UA - University of Alicante

Faculty of Artificial Intelligence

Department of Computer Science and Artificial Intelligence

Multi-Agent Systems

Lecturer:
Prof. Fraile Beneyto, Raúl

Author:
Christian Faccio

November 18, 2025

github.com/christianfaccio christianfaccio@outlook.it

Preface

As a student of Data Science and Artificial Intelligence, I've created these notes while attending the **Multi-Agent Systems** course.

The course provides a comprehensive introduction to the field of multi-agent systems, covering both theoretical concepts and practical applications. The notes encompass a variety of topics, including:

- Intelligent Agents and Multi-Agent Systems
- Communication in Multi-Agent Systems
- Interacting by Cooperating
- Competition and Negotiation
- Agent-Based Modeling (ABM)
- Learning in Multi-Agent Systems
- Bio-inspired Systems and Swarm Intelligence
- ABM Analysis

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in multi-agent systems.

Contents

1 Introduction **1**

2 Intelligent Agents and MAS **4**
2.1 Communication . . . 8
2.1.1 Communication Acts . . . 10
2.2 Interacting through cooperation . . . 12
2.2.1 Cooperation . . . 12
2.2.2 Coordination . . . 14
2.2.3 Planning . . . 15
2.2.4 Coalitions . . . 16
2.2.5 Voting in MAS . . . 16
2.2.6 Manipulation and Reputation . . . 17
2.3 Competition and Negotiation . . . 17
2.3.1 Game Theory . . . 18
2.3.2 Auctions . . . 19

3 Agent Based Modelling **20**
3.1 ODD Protocol . . . 22

4 Learning in MAS **24**
4.1 A little recall from single-agent RL . . . 24
4.2 Evolutionary Strategies . . . 25
4.3 Swarm Intelligence . . . 26
4.3.1 Metalearning . . . 27

5 How to use Gama Platform **29**
5.1 Platform . . . 29
5.1.1 Installation . . . 29
5.1.2 Workspace, Projects and Models . . . 30
5.1.3 Running Experiments . . . 32
5.2 Model Creation . . . 33

1 Introduction

Complex systems are evertwhere around us:

- Traffic systems;
- Disease spereading (e.g. COVID);
- dynamics of social networks;
- Natural ecosystems (take inspiration from nature evolution);
- Software systems.

Figure 1.1: Traffic.

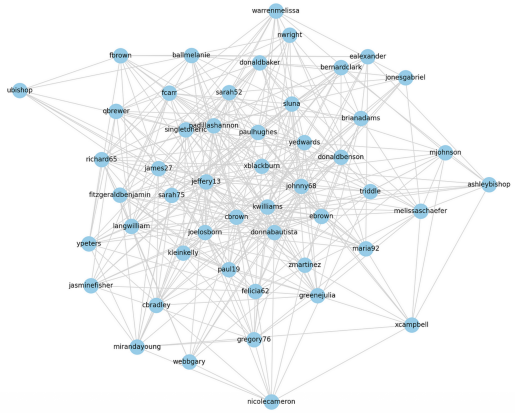


Figure 1.2: Social networks.

Definition: *Complex System*

A **Complex System** is composed by many components interactinv in a non-trivial way. A specific case could be: ”How can we analyze traffic in our city?”

- Traditional analytical methods;
- Simulation-based methods;
- Mathematical optimization;
- Directed graphs;
- ...

There are several approaches to the problem. We need to simplify the model first, otherwise there would be so many variables to consider that it would be impossible to analyze them all together. Then, we can focus on different aspects of the problem, using different models for each to capture the relevant dynamics. Finally, we can integrate these models to get a comprehensive understanding of the system as a whole. At the beginning it is a sort of *black-box*, in the sense that we do not know how the system works internally, but we can observe its inputs and outputs. The goal is to understand the internal mechanisms that drive the system’s behavior.

Definition: *Black-box models*

Black box models focus on modeling the relationships between inputs and outputs on a system, without explicitly considering the internal processes that generate those outputs from the inputs. They are widely used today and are based purely on observed input/output data from the system, without knowledge of the internal mechanism. See for example Neural Networks, statistical models or machine learning algorithms. They lack **explainability**.

We need something complex enough to capture the dynamics of the system, but simple enough to be tractable.

Agents are another way to model complex systems. Intelligent agents excel at modeling individual behavior and the emergence of global phenomena:

- Individual behavior is simple;
- Interactions are complex;
- Macroscopic behavior emerges from microscopic interactions.

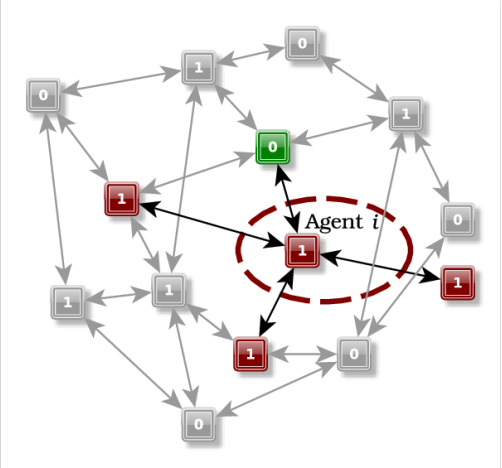


Figure 1.3: The world with agents

But how can we model a problem with agents? In the case of traffic, we can model each vehicle as an agent with individual behavior, represent traffic signals and lights as agents interacting with vehicles, incorporate pedestrians agents with their own behaviors and model the road environment as a graph of interconnected agents representing intersections and road segments.

Interactions emerge from local agent behavior and each agent models an individual component with autonomy.

Tip:

This is useful for simulation and comparing with real world scenarios.

At the end we can simulate different scenarios by modifying parameters, or analyze the emergence of traffic jams from individual behaviors, explore strategies to optimize circulation or calibrate and validate the model with real city data.

Intelligent Agent Systems are systems composed by multiple interacting agents, in which intelligence emerges from interactions among agents and between agents and the environment.

Such intelligence encompasses learning, reasoning, social interaction and other capabilities that are human-like.

Examples of intelligent agents can be found in various domains:

- An automated personal assistant that can make reservations, respond to emails and organize our schedule autonomously;
- An intelligent virtual tutor that can teach us new topics and adapt to our learning style;
- Autonomous cars or delivery drones.

Observation: ***Agents today***

Recent advancements include interactions with the user while maintaining **context**:

- **Memory Integration**: persistent memory to maintain long-term context;
- **Multi-modal Capabilities**: Understanding text and images;
- **Execution of actions**: interacting with external tools and APIs.

Advancements in cooperative AI has been also made, enabling agents to collaborate at solving problems in real-world applications such as in healthcare, logistics and resource management. Moreover, cognitive architectures are being developed to endow agents with human-like reasoning and decision-making capabilities, enhancing their ability to operate in complex and dynamic environments.

- **SOAR Cognitive Architecture**
- **ACT-R Cognitive Architecture**
- **Transformers**

2 Intelligent Agents and MAS

Nowadays, more and more devices are present in every imaginable context, along with an increasing number of interconnections between them. Moreover, smarter tasks and a tendency to make computers behave like humans are also being pursued.

A software, to support these trends, needs to:

- be independent;
- Seek the required goals autonomously;
- interact with other systems and humans.

To build agents capable of such tasks, they need to be endowed with some level of **intelligence** and **interaction**, not only independent, autonomous and able to solve the delegated task.

Definition: *Intelligent Agent*

An agent is a computer system capable of acting **autonomously** and **flexibly** in an environment to achieve the assigned objectives.

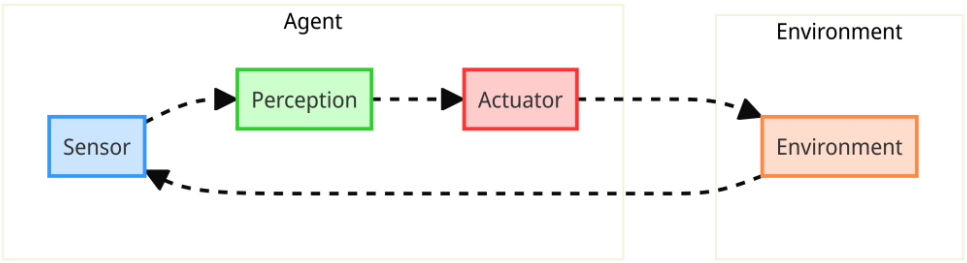


Figure 2.1: Intelligent Agent

An agent is made by:

- **Perception** *P*: agent's inputs. Perception adjusts the agent's state *S*;
- **State** *S*: internal representation of the world;
- **Action** *A*: agent's output, acting in the environment.

Moreover, an agent must be **flexible**, meaning reactive, proactive and social. Environmments, instead, must be:

- **Observable**: the agent can acquire complete information;
- **Deterministic**: given an action, we know the resulting state;
- **Episodic**: an action depends on a limited number of previous states;
- **Static**: the environment does not change while the agent is acting;
- **Discrete**: the environment can be divided into distinct states.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.2: Agent Environment Types

It can be mathematically formalized as a tuple of four components:

$$\text{Agent} = \langle P, A, S, f \rangle$$

where:

- P is the set of perceptions;
- A is the set of actions;
- S is the set of internal states;
- $f : (P \times S \times A)$ is the state transition function.

S is updated base on the actions A taken and the observations P made. P is used to update the agent's internal state S (internal model of the world). A is generated from the agent's state S and current perception P. The function responsible for this mapping is called the **agent function** and referred as π (often a probability distribution).

The agent can have a performance evaluation function that measures how well it is performing in achieving its goals. This function can be used to guide the agent's decision-making process and improve its performance over time. Usually a **reward function** R is used to provide feedback to the agent about the desirability of its actions.

The agent's objective is to maximize the cumulative reward over time, which can be formalized as:

$$\max_{\pi} \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$$

where:

- s_t is the state at time t;
- a_t is the action taken at time t;
- γ is a discount factor (between 0 and 1) that determines the importance of future rewards.

To achieve this, the state-value function $V(s)$ is used to estimate the expected cumulative reward starting from state s and following a particular policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s\right]$$

where $\mathbb{E}_{\pi}$ denotes the expected value given that the agent follows policy π.

The policy $\pi(a|s)$ is a function that assigns a state s and a perception p to an action a of the agent, i.e., $\pi : S \times P \to A$. The agent's objective is to find the optimal policy π^* that maximizes the

state-value function:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s)$$

for all states s.

The agent's interaction with the environment can be described in terms of a perception-action cycle. In each cycle, the agent receives a perception from the environment, updates its internal state based on the perception and the action taken, and selects a new action to perform in the environment.

The selection of the action can be made through a policy function that maps the agent's internal state to a probability distribution over the possible actions. The policy function can be either deterministic or stochastic, depending on whether the agent always selects the same action for a given state or selects actions based on a probability distribution.

Simple Reactive Agents A reactive agent has a state transition function thet depends solely on the current perception, i.e., $\pi : P \to S$. It may have (simple) states that will be used to generate actions. If it has no states, then it is called a **simple reflex agent** and $\pi : P \to A$. The agent's policy depends only on the current state of the world, ignoring the history of perceptions.

These agents select actions based on the current perceptions, ignoring the rest of the perceptual history. The agent uses a set of condition-action rules, often known as production rules, to determine its action.

Example: ***Simple Reflex Agent***

Consider a thermostat agent:

- S is boolean and indicates whether the room temperature is too high or too low;
- P is the room temperature;
- A is either "turn heater on" or "turn heater off";
- f compares the current room temperature P with the desired temperature stored in S to decide the appropriate action A.
- π decides the action based on the current perception and the internal state.

Model-based Reactive Agents They use a more complex model (S) than the previous ones. They take into account the decision of the action to take, maybe using historical data stored in S. A is generated from the state S and the agent's current perception P using the policy function $\pi : P \times S \to A \times S$. These agents maintain some type of internal model of the world, which allows them to take into account areas that they cannot currently perceive. The model also enables agents to anticipate the outcome of actions in situations where perceptions are insufficient.

Example: ***Model-based Reflex Agent***

Consider a chess-playing agent:

- P is the move text and turn;
- A is the valid piece moves;
- S is the internal representation of the chessboard and pieces;
- f is the chess rules to determine if an action is legal and to update the board state after a;
- π is the game model and the strategy to select the best move based on the current board state and perception.

Deliberative Agents They have an internal component that maintains a model of the world. They use that model to make decisions, being more ambitious than reactive agents. Their state transition function is more complex and depends on both the current perception and the agent's internal world model. The tuple can be written as $\langle P, A, S, f, M \rangle$, where M is the world model. They divide in:

- **Goal-based**: agents that act to achieve specific goals, using their internal model to evaluate the desirability of different states. Planning involves finding the best sequence of actions to achieve the agent's goals;
- **Utility-based**: agents that aim to maximize their expected utility, considering both the likelihood of different outcomes and their preferences over those outcomes.

It is important to note that these types of agents are not mutually exclusive, and many agents may combine elements from multiple categories to achieve their objectives effectively. Furthermore, each type of agent may be more suitable for specific tasks or environments, depending on the complexity of the problem and the available resources.

Learning Agents They have the ability to improve their performance and adapt to changes based on their experience. They usually have five main components:

- **Performance component**: Makes decisions, it is responsible for selecting actions based on perceptions;
- **Learning component**: Responsible for making modifications to the agent to improve performance;
- **Critique component**: This component provides feedback to the learning component about how the agent is performing;
- **Problem generator component**: This component suggests actions that will lead the agent to new and potentially informative experiences;
- **World model**: This component allows the agent to predict how its actions will change the world.

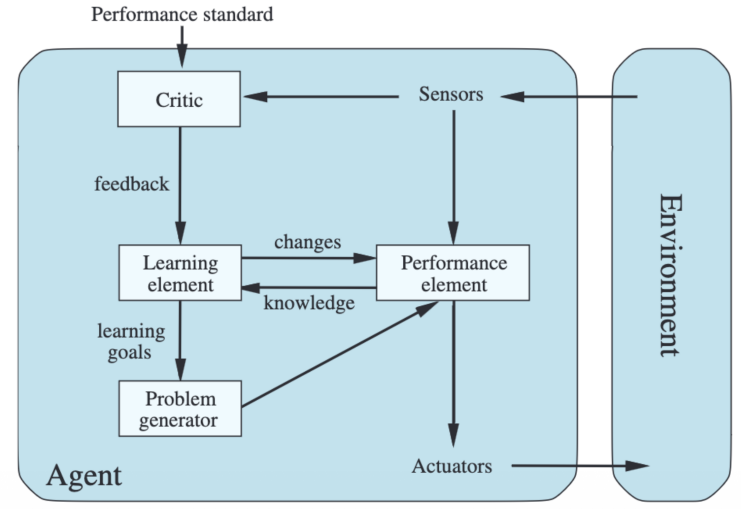


Figure 2.3: Learning Agent Architecture

Observation: ***Agents vs Objects***

Agents differ from traditional objects in several key ways:

- **Autonomy**: Agents operate independently and make decisions without direct human

intervention, while objects typically require external control.

- **Proactivity**: Agents can take initiative to achieve their goals, whereas objects are generally passive and respond only to external stimuli.
- **Social Ability**: Agents can interact and communicate with other agents or humans, while objects usually lack this capability.
- **Adaptability**: Agents can learn from their experiences and adapt their behavior over time, while objects typically have fixed behaviors defined by their programming.

Definition: ***BDI Agents***

BDI agents are a class of deliberative agents that take into account both the agent's knowledge of the world and its goals and plans to make decisions.

The main components of a BDI agent are:

- **Beliefs (what it knows)**: Information the agent has about the world, itself, and other agents. This knowledge can be incomplete or incorrect.
- **Desires (what it wants)**: The objectives or goals the agent aims to achieve. Desires represent the agent's motivations and can sometimes conflict with each other.
- **Intentions (what it plans to do)**: The plans and actions the agent has committed to in order to achieve its desires. Intentions guide the agent's behavior and decision-making process.

There are also behavioral structures that define how BDI agents operate:

- **Perception**: The agent perceives changes in the environment and updates its beliefs accordingly;
- **Rule**: A function that runs in each iteration to infer new desires or beliefs from the agent's current beliefs and desires;
- **Plan**: Defined intensions that the agent can execute to achieve its desires.

The BDI agent begins with a set of beliefs and desires. It then uses a reasoning process to generate intentions and select an action to perform based on its current beliefs and intentions. After executing the action, the agent updates its beliefs based on the new information received from the environment, and the cycle continues.

2.1 Communication

In communication, common terms are needed to describe domains and tasks. For example, what do we mean by 'size' or 'fast'? **Ontologies** offer a shared basis of understanding for agents to communicate effectively.

Definition: ***Ontology***

An ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts. It provides a shared vocabulary and structure for agents to communicate and reason about the domain. Ontologies are used in various fields, including artificial intelligence, knowledge representation, and information science, to facilitate interoperability and data sharing among different systems.

They allow for the estabilisment of common terminology and define relationships between concepts, helping to ensure that agents have a shared understanding of the message they exchange.

Ontologies typically contain a structural component that is the *taxonomy* of classes and subclass relationships, along with the definitions of the relationships between them.

Observation: ***Isn't it a database?***

Well... not exactly. An ontology captures the semantics and relationships within the domain, while the field desciption in the database focuses on the structure and basic data types needed to store the data. They have significant differences in terms of structure and purpose:

- **Purpose**: Ontologies are designed to represent knowledge and facilitate reasoning about a domain, while databases are primarily used for storing and retrieving data efficiently.
- **Structure**: Ontologies use classes, properties, and relationships to model complex relationships and hierarchies, while databases use tables, rows, and columns to organize data.
- **Semantics**: Ontologies capture the meaning and context of the data, enabling reasoning and inference, while databases focus on the syntactic representation of data without inherent semantics.

An ontology consists of several components that work together to capture the knowledge and structure of a specific domain. The main components are:

- Classes and Class Hierarchies;
- Instances;
- Properties;
- Constraints;
- Rules;
- Axioms.

Where the first two represent the different types of objects in a domain. An instance is an object of the class.

Properties They define attributes or relationships between classes in an ontology. Properties can have a *domain* (the class to which the property applies) and a *range* (the type of value that the property can have). There are two main types of properties:

- **Instances**: describe characteristics of individual instances of classes;
- **Relations**: describe relationships between different classes or instances.

Example: ***Properties***

```
1 Instance property: name (domain: Animal, range: string)
2 Relation property: hasParent (domain: Person, range: Person)
```

Restrictions Impose conditions or rules on the classes and properties in an oncology. These restrictions can include cardinality constraints, range restrictions, allowed value restrictions, among others.

Example: *Restrictions*

```
Cardinality restriction: A Person must have exactly two parents.
Range restriction: The age property of a Person must be an integer
    between 0 and 120.
```

Rules Rules allow for the definition of additional logic or inferences within an ontology. They can be logical rules or rules based on the semantics of concepts and relationships within the domain.

Example: *Rules*

```
If a Person is a Parent, then they must have at least one Child.
If an Animal is a Mammal, then it must give birth to live young.
```

Axioms Axioms are statements that establish fundamental truths or assumptions within an ontology. They provide additional information and constraints about classes, properties and relationships in the domain. Axioms can be of different types: *equivalence*, *inheritance* and *restriction*.

Example: *Axioms*

```
Equivalence axiom: A Bachelor is equivalent to an Unmarried Man.
Inheritance axiom: All Dogs are Animals.
Restriction axiom: A Person can have at most two biological parents
    .
```

2.1.1 Communication Acts

How do we exchange messages between agents? Communication acts are a way to structure the messages that agents exchange, i.e., 'request', 'report', 'confirmation', 'negotiation', etc. Ontologies can include communication acts. These definitions allow agents to understand the purpose and intention behind messages and to engage in negotiations and coordination of actions. By using the ontology to interpret communication acts, agents can make decisions based on the underlying semantics and coordinate their actions more effectively. They are typically defined using a communication protocol that specifies the syntax and semantics of the messages exchanged between agents.

Communication languages are a set of rules that agents use to communicate with each other. They define the syntax and semantics of the messages exchanged between agents, allowing them to understand each other's intentions and actions. Examples of communication languages include KQML (Knowledge Query and Manipulation Language) and FIPA ACL (Foundation for Intelligent Physical Agents Agent Communication Language). Communication languages specify how messages are structured and what the different communication acts mean.

- **Ontology** provides a common semantic structure and definitions of concepts;
- **Communication language** establishes the rules and format for message exchange.

Together, they enable semantic interoperability, precise communication and shared understanding between agents.

KQML Developed in the 1990s, it is still used toda in some systems. It defines a syntax for messages and a set of predefined communication acts. Here, structured messages are used to query and manipulate knowledge between agents.

Example: *KQML Message*

```
(Send
    :receiver AgentB
    :content (ask-one
                :receiver all
                :content (predicate "p" :arg1 "x")))

(Tell
    :sender AgentB
    :content (result
                :in-reply-to (ask-one :receiver all :content (
                   predicate "p" :arg1 "x")
                :content "Yes")))
```

In practice, various types of messages and actions can be used to perform more complex queries and manipulate knowledge between agents. KQML is a specific communication language: the agents involved must understand and process the messages in KQML format.

FIPA ACL ACL is another communication language that includes a broader set of communication acts and more sophisticated protocol mechanisms than KQML.

Example: *FIPA ACL Message*

```
(REQUEST
    :sender (agent-identifier :name AgentA :addresses (sequence
       AgentA-address))
    :receiver (set (agent-identifier :name AgentB :addresses (
       sequence AgentB-address)))
    :content "System status?")
(INFORM
    :sender (agent-identifier :name AgentB :addresses (sequence
       AgentB-address))
    :receiver (set (agent-identifier :name AgentA :addresses (
       sequence AgentA-address)))
    :content "All systems operational.")
```

FIPA is an international organization dedicated to promoting and standardizing technologies related to Intelligent Agents. It focuses on developing standards and specifications to facilitate interoperability and information exchange among intelligent agents.

- **Standards and specifications**: FIPA develops technical standards and specifications that establish a common framework;
- **Interoperability**: FIPA standards enable intelligent agents from different platforms and systems to communicate and collaborate effectively;
- It was a key organization in promoting and standardizing technologies related to intelligent agents.

Examples of FIPA protocols include:

Protocol	Description
FIPA Request	An agent requests another agent to perform a specific action or task.
FIPA Contract Net	A negotiation protocol where an agent announces a task and other agents bid to perform it.
FIPA Subscribe	An agent subscribes to receive updates or notifications from another agent about specific events or changes.
FIPA Query	An agent queries another agent for information or data.

Table 2.1: FIPA Protocols

Example: ***Request-proposal protocol***

An agent requests a task from another agent and receives a proposal on how the task will be carried out. The requesting agent sends a `request` message that includes the details of the task being requested. The agent receiving the request sends a `proposal` message that includes an offer to perform the task and the details of how it will be carried out.

Within a FIPA communication protocol, agents can perform different communication acts. These acts include requesting information, making proposals, accepting or rejecting offers, notifying events and more. Agents carry out these acts using the messages and rules defined in the protocol.

2.2 Interacting through cooperation

2.2.1 Cooperation

Cooperation among intelligent agents is essential for achieveing shared goals. Agents need to share **information**, **assign tasks**, **make joint decisions** and **coordinate actions**. Cooperation can be challenging due to differences in the capabilities and knowledge of the agents, as well as the possibility of **selfish** or **malicious behavior**.

Observation: ***Difference from traditional distributed systems***

In traditional distributed systems, components work together to achieve a common goal, often relying on centralized control and predefined protocols. In contrast, multi-agent systems consist of autonomous agents that interact and cooperate in a decentralized manner, often with varying degrees of knowledge and capabilities.

Cooperative Distributed Problem Solving (CDPS) studies how a network of solvers can work together to solve problems that are beyond their individual capabilities. Each node in the network is capable of solving problems in a sophisticated and independent manner, but cannot complete them without cooperation. **Benevolence** is assumed, meaning that the nodes implicitly share a common goal (no conflict among them).

- **Coherence** refers to how well the multi-agent system behaves as a unit across some dimension of evaluation;

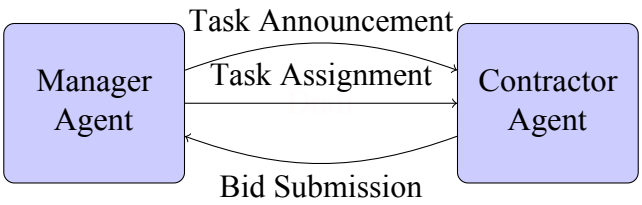
- **Coordination** refers to the degree to which agents can avoid individual activities by synchronizing and aligning their actions.

In a perfectly coordinated system, agents will not accidentally interfere with each other's sub-goals while attempting to achieve a common goal.

Task Sharing The problem is decomposed into subproblems and assigned to different agents. Task assignment to individual agents may involve agreements, neegotiations or auctions.

Task assignment is a key component of cooperation. Tasks must be divided and assigned efficiently to make the most of each agent's capabilities. A common approach is the **Contract Net Protocol (CNP)**, which is an auction protocol in which agents are either **manager** or **contractor**.

1. **Task Announcement**: The manager agent announces a task to be performed, specifying the requirements and constraints;
2. **Bids**: Contractor agents evaluate the task and submit bids, indicating their willingness to perform the task and the associated cost or resources required;
3. **Task Assignment**: The manager agent evaluates the bids and selects the most suitable contractor based on criteria such as cost, quality or timeliness. The task is then assigned to the selected contractor;
4. **Task Execution**: The contractor agent performs the assigned task and reports the results back to the manager agent upon completion.



The CNP is a decentralized mechanism, meaning there is no single point of control of failure. It is flexible and can adapt to a wide variety of situations and needs. However, the effectiveness of the CNP can be affected by the quality of the bids and the agents' ability to evaluate and select the best offers. The best strategy for bidding and task assignment may vary depending on the specific context and requirements of the system.

Result Sharing Agents share subproblem information proactively and reactively. This is useful when agents have different knowledge or capabilities.

Sharing results is a method in which agents cooperatively exchange information while developing a solution. These initial results are often the solution to small problems, which are then refined into larger and more abstract solutions. Problem solvers can improve the group's performance in sharing results:

- **Trust**: Independently derived solutinos can be verified against each other, highlighting potential errors and increasing confidence in the overall solution;
- **Completeness**: Sharing their local views to achieve a better global perspective;
- **Accuracy**: Sharing results to ensure that the accuracy of the global solution is increased;
- **Timeliness**: Although an agent could solve a problem by itself, by sharing a solution, the result could be derived more quickly.

Warning: *Inconsistencies*

Agents may have inconsistencies regarding their beliefs and their goals/intentions. Inconsistencies between goals generally arise because agents are assumed to be autonomous and therefore do not share common objectives and they are a major problem in cooperative systems. There are several sources of inconsistencies:

- The perspective that agents have will typically be limited;
- The sensors that agents have may be faulty, or the sources of information to which the agent has access may be faulty.

2.2.2 Coordination

It refers to the management of the interdependencies between the activities of agents. They may have a positive relationship (beneficial for at least one agent) or a negative relationship (destructive for at least one agent). Coordination in MAS is assumed to occur in real-time and agents must be able to recognize and manage these relationships during their activities. There are different types of coordination among agents:

Partial Global Planning Involves agents generating and following a global plan that only covers part of their activities. It is used when it is not feasible or efficient to plan all activities of all agents due to complexity of the system.

Example:

In public transportation, buses and trains may have their own schedules and routes, but they also need to coordinate with each other to ensure smooth transfers for passengers. Partial global planning allows each mode of transportation to plan its own activities while still considering the overall system's needs.

- **Pros**: Allows for high efficiency in task resolution and high coherence in decision making;
- **Cons**: Requires good communication and synchronization among agents and is not ideal for highly dynamic or uncertain environments where plans may need to be changed frequently.

Joint Intentions Agents form and act upon intentions that are shared among them. They commit to working together toward a common goal and to communicate with each other about their intentions and actions.

Example:

In a search and rescue operation, multiple agents (e.g., drones, robots, and human responders) may form joint intentions to locate and assist survivors. They share information about their findings and coordinate their actions to cover the search area effectively.

- **Pros**: Enhances cooperation and coordination among agents, leading to more effective problem-solving;
- **Cons**: Requires a high level of trust and communication among agents, which may not always be feasible in dynamic or uncertain environments, especially with a large number of agents or with conflicting goals.

Mutual Modeling Involves agents building and using models of each other to anticipate their actions and coordinate their own actions accordingly.

Example:

In a team of autonomous vehicles, each vehicle may use mutual modeling to predict the movements and intentions of other vehicles on the road. This allows them to coordinate their actions, such as merging or changing lanes, to ensure safe and efficient traffic flow.

- **Pros**: Enables agents to anticipate and adapt to each other's actions, leading to more effective coordination in dynamic and complex environments;
- **Cons**: Requires accurate and up-to-date models of other agents, which could be expensive in computational terme and lead to errors if the models are not accurate.

Social Norms and Laws Agents follow established social norms and laws to guide their behavior and interactions with each other.

Example:

In a multi-agent system for online marketplaces, agents representing buyers and sellers may adhere to social norms and laws regarding fair pricing, honest communication, and dispute resolution. This helps maintain trust and cooperation among participants.

- **Pros**: Provides a clear framework for agent behavior, promoting cooperation and reducing conflicts;
- **Cons**: May limit agent autonomy and flexibility, especially in dynamic or uncertain environments where norms and laws may need to be adapted.

2.2.3 Planning

Planning in multi-agent systems involves the coordination and collaboration of multiple agents to achieve shared goals. Each agent may have its own individual goals and plans, but they must also consider the actions and plans of other agents in the system. There are three main approaches to planning in multi-agent systems:

Centralized Planning for Distributed Plans Develops a plan for a group of agents, where the division and order of work are defined. This plan is distributed among the agents, who execute their part of the plan.

- **Pros**: Allows for coordination and efficiency, having a single centralized agent with a global view efficiently assigning tasks to agents;
- **Cons**: If the centralized planner fails, the entire system may be compromised; it may not scale well with a large number of agents or in geographically dispersed systems or those with private information that cannot be shared with a central planner.

Distributed Planning A group of agents cooperate to form a centralized plan. The agents forming the plan will not be the ones executing it.

- **Pros**: Planning is more robust to failures and is more suitable if agents have private or dispersed information;

- **Cons**: The challenge is to coordinate the actions of agents to avoid conflicts. Moreover, it is more difficult to achieve an efficient plan (lack of global view).

Distributed Planning for Distributed Plans A group of agents cooperates to form individual action plans, dynamically coordinating their activities. When coordination problems arise, they may need to be resolved through negotiation.

- **Pros**: More flexible and scalable, as agents can create and execute their own plans independently, is also more suitable for situations where agents have private information;
- **Cons**: Need for sophisticated mechanisms for coordination and conflict resolution among agents' plans.

2.2.4 Coalitions

Definition: *Coalition*

A coalition is a subset of agents that work together to achieve common goals. They allow for the distribution of tasks and collaboration to improve efficienct and achieve goals that cannot be reached individually.

Coalitions are situated in both coordination and planning of MAS:

- **Coordination**: Coalition agents must coordinate to work effectively together;
- **Planning**: Coalition agents must engage in joint planning to achieve their objectives.

Agents identify potential allies based on their capabilities and objectives. The terms of cooperation are negotiated, including the division of tasks and rewards. The challenges include determining **when** and **how** to form coalitions can be a complex problem and the **balancing** of the agents' interests with the benefits of cooperation.

Example: *Applications of coalitions in MAS*

- **Resource Management**, where agents cooperate to share resources;
- **Problem-solving**, where agents work toghether to solve complex problems;
- **In dynamics environments**, where agents form and reform coalitions in response to chenges in the environment.

2.2.5 Voting in MAS

Voting procedures are methods that allow intelligent agents to make collective decisions. Voting is crucial in MAS to facilitate cooperation and collective decision-making. Agents vote to make decisions that affect the entire coalition or the MAS as a whole.

Voting procedures can vary in **complexity**, from simple majority voting to weighted voting mechanisms or more complex procedures. Agents can cast their votes based on their **knowledge**, **goals** and **strategies**.

- **Majority Voting**: Each agent casts a vote for a particular option, and the option with the most votes wins;
- **Weighted Voting**: Each agent's vote is assigned a weight based on its importance or expertise, and the option with the highest total weight wins;
- **Consensus Voting**: Agents discuss and negotiate to reach a unanimous decision;

- **Ranked Voting**: Agents rank the options in order of preference, and the option with the highest overall ranking wins.

Voting facilitates cooperation and collective decision-making in MAS. It promotes **fairness** by giving all agents the opportunity to express their preferences and helps resolve conflicts and make decisions in uncertain situations.

Warning: *Challenges and considerations*

Agents may have incentives to manipulate the voting process to achieve their desired outcomes. Designing voting procedures that are robust against strategic manipulation is a significant challenge in MAS and so voting systems must be designed to be resilient to such manipulations. Moreover, the **computational efficiency** of the voting process must be considered. They should be **fair** and **transparent**.

2.2.6 Manipulation and Reputation

Manipulation and reputation are two crucial aspects of MAS. The first can be detrimental to the system's performance, while the second can promote cooperation and discourage selfish or malicious behaviors.

- **Manipulation**: occurs when an agent influences the decision-making of others for personal gain, for example by provising false information to change the outcome of a voting process in their favor; it can have negative consequences for the system as a whole, such as suboptimal decisions or reduced cooperation among agents. It is important to implement mechanisms to prevent manipulations, such as:
  - Manipulation-resistant voting procedures;
  - Secure communication protocols;
  - Anomaly detection systems.
- **Reputation** is a measure of an agent's past behavior: agents with good reputation are often seen as more reliable and cooperative, while agents with bad reputation may be considered potentially harmful or uncooperative. Reputation can be an effective mechanism to incentivize cooperation and discourage selfish or malicious behaviors. Reputation management involves collecting, storing and sharing information about the past behavior of agents, using a rating system, a trust system or a recommendation system.

Warning: *Challenges*

- The need to protect the **privacy** of agents while collecting and sharing reputation information;
- The possibility of false or manipulated ratings that can distort the reputation system;
- The need to make decisions based on **incomplete** or **uncertain** information about agents' behavior.

2.3 Competition and Negotiation

Competition and **negotiation** are two essential aspects in MAS. Agents may compete for limited resources or negotiate to achieve common goals.

Competition It occurs when the agents seek to maximize their individual benefits, often at the expense of others and happen when resources are scarce and agents have opposing interests.

Negotiation It is a communication and decision-making process in which agents try to reach an agreement. It can be:

- **Cooperative**, where agents work together to achieve a common goal;
- **Competitive**, where agents have opposing interests and try to maximize their individual benefits.

There are several negotiation techniques:

- Bilateral negotiation;
- Multilateral negotiation;
- Auction-based negotiation;

The choice of negotiation technique depends on the specific context and requirements of the MAS.

Warning: *Challenges in competition and negotiation*

- The need to balance individual interests with the overall goals of the system;
- The complexity of negotiation processes, especially in multilateral negotiations;
- The possibility of manipulation or deception by agents to gain an advantage.

Utility is a measure of the satisfaction or benefit an agent gains from a process. Agents make decisions based on maximizing their expected utility and an agent's preferences reflect their relative evaluation of different outcomes. **Preferences** can be represented as a utility function, which assigns a numerical value to each possible outcome.

Therefore, a **dominant strategy** is a strategy that provides the agent with the highest utility, regardless of the strategies chosen by other agents. In some games, a dominant strategy may not exist.

2.3.1 Game Theory

Dilemmas and **Games** are situations where agents interact and make decisions. **Game Theory** is an essential tool for modeling and analyzing these situations.

Definition: *Game Theory*

Game theory is the study of mathematical models of strategic interaction among rational decision-makers. It provides a framework for analyzing situations where the outcome of one agent's decision depends on the decisions made by other agents.

Games can be classified based on several criteria:

- **Number of players**: two-player games, multiplayer games;
- **Information availability**: perfect information games, imperfect information games;
- **Cooperation**: cooperative games, non-cooperative games;
- **Zero-sum vs non-zero-sum**: in zero-sum games, one player's gain is another player's loss, while in non-zero-sum games, both players can benefit from cooperation.

And exist different types of games:

- **Cooperative games**: players can form coalitions and share their payoffs;

- **Competitive games**: players compete against each other to maximize their own payoffs;
- **Symmetric games**: players have identical strategies and payoffs;
- **Asymmetric games**: players have different strategies and payoffs.

Definition: ***Nash Equilibrium***

A Nash Equilibrium is a solution concept in game theory where no player can improve their outcome by unilaterally changing their strategy, given the strategies of the other players. In other words, it is a stable state in which each player's strategy is optimal, considering the strategies chosen by the other players.

Each game can have one, several or no Nash equilibria at all. Finding the Nash equilibria of a game can be a complex task, especially in games with many players and strategies.

2.3.2 Auctions

Auctions are effective mechanisms for **allocating** scarce resources among multiple agents. They provide a structured way for agents to express their preferences and compete for resources. Intelligent agents can use them to determine the value of a resource and allocate it fairly and efficiently. There are different types of auctions:

- **Common Value**: The value of the item is the same for all bidders (may be uncertain or unknown prior to the auction), but bidders have different information about that value;
- **Private Value**: Each bidder has their own individual valuation of the item, which is independent of other bidders' valuations, and each bidder does not know the valuations of other bidders;
- **Correlated Value**: The value of the item is a blend of private and common value. Bidders have private information that helps them estimate the value of the item. Here, each agent has a private estimate of the value of the item, but these estimates are correlated because they are based on shared common information;
- **Open Cry**: Bidders openly bid against each other, with each bidder able to see the current highest bid and respond accordingly;
- **Sealed Bid**: Bidders submit their bids without knowing the bids of other participants. The highest bid wins, but bidders do not have the opportunity to adjust their bids based on others' offers.
- **One-shot Auction**: Bidders submit their bids only once, and the auction concludes after a single round of bidding;
- **Ascending Auction**: Bidders can place multiple bids, with each bid being higher than the previous one. The auction continues until no higher bids are made.

Each auction type has its own advantages and disadvantages, and the choice of auction type depends on the specific context and requirements of the MAS. For example, common value auctions can be useful in situations where the value of a resource is the same for all agents and each agent has an uncertain estimate of that value.

3
Agent Based Modelling

Multi-agent systems are applied in various domains to replicate commplex systems with:

- Multiple artificial or natural entities;
- Interactions among entities that produce emergent behaviors;
- The **microscopic** model is not sufficient to predict global system behavior;
- The focus is on **macroscopic** behavior.

Definition: ***Agent-Based Modelling (ABM)***

Agent-Based Modelling (ABM) is a computational approach for simulating the interactions of autonomous agents to assess their effects on the system as a whole.

It is:

- **Versatile and Heterogeneous**, since agents can represent different entities with diverse behaviors and attributes;
- **Agnostic**, since any architecture can be used and programmed in any language;
- **Generative**, since it focuses on how local interactions lead to global patterns.

A **model** is a simplified representation of a system, designed to facilitate understanding, analysis, and prediction of its behavior. The execution of the model is called a **simulation**. We require methods to capture and regenerate the complex characteristics of the environment. This is needed to describe a system, to explain its functioning and to predict its evolution. Scientific modeling requires mathematical and computational modeling to determine the consequences of model simplifications and assumptions.

- Formulate the appropriate questions;
- Hypotheses for essential processes and structures.

The modeling cycle consists of:

- **Observation**: Identify key components and interactions in the real-world system;
- **Model Design**: Define agents, environment, and interaction rules;
- **Implementation**: Develop the model using suitable software tools;
- **Simulation**: Run the model to observe emergent behaviors and patterns;
- **Analysis**: Evaluate the results, validate against real-world data, and refine the model as needed.

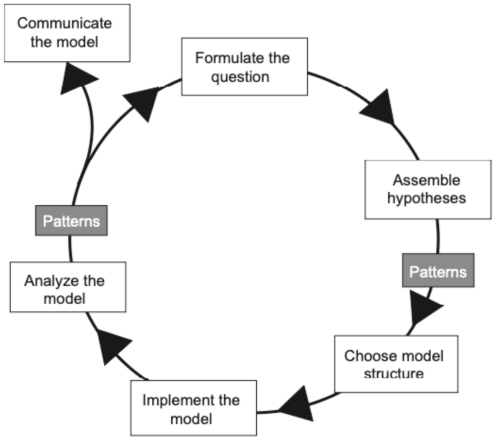


Figure 3.1: The modeling cycle.

The **world** is the environment where the simulation is executed. It contains not only the physical specifications but also the behavioral rules of the objects that compose it. Each decision influences the movement restrictions between agents and the interactions they can have with each other and with the environment itself. We have four types of environments:

- **Discrete space** (cellular space, grid), where space is divided into cells;
- **Continuous space**, where agents can move freely in a continuous environment
- **Geographic space** (GIS), which represents real-world geographical features;
- **Topological space** (network), where agents are nodes connected by edges.

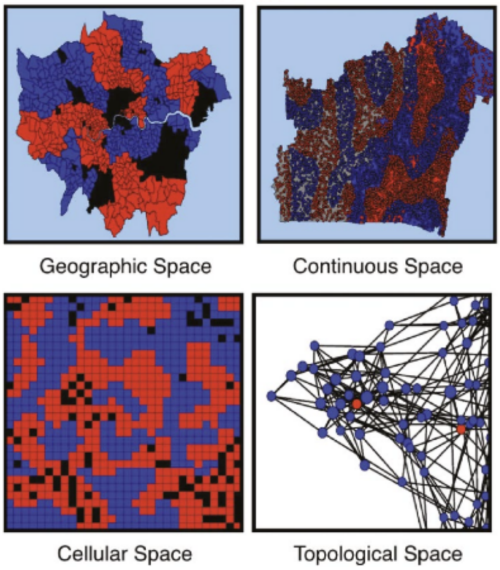


Figure 3.2: Types of environments in agent-based modeling.

A minimum time unit, the **step**, is assumed. In each step, agents can execute the actions they require, even if they don't have to follow the same order in each iteration of the simulation. A step:

- Must capture everything required by our simulation;
- Should be as short as possible;
- Is the same for every agent.

Agents can communicate with each other using **messages**, which can be direct (one-to-one) or broadcast (one-to-many). Communication can be synchronous (waiting for a response) or asynchronous (not waiting). The choice of communication method depends on the model's requirements and the desired level of interaction among agents. We have to keep in mind the distance, connectivity, information loss and other factors that can affect communication. Moreover, how are resources exchanged between agents? How is the behavior of agents modeled?

3.1 ODD Protocol

It is complicated to establish and communicate all the necessary characteristics of the model. An incomplete description eliminates **replicability**, and a non-reproducible model is not scientific. The **ODD Protocol** (Overview, Design concepts, Details) is a standardized framework for describing agent-based models. It provides a structured approach to document the key components and processes of the model, ensuring clarity and reproducibility. The ODD Protocol consists of three main sections:

- **Overview**: This section provides a high-level summary of the model, including its purpose, key entities, and processes. It outlines the main objectives of the model and the questions it aims to address.
- **Design Concepts**: This section delves into the theoretical underpinnings of the model. It discusses the fundamental concepts and principles that guide the design of the model, such as agent behaviors, interactions, and decision-making processes.
- **Details**: This section provides a comprehensive description of the model's implementation. It includes information on the model's structure, algorithms, parameters, and data sources. It also outlines the steps taken to validate and verify the model.

	Elements of the ODD protocol
Overview	1. Purpose and patterns
	2. Entities, state variables, and scales
	3. Process overview and scheduling
Design concepts	4. Design concepts • Basic principles • Emergence • Adaptation • Objectives • Learning • Prediction • Sensing • Interaction • Stochasticity • Collectives • Observation
Details	5. Initialization
	6. Input data
	7. Submodels

Figure 3.3: The ODD Protocol structure.

Pros of ABM Hypotheses are expressed at the individual level, allowing for detailed representation of agent behaviors and interactions. This bottom-up approach enables the emergence of

complex system dynamics from simple local rules. We can model the evolution of the system and models are experimental objects (simulations).

Cons of ABM Difficulty to reproduce the complexity of reality (micro/macro levels). High computational cost for large-scale simulations. Validation and verification can be challenging due to the stochastic nature of agent interactions.

ABM should be applied when:

- The system is composed of heterogeneous and autonomous entities;
- It is difficult to test hypothesis based solely on observations of the real system;
- It is possible to identify intermediate variables that link micro-level behaviors to macro-level outcomes;
- The relationships among entities are non-linear and dynamic;
- Changes at the macro level must be results, not inputs, of the model.

4
Learning in MAS

When we talk about learning we usually have in mind a single agent, which has to learn how to behave in an environment. He can learn in different ways but here we take into consideration only the learning through **experience**, generally mapped between the agent's internal state and its actions. This map can be reconstructed through different learning techniques, such as *reinforcement learning*, *Deep Learning* or *evolutionary algorithms*.

Extending the concept to a multi-agent system requires taking into account the actions from other agents, how to maximize the team outcome and how to coordinate the learning process. Therefore, it affects both the agent's perception within the group and the choice of actions considering the interaction within the group.

Warning: *Challenges*

Consider RL:

- The environment may be non-static;
- The states can be changed in unpredictable ways by other agents;
- But most importantly: **convergence is not guaranteed!**

An intuitive approach may be to consider the actions of other agents as part of the environment. This would allow us to use single-agent learning techniques, but it is not useful for complex systems, where the interaction between agents is relevant. So, learning can be seen as a search problem for all possible solutions in the joint action space of all agents. Problems here arise, since the complexity of the problem can increase exponentially just by adding a few agents.

Moreover, it is essential do design a **reward function** that encourages cooperation between agents, otherwise they may learn to behave selfishly, maximizing their own reward but not the global one.

Remember that for a single agent, if the environment is stationary and the agent experiments enough, convergence to the optimal policy is guaranteed. However, in MAS, multiple agents learn from the same environment facing unobservable actions and rewards of other agents, making the environment non-stationary from each agent's perspective. This means that convergence is not guaranteed anymore.

4.1 A little recall from single-agent RL

In single-agent RL, an agent interacts with an environment modeled as a Markov Decision Process (MDP). The agent observes the current state s, selects an action a, receives a reward r, and transitions to a new state s'. The goal is to learn a policy $\pi(a|s)$ that maximizes the expected cumulative reward over time.

$$\pi^* = \arg\max_{\pi} R(h_\pi)$$

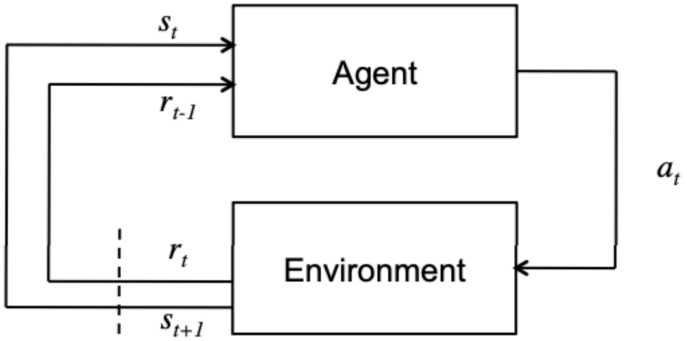


Figure 4.1: Single-Agent Reinforcement Learning Framework

Example: *Multi-Agent Deep RL*

Assume that:

- Agents must maximize the group's reward;
- Agents have partial and private observations;
- They cannot communicate explicitly;
- They must learn a cooperative task through obseration.

The first solution to this problem is to use a **Centralized Model**, meaning assuming a joint model of the actions and observations of all agents. Problems with this approach are the centralization of training and execution and the exponential growth in complexity with the number of agents. Even with a factorization of the joint policy, the complexity remains exponential ($O(|A|^n) \rightarrow O(n|A|)$).

A second solution is to use a **Concurrent Model** in which each agent learns its individual policy. Here we have private actions and states for each agent, independent policies and decentralized execution. This is easy to use in heterogeneous systems but suffers from limited scalability and dynamic environments (each agent is independent now).

Finally, a better model is to use **Shared Parameters**, assuming homogeneous agents using the same policy trained simultaneously with the experience of all agents. This allows for different behaviors since each agent perceives different observations.

4.2 Evolutionary Strategies

For a visual reference guide, follow this link.

Not everything is backpropagation and gradient descent. There are many problems where these techniques cannot be used, for example when the function to optimize is not differentiable or when the search space is discrete or highly non-linear. In these cases, **Evolutionary Strategies** (ES) can be a good alternative. This article from OpenAI gives a good overview of the topic. Evolutionary Strategies rival the performance of standard Reinforcement Learning techniques on modern RL benchmarks (Atari/MuJoCo), while overcoming many of RL's inconveniences. Advantages include high parallelizability, robustness, structured exploration and the ability to optimize non-differentiable reward functions.

Definition: *Evolutionary Strategies*

Evolutionary Strategies are optimization algorithms that provide a set of candidate solutions to evaluate a problem. The evaluation is based on an objective function that takes a given solution and returns a single fitness value. The algorithm will produce the next generation of candidate solutions based on the fitness values of the current generation, using operations inspired by natural evolution, such as selection, mutation and recombination. The iterative process will stop once the best-known solution is satisfactory to the user.

Simple Approximation Extract sample from a population defined by a normal distribution and keep the best individuals to create the next generation. Repeat until convergence.

Genetic Algorithm Create a population of individuals represented by chromosomes (bit strings). Evaluate their fitness and select the best ones to create a new generation through crossover and mutation. Repeat until convergence.

Observation: *CMA-ES*

A more advanced technique is the **Covariance Matrix Adaptation Evolution Strategy** (CMA-ES). It adapts the covariance matrix of the search distribution to increase the probability of generating better solutions. This allows the algorithm to learn the shape of the objective function and adapt its search strategy accordingly, leading to faster convergence and better performance on complex optimization problems.

How to apply Evolutionary Strategies to Multi-Agent Systems? One way is to evolve a population of agents, where each agent is represented by a set of parameters (e.g., weights of a neural network). The fitness of each agent is evaluated based on its performance in the environment, and the best-performing agents are selected to create the next generation through mutation and crossover. This process is repeated until a satisfactory level of performance is achieved.

Neuroevolution **Neuroevolution** is a specific application of Evolutionary Strategies where the parameters being optimized are the weights and architecture of neural networks. In a multi-agent context, neuroevolution can be used to evolve a population of neural network-based agents, allowing them to learn complex behaviors and strategies through evolutionary processes. This approach can be particularly effective in environments where traditional gradient-based learning methods struggle, such as in non-differentiable or highly dynamic settings.

Essentially we are searching among a population of policies (NN) to find the most suitable one. Operators like mutation and crossover can be applied to the weights of the neural networks, allowing for exploration of the policy space. The fitness of each policy is evaluated based on its performance in the multi-agent environment, and the best-performing policies are selected to create the next generation. Over time, this evolutionary process can lead to the emergence of sophisticated behaviors and strategies among the agents.

4.3 Swarm Intelligence

Swarm Intelligence (SI) is a subfield of artificial intelligence that focuses on the collective behavior of decentralized, self-organized systems, typically composed of simple agents interacting locally with one another and their environment. The inspiration for SI comes from natural systems, such as ant colonies, bird flocking, fish schooling, and bee swarming, where complex global behaviors

emerge from the interactions of simple individuals following simple rules. This intelligence comes from the collaboration and communication between lesser intelligent agents, leading to the emergence of intelligent behavior at the group level.

By self-organizing the colony reduces entropy without any external interaction. There are different key features:

- **Robustness**: The system can adapt to changes in the environment and recover from failures of individual agents;
- **Flexibility**: The system can adapt to different tasks and environments;
- **Scalability**: The system can function effectively with varying numbers of agents.

Particle Swarm Optimization It it a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It solves problems by having a group of candidate solutions, called particles, move through the solution space. Each particle adjusts its position based on its own experience and the experience of neighboring particles, allowing the swarm to converge on optimal or near-optimal solutions over time.

Ant Colony Optimization Inspired by the foraging behavior of ants, it is a probabilistic technique for solving computational problems that can be reduced to finding good paths through graphs. Ants deposit pheromones on the paths they take, and other ants are more likely to follow paths with stronger pheromone concentrations. Over time, this leads to the emergence of optimal or near-optimal paths as the collective behavior of the ant colony.

Bee Colony Optimization Similar to ACO, it is inspired by the foraging behavior of honeybees. In this algorithm, artificial bees search for food sources (solutions) and share information about their quality with other bees. The collective behavior of the bee colony leads to the discovery of optimal or near-optimal solutions through exploration and exploitation of the solution space. The difference with ACO is that honeybees dance to communicate the location of food sources, while ants use pheromone trails.

4.3.1 Metalearning

Meta-Learning is a subfield of machine learning that seeks to design models that can learn more quickly and efficiently by leveraging prior knowledge from previous learning experiences. In the context of swarm intelligence, meta-learning can be applied to enable agents within a swarm to adapt their learning strategies based on the collective experiences of the swarm. The goal is to create systems capable of adapting to new tasks with few training examples, rather than requiring large datasets for each new task.

- **Faster Learning**: By leveraging prior knowledge, agents can learn new tasks more quickly, reducing the time and data required for training;
- **Flexible Adaptation**: Agents can adapt their learning strategies based on the specific characteristics of the new task, allowing for more effective learning in diverse environments;
- **Improved Generalization**: Meta-learning can help agents generalize their knowledge to new tasks, improving their performance across a wider range of scenarios;
- **Decentralization**: In a swarm context, meta-learning can be implemented in a decentralized manner, allowing individual agents to learn from their own experiences while also benefiting from the collective knowledge of the swarm.

Warning:

Metalearning introduces weaknesses such as **overfitting** and **higher complexity** due to the additional layer of learning. Overfitting can occur when the meta-learning model becomes too specialized to the training tasks, leading to poor generalization on new tasks. The increased complexity can also lead to higher computational costs and longer training times, which may be a concern in resource-constrained environments.

Generally, meta-learning involves a two-level learning process:

1. **Task level**: Agents learn to perform specific tasks using traditional learning algorithms (e.g., reinforcement learning, supervised learning). This level focuses on optimizing the agents' performance on individual tasks;
2. **Meta level**: Agents learn to improve their learning strategies across multiple tasks. This level focuses on optimizing the agents' ability to learn new tasks quickly and efficiently by leveraging prior knowledge from previous tasks.

In each iteration, the meta-learning model attempts to improve its abilitiy to adapt to new tasks.

Example: *Model-Agnostic Meta-Learning (MAML)*

One popular approach to meta-learning is **Model-Agnostic Meta-Learning (MAML)**. MAML aims to find a set of model parameters that can be quickly adapted to new tasks with only a few gradient updates. The key idea is to optimize the model's initial parameters such that they are sensitive to changes in the task, allowing for rapid adaptation. The algorithm adjusts the model parameters in the direction that reduces the error on the largest possible number of tasks after a small number of training steps. This is achieved by computing the gradients of the loss function with respect to the model parameters and updating them accordingly.
In the context of swarm intelligence, MAML can be applied to enable agents within a swarm to quickly adapt their learning strategies based on the collective experiences of the swarm. This can lead to improved performance on new tasks and environments, as agents can leverage prior knowledge to learn more efficiently.

Finally, **Multi-Agent Meta-Learning** is an approach that combines concepts of meta-learning and multi-agent learning:

- **Individual agents as learners**: Each agent in the multi-agent system is treated as an individual learner that can adapt its learning strategy based on its own experiences and the experiences of other agents;
- **Learning through experience**: Agents learn to improve their learning strategies by interacting with the environment and other agents, allowing them to adapt to new tasks and environments more effectively;
- **Meta-Learning for rapid adaptation**: The multi-agent system employs meta-learning techniques to enable agents to quickly adapt their learning strategies based on the collective experiences of the swarm, leading to improved performance on new tasks and environments;
- **Cooperative and competitive learning**: Agents can learn to cooperate or compete with each other, depending on the nature of the task and the environment, allowing for more complex and dynamic interactions within the multi-agent system;
- **Communication and coordination**: Agents can share information about their learning experiences and strategies, enabling them to coordinate their actions and improve their overall

performance as a group;

- **Distributed learning**: The multi-agent meta-learning approach can be implemented in a distributed manner, allowing agents to learn from their own experiences while also benefiting from the collective knowledge of the swarm.

Observation: ***Future challenges of Meta-Learning***

- **Improve efficiency**: Develop more efficient meta-learning algorithms that can scale to larger multi-agent systems and complex environments;
- **Reduce overfitting**: Design techniques to mitigate overfitting in meta-learning models, ensuring better generalization to new tasks and environments;
- **Understandability**: Enhance the interpretability of meta-learning models, allowing for better understanding of how agents adapt their learning strategies.

5 How to use Gama Platform

In this chapter we will see how to use Gama Platform to create multi-agent simulations. Gama is an open-source platform for building spatially explicit agent-based simulations. It provides a user-friendly interface and a powerful modeling language that allows users to create complex models with ease. The generality of the agent-based approach advocated by GAMA is accompanied by a high degree of openness, which is manifested, for example, in the development of plugins designed to meet specific needs, or by the possibility of calling GAMA from other software or languages (such as R or Python)

- **Data-driven models**: GAMA offers the possibility to load and manipulate easily GIS (Geographic Information System) data in the models, in order to make them the environment of artificial agents;
- **GAML**: it is possible to create a simulated world, declare agent species, assign behaviors to them and display them and their interactions. GAML also offers all the power needed by advanced modelers: being an agent-oriented language coded in Java, it offers the possibility to build integrated models with several modeling paradigms, to explore their parameter space and calibrate them and to run virtual experiments with powerful visualization capabilities, all without leaving the platform;
- **Declarative User Interface**: The 3D displays are provided with all the necessary support for realistic rendering. A rich set of instructions makes it easy to define graphics for more dashboard-like presentations.

5.1 Platform

For more information and for visual examples, please refer to the official website: `https://gama-platform.github.io/`

GAMA consists of a single application that is based on the RCP architecture provided by Eclipse. Within this single application software, often referred to as a platform, users can undertake, without the need of additional third-parties softwares, most of the activities related to modeling and simulation.

5.1.1 Installation

1. **Download** → Go to the download page and pick the correct version for your operating system (Windows, MacOS, Linux);
2. **Install** → Follow the installation instructions for your operating system;
3. **Launch Gama**

Observation: *Docker*

Note that a Docker image of GAMA exists to execute GAMA Headless inside a container.

1. Install docker on your system;
2. Pull the GAMA's docker you want to use (e.g. `docker pull gamaplatform/gama:latest`);
3. Run this GAMA's docker with your headless command (e.g. `docker run gamaplatform/gama:latest -help`).

Note that GAMA can also be launched in two different other ways:

- In a so-called headless mode (i.e. without a user interface, from the command line, in order to conduct experiments or to be run remotely);
- From the terminal, using a path to a model file and the name or number of an experiment, in order to allow running this experiment directly.
  - `Gama.app/Contents/MacOS/Gama path_to_a_model_file#experiment_name_or_number` on Mac OS X
  - `Gama path_to_a_model_file#experiment_name_or_number` on Linux
  - `Gama.exe path_to_a_model_file#experiment_name_or_number` on Windows

GAMA will ask you to choose a workspace in which to store your models and their associated data and settings. The workspace can be any folder in your filesystem on which you have read/write privileges. If you want GAMA to remember your choice next time you run it (it can be handy if you run Gama from the command line), simply check the corresponding option.

The main window is then composed of several parts, which can be **views** or **editors**, and are organized in a **perspective**. GAMA proposes 2 main perspectives: *Modeling*, dedicated to the creation of models, and *Simulation*, dedicated to their execution and exploration. Other perspectives are available if you use shared models. The default perspective in which GAMA opens is *Modeling*.

It is composed of a central area where GAML editors are displayed, which is surrounded by a Navigator view on the left-hand side of the window, an Outline view (linked with the open editor), the Problems view, which indicates errors and warnings present in the models stored in the workspace and an interactive console, which allows the modeler to try some expressions and get an immediate result.

5.1.2 Workspace, Projects and Models

The workspace is a directory in which GAMA stores all the current projects on which the user is working, links to other projects, as well as some meta-data like preference settings, the current status of the different projects, error markers, and so on. Except when running in headless mode, GAMA cannot function without a valid workspace. The workspace is organized in 4 categories, which are themselves organized into projects.

The projects present in the workspace can be either directly stored within it (as sub-directories), which is usually the case when the user creates a new project, or linked from it (so the workspace will only contain a link to the directory of the project, supposed to be somewhere in the filesystem or on the network). A same project can be linked from different workspaces.

GAMA models files are stored in these projects, which may contain also other files (called resources) necessary for the models to function.

Observation: *Project and Model Status*

All the projects and models have an icon with a red or green circle on it. This eases to locate models containing compilation errors (red circle) and projects that have been successfully validated (green circle).

n the Navigator, models are organized in four different categories: Models library, Plugin models, Test models, and User models. This organization is purely logical and does not reflect where the models are actually stored in the workspace (or elsewhere). Whatever their actual location, model files need to be stored in projects, which may contain also other files (called resources) needed for the models to function (such as data files). A project may, of course, contain several model files, especially i

- **Library models**: This category represents the models that are shipped with each version of GAMA. They do not reside in the workspace but are considered as linked from it. This link is established every time a new workspace is created. Their actual location is within a plugin (msi.gama.models) of the GAMA application;
- **Plugin models**: This category represents the models that are related to a specific plugin (additional or integrated by default). The corresponding plugin is shown between parenthesis;
- **Test models**: These models are unit tests for the GAML language: they aim at testing each element of the language to check whether they produce the expected result. The aim is to avoid regression after evolutions of the software. They can be run from the validation view;
- **User models**: This category regroups all the projects that have been created or imported in the workspace by the user. Each project can be actually a folder that resides in the folder of the workspace (so they can be easily located from within the filesystem) or a link to a folder located anywhere in the filesystem (in case of a project importation). Any modification (addition, removal of files...) made to them in the file system (or using another application) is immediately reflected in the Navigator and vice-versa.

Tip:

Model files, although it is by no means mandatory, usually reside in a sub-folder of the project called `models`. Similarly, all the test models are located in the `tests` folder.

Warning:

It may happen, on some occasions, that the library of models is not synchronized with the version of GAMA that uses your workspace. This is the case if you use different versions of GAMA to work with the same workspace. In that case, it is required that the library be manually updated. This can be done using the ”Update library” item in the contextual menu.

Each model is presented as a node in the navigation workspace, including Experiment buttons and/or a Contents node and/or a Uses node and/or a Tags node and/or an Imports node.

Although GAML files are just plain text files, and can, therefore, be produced or modified in any text processor, using the dedicated GAML editor offers many advantages, among which the live display of errors and model statuses. A model can actually be in four different states, which are visually accessible above the editing area:

- Functional (grey color);
- Experimentable (green color);

- InError (red color);
- InImportedError (red color).

In its initial state, a model is always in the Functional state (when it is empty), which means it compiles without problems, but cannot be used to launch experiments. If the model is created with a skeleton, it is Experimentable.

5.1.3 Running Experiments

Running an experiment is the only way, in GAMA, to execute simulations on a model. Experiments can be run in different ways.

- The first, and most common way, consists in launching an experiment from the Modeling perspective, using the user interface proposed by the simulation perspective to run simulations;
- The second way, detailed on this page, allows to automatically launch an experiment when opening GAMA, subsequently using the same user interface;
- The last way, known as running headless experiments, does not make use of the user interface and allows to manipulate GAMA entirely from the command line.

All three ways are strictly equivalent in terms of computations (with the exception of the last one omitting all the computations necessary to render simulations on displays or in the UI).

There are two ways to run a GAMA experiment in headless mode: using a dedicated bash wrapper (recommended) or directly from the command line.

Bash Wrapper The file can be found in the headless directory located inside the GAMA's installed folder. It is named gama-headless.sh on macOS and Linux, or gama-headless.bat on Windows.

```
1     bash gama-headless.sh [m/c/t/hpc/v] $1 $2
```

Where:

- $1 input parameter file : an xml file determining experiment parameters and attended outputs;
- $2 output directory path : a directory which contains simulation results (numerical data and simulation snapshot);
- options `[-m/c/t/hpc/v]`
  - `-m` memory : memory allocated to gama
  - `-c` : console mode, the simulation description could be written with the stdin
  - `-t` : tunneling mode, simulation description are read from the stdin, simulation results are printed out in stdout
  - `-hpc` nb_of_cores : allocate a specific number of cores for the experiment plan;
  - `-v` : verbose mode. trace are displayed in the console

Java Command

```
1 java -cp \$GAMA_CLASSPATH -Xms512m -Xmx2048m -Djava.awt.headless=true
     org.eclipse.core.launcher.Main -application msi.gama.headless.id4 \
     $1 \$2
```

Where:

- `$GAMA_CLASSPATH GAMA` classpath: contains the relative or absolute path of jars inside the GAMA plugin directory and jars created by users;
- `$1` input parameter file: an XML file determining experiment parameters and attended outputs;
- `$2` output directory path: a directory which contains simulation results (numerical data and simulation snapshot).

5.2 Model Creation

A GAMA model is composed of three types of sections:

1. `global` : this section, that is unique, defines the ”world” agent, a special agent of a GAMA model. It represents all that is global to the model: dynamics, variables, actions. In addition, it allows to initialize the simulation (init block);
2. `species` and `grid` : these sections define the species of agents composing the model. Grid is defined in the following model step ”vegetation dynamic”;
3. `experiment` : these sections define the execution context of the simulations. In particular, it defines the input (parameters) and output (displays, files...) of a model.

Species A species represents a ”prototype” of agents: it defines their common properties. A species definition requires the definition of three different elements:

- the internal state of its agents (attributes). In addition to the attributes the modeler explicitly defines, species ”inherits” other attributes called ”built-in” variables;
- their behavior;
- how they are displayed (aspects). In order to display our prey agents we define two attributes: `size` and `color` .

Global The `global` section represents a specific agent, called `world` . Defining this agent follows the same principle as any agent and is, thus, defined after a species. The world agent represents everything that is global to the model: dynamics, variables... It allows to initialize simulations (`init` block): the world is always created and initialized first when a simulation is launched (before any other agents). The geometry (`shape`) of the `world` agent is by default a square with 100m for side size, but can be redefined if necessary.

The `init` section of the global block allows initializing the model which is executing certain commands, here we will create `nb_preys_init` number of prey agents. We use the statement `create` to create agents of a specific species: `create species_name +` :

- `number` : number of agents to create (int, 1 by default);
- `from` : GIS file to use to create the agents (optional, string or file);
- `returns` : list of created agents (list)

Experiment An `experiment` block defines how a model can be simulated (executed). Several experiments can be defined for a given model. They are defined using : `experiment exp_name type: gui/batch [input] [output]` :

- `gui` : experiment with a graphical interface, which displays its input parameters and outputs;

- `batch` : Allows to set up a series of simulations (w/o graphical interface).

Experiments can define (input) parameters. A parameter definition allows to make the value of a global variable definable by the user through the graphic interface.

```
1      parameter title var: global_var category: cat;
```

Where:

- `title` : the name of the parameter as it will appear in the interface (string);
- `var` : the name of the global variable associated with this parameter (string);
- `category` : the category in which this parameter will appear in the interface (string, optional).

Output blocks are defined in an experiment and define how to visualize a simulation (with one or more display blocks that define separate windows). Each display can be refreshed independently by defining the facet `refresh` nb (int) (the display will be refreshed every nb steps of the simulation). Each display can include different layers (like in a GIS):

- Agents species: `species my_species aspect: my_aspect` ;
- Agents lists: `agents layer_name value: agents_list aspect: my_aspect` ;
- Images: `image image_file` ;
- Charts.