*

In crisp logic, the premise *x is A* can only be true or false. However, in a fuzzy rule, the premise *x is A* and the consequent *y is B* can be true to a degree, instead of entirely true or entirely false. This is achieved by representing the linguistic variables *A* and *B* using fuzzy sets. In a fuzzy rule, modus ponens is extended to *generalised modus ponens:*

Premise: *x is A**
Implication: ***IF*** *x is A* ***THEN*** *y is B*
Consequent: *y is B**

The key difference is that the premise *x is A* can be only partially true. As a result, the consequent *y is B* is also partially true. Truth is represented as a real number between 0 and 1, where 0 is false and 1 is true.

Fuzzy inference:

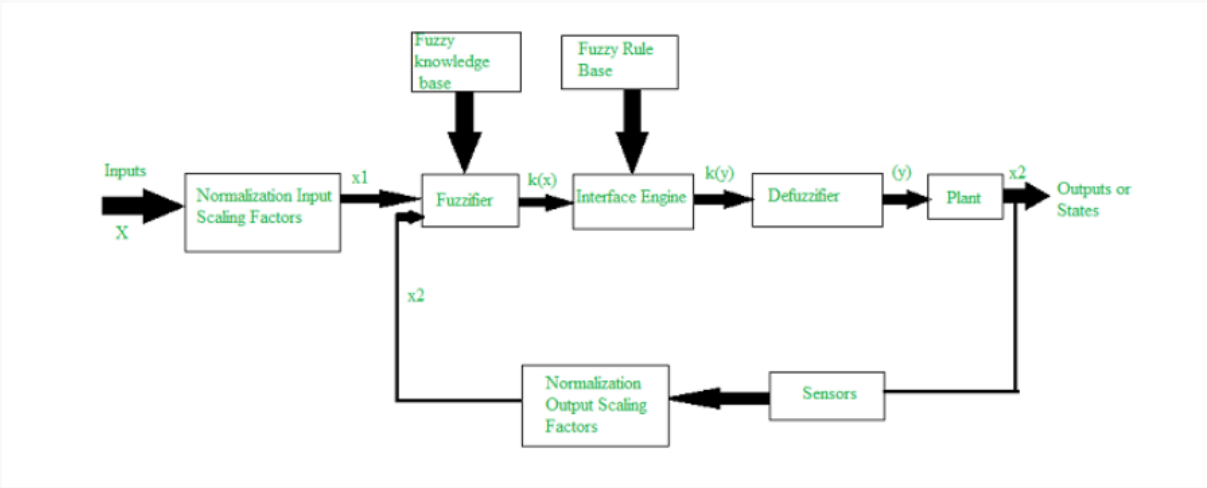
Fuzzy inference is a fundamental concept in **fuzzy logic**, which allows us to make decisions based on imprecise or uncertain information. Let's explore it further:

1. **Definition**:
    - **Fuzzy inference** is the process of **mapping** from a given input to an output using **fuzzy logic**.
    - It provides a basis for making decisions or discerning patterns when dealing with **inexact or vague data**.
2. **Components of Fuzzy Inference System (FIS)**:
    - **FIS** is the key unit of a fuzzy logic system responsible for decision-making.
    - It uses **IF…THEN** rules along with connectors such as **"OR"** or **"AND"** to draw essential decision rules.
    - Key characteristics of FIS include:
        - The output from FIS is always a **fuzzy set**, regardless of whether the input is fuzzy or crisp.
        - A **defuzzification unit** converts fuzzy variables into crisp variables when FIS is used as a controller.
3. **Functional Blocks of FIS**:
    - **Rule Base**: Contains fuzzy IF-THEN rules.
    - **Database**: Defines the membership functions of fuzzy sets used in fuzzy rules.
    - **Decision-making Unit**: Performs operations on rules.
    - **Fuzzification Interface Unit**: Converts crisp quantities into fuzzy quantities.
    - **Defuzzification Interface Unit**: Converts fuzzy quantities into crisp quantities.
4. **Methods of FIS**:
    - **Mamdani Fuzzy Inference System**:
        - Proposed by **Ebrahim Mamdani** in 1975.
        - Steps for computing the output:
            1. Determine a set of fuzzy rules.
            2. Fuzzify the input using input membership functions.
            3. Combine fuzzified inputs according to fuzzy rules to establish rule strength.
            4. Determine the consequent of the rule by combining rule strength and output membership function.
            5. Combine all consequents to obtain the output distribution.
            6. Finally, defuzzify the output distribution.
    - **Takagi-Sugeno Fuzzy Model (TS Method)**:
        - Proposed by **Takagi, Sugeno, and Kang** in 1985.
        - Format of rules: **IF x is A and y is B THEN Z = f(x, y)**.
        - Here, **A** and **B** are fuzzy sets in antecedents, and **Z = f(x, y)** is a crisp function in the consequent.
5. **Application Areas**:
    - FIS has been successfully applied in fields such as:
        - **Automatic control**
        - **Data classification**
        - **Decision analysis**
        - **Expert systems**
        - And more!

Remember that fuzzy inference allows us to handle uncertainty and imprecision, making it a powerful tool in various domains.

Architecture and Operations of FLC System:

The basic architecture of a fuzzy logic controller is shown in Figure. The principal components of an FLC system is a fuzzifier, a fuzzy rule base, a fuzzy knowledge base, an inference engine, and a defuzzifier. It also includes parameters for normalization. When the output from the defuzzifier is not a control action for a plant, then the system is a fuzzy logic decision system. The fuzzifier present converts crisp quantities into fuzzy quantities. The fuzzy rule base stores knowledge about the operation of the process of domain expertise. The fuzzy knowledge base stores the knowledge about all the input-output fuzzy relationships. It includes the membership functions defining the input variables to the fuzzy rule base and the out variables to the plant under control. The inference engine is the kernel of an FLC system, and it possesses the capability to simulate human decisions by performing approximate reasoning to achieve the desired control strategy. The defuzzifier converts the fuzzy quantities into crisp quantities from an inferred fuzzy control action by the inference engine.



The various steps involved in designing a fuzzy logic controller are as follows:

- **Step 1:** Locate the input, output, and state variables of the plane under consideration. I
- **Step 2:** Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.
- **Step 3:** Obtain the membership function for each fuzzy subset.
- **Step 4:** Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and the output of fuzzy subsets on the other side, thereby forming the rule base.
- **Step 5:** Choose appropriate scaling factors for the input and output variables for normalizing the variables between [0, 1] and [-1, I] interval.
- **Step 6:** Carry out the fuzzification process.

- **Step 7:** Identify the output contributed from each rule using fuzzy approximate reasoning.
- **Step 8:** Combine the fuzzy outputs obtained from each rule.
- **Step 9:** Finally, apply defuzzification to form a crisp output.

The above steps are performed and executed for a simple FLC system. The following design elements are adopted for designing a general FLC system:

1. Fuzzification strategies and the interpretation of a fuzzifier.
2. **Fuzzy knowledge base:** Normalization of the parameters involved; partitioning of input and output spaces; selection of membership functions of a primary fuzzy set.
3. **Fuzzy rule base:** Selection of input and output variables; the source from which fuzzy control rules are to be derived; types of fuzzy control rules; completeness of fuzzy control rules.
4. **Decision· making logic:** The proper definition of fuzzy implication; interpretation of connective "and"; interpretation of connective "or"; inference engine.
5. Defuzzification materials and the interpretation of a defuzzifier.

Applications:

FLC systems find a wide range of applications in various industrial and commercial products and systems. In several applications- related to nonlinear, time-varying, ill-defined systems and also complex systems – FLC systems have proved to be very efficient in comparison with other conventional control systems. The applications of FLC systems include:

1. Traffic Control
2. Steam Engine
3. Aircraft Flight Control
4. Missile Control
5. Adaptive Control
6. Liquid-Level Control
7. Helicopter Model

Fuzzy Rule-Based Systems:

Fuzzy Rule-Based Systems (FRBS) are a fascinating area of application within the realm of **fuzzy sets** and **fuzzy logic**. Let's dive into what they are and how they work:

1. **Conceptual Overview**:
    - FRBSs extend classical rule-based systems by incorporating fuzzy logic statements in their rules. Unlike conventional bivalent logic, which struggles with uncertainty and imprecision, fuzzy logic provides a more robust framework for reasoning in situations where human intuition often prevails.
    - In FRBSs, fuzzy sets and fuzzy logic serve as tools for representing various forms of knowledge about a problem. They also model interactions and relationships between variables.

- The heart of an FRBS lies in its **IF–THEN rules**, where both antecedents and consequents are composed of fuzzy logic statements.

2. **Components of an FRBS**:
   - **Antecedents**: These represent the conditions or inputs in the IF part of the rules. They consist of linguistic fuzzy variables.
   - **Consequents**: These define the outcomes or actions in the THEN part of the rules. Again, they involve fuzzy logic statements.
   - By using fuzzy statements, FRBSs can capture and handle the inherent uncertainty present in the represented knowledge.
3. **Structural Approaches**:
   - FRBSs can be structured in various ways:
     - **Flat FRBS**: Simple rule-based systems with a single layer of rules.
     - **Hierarchical FRBS**: Organized into multiple layers, allowing more complex interactions.
     - **Mamdani-Type FRBS**: Uses fuzzy inference to combine rules.
     - **Takagi-Sugeno-Kang (TSK) FRBS**: Employs linear combinations of fuzzy sets for consequents.
4. **Design and Applications**:
   - Designing an FRBS involves selecting appropriate linguistic variables, defining membership functions, and creating fuzzy rules.
   - FRBSs find applications in **classification**, **regression**, and other real-world problems where uncertainty, imprecision, and non-linearity play a significant role.

In summary, FRBSs provide a powerful way to reason and make decisions in complex scenarios, leveraging the flexibility and expressiveness of fuzzy logic

Genetic Algorithms

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random searches provided with historical data to direct the search into the region of better performance in solution space. **They are commonly used to generate high-quality solutions for optimization problems and search problems.**

Genetic algorithms simulate the process of natural selection which means those species that can adapt to changes in their environment can survive and reproduce and go to the next generation. In simple words, they simulate "survival of the fittest" among individuals of consecutive generations to solve a problem. **Each generation consists of a population of individuals** and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Genetic algorithms (GAs) are optimization techniques inspired by the process of natural selection. They mimic the principles of evolution to find optimal solutions to complex problems. Let's delve into the encoding strategies used in GAs:

1. **Binary Encoding**:
    - **Most common method**: Chromosomes are represented as strings of 1s and 0s.
    - Each position in the chromosome corresponds to a specific characteristic of the solution.
    - Well-suited for optimization problems in a **discrete search space**.
    - For example, in binary encoding, each gene controls a particular trait, just like genes in DNA encode traits in living organisms.
2. **Permutation Encoding**:
    - Useful for problems involving **ordering**, such as the **Traveling Salesman Problem (TSP)**.
    - In TSP, each chromosome is a string of numbers, where each number represents a city to be visited.
    - The order of cities in the chromosome determines the tour route.
3. **Value Encoding**:
    - Applied when complex values (e.g., real numbers) are involved.
    - Binary encoding may not suffice for such cases.
    - Requires specific crossover and mutation techniques tailored to these chromosomes.
    - Used in various domains, including engineering, finance, medical diagnostics, artificial intelligence, and logistics.

Remember, the choice of encoding method significantly impacts the GA's performance and convergence to optimal solutions.

genetic operators play a crucial role in guiding the algorithm toward finding solutions to specific problems. Let's explore these operators:

1. **Mutation Operator**:
    - Mutation is a **unary operator** that operates on **individual chromosomes** (solutions).
    - Its purpose is to introduce **genetic diversity** by randomly altering one or more genes within a chromosome.
    - By doing so, it prevents the algorithm from getting stuck in local optima and allows exploration of different regions of the solution space.
    - Think of it as a way to introduce small, random changes to the genetic makeup of an individual.
    - For example, if we're optimizing a set of parameters, mutation might tweak one of those parameters slightly.
2. **Crossover Operator (Recombination)**:
    - Crossover is a **binary operator** that combines **two parent chromosomes** to create a **new child chromosome**.
    - It mimics the process of genetic recombination in natural evolution.
    - By recombining portions of good solutions, the algorithm is more likely to create better offspring.
    - Different methods exist for combining parent solutions, such as **edge recombination**, **cut and splice crossover**, and **uniform crossover**.
    - The choice of crossover method often depends on the problem being solved and the representation of the solution.
    - For instance, if variables are grouped together as building blocks, a respectful crossover operator is essential to maintain their integrity.
3. **Selection Operator**:
    - Selection operators give preference to **better solutions** (chromosomes) based on some **fitness function**.
    - The best solutions are chosen to pass their genes to the next generation.
    - Methods like **fitness proportionate selection** and **tournament selection** help determine the best solutions.
    - Elitism, where the best solutions directly pass to the next generation without mutation, is also a form of selection.
4. **Inversion (Permutation) Operator** (less commonly used):
    - This operator is rarely discussed and its effectiveness remains uncertain.
    - It involves reversing a portion of a chromosome.
    - While it's not widely used, it's interesting to note its existence.

In summary, genetic operators work together to create and maintain genetic diversity, combine existing solutions, and select between them, ultimately guiding the genetic algorithm toward better solutions .

Genetic Algorithms (GAs) and explore the concepts of **fitness functions** and the GA cycle.

1. **Fitness Function**:

- The **fitness function** is a crucial component in GAs. It evaluates how "fit" or "good" a candidate solution (individual) is with respect to the problem being considered.
- Specifically, the fitness function takes a candidate solution as input and produces a quantitative measure of its fitness.
- In most cases, the fitness function aligns with the **objective function** of the problem. For optimization tasks, the goal is either to **maximize** or **minimize** the objective function.
- Characteristics of a good fitness function:
  - It should be **fast to compute**, as the fitness value is calculated repeatedly during the GA process.
  - It must **quantitatively measure** the fitness of a solution or the potential of producing fit individuals from that solution.
  - In complex problems, direct calculation of the fitness function may not be feasible due to inherent complexities. In such cases, **fitness approximation** is used to suit our needs.
- For example, consider the 0/1 Knapsack problem. A simple fitness function might sum the profit values of the selected items (those with a value of 1) until the knapsack is full.

2. **GA Cycle**:
   - The GA operates in a loop over a specified number of **generations**.
   - Key steps in each generation:
     - **Evaluation**: The fitness value of each individual in the population is assessed using the fitness function.
     - **Selection**: Individuals with better fitness scores have a higher chance of being selected for reproduction.
     - **Reproduction**: The current generation produces the next generation through genetic operators (such as crossover and mutation) based on the selected individuals.
   - The process continues iteratively, allowing the population to evolve toward better solutions.

Remember, GAs mimic natural selection and evolution, making them powerful tools for optimization and search problems.

Traveling Salesman Problem using Genetic Algorithm

A genetic algorithm is proposed to solve the travelling salesman problem. Genetic algorithms are heuristic search algorithms inspired by the process that supports the evolution of life. The algorithm is designed to replicate the natural selection process to carry generation, i.e. survival of the fittest of beings. Standard genetic algorithms are divided into five phases which are:

1. Creating initial population.

2. Calculating fitness.
3. Selecting the best genes.
4. Crossing over.
5. Mutating to introduce variations.

These algorithms can be implemented to find a solution to the optimization problems of various types. One such problem is the Traveling Salesman Problem. The problem says that a salesman is given a set of cities, he has to find the shortest route to as to visit each city exactly once and return to the starting city.

Algorithm:

```
1. Initialize the population randomly.
2. Determine the fitness of the chromosome.
3. Until done repeat:
    1. Select parents.
    2. Perform crossover and mutation.
    3. Calculate the fitness of the new population.
    4. Append it to the gene pool.
```

How the mutation works?

Suppose there are 5 cities: 0, 1, 2, 3, 4. The salesman is in city 0 and he has to find the shortest route to travel through all the cities back to the city 0. A chromosome representing the path chosen can be represented as:

This chromosome undergoes mutation. During mutation, the position of two cities in the chromosome is swapped to form a new configuration, except the first and the last cell, as they represent the start and endpoint.

Original chromosome had a path length equal to **INT_MAX**, according to the input defined below, since the path between city 1 and city 4 didn't exist. After mutation, the new child formed has a path length equal to **21**, which is a much-optimized answer than the original assumption. This is how the genetic algorithm optimizes solutions to hard problems.

A genetic algorithm (GA) is a search and optimization technique inspired by the process of natural selection and evolution. A GA works by creating and maintaining a population of candidate solutions (called individuals) to a given problem, and applying biologically inspired operators such as mutation, crossover, and selection to evolve them toward better solutions.

A typical GA diagram can be represented as follows:

The main steps of a GA are:

- **Initialization**: Generate an initial population of random individuals, each representing a possible solution to the problem.
- **Evaluation**: Calculate the fitness score of each individual, which measures how well it solves the problem.
- **Selection**: Select a subset of individuals from the current population, based on their fitness scores, to be the parents of the next generation.
- **Crossover**: Combine two or more parents to create new offspring, by exchanging some of their genetic information (such as bits, characters, or numbers).
- **Mutation**: Introduce some random changes in the offspring, by flipping, swapping, or altering some of their genetic information.
- **Replacement**: Replace the current population with the new offspring, or keep some of the best individuals from the current population.
- **Termination**: Check if a stopping criterion is met, such as reaching a maximum number of generations, finding an optimal solution, or reaching a convergence point. If not, go back to the evaluation step and repeat the process.

Here is an example of applying a GA to find the optimal values of a and b that satisfy the following expression:

The objective function is to minimize the value of the expression, which is zero when a = 3 and b = 5.

The steps are:

- **Initialization**: Generate six random pairs of a and b values between 1 and 10, and encode them as binary strings of length 8. For example, (a = 2.47, b = 6.84) can be encoded as 0010011001101100.

| Individual | Binary encoding | Decimal values | |---|---|---| | 1 | 0010011001101100| (2.47, 6.84) | | 2 | 0100100110000111| (4.77, 8.07) | | 3 | 0001110000011010| (1.87, 2.66) | | 4 | 0110001010100101| (6.21, 10.61) | | 5 | 0100000001110010| (4.00, 7.38) | | 6 | 0011100110101001| (3.51, 5.41) |

- **Evaluation**: Calculate the fitness score of each individual, which is the inverse of the value of the expression. For example, the fitness score of individual 1 is 1 / (2.47 - 3)^2 + (6.84 - 5)^2 = 0.33.

| Individual | Binary encoding | Decimal values | Fitness score | |---|---|---|---| | 1 | 0010011001101100| (2.47, 6.84) | 0.33 | | 2 | 0100100110000111| (4.77, 8.07) | 0.06 | | 3 | 0001110000011010| (1.87, 2.66) | 0.14 | | 4 | 0110001010100101| (6.21, 10.61) | 0.02 | | 5 | 0100000001110010| (4.00, 7.38) | 0.08 | | 6 | 0011100110101001| (3.51, 5.41) | 0.28 |

- **Selection**: Select three pairs of individuals to be the parents of the next generation, using a roulette wheel selection method, which gives a higher probability of selection to individuals with higher fitness scores.

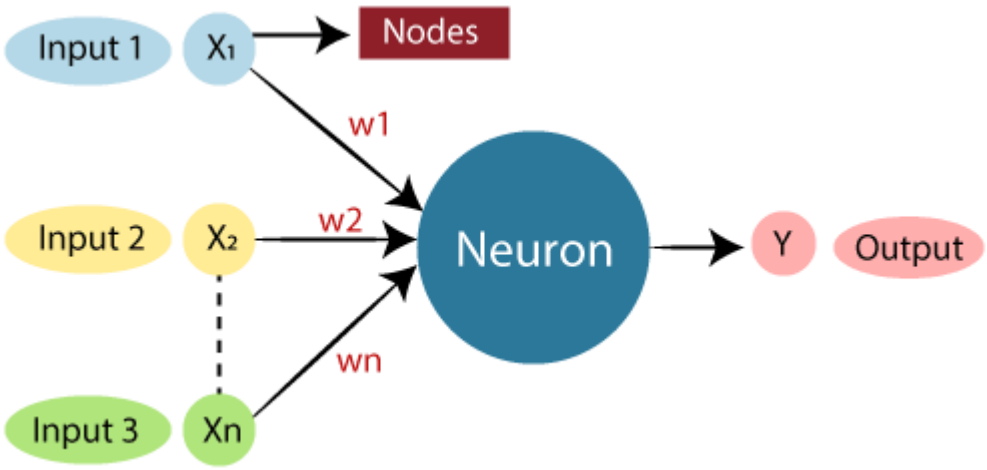
| Individual | Binary encoding | Decimal values | Fitness score | Selection probability | |---|---|---|---|---|---| | 1 | 0010011001101100| (2.47, 6.84) | 0.33 | 0.36 | | 2 | 0100100110000111| (4.77, 8.07) | 0.06 | 0.07 | | 3 | 0001110000011010| (1.87, 2.66) | 0.14 | 0.15 | | 4 | 0110001010100101| (6.21, 10.61) | 0.02 | 0.02 | | 5 | 0100000001110010| (4.00, 7.38) |

The table continues as:

| Individual | Binary encoding | Decimal values | Fitness score | Selection probability | |---|---|---|---|---|---| | 5 | 0100000001110010| (4.00, 7.38) | 0.08 | 0.09 | | 6 | 0011100110101001| (3.51, 5.41) | 0.28 | 0.31 | | Total | |

Artificial Neural Network (ANN)

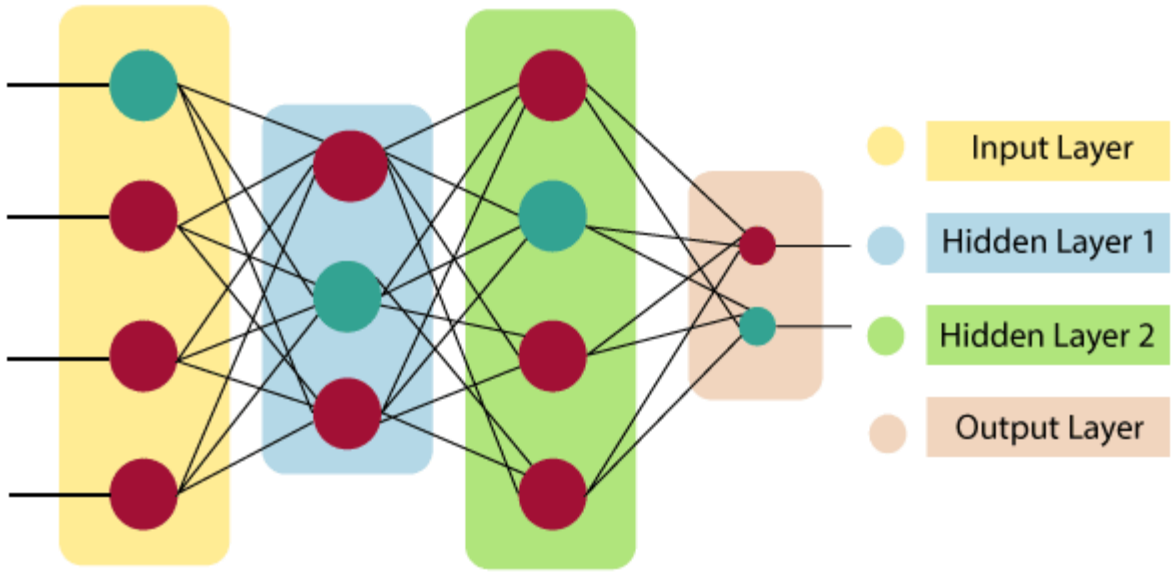
An Artificial Neural Network (ANN) is a **computational model that mimics the way nerve cells work in the human brain**. ANNs learn by examples and are configured for a specific application, such as pattern recognition or data classification, through a learning process. ANNs are also known as Feed-Forward Neural networks because inputs are processed only in the forward direction. ANNs use learning algorithms that can independently make adjustments as they receive new input.



The architecture of an artificial neural network:

To understand the concept of the architecture of an artificial neural network, we have to understand what a neural network consists of. In order to define a neural network that consists of a large number of artificial neurons, which are termed units arranged in a sequence of layers. Lets us look at various types of layers available in an artificial neural network.

Artificial Neural Network primarily consists of three layers:



Input Layer:

As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer:

The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

Output Layer:

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

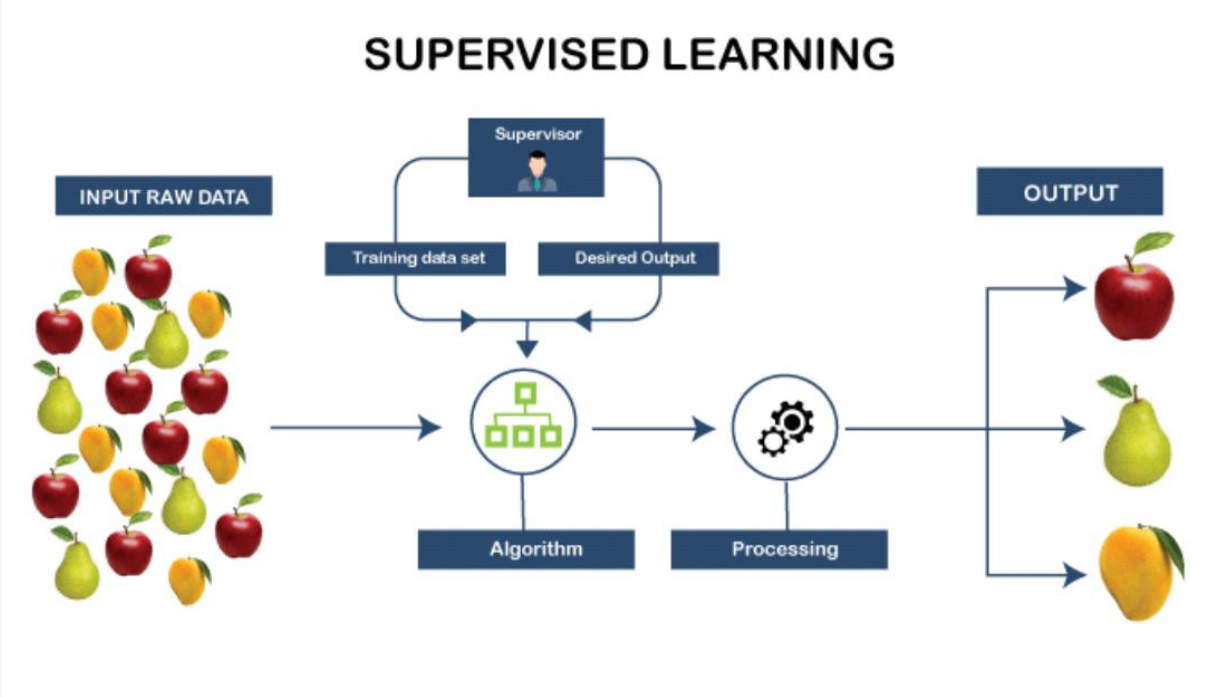
The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

$$\sum_{i=1}^{n} Wi * Xi + b$$

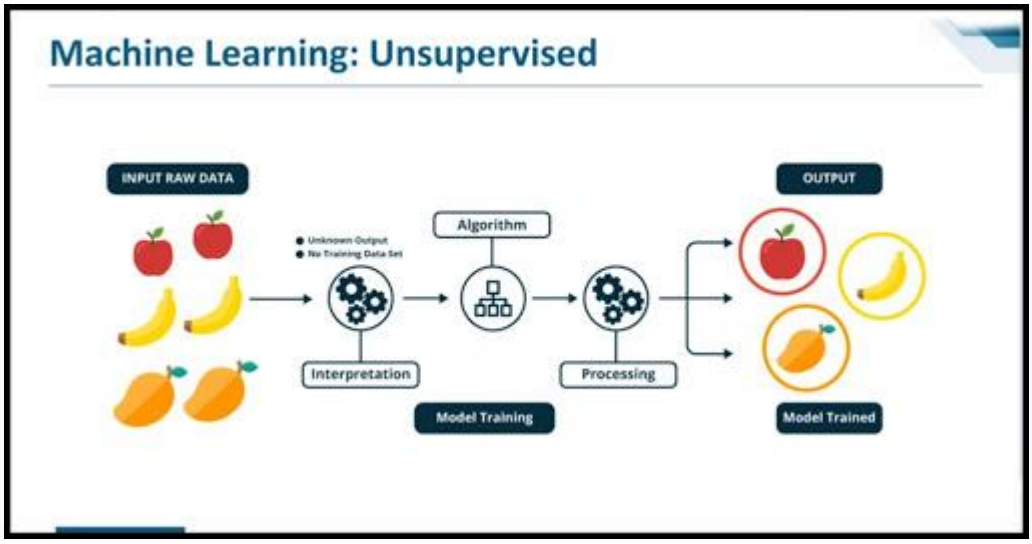
It determines weighted total is passed as an input to an activation function to produce the output. Activation functions choose whether a node should fire or not. Only those who are fired make it to the output layer. There are distinctive activation functions available that can be applied upon the sort of task we are performing.

supervised, **unsupervised**, and **reinforcement learning**.

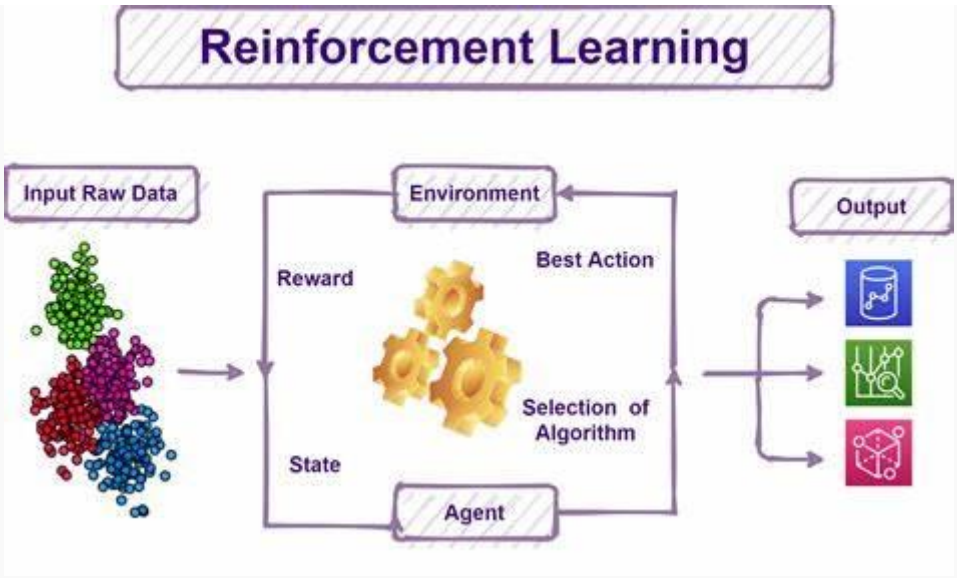
1. **Supervised Learning**:
    - In **supervised learning**, the machine learns from **labeled data**. Labeled data consists of examples paired with their correct answers or classifications.
    - Imagine a teacher guiding the machine during training. The process involves analyzing a set of labeled examples to learn the relationship between inputs (such as images) and outputs (such as labels).
    - Common tasks in supervised learning include **classification**, **regression**, and **object detection**.
    - For instance, if you have a dataset of fruit images labeled as “apple,” “banana,” or “orange,” a supervised learning algorithm can predict the fruit type for new, unlabeled images based on learned patterns.



2. **Unsupervised Learning**:
    - In **unsupervised learning**, the machine is trained on **unlabeled data**. There are no paired input-output examples.
    - The goal is to discover patterns, relationships, or structures within the data.
    - Common tasks in unsupervised learning include **clustering**, **dimensionality reduction**, and **anomaly detection**.
    - For example, given a collection of customer purchase data, unsupervised learning can group similar customers together without any predefined labels.



3. **Reinforcement Learning**:
    - **Reinforcement learning** is distinct from the other two. It deals with **sequential decision-making**.
    - Here, an agent interacts with an environment and learns by receiving **rewards or penalties** based on its actions.
    - Reinforcement learning is used in scenarios like game playing, robotics, and recommendation systems.
    - Think of it as teaching an AI to play chess: it explores different moves, receives feedback (rewards or penalties), and adjusts its strategy accordingly.



In summary:

- **Supervised learning** relies on labeled data.
- **Unsupervised learning** discovers patterns without labels.
- **Reinforcement learning** involves learning from rewards and penalties.

What is a Perceptron?

A perceptron is a type of artificial neuron or the simplest form of a neural network. It is a model of a single neuron that can be used for binary classification problems, which means it can decide whether an input represented by a vector of numbers belongs to one class or another. The concept of the perceptron was introduced by Frank Rosenblatt in 1957 and is considered one of the earliest algorithms for supervised learning.

At its core, a perceptron takes several binary inputs, multiplies each input by a weight, sums all the weighted inputs, and then passes that sum through a step function, which is a type of activation function, to produce a single binary output.

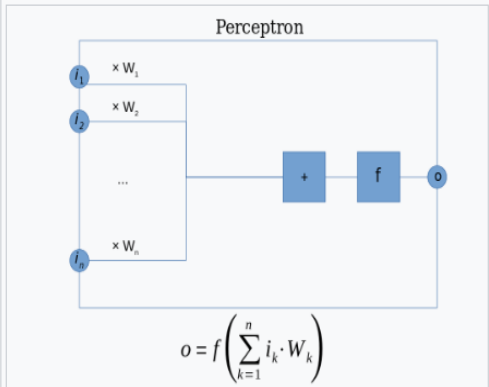
In the modern sense, the perceptron is an algorithm for learning a binary classifier called a threshold function: a function that maps its input $\mathbf{x}$ (a real-valued vector) to an output value $f(\mathbf{x})$ (a single binary value):

$$f(\mathbf{x}) = \theta(\mathbf{w} \cdot \mathbf{x} + b)$$

where θ is the heaviside step-function, $\mathbf{w}$ is a vector of real-valued weights, $\mathbf{w} \cdot \mathbf{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$, where m is the number of inputs to the perceptron, and b is the *bias*. The bias shifts the decision boundary away from the origin and does not depend on any input value.

Equivalently, since $\mathbf{w} \cdot \mathbf{x} + b = (\mathbf{w}, b) \cdot (\mathbf{x}, 1)$, we can add the bias term b as another weight $\mathbf{w}_{m+1}$ and add a coordinate 1 to each input $\mathbf{x}$, and then write it as a linear classifier that passes the origin:

$$f(\mathbf{x}) = \theta(\mathbf{w} \cdot \mathbf{x})$$



The appropriate weights are applied to the inputs, and the resulting weighted sum passed to a function that produces the output o.

The binary value of $f(\mathbf{x})$ (0 or 1) is used to perform binary classification on $\mathbf{x}$ as either a positive or a negative instance. Spatially, the bias shifts the position (though not the orientation) of the planar decision boundary.

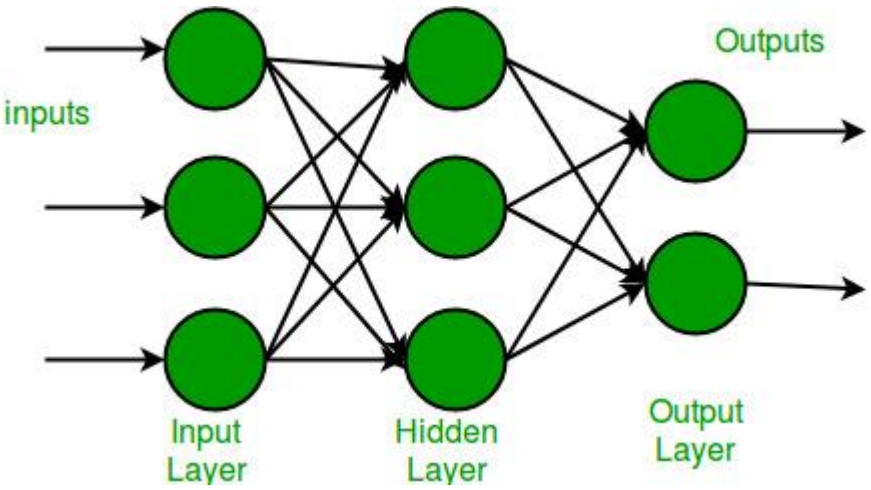
In the context of neural networks, a perceptron is an artificial neuron using the Heaviside step function as the activation function. The perceptron algorithm is also termed the **single-layer perceptron**, to distinguish it from a multilayer perceptron, which is a misnomer for a more complicated neural network. As a linear classifier, the single-layer perceptron is the simplest feedforward neural network.

Multi-layer Perceptron

Multi-layer perception is also known as MLP. It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.

A multi-layer perceptron has one input layer and for each input, there is one neuron(or node), it has one output layer with a single node for each output and it can

have any number of hidden layers and each hidden layer can have any number of nodes. A schematic diagram of a Multi-Layer Perceptron (MLP) is depicted below.



In the multi-layer perceptron diagram above, we can see that there are three inputs and thus three input nodes and the hidden layer has three nodes. The output layer gives two outputs, therefore there are two output nodes. The nodes in the input layer take input and forward it for further process, in the diagram above the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.

Every node in the multi-layer perception uses a sigmoid activation function. The sigmoid activation function takes real values as input and converts them to numbers between 0 and 1 using the sigmoid formula.

Self Organizing Map:

A **Self Organizing Map (SOM)**, also known as a **Kohonen Map**, is a type of **Artificial Neural Network (ANN)** inspired by biological models of neural systems from the 1970s. Let's dive into the details:

1. **Purpose and Approach**:
    - SOM follows an **unsupervised learning** approach.
    - It is trained using a **competitive learning algorithm**.
    - The goal is to map high-dimensional data onto a **lower-dimensional grid** while preserving the **topological structure** of the original data.
2. **Architecture**:
    - A SOM consists of two layers:
        - **Input layer**: Represents the input features.
        - **Output layer**: Comprises a grid of neurons (also called nodes or units).

- Each neuron in the output layer corresponds to a specific region in the input space.

3. **Training Process**:
   - Given an input data matrix of size *(m, n)* (where *m* is the number of training examples and *n* is the number of features), the SOM training proceeds as follows:
     1. **Weight Initialization**:
        - Initialize the weights *(w_ij)* randomly (usually small values).
        - Set the learning rate *(α)*.
     2. **Distance Calculation**:
        - Compute the squared Euclidean distance between each weight vector and the input sample.
        - Find the index *J* corresponding to the minimum distance (winning index).
     3. **Weight Update**:
        - Update the winning weight vector using: [w_{ij}^{new} = w_{ij}^{old} + α(t) \cdot (x_{ik} - w_{ij}^{old})]
          - *α(t)* is the learning rate at time *t*.
          - *i* denotes the feature index of the training example.
          - *k* denotes the training example index from the input data.
     4. **Learning Rate Update**:
        - Update the learning rate for the next iteration.
     5. **Repeat Steps 2-4** until convergence.
   - After training, the SOM's weights represent clusters in the input space.
4. **Use Cases**:
   - **Clustering**: SOM can group similar data points together.
   - **Dimensionality Reduction**: It maps high-dimensional data to a lower-dimensional grid for easier interpretation.

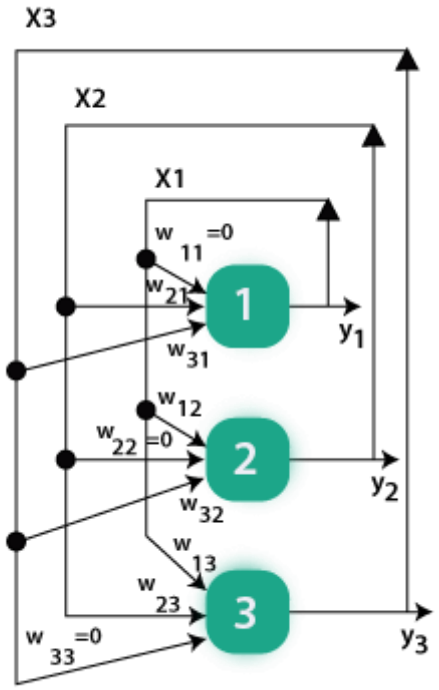
Remember, SOMs are powerful tools for understanding complex data patterns and visualizing high-dimensional data in a more manageable form!

Hopfield Network

Hopfield network is a special kind of neural network whose response is different from other neural networks. It is calculated by converging iterative process. It has just one layer of neurons relating to the size of the input and output, which must be the same. When such a network recognizes, for example, digits, we present a list of correctly rendered digits to the network. Subsequently, the network can transform a noise input to the relating perfect output.

A Hopfield network is a single-layered and recurrent network in which the neurons are entirely connected, i.e., each neuron is associated with other neurons. If there are two neurons i and j, then there is a connectivity weight $\mathbf{w_{ij}}$ lies between them which is symmetric $\mathbf{w_{ij} = w_{ji}}$.

With zero self-connectivity, $\mathbf{W_{ii} = 0}$ is given below. Here, the given three neurons having values **i = 1, 2, 3** with values $\mathbf{X_i = \pm 1}$ have connectivity weight $\mathbf{W_{ij}}$.



Updating rule:

Consider **N** neurons = **1, ... , N** with values $\mathbf{X_i}$ = **+1, -1**.

The update rule is applied to the node i is given by:

If $\mathbf{h_{i \geq 0}}$ then $\mathbf{x_i \rightarrow 1}$ otherwise $\mathbf{x_i \rightarrow -1}$

Where $\mathbf{h_i} = \sum_{j=1}^{N} \mathbf{wijxj}$ is called field at i, with b£ R a bias.

Thus, $\mathbf{x_i \rightarrow sgn(h_i)}$, where the value of **sgn(r)=1, if r ≥ 0**, and the value of **sgn(r)=-1, if r < 0**.

We need to put $b_i=0$ so that it makes no difference in training the network with random patterns.

We, therefore, consider $h_i=\sum_{j=1}^{N} w_{ij}x_j$.

We have two different approaches to update the nodes:

Synchronously:

In this approach, the update of all the nodes taking place simultaneously at each time.

Asynchronously:

In this approach, at each point of time, update one node chosen randomly or according to some rule. Asynchronous updating is more biologically realistic.

Hopfield Network as a Dynamical system:

Consider, $K = \{-1, 1\}^N$ so that each state $x \pounds X$ is given by $x_i \pounds \{ -1,1 \}$ for $1 \leq I \leq N$

Here, we get 2^N possible states or configurations of the network.

We can describe a metric on X by using the Hamming distance between any two states:

$P(x, y) = \#\{i: x_i \neq y_i\}$

N Here, **P** is a metric with $0 \leq H(x,y) \leq N$. It is clearly symmetric and reflexive.

With any of the asynchronous or synchronous updating rules, we get a discrete-time dynamical system.

The updating rule up: $X \rightarrow X$ describes a map.

And Up: $X \rightarrow X$ is trivially continuous.