# AI Questions & Answers

Marco Bernardi Andrea Auletta Aulo

March 7, 2024

**DISCLAIMER:** This document is a collection of questions and answers from the previous exams of the course of Artificial Intelligence. The answers are written by the authors and are not guaranteed to be correct.

If you find any mistake, please let us know by opening an issue on the GitHub repository of the project. GitHub repository

# Contents



# 1 Agents

**A.1** Explain in detail the meaning of the acronym PEAS in the context of the definition of an intelligent agent. According to the previous answer, introduce the various types of agents discussed in class.

**Answer:**

PEAS helps us define the task environment in which the agent is situated.

- **Performance**: This criterion defines how well the agent is performing in the environment. Performance measures could be a

combination of different criteria (e.g., an automated taxi driver's safety, legality, comfort, profit, time, etc.).

- **Environment**: It describes the task environment in which the agent is situated. An environment could be fully **observable** (e.g., a chess game) because you can see the whole state of the environment, or **partially observable** (e.g., poker).

  It could be **deterministic** (e.g., chess game) because the next state of the environment is completely determined by the current state and the action executed by the agent, or **stochastic** (e.g., poker) because the next state of the environment can't be predicted in a certain way.

  It could be **episodic** if the current action depends on the previous/next actions or **sequential** if the actions are independent.

  It could have only one agent or multiple agents.

  It could be **static** if the environment doesn't change while the agent is deliberating or **dynamic** if the environment can change while the agent is deliberating.

  Depending on the environment variables, the environment could be **discrete** or **continuous** (e.g., an automated taxi driver with roads and highways, other cars, pedestrians, traffic lights, etc.).

- **Actuators**: These are the mechanisms or devices through which the agent acts upon the environment. Actuators translate decisions made by the agent into actions that can be performed on the environment (e.g., an automated taxi driver's steering wheel, accelerator, brake, signal, etc.).

- **Sensors**: These are the mechanisms or devices through which the agent perceives the environment (e.g., an automated taxi driver's camera, GPS, speedometer, odometer, etc.).

An agent is a perceiving and acting entity; it is considered rational if it tries to achieve the best outcome given the available information.

Different types of agents exist:

- **Simple reflex agents**: They select actions based on the current percept, ignoring the rest of the percept history. A simple reflex agent acts according to a rule whose condition matches the current

state, as defined by the percept. The disadvantage of this type of agent is that if the environment is partially observable, the agent may select a wrong action.

- **Model-based reflex agents**: They select actions based on the current percept and some of the percept history. The disadvantage of this type of agent is that it has hard-coded rules, so it can't learn from experience and lacks flexibility.
- **Goal-based agents**: They select actions based on the goal they are trying to achieve, making the agent more flexible. The disadvantage of this type of agent is that it can have several goals that may conflict with each other.
- **Utility-based agents**: They select actions based on a utility function that measures the agent's performance. The utility function helps the agent choose the goal, keeping in mind the likelihood of success and the importance of the goals. There are no remarkable disadvantages.
- **Learning agents**: They select actions based on the knowledge gained from the environment and past experiences. There is a problem generator component that creates new situations to improve the agent's knowledge. There are no remarkable disadvantages.

# 2 Uninformed Search

**US.1** Describe the principal uninformed search strategies, and compare them in terms of correctness, completeness (Can we find a solution?), time and space complexity.

**Answer:** Search strategies are used to find a solution to a problem defined by four components: **initial state**, **successor function**, **goal test**, and **path cost**. Uninformed search strategies use only the information provided by the problem definition and operate systematically to find a solution. The solution is a sequence of actions that leads from the initial state to a goal state. There are different uninformed search strategies (We're working with trees):

(a) **Breadth-first search**: it explores the tree level by level, so the

fringe is implemented as a FIFO queue. The disadvantage of this strategy is that it requires a lot of memory because it has to visit all the nodes. It is **complete** (if the branching factor $b$ is finite) and **optimal** if the path costs are all equal. It has a **time complexity** of $O(b^d)$ and a **space complexity** of $O(b^d)$.

(b) **Uniform cost search**: if the path costs are all equal, it is equivalent to breadth-first search. Otherwise, it expands the least-cost unexpanded node (the node with the lowest path cost) and uses a priority queue (fringe) ordered by path costs. It is **complete** if the path costs are bounded below by a small positive constant and $b$ is finite. It is **optimal**. It has a **time complexity** of $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ (it can be greater than BFS time complexity because once it found a solution, it has to check if there is a better solution) and a **space complexity** of $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

(c) **Depth-first search**: it explores the tree by expanding the deepest node in the current frontier of the search tree. The fringe is implemented as a LIFO queue. It is **not complete** because it can get stuck in an infinite path. It is **not optimal**. It has a **time complexity** of $O(b^m)$ and a **space complexity** of $O(bm)$.

It requires less memory than breadth-first search, but it can get stuck in an infinite path.

(d) **Iterative deepening search**: it is a combination of depth-first search and breadth-first search. It performs depth-first search up to a certain depth, then it performs breadth-first search up to that depth, and so on, increasing the depth limit. It is **complete** if $b$ is finite and **optimal** if the path costs are all equal. It has a **time complexity** of $O(b^d)$ and a **space complexity** of $O(bd)$.

Its advantage is that it requires less memory than breadth-first search.

(e) **Depth-limited search**: it is a depth-first search with a depth limit that doesn't change. It isn't **complete** and not **optimal**. It has a **time complexity** of $O(b^l)$ and a **space complexity** of $O(bl)$.

(f) **Bidirectional search**: it performs two simultaneous searches, one forward from the initial state and one backward from the goal. It is **complete** if $b$ is finite and both searches use breadth-

first search. It is **optimal** if the path costs are all equal and both searches use breadth-first search. It has a **time complexity** of $O(b^{d/2})$ and a **space complexity** of $O(b^{d/2})$.

May not find the optimal solution, but if it's applicable, it is faster and lighter than breadth-first search.

# 3 Informed Search

**IS.1** Define the concept of informed search, and describe the greedy search and the A* algorithm. Discuss the properties (optimality, completeness, time and space complexity) and conditions of applicability of the two algorithms.

**Answer:**

Informed search algorithms use problem-specific knowledge beyond the definition of the problem itself. Greedy search and A* algorithm are some special cases of best-first search. Best-first search is an algorithm that explores a graph by expanding the most promising node chosen according to a specified rule. The rule is specific to the problem and defined by an evaluation function *f(n)* that estimates the cost of the path from the node $n$ to a goal node. Most best-first search algorithms include a heuristic function *h(n)* in *f(n)* that estimates the cost of the cheapest path from the node $n$ to a goal node.

- **Greedy search**: It expands the node that appears to be closest to the goal, according to the heuristic function *h(n)*. In this case, $f(n) = h(n)$. It is **not complete** because it can get stuck in a loop. In a finite space with repeated-state checking, it is complete. It is **not optimal**. It has a **time complexity** of $O(b^m)$ and a **space complexity** of $O(b^m)$.
- **A* algorithm**: The idea behind this algorithm is to avoid expanding paths that are already expensive. It expands the node that has the lowest value of $f(n) = g(n) + h(n)$, where:
  - $g(n)$ is the cost of the path from the initial state to the node $n$.
  - $h(n)$ is the heuristic function that estimates the cost of the cheapest path from the node $n$ to a goal node.

– *f(n)* is the evaluation function that estimates the cost of the cheapest solution through $n$.

A* uses an admissible heuristic function, a heuristic function that never overestimates the cost to reach the goal. It is **optimal** for tree search because it will never select a suboptimal goal $G_2$ over an optimal goal $G_1$, since $f(G_2) > f(n)$ where $n$ is an unexpanded node that leads to $G_1$. For graph search, the previous statement is not true, and we need another proof that uses a consistent heuristic. A heuristic is consistent if $h(n) \leq c(n, a, n') + h(n')$ where $n$ is the current node, $a$ is the action that leads to $n'$, and $n'$ is the successor of $n$. $f(n)$ is non-decreasing along any path. It is **complete**, unless there are infinitely many nodes with *f(n)* $\leq$ *f(G)*. It has a **time complexity** exponential and a **space complexity** of $O$(nodes count).

No other optimal algorithm is guaranteed to expand fewer nodes than A*. A* could be exponential in space $\Rightarrow$ Solutions:

– **Iterative deepening A* (IDA*)**: It acts like A* but uses a cutoff value instead of a depth limit. During an iteration, if $f(n) >$ cutoff, then the node is not expanded. When the queue is empty, the cutoff is increased to the lowest value of $f(n)$.
– **Recursive Best-First Search**: It imitates a deep search, using only linear space. It keeps track of the best alternative path available from any ancestor of the current node. When a node is expanded, the algorithm updates the value for the best alternative path. If the value of the best alternative path is smaller than the value of the current node, recursion goes back to the alternative path. On the return from the recursion, the algorithm updates the f-value of the best child node. It is **optimal** if the heuristic is consistent. Space complexity is $O(bd)$, and time complexity is exponential in the worst case. RBFS has a problem; it uses too little memory.
– **Memory-bounded A* (MA*)**:
– **Simplified memory-bounded A* (SMA*)**: It expands the best node until the memory is full. It removes the worst node from the memory to make space for a new node and backs up the f-value of the removed node on the parent node. The

parent node will eventually be expanded if the other paths are worse. It is **complete** only if the solution can be kept in memory. It is **optimal** if any optimal solution is reachable; otherwise, it returns the best solution found.

**IS.2** Introduce the A$^*$ algorithm in detail and exhaustively. Formally prove its optimality prop-erties. Finally, discuss the memory usage issues of A$^*$ and how this can be solved/reduced.

**IS.3** Describe the concept of admisible heuristic function, and give the formal definition of heuristic admisible and consistent. Choose a domain and give an example of two heuristic functions $h_1$ and $h_2$, they are both admisible and $h_1$ dominates $h_2$. Discuss how a heuristic function can be systematically constructed.

**Answer:**

A heuristic function is a function that estimates the cost of the cheapest path from the node $n$ to a goal node. A heuristic function is **admissible** if it never overestimates the cost to reach the goal ($h(n) \leq h^*(n)$). A heuristic function is **consistent** if $h(n) \leq c(n, a, n') + h(n')$, where $n$ is the current node, $a$ is the action that leads to $n'$, and $n'$ is the successor of $n$. Given two heuristic functions $h_1$ and $h_2$, $h_1$ dominates $h_2$ if $h_1(n) \geq h_2(n)$ for all nodes $n$; for research purposes, it's better to use the dominant heuristic function (but it should not be too expensive to compute).

Example 8-puzzle:

- $h_1(n)$ = total Manhattan distance
- $h_2(n)$ = number of misplaced tiles

A heuristic function can be systematically constructed by relaxing the problem. In the 8-puzzle example, the relaxed problem is to move the tiles without considering the other tiles. The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.

# 4 Iterative Improvement Algorithms

**II.1** Introduce hill-climbing search, appropriately placing it among the various categories of problem solving approaches we discussed in class. Present the variants, the properties, and in the case of a search with restart, formally demonstrate the result relative to the number of expected searches before finding an optimal solution.

**Answer:** Hill-climbing search is an iterative improvement algorithm that is part of local search algorithms. Hill-climbing follows the iterative improvement paradigm, where the goal state itself is the solution, and the path to reach it is irrelevant. As a local search algorithm, it doesn't keep track of the search tree but only of the current state and tries to improve it. Although local search algorithms are not systematic, they have two key advantages:

(a) They use very little memory, usually a constant amount.

(b) They can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

Hill-climbing search, at each iteration, selects the best successor among the neighbors of the current state and replaces the current state with it (or the lowest heuristic cost if a heuristic function is used). Hill-climbing is **not complete** because it never makes downhill moves, so it can get stuck in a local maximum.

Problems:
If the neighboring solutions are equivalent, it can get stuck in a shoulder; otherwise, if the neighboring solutions are worse, it can get stuck in a local maximum. If there are ridges (a sequence of local maxima not directly connected to each other), it will be difficult to move from one local maximum to another. In continuous space, it is challenging to pick the "right" step size, and convergence can be very slow.

To overcome these problems, we can use:

- Plateaux (flat values for $h$):
  - Allow sideways moves (moves to neighbors with equal $h$ value), but if there are no uphill moves (flat maximum), an infinite

loop will occur. To address this, a limit on the number of sideways moves can be imposed.

- Local maxima:
  - **Stochastic hill climbing**: Chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than regular hill climbing, but it can lead to better solutions.
  - **Random restart hill climbing**: Conducts a series of hill-climbing searches from randomly generated initial states. If $p$ is the probability of finding an optimal solution in a single search, the expected number of searches before finding an optimal solution is $1/p$. We can derive the probability $1/p$ as follows:

$$x_i = \begin{cases} 0 & \text{if the } i\text{-th search doesn't find an optimal solution} \\ 1 & \text{if the } i\text{-th search finds an optimal solution} \end{cases} \tag{1}$$

    We know that $\forall i$, $p = P(x_i = 1) = p$, and $P(x_i = 0) = 1 - p$. Variables $x_i$ are mutually independent; we have a Bernoulli Process for which we can use the geometric distribution to calculate the expected number of searches before finding an optimal solution. The probability that the $k$-th search finds an optimal solution is:

$$P\left(\left(\sum_{j=1}^{k-1} x_j = 0\right) \wedge (x_k = 1)\right) = (1-p)^{k-1}p \tag{2}$$

    The expected value of the quantity we are interested in is

then:

$$\begin{aligned}\sum_{k=1}^{\infty} k(1-p)^{k-1}p =& p\sum_{k=1}^{\infty} k(1-p)^{k-1} \\ =& -p\sum_{k=1}^{\infty}\frac{d}{dp}\left((1-p)^k\right) \\ =& -p\frac{d}{dp}\left(\sum_{k=0}^{\infty}(1-p)^k - 1\right) = -p\frac{d}{dp}\left(\frac{1}{p}-1\right) \\ =& p\frac{1}{p^2} = \frac{1}{p}\end{aligned} \tag{3}$$

For the random walk algorithm, it is **complete** but extremely inefficient.

**II.2** Local search algorithms:

**Answer:**

- **hill climbing**: Refer to the previous question **II.1**.
- **simulated annealing (gradient descent)**: It's a combination of hill climbing and random walk. It allows "bad" moves but gradually decreases their size and frequency. The probability of selecting a "bad" move is controlled by a parameter $T$ called "temperature," and $T$ is gradually decreased. The algorithm halts when a certain criterion is met (e.g., $T$ is less than a certain threshold). Bad moves are more likely to be selected at the beginning and less likely as the temperature decreases. If $T$ decreases slowly enough $\Rightarrow$ always reach the best state $x^*$.
- **local beam search**: It keeps track of $k$ states instead of one. It starts with $k$ randomly generated states, and at each iteration, it generates all the successors of the $k$ states and selects the best $k$ successors. It halts when a goal state is found or when a certain criterion is met. Otherwise, it selects the best $n$ (beam width) successors and repeats the process.

  **Problem**: Often, all $k$ states end up on the same local hill.

  **Idea**: Choose $k$ successors randomly, biased towards better states: stochastic beam search.

- **genetic algorithms**: It's a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state.
  (a) Are considered the best states of the current population.
  (b) They are combined to generate a new population.
  (c) There is a mutation probability $p_m$ that a state is randomly modified to explore new areas of the search space.
  (d) The algorithm tries to converge to an acceptable solution.

# 5 Online Search

**OS.1** Introduce the concept of online search, and describe the algorithms we discussed in class.

**Answer:** Online search algorithms are used when the environment is partially observable, or it is dynamic/semidynamic, and the agent needs to interact with the environment to get information about it. It's not possible to compute a complete solution before starting to act, so the agent has to interleave computation and action. Online search works well for exploration problems.

**Problem**: If some actions are irreversible, the agent may get stuck in a dead end (THIS IS NOT AVOIDABLE).

The performance of an online search algorithm is measured by the competitive ratio:

$$\frac{\text{total path cost online search}}{\text{total path cost knowing state space in advance}} \tag{4}$$

Types of online search algorithms:

- **Depth-first online search**: It's a depth-first search that keeps track of the current path. When the agent reaches a dead end, it backtracks until it finds a node with an unexplored successor. For backtracking, the algorithm uses a table that maps each node to its unbacktracked parent node.

  It iterates this process until it reaches the goal state. It is **complete** if the state space is finite and **optimal**. It has a **time complexity** of $O(b^m)$ and a **space complexity** of $O(bm)$.

- **Random search**: It's like hill climbing, but it chooses a random successor instead of using random restart. With this approach, the agent could escape from local maxima. Random walk will eventually find a solution if it exists (if finite state spaces), but it could take a long time.
- **LRTA$^*$ search (Learning Real-Time A$^*$)**: It's a combination of hill climbing with memory + a strategy to overcome local optima. It stores a "current best estimate" $H(s)$ of the cost of reaching the goal from each state that has been visited. $H(s)$ starts out being just the heuristic estimate $h(s)$. Based on the current best estimate, the agent chooses the action that appears to lead most quickly to the goal. When the agent reaches a new state, it updates the cost estimate for the state it has just left with the actual node cost estimate plus the cost to reach the actual node.

  This optimism under uncertainty encourages the agent to explore new, possibly promising paths. It is **complete** if the state space is finite. It can explore an environment in $O(n^2)$ steps in the worst case.

# 6 Adversarial Search

**AS.1** In the context of adversarial search, explain in detail how games with elements of chance can be deal with. In the case of resources limit, explain what is the property of the evaluation function that allows the search to preserve optimal decision (i.e. obtained with no resources limit). Finally, explain why the same approach is not returning an optimal strategy in the case of partially observable games.

**Answer:** In the context of adversarial search, nondeterministic games are games with elements of chance introduced by events like dice rolls or card shuffling. There are two types of nondeterministic games: **perfect information** (backgammon or Monopoly) and **imperfect information** (card games). We can handle nondeterministic games by using $\alpha - \beta$ pruning, a generalization of the minimax algorithm.

**Minimax algorithm** is a recursive algorithm for choosing the next move in an n-player game. It achieves the best payoff assuming that the opponent is also playing optimally. The minimax algorithm is based

on the idea of **minimizing the maximum loss**, meaning that the best move is chosen to minimize the maximum potential loss. Minimax is **complete** if the tree is finite and **optimal** if both players play optimally. It has a **time complexity** of $O(b^m)$ and a **space complexity** of $O(bm)$ (depth-first exploration).

**Minimax** can explore the entire tree, even if it's not necessary. To address this, we can use $\alpha - \beta$ pruning. If it finds a value $n$ less than the upper bound of another path, it halts the exploration of the current path.

$\alpha$ is the best choice for the max player along the path to the root, and $\beta$ is the best choice for the min player along the path to the root.

**Properties of $\alpha - \beta$ pruning:**

- Pruning doesn't affect the final result.
- A good ordering of the moves improves the effectiveness of pruning.
- Perfect order (MAX: from highest to lowest, MIN: from lowest to highest) $\Rightarrow$ time complexity (almost) $O(b^{m/2})$.
- Doubles the depth of exploration.

$\alpha - \beta$ pruning suits well for games with deterministic outcomes and also works for nondeterministic games. In the case of nondeterministic games, the states don't have definite values; we can only compute the expected value of a state. The **Expectiminimax algorithm** helps us handle this problem. It works like minimax but also handles chance nodes. It computes the expected value of a state by multiplying the probability of each outcome by the utility of that outcome. The upper bound of a node is the sum of the expected values of its children. Stronger pruning can be obtained if it's possible to bound the values of the leaves.

**Problem:** Resource limits. In the real world, we have limited resources, so we can't explore the entire tree.

**Solution:** Evaluation function & cutoff test.

An evaluation function is used to estimate the expected utility of the game from a given state. An evaluation function should order the

terminal states, where states that are wins should evaluate higher than states that are draws or losses. The computation must not take too long. Most evaluation functions work by calculating various features of the states. With a good evaluation function, we can explore the tree to a limited depth (CUTOFF), obtaining a good approximation of the true minimax value. The cutoff should be applied only to quiescent states (states in which the game is unlikely to change drastically in the near future). Non-quiescent states can be expanded further until a quiescent state is reached.

With this approach, there is still a difficult problem to overcome: the horizon effect. The horizon effect is the problem that the agent can't see beyond a certain depth, so it can't see a threat or an opportunity. This cannot be avoided, but we can temporarily mitigate it by delaying tactics; to mitigate it, we can use the **SINGULAR EXTENSION** (Find and remember a move that is clearly better than others and consider it if it is legal when the search reaches the normal depth limit).

The previous approaches do not return an optimal strategy in the case of partially observable games. In these types of games, the agent doesn't know the initial state of the game. We can try to predict the optimal move with $N$ deals. During this process, all possible moves are considered and weighted by their probability to get the expected value of the state. The action that wins most tricks on average will be picked; this will lead most of the time to a non-optimal strategy. In these types of games, the value of an action depends on the information state or belief state the agent is in.

**AS.2** In the context of adversarial search, discuss how huge search spaces, such as those gen-erated by the game of chess or Go, can be computationally managed. Also explain how the algorithm $\alpha - \beta$ pruning can help in such cases.

**Answer:**

**AS.3** Describe the $\alpha - \beta$ pruning algorithm in detail. Prove its complexity in the best case.

**Answer:**

# 7 Propositional Logic

**PL.1** Give the abstract definition of knowledge-based agent, discussing its various components. In addition, in the case of propositional logic, introduce how inference can be made by enumeration, discussing its algorithmic and computational complexity aspects.

**Answer:** A knowledge-based agent is an agent that uses an internal representation of the world to decide what actions to perform. The central component of a knowledge-based agent is the **knowledge base**, which is a set of sentences in a knowledge representation language representing the agent's knowledge about the world and may initially contain some background knowledge.

A knowledge-based agent also has an **inference engine**, a component that uses the knowledge base to infer new information. Typically, you can communicate with the agent through operations such as *tell* and *ask*.

A knowledge-based agent must be able to:

- Represent states, actions, etc.
- Incorporate new percepts
- Update internal representations of the world
- Deduce hidden properties of the world
- Deduce appropriate actions

In the case of propositional logic, inference can be made by enumeration, a general algorithm for enumerating all the models of a sentence. The algorithm involves enumerating all models and checking that the sentence $\alpha$ is true in every model in which $KB$ is true ($KB \models \alpha$). Models are assignments of *true* or *false* to each propositional symbol. The algorithm is **sound**[1] because it directly implements the definition of entailment, and it is **complete**[2] because it works for any $KB$ and $\alpha$ and always terminates.

[1] Soundness: $i$ is sound if whenever $KB \vdash_i \alpha$ (sentence $\alpha$ can be derived from $KB$ by the inference procedure $i$), it is also true that $KB \models \alpha$

[2] Completeness: $i$ is complete if whenever $KB \models \alpha$, it is also true that $KB \vdash_i \alpha$

If $KB$ and $\alpha$ contain $n$ symbols in total, then there are $2^n$ models. Thus, the time complexity of the algorithm is $O(2^n)$ and the space complexity is $O(n)$ because the enumeration is depth-first. Unfortunately, this algorithm is co-NP-complete, so every known inference algorithm for propositional logic has a worst-case time complexity that is exponential in the size of the input.

**PL.2** In the context of propositional logic, explain in detail the Forward-chaining algorithm, discussing its algorithmic and computational complexity aspects.

**Answer:**

Given a knowledge base $KB$, the idea is to fire any rule whose premises are satisfied in $KB$ and add the conclusion to $KB$ until the query is found.

i. Set the known leaves ($KB$).

ii. Inference propagates up the graph as far as possible.

iii. If conjunctions are found, wait until all conjuncts are known before proceeding.

The algorithm is **sound** (every inference is an application of modus ponens, as we are working with Horn clauses) and **complete** (every entailed atomic sentence will be derived). Proof of completeness:

(a) FC reaches a fixed point (no new atomic sentences can be inferred).

(b) Consider the final state as a model $m$, assigning true/false to symbols (inferred table: the table contains true for every inferred symbol, false otherwise).

(c) Every clause in the original KB is true in $m$ (because it's a fixed point). Proof: assume the opposite; then, there is a clause that is false in $m$:

$a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in $m$.

$a_1 \wedge \ldots \wedge a_k$ in the model must be true, so $b$ must be false. However, $b$ is in the inferred table, so it must be true $\Rightarrow$ contradiction: the algorithm has not reached a fixed point.

(d) $m$ is a model of $KB$.

(e) If $KB \models q$, then $q$ is true in every model of $KB$, including $m$.

It runs in **linear time**.

Forward-chaining is an example of the general concept of **data-driven** (unconscious processing) reasoning, where reasoning is driven by known data. One disadvantage of this approach is that it may perform a lot of work that is irrelevant to the goal.

**PL.3** In the context of propositional logic, explain in detail the Backward-chaining algorithm, discussing its algorithmic and computational complexity aspects.

**Answer:** Truth methods, which are useful for testing the validity of a sentence, divide into two classes:

- **Model checking**: This method checks if a given model satisfies a sentence. (Truth table enumeration is an example of model checking.)
- **Application of inference rules**: This method uses logical entailment to determine the truth of a sentence. (It involves the generation of new sentences from old ones.)

Truth table enumeration has a problem: it is exponential in the number of propositional symbols, and in many cases, the full power of resolution is not needed. We need to find a more efficient way to determine if $\alpha$ is entailed by $KB$. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, enabling them to use a more restricted and efficient inference algorithm. One such restricted form is the **Horn form**.

With the Horn form, $KB$ is a conjunction of Horn clauses[3]:

- proposition symbol
- (conjunction of symbols) $\Rightarrow$ symbol

[3]Horn clauses are a disjunction of literals with at most one positive literal

Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal.

**Modus Ponens** is an inference rule that can be used with the Horn form:

$$\frac{\alpha_1, \ldots, \alpha_n, \qquad \alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \beta}{\beta} \tag{5}$$

The Horn form simplifies the inference process, lowering the complexity from exponential to polynomial. Although this reduction in complexity is a significant improvement, it comes at a cost: reduced expressive power.

The Horn form can be utilized with the **forward chaining** and **backward chaining** algorithms.

**PL.4** In the context of inference in propositional logic, introduce the resolution rule. Then, explain how the resolution algorithm works. Finally, prove its completeness.

**Answer:**

Resolution yields a complete inference algorithm when coupled with any complete search algorithm. This rule applies only to clauses that are a disjunction of literals: every sentence of propositional logic can be converted into an equivalent conjunction of clauses, leading to conjunctive normal form (CNF).

**Resolution rule:**

$$\frac{\ell_1 \vee \ldots \vee \ell_k, \quad m_1 \vee \ldots \vee m_n}{\ell_1 \vee \ldots \vee \ell_{i-1} \vee \ell_{i+1} \vee \ldots \vee \ell_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n} \tag{6}$$

**Resolution algorithm:**

1. Convert $KB \wedge \neg\alpha$ to CNF.
   i. Eliminate $\Leftrightarrow$ by replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
   ii. Eliminate $\Rightarrow$ by replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
   iii. Move $\neg$ inwards using De Morgan's laws and double negation.
   iv. Apply the distributivity law ($\vee$ over $\wedge$) and flatten.
2. Apply the resolution rule to resulting clauses.

3. Each pair containing complementary literals is resolved to produce a new clause.
4. The process is repeated until one of two things happens:
   - There are no new clauses that can be added, indicating that $KB$ does not entail $\alpha$.
   - Two clauses resolve to yield the empty clause $\Rightarrow \alpha$ is entailed by $KB$ (The empty clause is equivalent to false because disjunction is true only if at least one of the disjuncts is true).

The algorithm is sound because it directly implements the definition of entailment, and it is complete because it works for any $KB$ and $\alpha$ and always terminates.

**Proof of completeness:**

The completeness theorem for resolution in propositional logic is called the ground resolution theorem:

*If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.* *(Resolution Closure: the set of all clauses that can be derived by repeated application of the resolution rule)*

The proof of this theorem is demonstrated by proving its contrapositive:

*If the resolution closure of a set of clauses does not contain the empty clause, then the set of clauses is satisfiable.*

(a) Construct a model for $S$ with a suitable truth value for $P_1, \ldots, P_k$: For $i$ from 1 to $k$:
   - If a clause in $RC(S)$ contains $\neg P_i$ and all its other literals are false under the assignment chosen for $P_1, \ldots, P_{i-1}$, then assign false to $P_i$.
   - Otherwise, assign true to $P_i$.

(b) Show that the previous point always ends up with a model for $S$, which is done by induction:

- **Base case: $i = 1$:**
  It cannot be that $\neg P_1$ and $P_1$ simultaneously appear in $S$, otherwise the empty clause will appear in $RC(S)$. So $P_1 \leftarrow false$ if $\neg P_1$ occurs in $S$, otherwise $P_1 \leftarrow true$.
- **Case $i$:** Assume a partial model $m_{(i-1)}$ for symbols $P_1, \ldots, P_{i-1}$. It is not possible to assign a truth value to $P_i$ if $RC(S)$ contains clauses:
  - $C \equiv B \vee P_i$ and $C' \equiv B' \vee \neg P_i$, where $B$ and $B'$ only contain symbols in $\{P_1, \ldots, P_{i-1}\}$.
  - Both $B$ and $B'$ are false under $m_{i-1}$.

  In the case we had these clauses, the resolution of them will return $B \vee B'$, so either $B$ or $B'$ should be true by the induction hypothesis. This means that:
  - If $B$ is true $\Rightarrow$ $P_i$ is false.
  - If $B'$ is true $\Rightarrow$ $P_i$ is true.

  Thus, it will be possible to extend model $m_{i-1}$ to either $m_i = m_{i-1} \cup \{P_i \leftarrow false\}$ or $m_i = m_{i-1} \cup \{P_i \leftarrow true\}$. So it's possible to extend the model up to $k$, and $S$ is satisfiable.

# 8 First-Order Logic

**FOL.1** Describe the fundamental elements to be able to carry out inference in the first-order logic through the resolution rule. Discuss the various strategies used by the resolution rule, highlighting the strengths and weaknesses. At the end discuss the computational aspects.

**Answer:** The fundamental elements of first-order logic are:

- **Constants**: These represent objects in the domain of discourse.
- **Predicates**: These represent properties of objects in the domain of discourse.
- **Functions**: These represent relations between objects in the domain of discourse.
- **Variables**: These represent unspecified objects in the domain of discourse.
- **Connectives**: $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$.

- **Quantifiers**: $\forall, \exists$.
- **Equality**: $=$.

Sentences are true with respect to a model and an interpretation.

The idea is to use quantifiers to perform inference and reduce it to inference in propositional logic. For example:

- **Universal quantification**: $\forall x P(x)$ is true in a model $m$ if and only if $P$ is true with $x$ representing every possible object in the model.

  **Universal instantiation**: Every instance of a universally quantified sentence is entailed by:

  $$\frac{\forall v \quad \alpha}{SUBST(\{v/g\}, \alpha)} \tag{7}$$

  for any variable $v$ and ground term $g$. It can be applied several times to add new sentences, and the new $KB$ is equivalent to the original one.

- **Existential quantification**: $\exists x P(x)$ is true in a model $m$ if and only if $P$ is true with $x$ representing at least one object in the model.

  **Existential instantiation**: For any sentence $\alpha$, variable $v$, and constant symbol $k$ that does not appear elsewhere in the KB:

  $$\frac{\exists v \quad \alpha}{SUBST(\{v/k\}, \alpha)} \tag{8}$$

  EI can be applied only once to replace the existential sentence, and the new $KB$ is not equivalent to the original one, but it is satisfiable if the original one is.

**FOL.2** In the context of first order logic, describe in a complete and exhaustive way the Unification algorithm, further explaining why this is particularly useful for logical inference

**Answer:**

Propositionalization seems to generate lots of irrelevant sentences (Instantiate the universal sentence in all possible ways; this can generate infinitely many ground terms with function symbols, but we can solve this problem by putting a limit):

With $p$ $k$-ary predicates and $n$ constants, there are $p \cdot n^k$ instantiations.

If there is some substitution $\theta$ that makes each of the conjuncts of the premise of the implication identical to sentences already in the KB, then we can assert the conclusion of the implication. This inference rule is called **Generalized Modus Ponens**:

$$\frac{p'_1, p'_2, \ldots, p'_n, \quad (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{q\theta} \tag{9}$$

Soundness GMP:

For any definite clause $p$, we have $p \vDash p\theta$ by UI.

- $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \vDash (p_1 \wedge \ldots \wedge p_n \Rightarrow q)\theta = (p_1\theta \wedge \ldots p_n\theta \Rightarrow q\theta)$
- $p'_1, \ldots, p'_n \vDash p'_1 \wedge \ldots \wedge p'_n \vDash p'_1\theta \wedge \ldots \wedge p'_n\theta$
- From 1 and 2, $q\theta$ follows by ordinary Modus Ponens.

**Unification** is a process that finds substitutions for variables that make two sentences identical. We can get the inference immediately if we can find a substitution $\theta$ such that $\alpha\theta = \beta\theta$. To avoid failures, we need to eliminate the overlap of variables in $\alpha$ and $\beta \Rightarrow$ **Standardizing apart**: each sentence has to use different variable names (we can rename variables in a sentence without changing its meaning).

**Unification algorithm**: The algorithm works by comparing the structures of the inputs, element by element. The substitution $\theta$ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. At the beginning, it checks if the two sentences have the same function or predicate and the same number of arguments. Then it checks arguments one by one:

- If those arguments are variables, it adds the binding to $\theta$.
- If those arguments are compounds, it recursively calls UNIFY on them.
- If those arguments are lists, it recursively calls UNIFY on them.

**Problem**: there could be more than one such unifier $\Rightarrow$ for each pair of sentences, we need to find the most general unifier (MGU: that one which places the fewest restrictions on the variables), which is unique up to renaming and substitution of variables.

This algorithm is particularly useful because it is used in algorithms for logical inference like forward/backward chaining and resolution.

**FOL.3** In the context of inference in first-order logic, introduce the resolution rule. Discuss in what form the clauses need to be represented in order to efficiently enforce the resolution and how that form can be achieved. Finally, present the various strategies that have been proposed regarding the choice of clauses to be used during the inference.

**Answer:**

The resolution rule is an inference procedure based on resolution for propositional logic and extended to first-order logic. It requires sentences to be in **conjunctive normal form** (CNF: a conjunction of clauses, where a clause is a disjunction of literals). To achieve this form, we must follow these steps:

(a) **Eliminate biconditionals and implications**: $p \Rightarrow q$ is equivalent to $\neg p \vee q$.

(b) **Move negation inwards**: we need rules for negated quantifiers:

- $\neg \forall x P(x)$ is equivalent to $\exists x \neg P(x)$
- $\neg \exists x P(x)$ is equivalent to $\forall x \neg P(x)$

(c) **Standardize variables**: rename variables to avoid conflicts.

(d) **Skolemize**: eliminate existential quantifiers by replacing them with Skolem functions. In the existential instantiation rule, we drop the quantifier and substitute the variable with another one, creating only ONE new sentence. When the existential instantiation can't be applied (The sentence doesn't match the pattern $\exists v\ \alpha$), we replace the existential quantifier variables with Skolem functions, which have as arguments the universally quantified variables in whose scope the existential quantifier appears.

(e) **Drop universal quantifiers**: we can drop universal quantifiers because they are implicit in CNF.

(f) **Distribute** $\vee$ **over** $\wedge$: $p \vee (q \wedge r)$ is equivalent to $(p \vee q) \wedge (p \vee r)$.

Once the sentences are in CNF, we can apply the resolution rule:

Two sentences (standardizing apart) are resolvable if they contain complementary literals: propositional literals are complementary if one unifies with the negation of the other. Thus, we have:

$$\frac{\ell_1 \vee \ldots \vee \ell_k, \quad m_1 \vee \ldots \vee m_n}{SUBST(\theta, \ell_1 \vee \ldots \vee \ell_{i-1} \vee \ell_{i+1} \vee \ldots \vee \ell_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n)} \quad (10)$$

where $UNIFY(\ell_i, \neg m_j) = \theta$.

This rule is called binary resolution because it combines two clauses at a time.

The various strategies that have been proposed regarding the choice of clauses to be used during the inference are:

- **Unit preference**: select a unit clause (a clause with only one literal) if possible. The idea is that we're trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses.
- **Unit resolution**: is a restricted form of resolution in which every resolution step must involve a unit clause. It is incomplete in general, but it is complete for Horn clauses.
- **Set of support**: select a set of clauses (the set of support) and use at least one element of the set in each step. It is incomplete if the wrong set of support is chosen (Example of a good set: the negated query).
- **Input resolution**: every resolution combines one of the input sentences (from the KB or the query) with some other sentence. It is complete if KB is in Horn form but incomplete in the general case. Complete if modified: allows $P$ and $Q$ to be resolved together either if $P$ is in the original KB or if $P$ is an ancestor of $Q$ in the proof tree.
- **Subsumption**: eliminates all sentences that are subsumed by other sentences in the KB.

# 9 Uncertainty

Agents may need to handle uncertainty, whether due to partial observability, nondeterminism, or a combination of the two. Agents that never fail must know everything about the world and various exceptions, but this is not possible in the real world:

- **Laziness**: It's too expensive to enumerate all the possible cases.
- **Theoretical ignorance**: Most of the time, we don't know the complete domain of the problem.
- **Practical ignorance**: Even if we know all the possible cases, there is uncertainty in applying these rules in some specific cases.

Probability provides a way of summarizing the uncertainty that comes from our laziness and ignorance, thereby solving the qualification problem (the problem of how to represent all the knowledge that an agent needs to know about the world). In **subjective** (degree of belief) or **Bayesian** (degree of belief given the evidence), probability relates propositions to one's own state of knowledge (it might be learned from past experience of similar situations). Decision under uncertainty $\Rightarrow$ Decision theory = Probability theory + Utility theory (used to represent and infer preferences).

**U.1** In the context of treatment of uncertainty, discuss what is the main computational challenge when using the joint probability distribution of the main involved variables.

**Answer:**

Joint probability distribution (completely determines a probability model) is used for inference by enumeration: for any proposition $\phi$, sum the atomic events where it is true:

$$P(\phi) = \sum_{\omega:\omega\models\phi} P(\omega) \tag{11}$$

It works also for conditional probability:

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \text{ if } P(b) \neq 0 \tag{12}$$

In 12, the denominator acts as a normalization factor; it is the probability of the event $b$.

Inference procedure:

- $X$ is a set of random variables.
- $Y$ is the set of query variables.
- $E$ is the set of evidence variables given specific values $e$.
- $H$ is the set of hidden variables ($H = X - Y - E$).

Then the required summation of joint entries is done by summing out the hidden variables:

$$P(Y|E = e) = \alpha P(Y, E = e) = \alpha \sum_h P(Y, E = e, H = h) \qquad (13)$$

The terms in the summation are joint entries because $Y$, $E$, and $H$ together exhaust the set of random variables.

The main computational challenge is that the space grows with the number of variables. Worst-case time complexity $O(d^n)$, where $d$ is the maximum number of values that any variable can take on (arity). Space complexity $O(d^n)$ because we need to store the probability of each possible combination of values. To reduce complexity, we can use conditional independence and Bayesian networks (which can exploit conditional independence relationships and improve computational efficiency). Ref **U.2** and **U.3**.

**U.2** In the context of the treatment of uncertainty, formally define the concept of conditional independence and how this can be used. In particular, discuss its role in Bayesian Networks.

**Answer:**

**Definition:** Two random variables $X$ and $Y$ are independent given a set of random variables $Z$ if and only if:

$$P(X|Y) = P(X) \text{ or } P(Y|X) = P(Y) \text{ or } P(X, Y) = P(X)P(Y) \quad (14)$$

This means that in a given set of random variables, if we can find independent variables, then the full joint distribution can be factored into

smaller separated joint distributions on the subsets composed of the variables that are related to each other (size of the representation from exponential in $n$ to linear in $n$). In this way, the full joint distribution can be represented as the product of the smaller joint distributions. In real-world problems, it's difficult to find a clean separation of variables by independence. Conditional independence is our most basic and robust form of knowledge about uncertain environments.

Conditional independence:

$$P(X, Y|Z) = P(X|Z)P(Y|Z) \tag{15}$$

Full joint distribution:

$$P(\mathit{Cause},\ \mathit{Effect}_1, \ldots, \mathit{Effect}_n) = P(\mathit{Cause}) \prod_i P(\mathit{Effect}_i|\mathit{Cause}) \tag{16}$$

Such probability distributions are called **naive Bayesian** models.

In a Bayesian network, conditional independence is used to define the network topology. The topological semantics specify that each variable is conditionally independent of its nondescendants, given its parents.

**U.3** What does a Bayesian Network consist of? Why is it useful? What is its computational complexity in space and time (also discuss special cases) for the exact inference?

**Answer:**

A Bayesian network is a directed graph in which each node is annotated with quantitative probability information. The full specification is as follows:

(a) Each node corresponds to a random variable, which may be discrete or continuous.

(b) A set of directed links or arrows connects pairs of nodes. If there is an arrow from node $X$ to node $Y$, $X$ is said to be a parent of $Y$. The graph has no directed cycles and hence is a directed acyclic graph, or DAG.

(c) Each node $X_i$ has a conditional probability distribution $P(X_i|\text{Parents}(X_i))$ that quantifies the effect of the parents on the node.

Bayesian networks are useful because they provide a compact representation of a full joint distribution, given the combination of the topology (conditional independence) and the conditional distributions. In the simplest case, the conditional distribution is represented as a Conditional Probability Table (CPT), giving the distribution over $X_i$ for each combination of values for the parents. A CPT for a boolean $X_i$ with $k$ boolean parents has $2^k$ rows for the combinations of parent values. If each variable has at most $k$ parents, the total number of parameters is $O(n2^k)$, where $n$ is the number of variables, i.e., it grows linearly with $n$ vs $O(2^n)$ for the full joint distribution.

In Bayesian networks, we can perform inference by computing the probability distribution of a set of query variables given some observed evidence variables.

The complexity (ref **U.1**) of exact inference in Bayesian networks depends strongly on the structure of the network.

- In singly connected networks (or polytrees), any two nodes are connected by at most one (undirected) path, so the complexity of inference is linear in the size of the network ($O(d^k n)$).
- In multiply connected networks, we can reduce 3SAT to exact inference, which is NP-Hard and is equivalent to counting 3SAT solutions, thus #P-complete.

We can also perform inference by stochastic simulation:

- Draw $N$ samples from a sampling distribution $S$.
- Compute an approximate posterior probability $\hat{P}$.
- Show this converges to the true probability $P$.

Given the intractability of exact inference in large, multiply connected networks, it is essential to consider approximate inference methods.

- Sampling from an empty network
- Rejection sampling: reject samples disagreeing with evidence
- Likelihood weighting: weight samples by the likelihood of evidence
- Markov Chain Monte Carlo (MCMC): sample from a stochastic process whose stationary distribution is the true posterior

# 10 Machine Learning

**ML.1** Introduce the main paradigms of machine learning, describing in particular the fundamental ingredients of the supervised paradigm, and how the complexity of an hypothesis space can be measured in a useful way in the case of a binary classification task.

**Answer:**

Machine learning is the study of computer algorithms capable of learning from data. A learning algorithm must have the following components:

- **Tasks**: Define how the machine learning algorithm should process an example.
- **Performance measure**: Evaluate the accuracy of the function/model returned by the learning algorithm.
- **Experience**: Refers to the dataset.

There are different paradigms of machine learning:

- **Supervised learning**: Given pre-classified examples (training set) $Tr = \{(x^{(i)}), f(x^{(i)})\}$, learn a general description $h(x)$ (hypothesis) that captures the information content of the examples. Then, given a new example $\tilde{x}$, we can predict the corresponding output $h(\tilde{x})$. It's called supervised because it assumes that an expert provides the value of $h$ for the corresponding training instance $x$.
- **Unsupervised learning**: Given a set of examples $Tr = \{x^{(i)}\}$, discover regularities and/or patterns in the data. In this case, there is no expert to provide the correct answer.
- **Reinforcement learning**: The agent learns by interacting with the environment. The agent receives a reward that can be positive, negative, or neutral for each action, and the goal is to maximize the total reward.

The fundamental ingredients of the supervised paradigm are:

- **Training data**: Data drawn from the Instance Space, $X$.

- **Hypothesis space**: The set of functions that the learning algorithm can choose from to approximate the function $f$ (Function to be learned).
- **Learning algorithm**: A search algorithm into the hypothesis space.

$H \neq$ *set of possible functions*; searching into $H$ exhaustively can lead to overfitting, where the algorithm learns the training data too well and doesn't generalize well to new examples.

There is an inductive bias on $H$ and the search algorithm: a set of assumptions that the learning algorithm uses to predict outputs of new instances.

The complexity of a hypothesis space can be measured effectively by the **VC dimension**:

- **Definition**: The VC dimension of a hypothesis space $H$ is the size of the largest set of points that can be shattered by $H$.
- **Shattering**: A set of points $S$ is shattered by $H$ if for every possible labeling of the points in $S$, there exists a function $h$ in $H$ that correctly classifies the points in $S$.

In the case of a binary classification task, we have only two possible labels. If we work with linear classification, we can classify the points in the plane with the sign of the position with respect to a line (positive or negative). The VC dimension of the hypothesis space of linear classifiers is 3 because we can shatter 3 points but not 4. It is not possible to find a line that separates 4 points in all possible ways; there always exist two couples of points such that if we connect the two members by a segment, the two resulting segments will intersect, so a curve is needed.

**ML.2** Explain in detail the supervised learning paradigm, describe the role of the training set, the validation set and the test set (how to use data in our hands). Give the definition of true error and empirical error, highlighting the role of them during the learning process.

**Answer:**

Supervised Learning, referring to answer **ML.1**.

On learning tasks, we have a set of data that can be split into:

- **Training set**: Used to train the model.
- **Validation set**: A subset of the training set used to tune the hyperparameters of the model (hold-out, cross-validation).
- **Test set**: Used to evaluate the selected model.

Model selection is the process of choosing the best model for a given task by selecting the best hyperparameters:

- **Hold-out procedure**: Split the training set into two parts; the first one is used to train the model, and the second one (validation set) is used to test the trained model with different hyperparameters.
- **Cross-validation**: K-different classifiers/regressors are trained on K-different subsets of $Tr$ ($Va_1, \ldots, Va_k$), and then the Hold-out procedure is iteratively applied to the k-pairs ($Tr_i = Tr - Va_i, Va_i$).

The empirical error ($error_{Tr}(h)$) of hypothesis $h$ with respect to $Tr$ is the number of examples that $h$ misclassifies:

$$error_{Tr}(h) = \frac{\#\{(x, f(x)) \in Tr | f(x) \neq h(x)\}}{|Tr|} \tag{17}$$

The true error ($error_D(h)$) of hypothesis $h$ with respect to the target concept $c$ and distribution $D$ is the probability that $h$ will misclassify an instance drawn at random according to $D$:

$$error_D(h) \equiv \Pr_{x \in \mathcal{D}}[c(x) \neq h(x)] \tag{18}$$

We can say that $h \in \mathcal{H}$ overfits $Tr$ if $\exists h' \in \mathcal{H}$ such that $error_{Tr}(h) < error_{Tr}(h')$ and $error_D(h) > error_D(h')$.

The goal of machine learning is to solve a task with the lowest possible true error, but a classifier learns on training data, so it generates empirical error and not true error. It's possible to have a bound on the true error from the empirical error with a probability of $1 - \delta$:

$$error_D(h^*_w) \leq \underbrace{error_{Tr}(h^*_w)}_{A} + \underbrace{\epsilon(n, VC(\mathcal{H}), \delta)}_{B} \tag{19}$$

B (VC-confidence) depends on the ratio between $VC(\mathcal{H})$ and $n$ (number of training examples) and on $1 - \delta$ (confidence level).

Problem: As the VC-dimension grows, the empirical risk (A) decreases; however, the VC confidence (B) increases! To minimize the right hand of the confidence bound, we can use the principle of **Structural Risk Minimization**: we get a tradeoff between A and B, aiming to select the hypothesis with the lowest bound on the true risk.

**ML.3** In the context of machine learning, explain the fundamental ingredients of perceptron. Provide a brief introduction of how this model can be extended by creating a multi-layer architecture.

**Answer:**

A perceptron, given an input vector $\vec{x}$ and a weight vector $\vec{w}$, calculates $f(\sum w_i * x_i)$, which is the activation function of the perceptron. A Neural Network is a system consisting of interconnected units that compute nonlinear functions.

In a Neural Network, we can find:

- **Input Units**: Represent input variables.
- **Output Units**: Represent output variables.
- **Hidden Units**: Represent internal variables that codify correlations between input and output variables.
- **Weights**: Are associated with connections between units.

Having decided on the mathematical model for individual "neurons," the next task is to connect them together to form a network. **Feed-forward networks** have information flowing in one direction, from the input units, through the hidden units (if any), and to the output units.

For gradient descent in feed-forward networks, we need to optimize the weights of the network to minimize the error on the training set using the backpropagation algorithm. The process involves the following steps:

(a) Define a loss function that measures the error of the network on the training set.

(b) Randomly initialize the weights.

(c) Execute the forward pass: propagate the input through the network, and compute the output.

(d) Calculate the gradient descent with respect to the weights.

(e) Update the weights in the opposite direction of the gradient, multiplied by a learning rate.

(f) Repeat steps c and e until a stop condition is reached (e.g., the error is below a threshold or the number of epochs is reached).

# 11 Reinforcement Learning

**RL.1** Introduce the Reinforcement Learning paradigm. Moreover, present the Q-Learning algorithm explaining its theoretical basis.

**Answer:**

Reinforcement Learning is a paradigm in which the agent learns by interacting with the environment. The agent receives a reward that can be positive, negative, or neutral for each action, and the goal is to learn to choose actions that maximize the total reward:

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$$

Here:

- $r_i$ is the reward received after the action at $a_i$ in the state $s_i$.
- $\gamma$ is the discount factor, a value between 0 and 1 that represents the importance of future rewards.

We have a finite set of states $S$ and a finite set of actions $A$. At each discrete time, the agent observes state $s_t \in S$ and selects action $a_t \in A$. The agent receives a reward $r_t$, and the environment moves to a new state $s_{t+1}$. This works under the Markov assumption: $s_{t+1} = \delta(s_t, a_t)$ and $r_t = r(s_t, a_t)$, where:

- $r_t$ and $s_{t+1}$ depend only on the current state and action.
- Functions $\delta$ and $r$ may be nondeterministic and not necessarily known to the agent.

The agent's goal is to execute actions in the environment, observe results, and learn an action policy $\pi() : S \rightarrow A$ that maximizes $E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots]$ from any starting state in $S$ (training examples of the form $((s, a)r)$).

For deterministic environments, we define an evaluation function over states for each possible policy $\pi$:

$$V^{\pi}(s) \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+1}$$

The task is to learn the optimal policy $\pi^*$ that maximizes $V^{\pi^*}(s)$ for all $s \in S$.

In cases where the agent doesn't know the transition function $\delta$ and the reward function $r$, we can define the Q-function, similar to $V^*$:

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a))$$

If the agent learns Q, it can choose the optimal action even without knowing $\delta$:

$$\pi^*(s) = \arg\max_a Q(s, a)$$

Q is the evaluation function the agent will learn. Note that Q and $V^*$ are related by:

$$V^*(s) = \max_{a'} Q(s, a')$$

which allows us to write Q recursively as:

$$Q(s, a) = r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a')$$

So, we can define a training rule to learn Q:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

The Q-learning algorithm is as follows:

(a) Initialize table $\hat{Q}(s, a) \leftarrow 0$.

(b) Observe the current state $s$.

(c) Do forever:

i. Select an action $a$ and execute it (it could be randomly selected or using $\arg\max_a \hat{Q}(s,a)$).

ii. Receive immediate reward $r$.

iii. Observe new state $s'$.

iv. Update the table entry for $\hat{Q}(s,a)$:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

v. $s \leftarrow s'$

It can be shown that $\hat{Q}$ converges to $Q$.

For non-deterministic environments, we redefine $V$ and $Q$ by taking expected values:

$$V^{\pi}(s) \equiv E[\sum_{i=0}^{\infty} \gamma^i r_{t+1}]$$

$$Q(s,a) \equiv E[r(s,a) + \gamma V^*(\delta(s,a))]$$

And the training rule becomes:

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n(r + \gamma \max_{a'} \hat{Q}_{n-1}(s',a'))$$

# 12 Constraint satisfaction problems

1. Give the formal definition for a constrain satisfaction system, and discuss the approaches presented in class on how to find solutions.

   **Answer:**

   A Constraint Satisfaction System (CSP) is used to solve constraint satisfaction problems. It is defined by:

   - **State**: A set of variables $X_i$ with values from domains $D_i$.
   - **Goal Test**: A function that determines whether a given state is a goal state.

It can be helpful to visualize a CSP as a constraint graph, where each node represents a variable, and each arc represents a constraint (in binary CSP, at most two vars for each constraint). A solution to a CSP is an assignment of values to all variables such that all constraints are satisfied.

A constraint satisfaction problem can have **discrete** or **continuous** variables.

- **Discrete Variables**: Finite domains and infinite domains (integers, strings, etc.). For infinite domains, a constraint language allowing for compact representation of constraints is needed.
- **Continuous Variables**: Common in real-world problems, with linear programming (LP) being a well-known example.

The constraints could be:

- **Unary**: Involve a single variable.
- **Binary**: Involve pairs of variables.
- **Higher-order**: Involve 3 or more variables.
- **Preferences**: Soft constraints, not required to be satisfied $\rightarrow$ constrained optimization problem.

The first approach to finding solutions is the **Standard Search Formulation** $\rightarrow$ **Min-Conflicts**:

- **Initial State**: The empty assignment, {}
- **Successor Function**: Assign a value to an unassigned variable that does not conflict with the current assignment.
- **Goal Test**: The current assignment is complete and satisfies all constraints.

It uses depth-first search.

There is an important property not considered by the previous approach: **Commutativity**. The commutativity property states that the order in which we assign values to variables does not matter; the result is the same.

The term backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. It repeatedly chooses an unassigned variable and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value. We can improve the backtracking search with general-purpose methods:

- **Variable Ordering**: Choose the next variable to be assigned a value (most constrained variable, most constraining variable, least constraining).
- **Variable Assignment**: Forward checking: Keep track of remaining legal values for unassigned variables and terminate the search when any variable has no legal values.
- **Detecting Failure**: Constraint propagation (arc and node consistency).
- **Problem Structure**: The problem structure can give some advantages in the search:
  - **Independent Subproblems**: Independence can be ascertained by finding the connected components of the constraint graph. Each component can be solved independently of the others in $O(d^c n/c)$ time; otherwise $O(d^n)$.
  - **Tree-Structured CSPs**: A graph is a tree when any two variables are connected by only one path and can be solved in linear time, $O(nd^2)$. Algorithms for tree-structured CSPs:
    (a) Choose a variable as the root, order the variables from root to leaves such that every node's parent precedes it in the ordering.
    (b) For $i$ from $n$ down to 2, impose arc consistency on the arc from $X_i$ to its parent.
    (c) For $i$ from 1 up to $n$, assign $X_i$ consistently with all its ancestors.

The last approach is based on the use of iterative algorithms that repeatedly improve the quality of the current assignment (e.g., hill climbing, simulated annealing, genetic algorithms). To apply to CSPs:

- Allow states with unsatisfied constraints.
- Operators reassign variable values.

Variable selection: Randomly select any conflicted variable. Value selection by min-conflicts heuristic: choose a value that violates the fewest constraints, i.e., hill climb with $h(n)$ = the total number of violated constraints.

# 13 NLP

NLP (Natural Language Processing) is the field of AI that is concerned with the processing and understanding of natural language, acquiring knowledge and communicating with humans.

**NLP.1** In the field of NLP, explain the difference between the n-gram (in particular unigram and bigram) and bag-of-words models. Show how the n-gram model can be used for classification and information retrieval problems, detailing the relations and possible differences with models such as bag-of-words.

**Answer:**

For NLP-related tasks, we need language models that predict the probability distribution of linguistic expressions.

**Definition:** A language model is a probability distribution over sequences of words.

One such model is the **n-gram** model, defined as a probability distribution on sequences of characters.

$$P(c_{1:N}) = \textit{the probability of a sequence of } N \textit{ characters } c_1, \ldots, c_N \tag{20}$$

A sequence of written symbols of length N is called an N-gram (e.g., unigram, bigram, trigram, etc.).

*A pattern n-gram is a Markov chain of order n-1 → the probability depends only on the previous n-1 symbols.*

For a trigram:

$$P(c_{1:N}) = \prod_{i=1}^{N} P(c_i|c_{i-2:i-1}) \tag{21}$$

The main problem with the n-gram model is that the training corpus provides only an estimate of the true probability distribution, which could lead to zero probability for unseen n-grams. Two main solutions to this problem are:

- **Laplace Smoothing**: Assign a non-zero probability to unseen n-grams.
- **Backoff**: Start by estimating n-gram counts, but for any particular sequence with a low (or zero) count, back off to $(n-1)$-grams.

We can evaluate a model with cross-validation, but we would like to define a task-independent quality model.

$$\textit{Perplexity}(c_{1:N}) = P(c_{1:N})^{-\frac{1}{N}} \tag{22}$$

Low perplexity means that the model is good at predicting the test set. Perplexity is the inverse probability of the test set, normalized by the number of words.

N-gram models can also be used for words, where the vocabulary is significantly larger. The objective is to calculate the probability of a sentence $P(w_1, \ldots, w_n)$. The chain rule can be used to calculate the joint probability of words:

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i|w_1, \ldots, w_{i-1}) \tag{23}$$

We will never see enough data to estimate all these probabilities accurately, so we need to make some approximation. By the Markov assumption, we can approximate the probability of a word with the probability of the previous word:

$$P(w_1, \ldots, w_n) \approx \prod_{i=1}^{n} P(w_i|w_{i-k:i-1}) \tag{24}$$

**Bag-of-Words** (also known as a vector model): It is a simplifying representation used in natural language processing, representing the frequency of occurrence of each word in a document.

Differences between n-gram and bag-of-words:

- Feature vectors are large and scattered in BoW.
- BoWs and unigram return the same result.
- The order of words is lost in BoW, while a local notion of order is preserved in higher-order n-grams.
- In n-grams, computational complexity increases with n.

A simple representation of a document is one-hot encoding, where each word present in the vocabulary is represented by one or zero in the vector (the word is present or not). The problem is that one-hot vectors are orthogonal, so we can't capture any similarity between words. Some possible solutions are:

- **Knowledge Representation Way**: Rely on a list of synonyms or taxonomies of words to obtain similarity.
- **Machine Learning Way**: Learn to code the similarity in the vectors themselves. The idea is to construct a vector space where the distance between vectors is a measure of similarity between the words. An example of this approach is the **word2vec** model: it uses context to predict a target word (CBOW) or uses a word to predict the context (skip-gram).

N-gram models can be used for classification problems, such as spam detection. There are two main approaches:

- **Rules-Based**: Define a set of hard-coded rules that classify the text based on a combination of words (or other features). **Problem**: Rule-based approaches can lead to high accuracy results, but building such rules is very expensive and not always possible.
- **Linguistic Modeling and Machine Learning**: Define an n-gram pattern for each class and calculate the probability of the text belonging to each class with Naive Bayes (or other classifiers).