ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

Master Degree in Computer Science and Engineering

Multi-Agent Reinforcement Learning of Swarm Behaviours with Graph Neural Networks: prototype and first experiments

Master thesis in:
ADVANCED SOFTWARE MODELLING AND DESIGN

Supervisor
Prof. Mirko Viroli

Cosupervisors
Dott. Gianluca Aguzzi
Dott. Davide Domini

Candidate
Filippo Venturini

II Graduation Session
Academic Year 2023-2024

Abstract

In a distributed multi-agent system involving intelligent agents, a typical problem consists of coordinating agents to perform complex global goals. In this thesis, we consider a swarm of drones that need to enact certain swarming scenarios using a Multi-Agent Reinforcement Learning approach.

In general, three different approaches are proposed for designing this kind of systems: manual design, in which developers design all the necessary algorithms to achieve the desired behavior; automatic design, which involves employing machine learning techniques to learn the correct policy to apply; and a hybrid approach that combines both.

In this work we consider the automatic approach. Specifically, we use a variation of the Deep Q-Network (DQN) algorithm, which combines Q-learning with Graph Neural Networks (GNNs) to enable efficient decision-making in complex environments through Multi-Agent Reinforcement Learning. To analyze the effectiveness of this technique, we replicate some swarming scenarios and examine how accurately they are reproduced. Another important aspect of this article is testing the scalability of this technique to clarify how many agents a system can handle with this design.

"Now, bring me that horizon."

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The rapid advancement of technology is paralleled by the increasing complexity of software systems. In this historical period, there is a pursuit of autonomous systems, characterized as “intelligent”, capable of making decisions or selecting actions independently, and learning from their experiences and errors. These types of systems can be effectively applied across numerous domains, ranging from everyday tasks to space exploration.

Another aspect that escalates the complexity of a system is the multitude of devices that can now be integrated, interacting with each other, and either cooperating or competing to achieve individual or shared objectives. In particular, Distributed Systems and the Internet of Things (IoT) are currently integrating Artificial Intelligence (AI) into their inherently complex systems.

Typically, an IoT system comprises numerous cyber-physical devices, and in the context of a multi-agent system [SV00] each device is controlled by a software agent. We now consider replacing these software agents, traditionally governed by manually designed algorithms, with AI models, transforming them from conventional agents to autonomous agents [WJ95].

This thesis addresses the scenario of self-organizing systems, which are complex cyber-physical systems composed of numerous devices, each controlled by an agent we aim to render autonomous. Specifically, we intend to model a swarm of devices capable of autonomously performing complex swarming behaviors, such as reaching a target position, flocking, or avoiding obstacles.

The field of AI that perfectly aligns with our requirements is Multi-Agent Reinforcement Learning (MARL), which involves applying the traditional Reinforcement Learning approach to a Multi-Agent system. Using this technique, each agent operates within an environment, interacts with it, and receives rewards based on its actions. Each Autonomous Agent must enhance its performance and learn optimal actions to maximize the obtained rewards. If the reward structure is well-engineered, the individual policies learned by the agents will collectively result in a global swarming behavior.

More specifically, we leverage an emerging model well-suited to our problem: the Graph Neural Network (GNN). GNNs are a powerful technique for applying machine learning to graph-structured data. In our system, each agent is represented as a node in a graph, with connections between agents as edges. This structure enables the swarm of autonomous agents to fully exploit the capabilities of the GNN model.

This thesis is structured as follows. In chapter 2, we introduce all the necessary concepts for understanding this work, covering Cyber-Physical Systems, Multi-Agent Reinforcement Learning, and Graph Neural Networks. In chapter 3, we describe the system design, including the organization of software component interactions, the design of the GNN, and the learning algorithm. chapter 4 provides detailed implementation information, offering a low-level description of the software system. Finally, in chapter 5, the system is evaluated using various metrics to assess the effectiveness of this approach.

More details about the software artifact are available in a Github public repository. [1]

[1] `https://github.com/Filippo-Venturini/vmas-workspace`

Chapter 2

Background

2.1 Concepts

This section discusses all the theoretical elements necessary to understand the contribution of this thesis. First, it explores the concept of **Cyber-Physical Swarms (CPS)**, inspired by natural swarms where agents work together to achieve collective goals. In CPS, autonomous agents, such as drones, collaborate to perform tasks through coordinated behavior.

We first formalize the **swarming model**, which defines agents' local interactions and introduces the **SwarMDP** model, a compact framework for decentralized decision-making in swarm systems. This formalization helps establish the basis for agent cooperation.

Next, we discuss **Graph Neural Networks (GNNs)**, which allow agents to communicate and learn from local neighbors by propagating information through a graph structure. GNNs enable decentralized decision-making, essential for swarm behavior.

Finally, we introduce **Multi-Agent Reinforcement Learning (MARL)**, where agents learn local policies through trial and error. Using the Centralized Training Decentralized Execution (CTDE) approach, agents learn globally during training and act independently in execution.

Together, CPS, swarming models, GNNs, and MARL provide a comprehensive framework for developing autonomous swarm systems. The following subsections detail each concept and their role in achieving coordinated swarm behavior.

2.1.1 Cyber-Physical Swarms

In nature is common to find swarms of animals or insects that perform a certain coordinated group behaviour to reach an objective.

For example, we often see birds that are flocking, for confusing predators, sharing information about food sources or also reducing air resistance. Similar to birds also fish can exhibit coordinated movements for puzzle predators and improve the swimming efficiency.

Given that, also in the computer science field we can build a swarm of devices that has to achieve a global objective such as animals or insects, these swarms are called Cyber-Physical Swarms (CPSs or *swarm-like systems*)[AS24] [BF16]. Swarm-like systems can model a lot of real-life problems and scenarios such as: environmental monitoring [MSF16], UAV coordination [SAB+19], social systems [ZYCK20] or more in general sensors networks [PPCE22]. In the case of this thesis we consider a swarm of drones in which each drone is equal, and by moving in the environment has to learn a local behaviour that merged with the others, result in a collective swarming movement. Some example of typical swarming behaviours of drones can be: go to a position, flocking, pattern formation, obstacle avoidance and so on [ACV24].

A crucial aspect of developing swarm-like systems involves devising decentralized controllers for the agents, enabling them to execute sophisticated group tasks. There are three primary strategies for tackling this issue: manual design, automatic design and an hybrid approach. In the manual approach, developers directly construct the controllers, leveraging their expertise and utilizing programming languages or frameworks that facilitate distributed computation and communication, such as macro-programming methods [Cas23]. Conversely, in the automated approach, machine learning techniques, like multi-agent reinforcement learning and evolutionary algorithms, are employed to generate programs or policies for the agents based on high-level task descriptions or goals. Each method has its pros and cons. Manual creation allows for the declarative specification of desired system properties but can be laborious and prone to mistakes. Automated generation can mitigate these drawbacks by learning from data and experience, yet it also faces significant challenges, including the difficulty of learning suitable representations for agent communication. There is also a third possibility that is called Field-Informed Reinforcement Learning (FIRL) [AVE23], that mix the manual design using macro-programming technique such as aggregate computing [BPV15], with automatic design utilizing a reinforcement learning approach.

Specifically, in this thesis we set the focus on the automatic approach, so our swarm-like system of drones has one autonomous agent for each drone that has to learn a local policy via Reinforcement Learning (RL), for lead to a collective swarm behaviour, more details will be explored in the next sections.

2.1.2 Swarming Model formalization

In this section is formalized the swarming model with the aim of establishing a more robust foundation for defining a formal concept of swarm.

We define a swarm system as a group of agents characterized by two main attributes:

- **Locality**: Agents are limited to observing only portions of the system within a specified range, as dictated by their observational capabilities. Consequently, their decision-making is based solely on their immediate surroundings, rather than the state of the entire swarm.

- **Homogeneity**: All agents share an identical architecture, meaning they possess the same dynamics, degrees of freedom, and observational capacities, making them interchangeable.

Any system exhibiting these features can, in theory, be represented as a decentralized partially observable Markov decision process (Dec-POMDP) [BPZ19]. However, the homogeneity aspect, which is crucial for scalable inference, is not explicitly incorporated in this model. Given that swarms typically consist of a large number of agents, it is advantageous to adopt a more compact system representation that leverages these inherent symmetries.

So we refer to SwarMDP [SKZK17], that is a subclass of Dec-POMDP. Accordingly, we can define a prototype of our autonomous agent within the framework of SwarMDP as:
$\mathcal{A} = (\mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \pi)$, where:

- $\mathcal{S}$ is the set of all local states.
- $\mathcal{O}$ is the set of all agent observations.
- $\mathcal{A}$ is the set of agent actions.
- $\mathcal{R} : \mathcal{O} \to \mathcal{R}$ is the reward function of the agent.
- π is the local policy of the agent, which, when combined with the policies of other agents, results in the learned swarm behavior.

Once the swarming agent is defined, the concept of a SwarMDP can be described as a tuple $(\mathcal{N}, \mathcal{A}, \mathcal{T}, \mathcal{E})$, where:

- $\mathcal{N}$ is the number of swarming agents in the system.
- $\mathcal{A}$ is the swarming agent defined above.
- $\mathcal{T} : \mathcal{S}^{\mathcal{N}} \times \mathcal{A}^{\mathcal{N}} \times \mathcal{S}^{\mathcal{N}} \to \mathbb{R}$ is the global transition function of the system. Specifically, $\mathcal{T}$ represents the transition from a global state (composed of all agents' states) to a new global state as a result of executing all local actions of the agents.
- $\mathcal{E} : \mathcal{S}^{\mathcal{N}} \to \mathcal{O}^{\mathcal{N}}$ is the observation model of the system.

The observation model ξ indicates which segments of a system state $s \in S^N$ are visible to each agent. Specifically, $\xi(s) = (\xi^{(1)}(s), \ldots, \xi^{(N)}(s)) \in O^N$ represents the ordered set of local observations provided to agents at state s. For instance, in a fish school, $\xi^{(n)}$ could represent the local alignment of a fish relative to its nearby neighbors. It's important to note that agents do not have direct access to their local states $s^{(n)} \in S$. Instead, they only receive their local observations $\theta^{(n)} = \xi^{(n)}(s) \in O$.

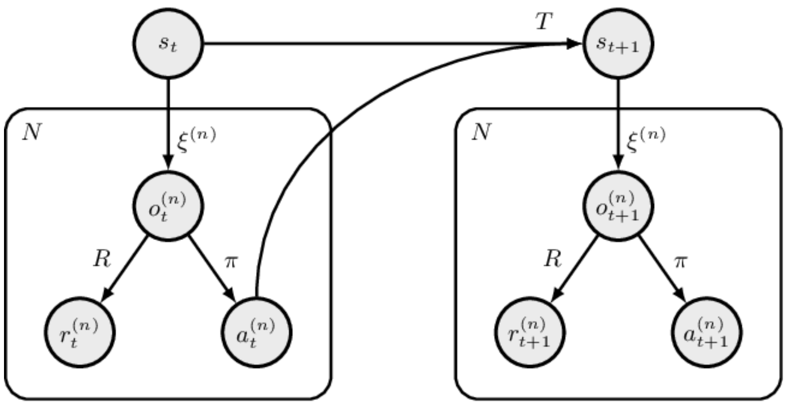


Figure 2.1: The SwarMDP model formalization

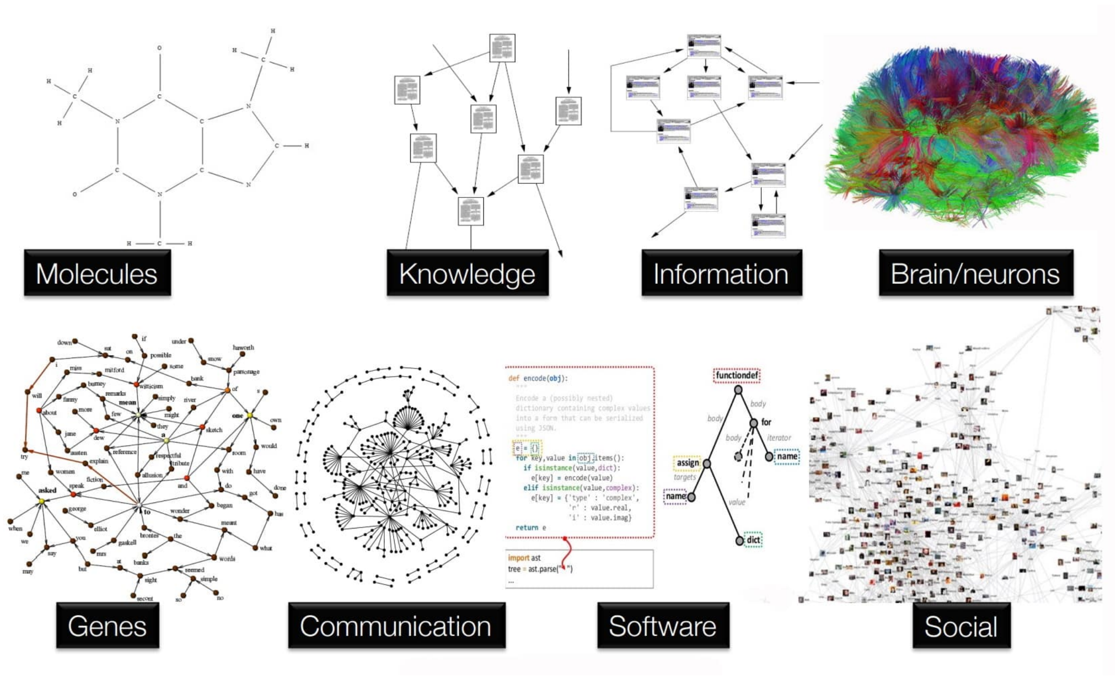
Moreover, is possible to use a global observation model, so we can encode all properties in a single object, yielding a more compact system description. But in this work the observation model is defined locally at the agent level, since the observations are agent-related quantities. However, this would still require a global notion of connectivity between the agents, which is provided by our Graph Neural Network Model (GNN) (see Section 2.3).

2.1.3 Graph Neural Networks

Graph Neural Networks (GNNs) represent a transformative class of neural networks designed to handle data structured as graphs, which are ubiquitous in various domains including physical models, chemical compounds, images, and text. Several works explore and highlight the key areas where GNNs make a significant impact [GMP21] [ZCH+20].

Unlike traditional neural networks that operate on grid-like structures, GNNs can effectively capture complex relationships and dependencies inherent in graph data. At their core, GNNs propagate information across nodes and edges of the graph to learn representations that encode both node attributes and graph topology. The structure of a GNN typically involves several layers where each layer iteratively aggregates information from neighboring nodes, enabling the network to refine its understanding of each node's context within the graph.

One prominent type of GNN is the **Graph Convolutional Network (GCN)**



```
def encode(obj):
    """
    Encode a (possibly nested)
    dictionary containing complex values
    into a form that can be serialized
    using JSON.
    """
    e = {}
    for key,value in obj.items():
      if isinstance(value,dict):
        e[key] = encode(value)
      elif isinstance(value,complex):
        e[key] = {'type' : 'complex',
                  'r' : value.real,
                  'i' : value.imag}
    return e
```

```
import ast
tree = ast.parse(" ")
...
```

Figure 2.2: GNN applications

[KW17]. GCNs generalize the concept of convolution from regular grids to irregular graph structures by applying a linear transformation of node features combined with neighborhood aggregation. This approach allows GCNs to effectively capture localized graph structures and has been widely adopted in tasks such as node classification and link prediction.

Another notable variant is the **Graph Attention Network (GAT)** [VCC+18], which enhances GCNs by introducing attention mechanisms. GATs assign different importance weights to neighboring nodes dynamically during each layer's aggregation step, allowing the network to focus more on relevant nodes and thus improving performance in tasks requiring nuanced relationships within the graph.

But, apart from variations, now we will understand how a general GNN works with a more formal approach, by introducing some notations. [SGT+09]. Let $G = (V, E)$ be the graph of the GNN, where V is the set of nodes and E is the set of edges connecting the nodes. Each node $v \in V$ has a corresponding feature set f_v, which is a collection of characteristics specific to that node (e.g., if a person is modeled as a node, the node features might include name, age, height, etc.).

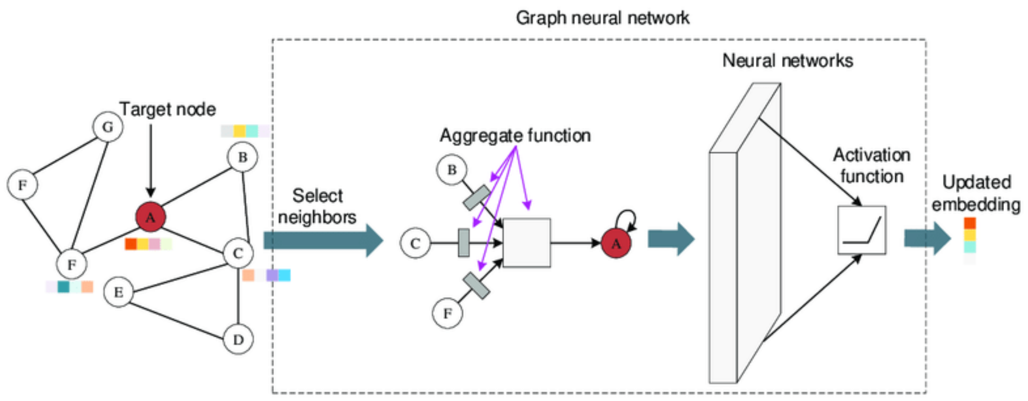


Figure 2.3: Graph neural network behaviour

Each edge $e \in E$ connects two nodes and represents the relationship between them. The concept of a node's neighborhood is straightforward and is considered during the learning phase. The aim of a GNN is to learn an embedding h_v for each $v \in V$, which encodes the node's features, capturing both its intrinsic properties and its contextual information within the graph.

Specifically, h_v is computed iteratively through various phases of information aggregation from the neighbors $N_G(v)$ and combined with the current embedding of the node h_v. This process can be seen as a communication process between nodes, referred to as *message passing* [SZWL23].

A GNN consists of k message passing layers, with each layer responsible for computing the embedding h_v^k for each $v \in V$.

Formally:

$$m_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in N_G(v)\right\}\right) \tag{2.1}$$

$$h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, m_v^{(k)}\right) \tag{2.2}$$

where $N_G(v)$ is the set of all direct neighbors of node v, h_v^k is the embedding of node v at layer k, and $h_v^{(k-1)}$ is the embedding of the node at the previous layer. The $AGGREGATE(\cdot)$ function accumulates information from the neighbors and must be permutation-invariant. The aggregation can be simple, such as sum,

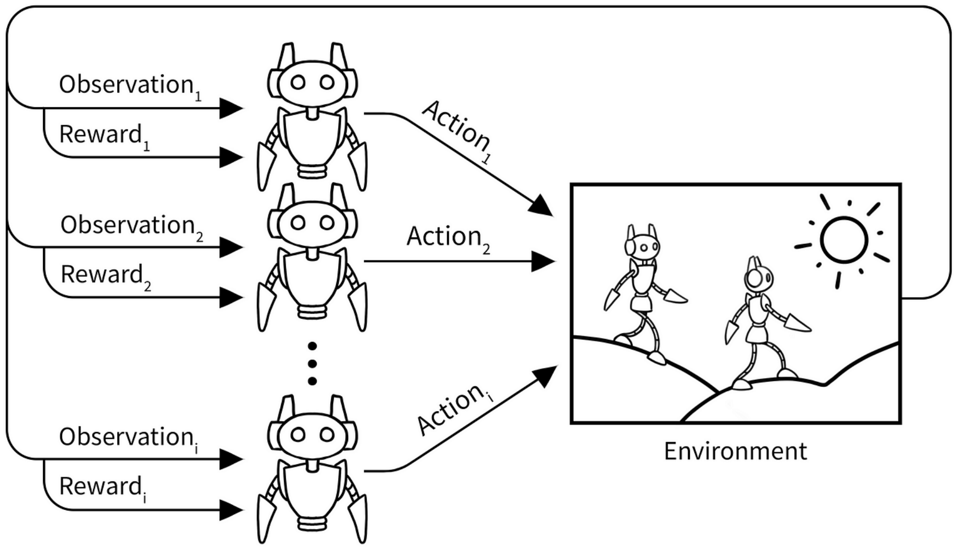


Figure 2.4: Multi-Agent Reinforcement Learning

max pooling, or sum of products, but can also be more complex and articulated [PTF$^+$21].

Through the aggregation process, the *message* $m_v^{(k)}$ is obtained, which is then merged with the individual node information using the $COMBINE(\cdot)$ operator.

In this work, we use a GNN to enable each swarming agent to learn local behaviors in a Multi-Agent Reinforcement Learning (MARL) scenario (more details in the next section).

2.1.4 Multi-Agent Reinforcement Learning

Another important concept to introduce is **Multi-Agent Reinforcement Learning (MARL)** which is composed by two distinct parts: Multi-Agent Systems and Reinforcement Learning.

Multi-agent systems (MAS) represent a paradigm in artificial intelligence where multiple autonomous agents interact within an environment to achieve individual or collective goals [SV00]. These agents can be software entities, robots, or even biological organisms, and they operate based on a set of rules or learning algorithms. MAS are particularly powerful in solving complex problems that are difficult or impossible for a single agent to handle, such as resource allocation, swarm robotics, and distributed sensor networks. The interactions among agents in an MAS can be cooperative, competitive, or neutral, and these interactions are governed by protocols that define communication, coordination, and negotiation mechanisms.

The second component is **Reinforcement Learning (RL)** which is a branch of machine learning where agents learn to make decisions by performing actions in an environment to maximize cumulative rewards. Unlike supervised learning, which relies on labeled data, RL involves an agent exploring its environment and learning from the consequences of its actions through trial and error. Techniques such as Q-learning [WD92], policy gradients [SMSM99], and deep reinforcement learning [HR16] have been developed to address various challenges in RL, such as the balance between exploration and exploitation and the scalability to high-dimensional state and action spaces. RL has garnered significant interest due to recent achievements in various domains, from mastering complex games such as Go, Chess, Starcraft, and Dota2 [SHM+16][SHS+17][VBC+19][BBC+19] to applications in robotics [ABC+20] [OAA+19].

By combining these two concepts, we arrive at MARL, an approach where all agents in a MAS are trained to achieve a collective global goal using RL techniques [BBS08]. In this work, we consider *homogeneous MARL*, where each agent in the system is *indistinguishable* and *interchangeable*. Thus, all agents are at the same level and can be viewed as identical computational units.

Additionally, the approach used in this work is known as **Centralized Training Decentralized Execution (CTDE)**. In this approach, the training phase considers the global state of the system (including all agents' observations, actions, rewards, etc.) to learn a policy. During the execution phase, each agent operates using only its local perception of the environment [Che20] [ABZ23].

2.2 Technologies

The aim of this section is to provide a brief description of all the main technologies used in this work, for designing and implementing the contribution described in the next chapter. Along with the descriptions, examples of usage and known applications that leverage these technologies will also be highlighted.

2.2.1 PyTorch

PyTorch [PGM+19] is an open-source deep learning framework developed by Meta's AI Research lab. It provides a flexible and efficient platform for developing machine learning models, offering dynamic computation graphs that facilitate intuitive and immediate construction and manipulation of neural networks.

This is particularly beneficial for research and prototyping, as it allows for easy debugging and real-time updates. PyTorch supports automatic differentiation, which is essential for gradient-based learning algorithms. It is built on the Torch library and integrates seamlessly with Python, making it accessible for both beginners and experienced practitioners.

The framework includes a rich ecosystem of tools and libraries for reinforcement learning and it has distributed training capabilities for enabling scalable and efficient training on multiple GPUs and across nodes, making it suitable for large-scale machine learning projects. For instance, PyTorch has been employed in various large-scale applications such as Tesla's Autopilot and Meta's speech recognition systems. Furthermore, PyTorch is used in Uber's Pyro for probabilistic programming, showing its versatility across industries [BCJ+19].

Additionally, PyTorch is compatible with the ONNX (Open Neural Network Exchange) format, which allows models to be transferred between different frameworks, enhancing its versatility in various deployment scenarios. The active and growing community around PyTorch contributes to a wealth of resources, tutorials, and third-party libraries, further supporting its adoption in both academia and industry. PyTorch is widely used in prominent academic research, including areas like natural language processing (e.g., BERT)[DCLT19], computer vision (e.g., ResNet) [HZRS15], and generative models (e.g., GANs) [GPAM+14].

In this project, this framework is essential for handling complex and structured data such as observations of the agents, rewards, states, policies, and so on. It is also a key component on which the library for designing Graph Neural Networks, discussed in the next section, is built.

2.2.2 PyTorch Geometric

PyTorch Geometric (PyG) [PyG23] is a pivotal Python library in the realm of geometric deep learning, designed to tackle the complexities of analyzing and processing data represented as graphs and meshes.

At its core, PyG empowers researchers and practitioners to extend traditional deep learning techniques beyond Euclidean domains, effectively bridging the gap to non-linear, irregular data structures. By leveraging the robust capabilities of PyTorch, it provides a comprehensive suite of tools and implementations for constructing and training graph neural networks (GNNs). These include cutting-edge models such as Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and Graph Isomorphism Networks (GINs), each tailored to handle the intricate relationships and inherent complexities within graph data.

PyG is widely used in both academia and industry for graph-based tasks such as drug discovery, fraud detection in financial transactions, and recommendation systems. For example, it has been used in Pinterest's PinSage for content recommendation by modeling the relationships between users and pins through a graph [YHC+18]. The versatility of PyG extends to social network analysis, molecular graph learning, and 3D vision tasks, demonstrating its relevance across numerous fields where data is naturally represented as graphs.

In this work, PyG is used for designing and training the GNN used as a model to learn complex collective swarming behaviors in our multi-agent reinforcement learning (MARL) scenario.

2.2.3 Vectorized Multi-Agent Simulator

A pivoting component of a MARL scenario is the `Simulator` used to train the agents. Multiple alternatives were explored, but the one that suits our requirements best is the Vectorized Multi-Agent `Simulator` (VMAS).

VMAS stands as a vectorized differentiable `Simulator` crafted for efficient benchmarking in MARL. It comprises a fully-differentiable 2D physics engine implemented in PyTorch, alongside a collection of intricate multi-robot scenarios. Creating scenarios is designed to be straightforward and modular, encouraging contributions. VMAS simulates agents and landmarks of varying shapes, supporting rotations, elastic collisions, joints, and customizable gravity settings.

Agents utilize holonomic motion models to streamline simulation, while custom sensors like LIDARs are available, facilitating inter-agent communication. Leveraging PyTorch's vectorization capability enables VMAS to execute simulations in batch mode, effortlessly scaling to tens of thousands of parallel environments on accelerated hardware. This makes it particularly efficient for scaling up experiments in multi-agent systems, where rapid simulation of interactions between agents is essential for reinforcement learning tasks.

VMAS boasts compatibility with OpenAI Gym [BCP$^+$16], RLlib [LLM$^+$18], torchrl [BBD$^+$23], and the MARL training library BenchMARL [BPM23], ensuring seamless integration with a broad spectrum of RL algorithms. Its design draws inspiration from OpenAI's Multi-Agent Particle Environment (MPE) framework, complemented by the porting and vectorization of all MPE scenarios within VMAS.

Another alternative considered for the simulation environment is Magent2 [ZYC$^+$18]. This `Simulator` is integrated with multiple PettingZoo [TBG$^+$21] scenarios, and is also compatible with OpenAI Gymnasium [TTK$^+$23]. However, Magent2 is more suitable for creating battle games or competitive scenarios, which is not aligned with our specific case study focusing on cooperative swarming behavior.

Chapter 3

Contribution

This chapter outlines the design of the primary contribution of this work. First, a formal analysis of the problem formulation and requirements is conducted. Next, the overall design of the system architecture is examined in detail, focusing on the integration of all components, such as the `Simulator`, the `Trainer`, and others, at a high level of abstraction. Following this, the core software artifact is analyzed, particularly the design of the Graph Neural Network and the algorithm used during the learning phase. The final section provides a theoretical explanation of the tasks addressed in this work, including a formal analysis of the reward structure designed to achieve the goals.

3.1 Analysis

After introducing all the background concepts necessary to understand this thesis, it is essential to perform an analysis to outline the key aspects of this contribution, as well as to define the objectives and goals for the software developed.

The overall goal is to create a software artifact that results in a clear and well-engineered system capable of training Graph Neural Network models in a Multi-Agent Reinforcement Learning scenario, to simulate an autonomous swarm of drones performing various tasks.

We aim to design an approach that effectively exploits the potential of GNNs to enable agents to learn appropriate behaviors in various situations. The goal is to develop both the model structure and the learning algorithm to teach agents individual policies that collectively result in the desired swarming behavior. The proposed approach should scale efficiently with the number of agents and the complexity of tasks the swarm can successfully accomplish.

Below there is a structured list of all the objectives of this thesis:

1. Smoothly integrate different technologies to compose the software architecture.

2. Design a Reinforcement Learning model and a learning algorithm suitable for our problem, utilizing Graph Neural Networks.

3. Engineer various scenarios, including reward structures, observation and action spaces, in which the swarm must perform different tasks such as moving to a position, avoiding obstacles, and flocking.

4. Evaluate the performance and behavior of the chosen techniques and approaches during the execution of these different tasks.

This section details the overall design of the system, from the software architecture to the core components, including the design of the Graph Neural Network and the learning algorithm.

3.2 Architecture

The system architecture integrates various essential components efficiently. Each component employs distinct technologies, as delineated in Section 2.2, and they must synergistically collaborate to train the multi-agent system, enabling it to perform diverse tasks.

3.2.1 Simulator

A fundamental component of the system architecture is the `Simulator`, which is crucial for allowing our swarm of agents to conduct experiments to learn optimal behaviors.

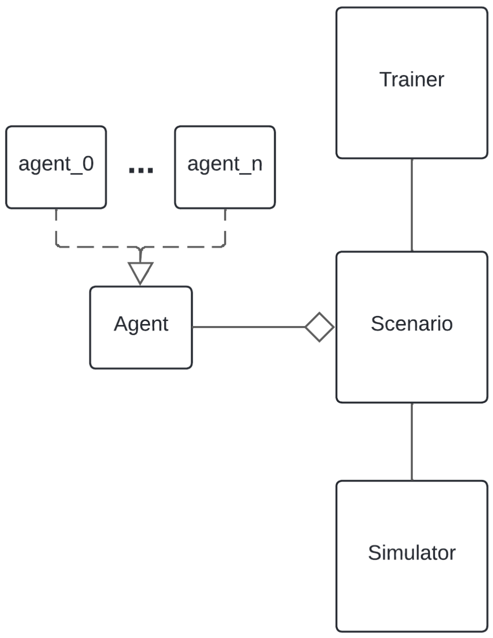


Figure 3.1: System Architecture

The `Simulator`'s objective is to provide an environment in which the agents are situated, enabling them to perform actions and interact with the environment. Specifically, this component must incorporate all necessary logic to run simulations where the system state evolves and changes, potentially incorporating basic physics concepts such as movement, collisions, and time. Additionally, this component includes a GUI to render the simulation results, which is essential for both the training and evaluation phases.

3.2.2 Scenario

A critical component of any Reinforcement Learning system is the environment. In this architecture, the `Scenario` component defines the conceptual environment in which the agents operate. The design of the `Scenario` component is pivotal to ensuring that the agents learn effective behaviors that align with our objectives. Specifically, the `Scenario` aims to specify the environment's structure: initial agent positions, obstacle placements, goal locations, etc. Moreover, the `Scenario` must also define how agents perceive their surroundings (observations) and how their actions impact the system (rewards). The state of the scenario defined by this component is subsequently rendered by the `Simulator`.

3.2.3 Agent

The `Agent` component represents the individual autonomous agents within our system. Each agent must interact with the environment by executing actions and receiving rewards. All agents are contained within the `Scenario` and can interact with the environment according to their configuration, which will be detailed in Chapter 4.

3.2.4 Trainer

The `Trainer` component is the core of the system. It should encapsulate the GNN model and provide an implementation of a learning algorithm. This component represents the primary learning aspect of the system, enabling the construction of graph data from agent observations, training the agents, and producing a trained model, which can be used to run simulations via the `Simulator` component.

3.2.5 Component Integration

When these subsystems are correctly integrated, the system functions as follows.

As depicted in fig. 3.2, the workflow of the designed architecture begins with an agent computing an action rendered by the `Simulator` component. After rendering the action, the `Simulator` provides the agent with observations of its surroundings, which are then passed to the `Scenario` component. In the `Scenario` component, all agent observations are aggregated and sent to the `Trainer` component. Here, a graph data structure is constructed from the agent observations, and a training step of the model is performed, resulting in an action for each autonomous agent. These actions are evaluated in the `Scenario`, representing the environment in which the agents operate, and the corresponding rewards are computed and sent to the agents. Finally, the actions to be rendered are sent to the `Simulator`, which updates the environment and agent states graphically.

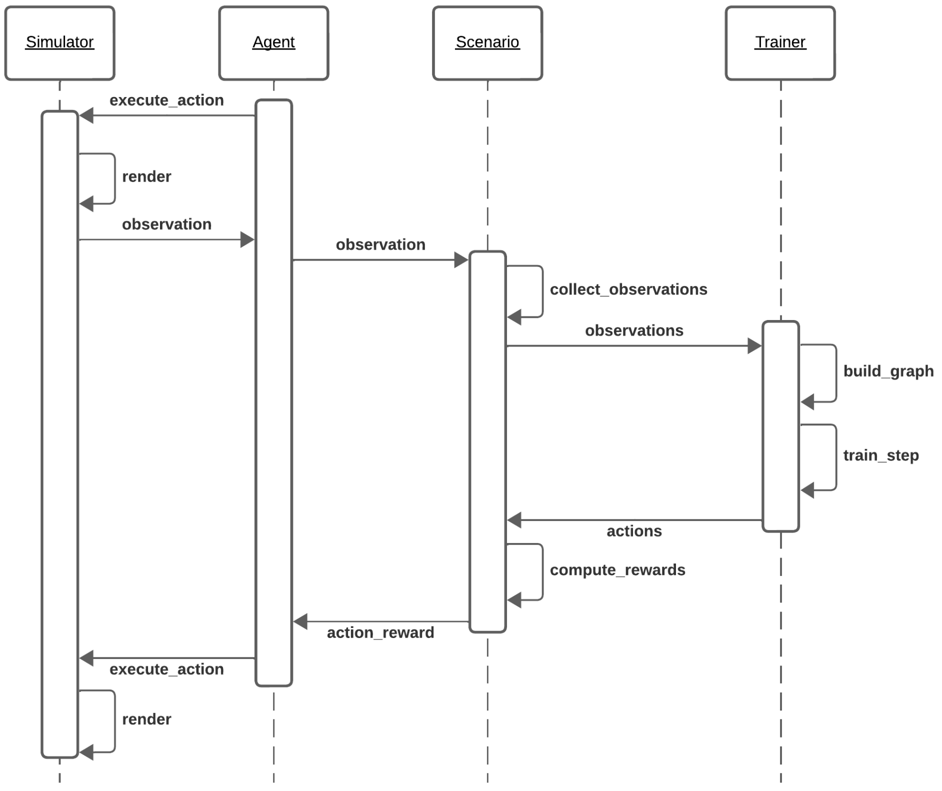


Figure 3.2: Architecture Component Integration

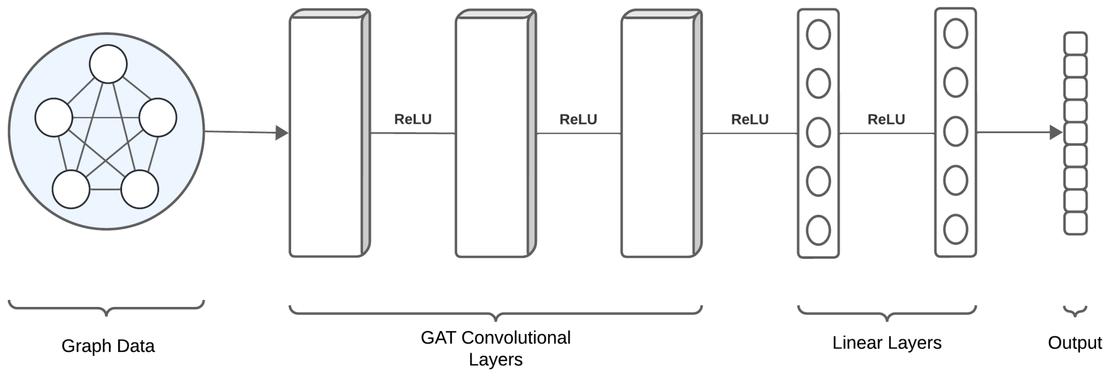


Figure 3.3: Graph Neural Network model

3.3 Model: Graph Neural Network

The designed Graph Neural Network model represented in fig. 3.3 is structured to process graph-structured data through a series of graph convolutional layers, followed by fully connected layers to transform and extract meaningful features from the input data.

At the core of the model are three graph attention convolutional (GAT) layers, which enhance the representation of each node by aggregating information from its neighbors. Each of these layers leverages the attention mechanism to weigh the importance of neighboring nodes, allowing the model to focus on the most relevant connections within the graph.

After each convolutional layer, a rectified linear unit (ReLU) activation function is applied, introducing non-linearity into the model and enabling it to capture more complex patterns. Following the convolutional layers, the model transitions to two fully connected (linear) layers. The first linear layer further processes the features extracted by the convolutional layers, with another ReLU activation ensuring the continuation of non-linear transformations. The final linear layer projects the processed features into the desired output dimension, producing the final output of the model.

This design, combining graph attention mechanisms and multi-layer perceptrons, aims to effectively capture and utilize the intricate relationships and structures within the graph data.

3.4 Building Graph Data

Given the model of the GNN described above, we need to engineer a way for integrate the raw observations of the agents in the environment with the model, so we need to build a graph data that the model can exploit during the learning phase.

To build a graph from the observations of multiple agents in the environment, we follow a structured process. Each agent's observations are collected and transformed into a feature vector. Let $\mathbf{O} = \{\mathbf{o}_i \mid i = 1, \ldots, N\}$ represent the set of observations for N agents, where $\mathbf{o}_i$ denotes the observation vector for agent i.

First, we construct a node feature matrix $\mathbf{X}$, where each row corresponds to an agent's observation vector augmented with the agent's identifier. Formally, the node feature matrix is given by:

$$\mathbf{X} = \begin{bmatrix} \mathbf{o}_1 & id_1 \\ \mathbf{o}_2 & id_2 \\ \vdots & \vdots \\ \mathbf{o}_N & id_N \end{bmatrix}$$

where id_i is the unique identifier for agent i.

Next, we establish the connectivity between nodes by defining the edges of the graph. To ensure the graph is fully connected, we create bidirectional edges between every pair of distinct agents. This results in an edge index matrix $\mathbf{E}$ that captures these connections. The edge index matrix can be represented as:

$$\mathbf{E} = \begin{bmatrix} i_1 & j_1 \\ i_2 & j_2 \\ \vdots & \vdots \\ i_m & j_m \end{bmatrix}$$

where each pair (i_k, j_k) indicates a directed edge from node i_k to node j_k. Additionally, we include self-loops to ensure each node has a direct connection to itself, represented as (i, i) for each agent i.

The resulting graph $\mathcal{G}$ is then composed of the node feature matrix $\mathbf{X}$ and the edge index matrix $\mathbf{E}$. This graph structure, $\mathcal{G} = (\mathbf{X}, \mathbf{E})$, provides a rich representation of the agents' observations and their interactions, making it suitable for processing by the GNN. The GNN can exploit this structure to learn and infer complex relationships between the agents, leveraging both the node features and the connectivity information encapsulated in the graph.

3.5 Learning Algorithm: DQN

In this section the design of the learning algorithm will be discussed. A custom implementation of the The Deep Q-Network (DQN) algorithm is performed (see algorithm 1), with the using of the replay buffer technique for make the agents capable of exploiting past experiences during the learning phase [HR16].

The DQN algorithm involves training a neural network to approximate the Q-function, which estimates the value of taking a particular action in a given state, considering future rewards. The model interacts with an environment to learn an optimal policy.

The algorithm starts by initializing the environment and two neural networks: the model Q and the target model Q', both initialized with the same random weights. The model Q is used to select actions, while Q' provides stable target values. An optimizer is set up to update the model's weights, and a replay buffer B with a specified capacity is created to store experiences.

For each episode, the environment is reset to obtain initial observations. Graph data is created from the current observations with the methodology described in section section 3.4 and fed into the model Q to get the logits, which represent the predicted Q-values for each action.

An epsilon-greedy policy is used to balance exploration and exploitation. A random action is selected with a probability of ϵ; otherwise, the action with the highest Q-value is chosen. The selected actions are executed in the environment, resulting in new observations and rewards. The transition (current state, action, reward, next state) is stored in the replay buffer B.

Then is performed a sample of a batch of transitions from the replay buffer for executing the training step. After, the Q-values and the target Q-values are computed for eventually computes the loss (calculated using the target model Q').

Algorithm 1 Deep Q-Network (DQN) with Graph Neural Network and Graph Replay Buffer

```
 1: **Initialize** environment $env$;
 2: **Initialize** exploration strategy $\epsilon$;
 3: **Initialize** model $Q$ with random weights;
 4: **Initialize** target model $Q'$ with weights $\theta' \leftarrow \theta$;
 5: **Initialize** replay buffer $B$ with random entries;
 6: **for** episode = 1 to $max\_episodes$ **do**
 7:     Get current observations $O$ from $env$
 8:     **for** step = 1 to $max\_steps$ **do**
 9:         **Build** graph data $G$ from $O$
10:         $logits \leftarrow Q(G)$
11:         **if** random $< \epsilon$ **then**
12:             Select random actions $A$
13:         **else**
14:             $A \leftarrow \arg\max(logits)$
15:         **end if**
16:         $O', R \leftarrow env.step(A)$
17:         **Build** graph data $G'$ from $O'$
18:         **Store** experience $(G, A, R, G')$ in $B$
19:         **Sample** a batch of experiences $(G_b, A_b, R_b, G'_b)$ from $B$
20:         **Compute** the current Q-values: $Q\_values \leftarrow Q(G_b)[A_b]$
21:         **Compute** the target Q-values: $next\_Q\_values \leftarrow \max Q'(G'_b)$
22:         $target\_Q\_values \leftarrow R_b + \gamma \cdot next\_Q\_values$
23:         **Compute** loss: $loss \leftarrow \text{SmoothL1Loss}(Q\_values, target\_Q\_values)$
24:         **Perform** gradient descent step
25:         **Perform** the backpropagation of $loss$
26:         Every C steps, update the target network weights: $\theta' \leftarrow \theta$
27:     **end for**
28: **end for**
```

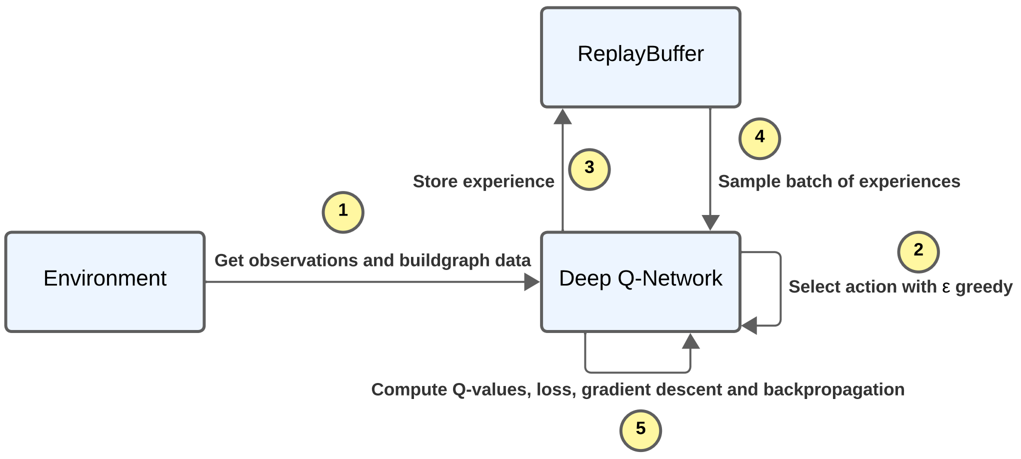

ReplayBuffer
4
3
Store experience
Sample batch of experiences
1
2
Environment
Get observations and buildgraph data
Deep Q-Network
Select action with ε greedy
Compute Q-values, loss, gradient descent and backpropagation
5

Figure 3.4: DQN step visualisation

Then is performed a gradient descent step and the model weights are updated using backpropagation.

After each step, the epsilon value is decayed to reduce the exploration rate over time, and episode metrics are logged.

Periodically, the weights of the target model Q' are updated to match the model Q to stabilize training.

Once all episodes are completed, the trained model's weights are saved to be loaded during the simulation phase.

In fig. 3.4 is available a visualization of one step of the DQN algorithm for better understand how its behaviour.

Chapter 4

Implementation

In this chapter will be detailed the implementation built upon the design described in the previous chapters. Starting from the practical definition of all the necessaries components for a Multi-Agent Reinforcement Learning system such as: Agent, Observation Space, Action Space, Rewards, Policy etc. Followed by the detailed description of the implementation of all the components of the architecture such as the `Simulator`, the `Trainer`, the GNN model and so on.

4.1 MARL components

The aim of this section is to analyze with a low level of abstraction, how the theoretical components of our Multi-Agent Reinforcement Learning system are implemented.

4.1.1 Agent

As already explained in the previous sections an agent is an entity that can observe the environment and perform some actions to interact with it. In our case the practical implementation of it it's provided by the `Agent` class of the `VMAS` library (see section section 2.2).

In general, our agent have the following important attributes:

- `name`: the name of the agent, for distinguish it from the others in the multi-agent system.

- `position`: represent the position of the agent in a 2-dimension environment, with a `Tensor` of two coordinates (x, y).
- `velocity`: indicates the velocity vector of the agent in the environment also with a 2D `Tensor` (x, y), so it provides the direction in which the agent is moving.
- `goal`: represent the goal of the agent, it consist in a simple point in the environment.

4.1.2 Observation space

As **observation space** we mean the portion of the environment that the agent can perceive, and it's definition is fundamental in every Reinforcement Learning system.

In our case the agent observations are defined basing on the task that the swarm has to perform, but in general, we can imagine a single agent observation as a `Tensor` with for example its position and its velocity.

Given we are implementing a Multi-Agent System our observation space is composed by all the single observations of all the agents, structured in a `Dict` that match each agent's name with its observation.

```
{'agent0': tensor([[-1., -1.,  0.,  0.]]),
 'agent1': tensor([[ 0., -1.,  0.,  0.]]),
 'agent2': tensor([[ 0.,  1.,  0.,  0.]]),
 'agent3': tensor([[ 0.,  0.,  0.,  0.]]),
 'agent4': tensor([[ 1.,  1.,  0.,  0.]]),
 'agent5': tensor([[ 1., -1.,  0.,  0.]]),
 'agent6': tensor([[-1.,  1.,  0.,  0.]]),
 'agent7': tensor([[ 1.,  0.,  0.,  0.]]),
 'agent8': tensor([[-1.,  0.,  0.,  0.]])}
```

4.1.3 Action space

Another key space to define in a MARL System is the **action space**. It's a similar concept to the previous one, but in this case it define the space of actions possible for our agents. In our case the agents are located in a 2D environment and for simplicity we choose to make the agent execute only discrete actions.

In this case we refer to the spaces implemented in Openai Gymnasium [TTK+23], and we can model the action space as a `Discrete(9)`. This happens because our agent in each iteration can execute just one discrete movement out of nine possible (up-left, up, up-right, left, idle, right, down-left, down, down-right).

Same as before, the complete action space result in a `Dict` that match each agent name with its action space.

```
{'agent0': Discrete(9),
 'agent1': Discrete(9),
 'agent2': Discrete(9),
 'agent3': Discrete(9),
 'agent4': Discrete(9),
 'agent5': Discrete(9),
 'agent6': Discrete(9),
 'agent7': Discrete(9),
 'agent8': Discrete(9)}
```

4.1.4 Reward

Now let's define how it's implemented the concept of reward. As defined in the previous chapters, the reward is essential for the agent to learn if the action that has performed is good or not in terms of completing a task. In this implementation, the reward is a float number that is computed basing on different tasks following the reward structures defined in section section 5.1.

Same as before, given we are in a Multi-Agent system, the rewards are structured in a `Dict` that map each agent with the reward obtained in that iteration.

```
{'agent0': tensor(-2.5784),
 'agent1': tensor(-1.9861),
 'agent2': tensor(-1.9861),
 'agent3': tensor(-1.1642),
 'agent4': tensor(-2.5784),
 'agent5': tensor(-2.5784),
 'agent6': tensor(-2.5784),
 'agent7': tensor(-1.9861),
 'agent8': tensor(-1.9861)}
```

4.1.5 Policy

Given the previous definitions, now the policy that the agent learn during the training consists in a data structure that for each possible state of the agent return the best action that the agent has to perform to complete the task.

4.2 System components implementation

Referring to Figure 4.1, the following sections will detail the implementations of the components of our system.

In general, these sections will provide a lower-level abstraction of all the implementations corresponding to the component designs described in the previous chapter.

Examining the overall system implementation depicted in Figure 4.1, we can identify all the previously described components, such as:

- `GNN`: for represent the model of the Graph Neural Network.
- `DQNTrainer`: that represent the trainer of the model, with the methods for build the graph data and implement the DQN algorithm.
- `GraphReplayBuffer`: for encapsulate the concept of buffer used inside the DQN algorithm.
- `BaseScenario`: for represent the environment in which our swarm of agent is located.
- `Simulator`: for simulating the agent behaviour during the training and the evaluation phase.

In the following sections each of these components' implementation is going to be described and detailed.

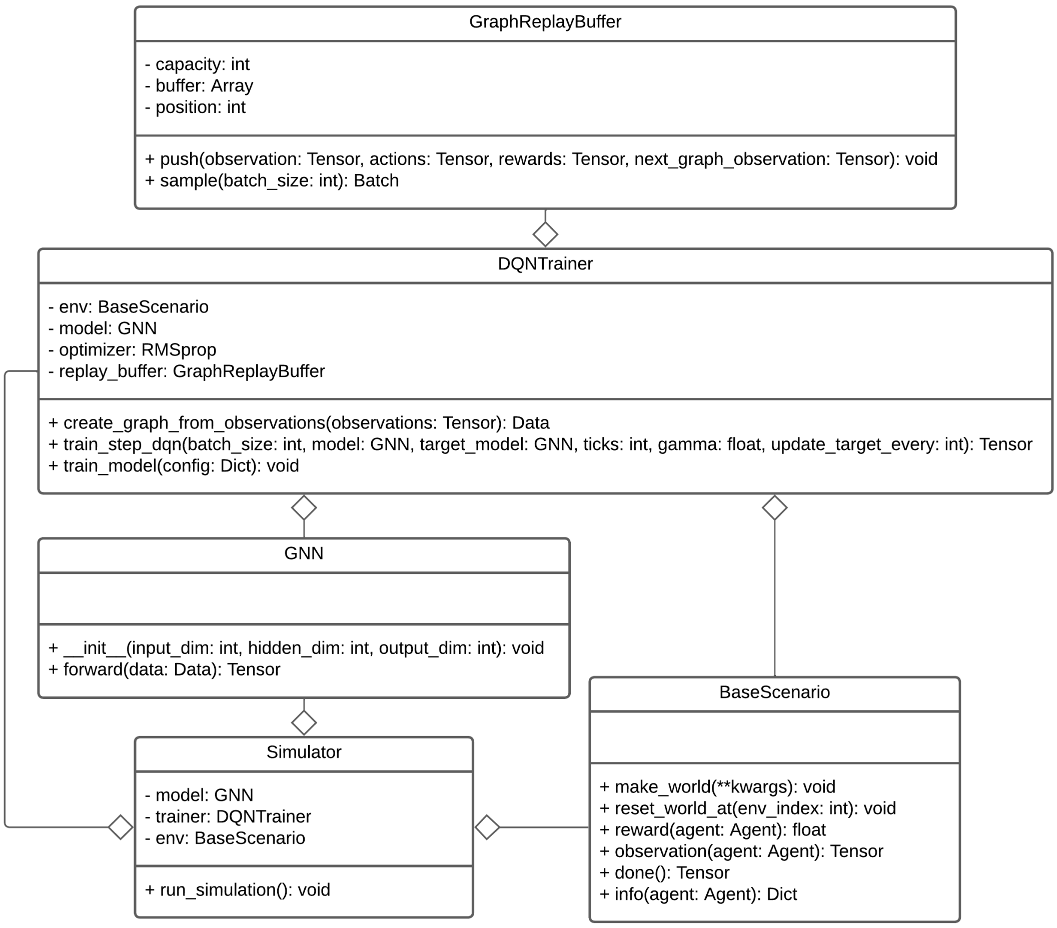

GraphReplayBuffer
- capacity: int
- buffer: Array
- position: int
+ push(observation: Tensor, actions: Tensor, rewards: Tensor, next_graph_observation: Tensor): void
+ sample(batch_size: int): Batch
DQNTrainer
- env: BaseScenario
- model: GNN
- optimizer: RMSprop
- replay_buffer: GraphReplayBuffer
+ create_graph_from_observations(observations: Tensor): Data
+ train_step_dqn(batch_size: int, model: GNN, target_model: GNN, ticks: int, gamma: float, update_target_every: int): Tensor
+ train_model(config: Dict): void
GNN
+ __init__(input_dim: int, hidden_dim: int, output_dim: int): void
+ forward(data: Data): Tensor
Simulator
- model: GNN
- trainer: DQNTrainer
- env: BaseScenario
+ run_simulation(): void
BaseScenario
+ make_world(**kwargs): void
+ reset_world_at(env_index: int): void
+ reward(agent: Agent): float
+ observation(agent: Agent): Tensor
+ done(): Tensor
+ info(agent: Agent): Dict

Figure 4.1: UML class diagram of the system

4.3 GNN

Following the design of the Graph Neural Network model performed in section section 3.3, the implementation of the class `GNN` is straightforward.

In the `init` method, extending the neural network model of `PyTorch`, three graph attention convolutional (GAT) layers are created followed by two linear layers, all with the correct dimension.

Then, the `forward` method of the network is implemented just by taking in input the graph data created and using all the layers created followed by the ReLU activation function as decided in the model design phase.

```python
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GCN, self).__init__()
        self.conv1 = GATConv(input_dim, hidden_dim, add_self_loops=False, bias=
            True)
        self.conv2 = GATConv(hidden_dim, hidden_dim, add_self_loops=False, bias=
            True)
        self.conv3 = GATConv(hidden_dim, hidden_dim, add_self_loops=False, bias=
            True)
        self.lin1 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.lin2 = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        x = torch.relu(x)
        x = self.conv3(x, edge_index)
        x = torch.relu(x)
        x = self.lin1(x)
        x = torch.relu(x)
        x = self.lin2(x)
        return x
```

4.4 DQNTrainer

The implementation of the class `DQNTrainer` represent the core of the system, it includes all the feature necessary for train the model, starting from the building of the graph data to the train step of the DQN algorithm explained in section 3.5.

4.4.1 create_graph_from_observations

```python
def create_graph_from_observations(self, observations):
    node_features = [observations[f'agent{i}'] for i in range(len(observations))]
    node_features = torch.stack(node_features, dim=0).squeeze(dim=1)

    agent_ids = torch.arange(len(observations)).float().unsqueeze(1)
    node_features = torch.cat([node_features, agent_ids], dim=1)

    num_agents = self.env.n_agents
    edge_index = []
    for i in range(num_agents):
        for j in range(i + 1, num_agents):
            edge_index.append([i, j])
            edge_index.append([j, i])
    edge_index.append([0,0])
    edge_index = torch.tensor(edge_index, dtype=torch.long).t().contiguous()
    graph_data = Data(x=node_features, edge_index=edge_index)
    return graph_data
```

This function is the implementation of the idea described in section 3.4, so it transforms agents' observations into a graph structure suitable for training the GNN model. It begins by extracting node features from the observations dictionary for each agent and stacking them into a tensor. It then adds an identifier for each agent to the node features, for ensure that the GNN differentiate the observations. The function constructs an undirected graph by defining edges between each pair of agents, ensuring both directions (i.e., agent ii to agent jj and agent jj to agent ii) are included, and appends a self-loop for the first agent. These edges are organized into an edge index tensor. Finally, the node features and edge index are combined into a `Data` object, which encapsulates the graph structure for GNN training.

4.4.2 train_model

This method of the `DQNTrainer` class, represent the external shell of the DQN algorithm presented in section 3.5. It contains the iteration all over the episodes and the steps, and the transformation in graph data (using the function explained in section 4.4.1) of the raw observations of the agents obtained from the environment.

After the graph data is structured, the step of the DQN algorithm (see section 4.4.3) is executed, and also the actions execution and the storing in the replay buffer are performed.

```python
def train_model(self, config):
    ...
    for episode in range(episodes):
        ...
        for _ in range(self.env.max_steps):
            ...
            graph_data = self.create_graph_from_observations(observations)
            self.model.eval()
            logits = self.model(graph_data)

            if random.random() < epsilon:
                actions = torch.tensor([random.randint(0, 8) for _ in range(len(
                    self.env.agents))])
            else:
                actions = torch.argmax(logits, dim=1)

            actions_dict = {f'agent{i}': torch.tensor([actions[i].item()]) for i
                in range(len(self.env.agents))}

            newObservations, rewards, done, _ = self.env.step(actions_dict)

            rewards_tensor = torch.tensor([rewards[f'agent{i}'] for i in range(len
                (self.env.agents))], dtype=torch.float)

            self.replay_buffer.push(graph_data, actions, rewards_tensor, self.
                create_graph_from_observations(newObservations))

            loss = self.train_step_dqn(128, self.model, self.target_model, ticks,
                update_target_every=10)
            episode_loss += loss
            total_episode_reward += rewards_tensor
            observations = newObservations

        epsilon = max(min_epsilon, epsilon * epsilon_decay)

        average_loss = episode_loss / 100
```

4.4.3 train_step_dqn

This method contains the core of the DQN algorithm presented in section 3.5.

It reflect all the steps already presented, from the batching of the past experiences from the `GraphReplayBuffer` to the calculation of the actual and target Q-values. Then the loss calculation is performed, terminating with a backpropagation phase.

```
def train_step_dqn(self, batch_size, model, target_model, ticks, gamma=0.99,
    update_target_every=10):
    ...
    (obs, actions, rewards, nextObs) = self.replay_buffer.sample(batch_size)

    values = model(obs).gather(1, actions.unsqueeze(1))
    nextValues = target_model(nextObs).max(dim=1)[0].detach()
    targetValues = rewards + gamma * nextValues
    loss = nn.SmoothL1Loss()(values, targetValues.unsqueeze(1))
    self.optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_value_(model.parameters(), 1)
    self.optimizer.step()

    if ticks % update_target_every == 0:
        target_model.load_state_dict(model.state_dict())
    self.writer.add_scalar('Loss', loss.item(), ticks)
    return loss.item()
```

4.5 GraphReplayBuffer

```
class GraphReplayBuffer:
    ...
    def push(self, graph_observation, actions, rewards, next_graph_observation):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (graph_observation, actions, rewards,
            next_graph_observation)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        sample = random.sample(self.buffer, batch_size)
        observations = [s[0] for s in sample]
        actions = [s[1] for s in sample]
        rewards = [s[2] for s in sample]
        next_graph_observations = [s[3] for s in sample]
        return (Batch.from_data_list(observations), torch.cat(actions), torch.cat(
            rewards), Batch.from_data_list(next_graph_observations))
```

The `GraphReplayBuffer` is a specialized component designed to support the training process of our DQN algorithm, performed by the already discussed `DQNTrainer`, using graph-structured data.

This buffer operates by storing and managing a fixed-size collection of past experiences, which include observations of graph states, corresponding actions taken, received rewards, and subsequent graph states. Upon initialization, the buffer is allocated a specified maximum capacity. When new experiences are pushed into the buffer via the `push` method, they are either appended to the buffer if there is still space available, or they replace the oldest experiences once the buffer has reached its capacity, ensuring a continuous recycling of memory. This mechanism is crucial for maintaining a diverse and representative set of experiences that the DQN can learn from over time.

The buffer also includes the `sample` method to randomly sample a batch of experiences, which is essential for breaking the temporal correlations in the training data and stabilizing the learning process. This sampling returns batches of observations, actions, rewards, and next observations in a format that is readily usable by the DQN algorithm.

4.6 Simulator

```python
class Simulator:
    ...
    def run_simulation(self):
        ...
        for step in range(self.env.max_steps):
            ...
            graph_data = self.trainer.create_graph_from_observations(observations)

            with torch.no_grad():
                logits = self.model(graph_data)
                actions = torch.argmax(logits, dim=1)

            actions_dict = {f'agent{i}': torch.tensor([actions[i].item()]) for i
                in range(len(self.env.agents))}

            observations, rewards, _, _ = self.env.step(actions_dict)

            total_reward += sum(rewards.values())

            ...
```

The `Simulator` is the last important component of the implementation. As mentioned during the description of the overall architecture, the `Simulator` is responsible of executing a simulation with an already trained model, in a defined `Scenario` and render the result of the simulation.

As we can see from the code, the `Simulator` has the references to: the `DQNTrainer`, the `GNN` model already trained and the `VMAS` scenario.

The method `run_simulation` perform a sequence of instructions similar to the training phase, it just iterate over the steps, build the graph data from the agents observations and get the actions from the GNN model, which corresponds to the optimal policies learned by the agents. Eventually it renders the result of each step of the simulation.

Chapter 5

Evaluation

An essential question arises: *"Is this approach effective?"* To address this, in this chapter is described our scientific approach to verify and evaluate the effectiveness, the accuracy and the scalability of the implemented work.

First of all, a batch of tasks is designed and implemented, to verify the accuracy of the swarm learned behaviours in different situations like: reaching a position, flocking or avoiding an obstacle. Afterwards, a fine-tuning phase is required to identify hyperparameter values that lead to improved and stable training. Subsequently, for each task outlined in the preceding section, data can be collected and analyzed during the training phase (e.g., rewards obtained by the agents) to provide insights into the model's behavior throughout the learning process. Following this, the learned policy of the swarm can be evaluated through pseudo-random simulations. Finally, the scalability of the policies can be assessed by observing their performance with an increased number of agents.

5.1 Tasks design

In this section is described the design of a set of tasks, engineered for make the swarm of agents perform different swarming behaviour for address a complex global goal. Here the main focus is on the objective of the task but most important on the design of the correspondent reward structure, which is a key element for each agent, because it defines how it's action affect the environment in a positive or

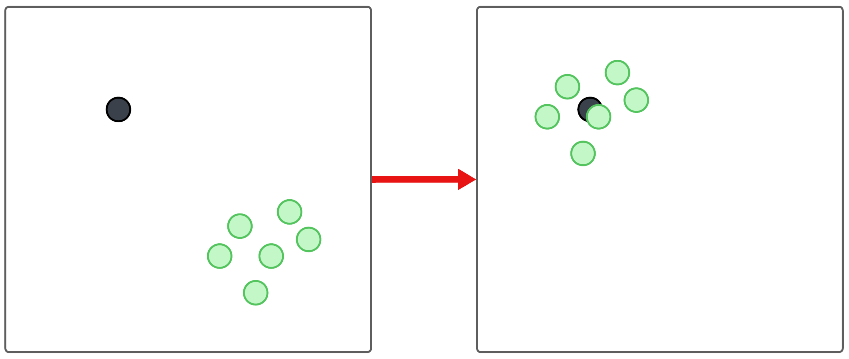


Figure 5.1: Go to position task

negative way, and consequently affect the learning of the correct behaviour.

5.1.1 T1: Go to position

This first task consists in make the swarm reach a position in the environment. As we can see in fig. 5.1 we have a set of agents (green dots) that for now doesn't match any particular formation, that has to learn how to reach a point in the environment (black dot).

The reward structure is simple and defined as follows.

Let:

$$\begin{aligned} d_i &= \|p_i - g_i\| \quad \text{(distance of agent } i \text{ to the goal)} \\ d_i^{prev} &= \text{previous distance of the agent } i \text{ to the goal} \\ \alpha &= \text{shaping factor} \\ r_i &= \alpha(d_i^{\text{prev}} - d_i) \quad \text{(positional reward for agent } i\text{)} \\ N &= \text{number of agents} \end{aligned}$$

If the agent i is on the goal, with r_{goal} as the radius of the target:

$$\text{on\_goal}_\mathbf{i} = \begin{cases} 50 & \text{if } d_i < r_{\text{goal}} \\ 0 & \text{otherwise} \end{cases}$$

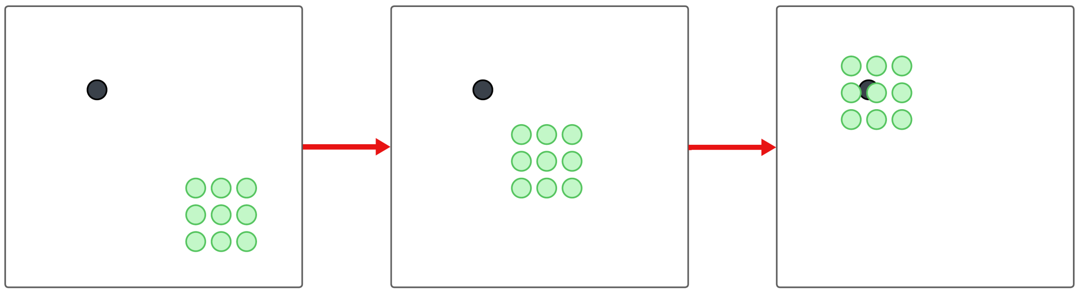


Figure 5.2: Flocking task

The reward R_i for each agent i is:

$$R_i = \sum_{i=1}^{N} (\alpha(d_i^{\text{prev}} - d_i) + \text{on\_goal}_{\mathbf{i}})$$

So, the idea is to give to each agent the same collective reward which is influenced by the performance of all the agents. In details, each reward takes into account the actual distance between the agent and the goal positions. If during the simulation this distance decreases the reward increases and vice versa. In this way we incentives the agent to go towards the goal, and we penalize it if it goes away from it. Furthermore, a bonus reward is given to an agent if it's exactly on the goal and since the reward is shared, we incentives the collaboration between agents for make at least one agent to reach the goal.

5.1.2 T2: Flocking

In this second task, we expect the swarm to keep a formation that is already structured such as a square. The idea is to reach a goal such as in the previous task, but this time performing a flocking behaviour [Rey87].

Flocking consist in move toward a position but without colliding with other agents, and at the same time keeping the cohesion for maintain the structure.

Let's analyze formally the reward structure which is composed by three parts, let:

$$\begin{aligned} d_{ij} &= \|p_i - p_j\| \quad \text{(distance between agent } i \text{ and agent } j) \\ d_{\text{desired}} &= \text{desired distance between agents} \\ \alpha_{\text{dist}} &= \text{distance shaping factor} \\ \beta &= \text{agent collision penalty} \end{aligned}$$

Distance to the goal

This part of the reward $R_{distance}$ is calculated exactly as in the previous task, see section 5.1.1. So basically, it consider if the agent is going towards or away from the goal, and it assign a bonus if the agent reach the exact position of the target.

Cohesion

For the cohesion part that has the aim to make the swarm keeps the formation, first is calculated for each agent i a mean shaped distance to the other agents:

$$D_i = \alpha_{\text{dist}} \left(\frac{1}{N-1} \sum_{\substack{j=1 \\ j \neq i}}^{N} (d_{ij} - d_{\text{desired}})^2 \right)$$

Then the reward for maintain the formation is:

$$R_{\text{cohesion},i} = D_i^{\text{prev}} - D_i$$

So, this part takes into account the desired distance between agents, and with a similar logic to the previous point, makes the agents keep the correct distance from each other for maintain the formation.

Collisions between agents

Let:

$$\text{collision}_{ij} = \begin{cases} 1 & \text{if } d_{ij} < d_{\text{collision}} \\ 0 & \text{otherwise} \end{cases}$$

Then the reward for collisions is:

$$R_{\text{agents\_collision},i} = \sum_{\substack{j=1 \\ j \neq i}}^{N} \beta \cdot \text{collision}_{ij}$$

This last part consider the collisions between agents and assign negative rewards β to the agents that collides with others.

So the final reward for each agent i is defined as:

$$R_i = \sum_{j=1}^{N} R_{distance,j} + R_{cohesion,j} + R_{agents\_collision,j}$$

As we can see for the formulas, also in this task the reward is shared. So each agent receive the same reward that is the result of the sum of the already explained three different component: distance from the goal, maintenance of the formation and collision between agents.

5.1.3 T3: Obstacle avoidance

This task is a natural progression of the first two and add more complexity to the scenario. Here we want the swarm of agents to reach a goal, maintain the formation without colliding with each others (flocking), and in addition to avoid an obstacle (red dot) that is placed on the trajectory for reach the target.

Here the reward calculation is straightforward, it exploit the goal reaching, the agent cohesion and the collision avoidance definitions of the previous task, but then a new component is added. These new part works as the collision between agents but this time it consider the collision with the obstacle, giving a negative reward in case an agent collide with it.

Let:

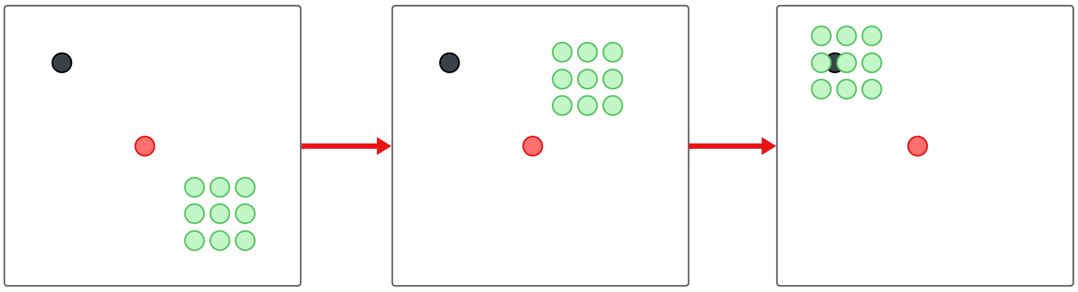


Figure 5.3: Obstacle avoidance task

$$d_{i0} = \|p_i - p_o\| \quad \text{(distance between agent } i \text{ and the obstacle)}$$
$$\gamma = \text{obstacle collision penalty}$$

$$\text{obstacle\_collision}_i = \begin{cases} 1 & \text{if } d_{io} < d_{\text{collision}} \\ 0 & \text{otherwise} \end{cases}$$

Then the reward for colliding with the obstacle is:

$$R_{\text{obstacle\_collision},i} = \gamma \cdot \text{obstacle\_collision}_i$$

Eventually the final shared reward that consider reaching the goal, flocking and avoiding the obstacle is:

$$R_i = \sum_{j=1}^{N} R_{distance,j} + R_{cohesion,j} + R_{agents\_collision,j} + R_{obstacle\_collision,j}$$

5.2 Scenarios implementation

The implementation of the environment is provided by the `BaseScenario` class of `VMAS` which is extended by each custom scenario that is responsible of building the environment for solve a specific task, defining the initial configuration of the

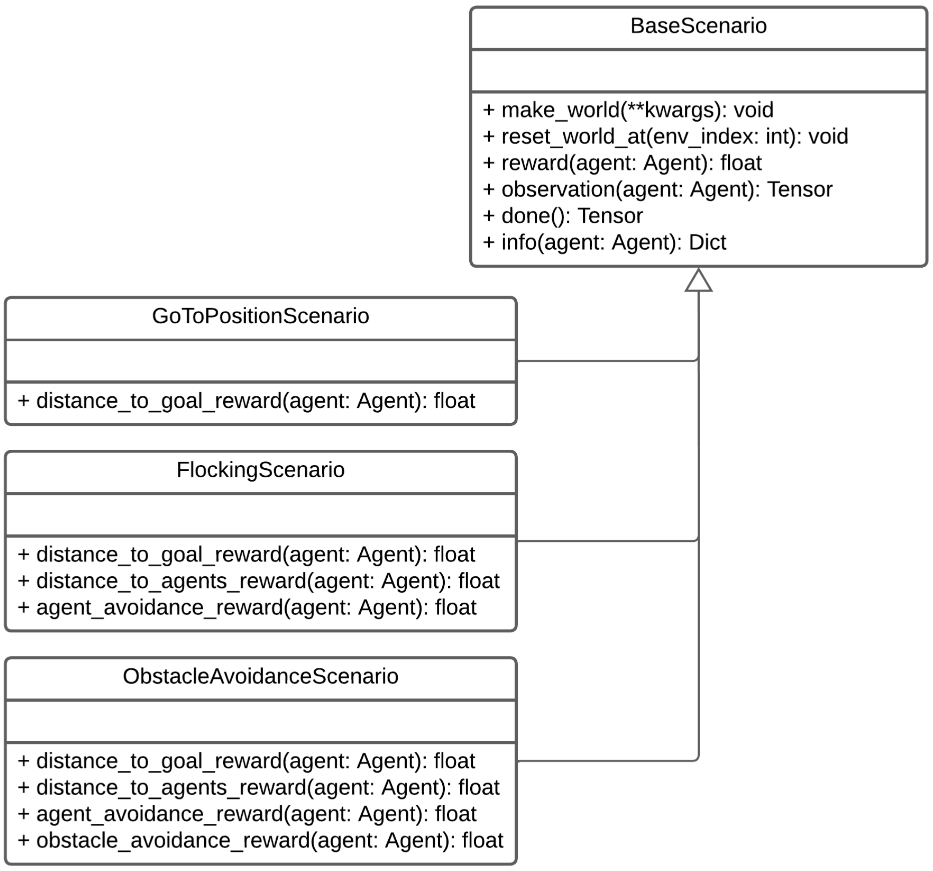

BaseScenario
+ make_world(**kwargs): void
+ reset_world_at(env_index: int): void
+ reward(agent: Agent): float
+ observation(agent: Agent): Tensor
+ done(): Tensor
+ info(agent: Agent): Dict
GoToPositionScenario
+ distance_to_goal_reward(agent: Agent): float
FlockingScenario
+ distance_to_goal_reward(agent: Agent): float
+ distance_to_agents_reward(agent: Agent): float
+ agent_avoidance_reward(agent: Agent): float
ObstacleAvoidanceScenario
+ distance_to_goal_reward(agent: Agent): float
+ distance_to_agents_reward(agent: Agent): float
+ agent_avoidance_reward(agent: Agent): float
+ obstacle_avoidance_reward(agent: Agent): float

Figure 5.4: UML class diagram of the scenarios hierarchy

agents, and the reward structure accordingly to the ones detailed in section 5.1.

5.2.1 GoToPositionScenario

The aim of this scenario is to provide the environment for accomplish the task described in section 5.1.1, so here is defined the starting configuration of the agents and the target point to reach. In the following code is implemented the reward structure as defined in the previous sections.

```python
def distance_to_goal_reward(self, agent: Agent):
    agent.distance_to_goal = torch.linalg.vector_norm(
        agent.state.pos - agent.goal.state.pos,
        dim=-1,
    )
    agent.on_goal = agent.distance_to_goal < agent.goal.shape.radius

    shaped_distance_to_goal = agent.distance_to_goal * self.pos_shaping_factor
    agent.pos_rew = agent.previous_distance_to_goal - shaped_distance_to_goal
    agent.previous_distance_to_goal = shaped_distance_to_goal

    reward = agent.pos_rew

    if agent.on_goal:
        reward = reward + 50

    return reward
```

5.2.2 FlockingScenario

This scenario is designed for represent the environment correspondent to the second task presented in section 5.1.2.

As in the previous one, in this scenario is defined a target point that the swarm has to reach, but also the agents are already configured in the formation that they have to maintain (a grid in this case) during the flocking.

In the following code there is the implementation of the reward structure of discussed in section 5.1.2, with all its components: the distance to the goal presented in the previous section, cohesion and collision between agents.

```python
def distance_to_agents_reward(self, agent: Agent):
    distance_to_agents = (
        torch.stack(
            [
                torch.linalg.vector_norm(agent.state.pos - a.state.pos, dim=-1)
                for a in self.world.agents
                if a != agent
            ],
            dim=1,
        )
        - self.desired_distance
    ).pow(2).mean(-1) * self.dist_shaping_factor
    agent.dist_rew = agent.previous_distance_to_agents - distance_to_agents
    agent.previous_distance_to_agents = distance_to_agents

    return agent.dist_rew

def agent_avoidance_reward(self, agent: Agent):
    reward = 0

    reward = sum(
        self.agent_collision_reward for other_agent in self.world.agents
        if agent.name != other_agent.name and self.world.get_distance(agent,
            other_agent) <= self.min_collision_distance
    )

    return reward
```

5.2.3 ObstacleAvoidanceScenario

This scenario is quite similar to the others but it has the addition of the implementation of the reward structure that makes the agents avoid the obstacle, as described in section 5.1.3.

```
def obstacle_avoidance_reward(self, agent: Agent):

    for i in range (1, self.n_obstacles + 1):
        if self.world.get_distance(agent, self.world.landmarks[i]) <= self.
            min_collision_distance :
            return self.obstacle_collision_reward

    return 0
```

5.3 Model Tuning

Before evaluating the technique, it is necessary to tune the hyperparameters to enhance the stability and effectiveness of the model's training. The training phase is structured into 800 episodes, each consisting of 100 steps, resulting in a total of 80,000 experiences per agent. This volume of experiences is crucial for enabling the swarm to learn complex tasks such as flocking or obstacle avoidance.

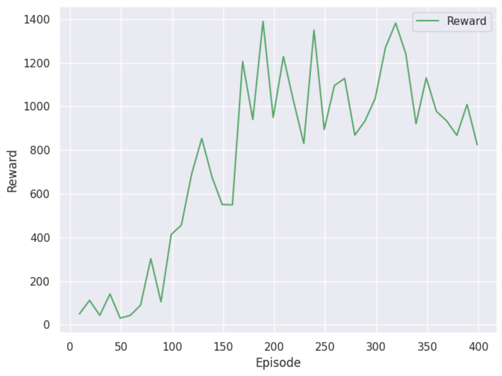
In all tasks, the model is trained using a swarm of 5 agents, which are semi-randomly initialized in a grid-like formation at the start of each episode (as illustrated in Figure 5.2). During training, an *epsilon-greedy* strategy is employed with $\epsilon_{\text{min}} = 0.05$, $\epsilon_{\text{max}} = 0.99$, and $\epsilon_{\text{decay}} = 0.9$. This strategy facilitates initial exploration of the environment through random actions, followed by exploitation of acquired knowledge. Additionally, γ is set to 0.99 to prioritize future rewards and mitigate suboptimal behaviors. The replay buffer, which stores past experiences, is configured with a size of 6000 and a batch size of 256. The model architecture for the GNN is detailed in Section 3.3, and RMSprop with a learning rate of 0.0001 is utilized as the optimizer for the learning process.

5.4 Training and Simulation Phases

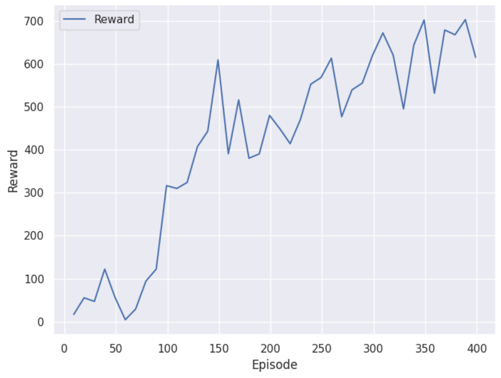
An initial evaluation can be conducted during the swarm system's training. Semi-random experiments can be configured by setting seeds to ensure reproducibility of each simulation. Given that rewards are shared in each task, the reward values can be aggregated across episodes to assess the model's performance in various tasks.

As illustrated in Figure 5.5, the reward trend depicted in the charts represents the mean of multiple pseudo-random trainings. The results show an expected pattern: during the training phase, the shared reward increases rapidly in the early episodes and then stabilizes with minor fluctuations. Notably, the reward growth is more pronounced in the *Flocking* and *Obstacle Avoidance* tasks compared to the *Go To Position* task. An additional factor of interest in the *Obstacle Avoidance* task is the frequency of obstacle collisions by the agents. As expected, the number of collisions decreases as training progresses, as shown in Figure 5.5d.

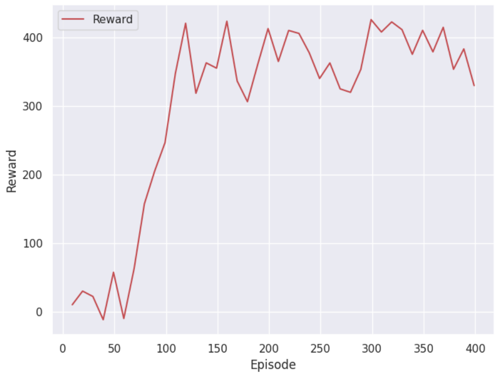
After analyzing the agents' behavior during training, it is also pertinent to examine the model's performance post-training. Following the execution of multiple pseudo-random simulations using the trained models and a swarm of *5 agents*, the results are presented in Figure 5.6. This figure includes boxplots that depict



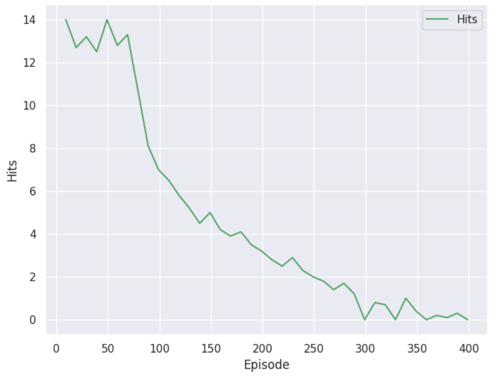
(a) Go to position.



(b) Flocking.



(c) Obstacle avoidance.



(d) Hits of the obstacle.

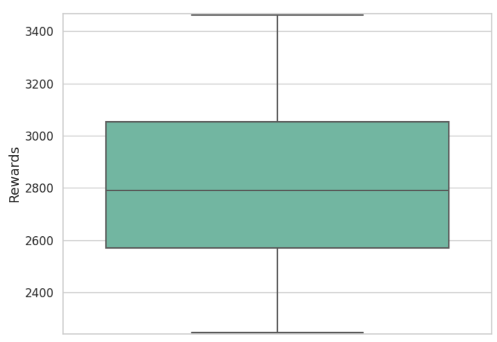
Figure 5.5: Reward trends during training.

the computed mean reward trends for each task, clearly illustrating the reward distribution for this case study. It is observed that the mean rewards obtained in the post-training simulations are slightly higher than those measured during the training phase.

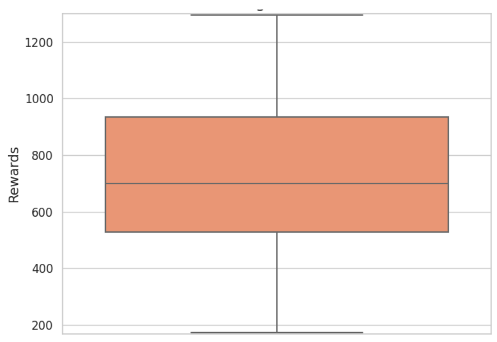
5.5 Scalability

Given that our approach performs well in both the training and simulation phases with *5 agents*, we proceed with further analysis to evaluate the scalability of this technique.

To assess the scalability of the proposed approach, it is essential to examine the system's adaptability to varying numbers of agents under different tasks. This



(a) Go to position.



(b) Flocking.

Figure 5.6: Reward distribution with 5 agents.

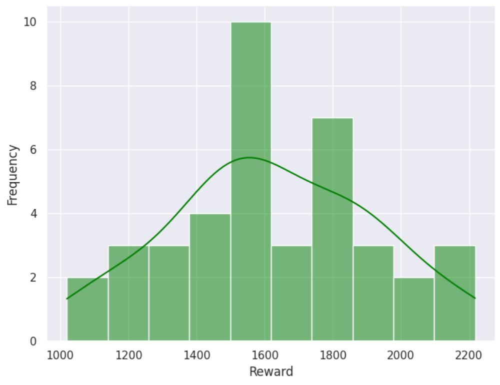
evaluation involves analyzing the model's performance as the number of agents increases, which is crucial for understanding the robustness and efficiency of the approach in practical applications.

First, we increased the swarm size from *5 agents* to *9 agents*, conducted a new training phase, and then gathered the reward trends using the same methodology as before. The results are presented in the distribution charts of Figure 5.7, which indicate that the agents successfully completed the tasks.

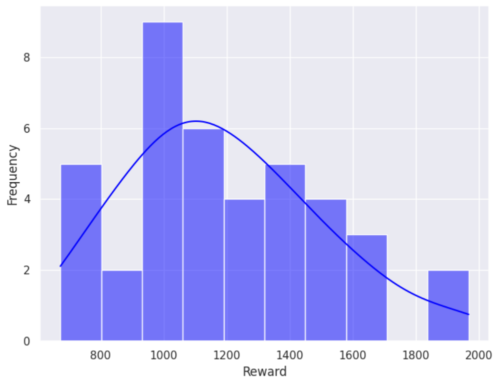
Subsequently, we further increased the swarm size to *12 agents* and applied the same procedure, obtaining the results shown in Figure 5.8. It is evident that as the swarm size increases, the mean reward also increases. This is due to the reward structure design described in Section 5.1, where the final reward is proportional to the number of agents.

Based on the evaluations performed, we can conclude that this approach effectively scales across scenarios with *5, 9, and 12 agents*. Specifically, in tasks requiring agents to navigate to specific positions and in flocking scenarios, the GNN-based model maintains its performance metrics, such as task completion efficiency, as the number of agents increases.

This suggests that the model is capable of handling larger swarms with minimal degradation in performance. The importance of this evaluation lies in its implications for real-world applications, where swarm size significantly impacts the complexity of coordination and control. A scalable approach ensures that the model remains practical and efficient, even as operational demands increase.

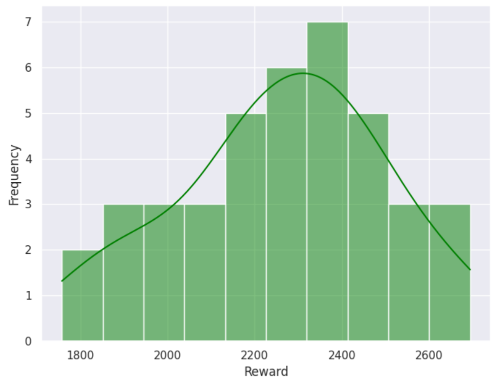


(a) Go to position.

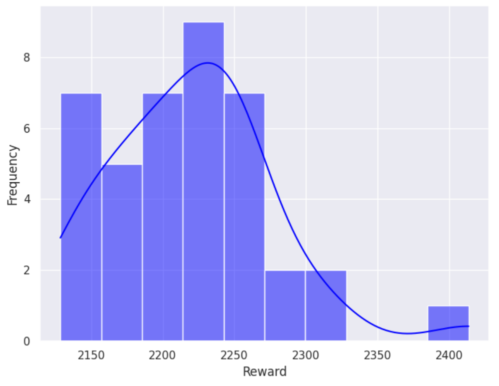


(b) Flocking.

Figure 5.7: Scalability evaluation with 9 agents.



(a) Go to position.



(b) Flocking.

Figure 5.8: Scalability evaluation with 12 agents.

Chapter 6

Conclusion and Future Work

In this thesis, we explore a novel approach for engineering a MARL scenario using a fully automated methodology: GNN models coupled with the DQN learning algorithm. We have developed a system capable of training multi-agent systems to perform complex swarming behaviors to achieve a global objective. Concepts related to MARL, GNN, Multi-Agent Systems, and Cyber-Physical Systems are presented comprehensively. Subsequently, we design a well-engineered system that considers both the specifics of a pure software system and the features of machine learning. A set of tasks is modeled alongside reward structures to enable the swarm to learn to reach a position, flock, and avoid obstacles. In the concluding part, a structured approach is applied to evaluate various aspects of the proposed solution, including its effectiveness and scalability.

Future work offers numerous possibilities. We can consider testing different algorithms with this GNN approach to compare performances or optimizing the current system to enhance its scalability. Another option could involve engineering different tasks or behaviors for the swarm to perform.

This thesis establishes a foundation for a promising approach that can be further explored and refined to realize self-organizing systems using a comprehensive machine learning methodology.

Bibliography

[ABC^{+}20] Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *Int. J. Robotics Res.*, 39(1), 2020.

[ABZ23] Rana Azzam, Igor Boiko, and Yahya Zweiri. Swarm cooperative navigation using centralized training and decentralized execution. *Drones*, 7(3), 2023.

[ACV24] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. Macroswarm: A field-based compositional framework for swarm programming. *CoRR*, abs/2401.10969, 2024.

[AS24] Gianluca Aguzzi and Claudio Savaglio. Engineering distributed collective intelligence in cyber-physical swarms. In *2024 20th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 570–575, 2024.

[AVE23] Gianluca Aguzzi, Mirko Viroli, and Lukas Esterle. Field-informed reinforcement learning of collective tasks with graph neural networks. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2023, Toronto, ON, Canada, September 25-29, 2023*, pages 37–46. IEEE, 2023.

[BBC$^+$19] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[BBD$^+$23] Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. Torchrl: A data-driven decision-making library for pytorch, 2023.

[BBS08] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Trans. Syst. Man Cybern. Part C*, 38(2):156–172, 2008.

[BCJ$^+$19] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019.

[BCP$^+$16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[BF16] Giacomo Barbieri and Cesare Fantuzzi. Design of cyber-physical systems: Definition and metamodel for reusable resources. In *21st IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2016, Berlin, Germany, September 6-9, 2016*, pages 1–9. IEEE, 2016.

[BKBP22] Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. Vmas: A vectorized multi-agent simulator for collective robot learning. *The 16th International Symposium on Distributed Autonomous Robotic Systems*, 2022.

[BPM23] Matteo Bettini, Amanda Prorok, and Vincent Moens. Benchmarl: Benchmarking multi-agent reinforcement learning. *arXiv preprint arXiv:2312.01472*, 2023.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[BPZ19] David Baldazo, Juan Parras, and Santiago Zazo. Decentralized multi-agent deep reinforcement learning in swarms of drones for flood monitoring. In *27th European Signal Processing Conference, EUSIPCO 2019, A Coruña, Spain, September 2-6, 2019*, pages 1–5. IEEE, 2019.

[Cas23] Roberto Casadei. Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. *ACM Comput. Surv.*, 55(13s):275:1–275:37, 2023.

[Che20] Gang Chen. A new framework for multi-agent reinforcement learning - centralized training and exploration with decentralized execution via policy distillation. In Amal El Fallah Seghrouchni, Gita Sukthankar, Bo An, and Neil Yorke-Smith, editors, *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, pages 1801–1803. International Foundation for Autonomous Agents and Multiagent Systems, 2020.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[GMP21] Atika Gupta, Priya Matta, and Bhasker Pant. Graph neural network: Current state of art, challenges and applications. *Materials Today: Proceedings*, 46:10927–10932, 2021. International Conference on Technological Advancements in Materials Science and Manufacturing.

[GPAM$^+$14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[HR16] Ionel-Alexandru Hosu and Traian Rebedea. Playing atari games with deep reinforcement learning and human checkpoint replay. *CoRR*, abs/1607.05077, 2016.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* OpenReview.net, 2017.

[LLM$^+$18] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning, 2018.

[MSF16] George Dan Mois, Teodora Sanislav, and Silviu C. Folea. A cyber-physical system for environmental monitoring. *IEEE Trans. Instrum. Meas.*, 65(6):1463–1471, 2016.

[OAA$^+$19] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113, 2019.

[PGM$^+$19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward

Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. `https://pytorch.org`, 2019. Accessed: 2024-06-28.

[PPCE22] Danilo Pianini, Federico Pettinari, Roberto Casadei, and Lukas Esterle. A collective adaptive approach to decentralised k-coverage in multi-robot systems. *ACM Trans. Auton. Adapt. Syst.*, 17:4:1–4:39, 2022.

[PTF$^+$21] Giovanni Pellegrini, Alessandro Tibo, Paolo Frasconi, Andrea Passerini, and Manfred Jaeger. Learning aggregation functions. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 2892–2898. ijcai.org, 2021.

[PyG23] PyGeometric Development Team. Pygeometric: A python library for geometric deep learning, 2023. Accessed: 2024-06-28.

[Rey87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In Maureen C. Stone, editor, *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, Anaheim, California, USA, July 27-31, 1987*, pages 25–34. ACM, 1987.

[SAB$^+$19] Reza Shakeri, Mohammed Ali Al-Garadi, Ahmed Badawy, Amr Mohamed, Tamer Khattab, Abdulla K. Al-Ali, Khaled A. Harras, and Mohsen Guizani. Design challenges of multi-uav systems in cyber-physical applications: A comprehensive survey and future directions. *IEEE Commun. Surv. Tutorials*, 21(4):3340–3385, 2019.

[SGT$^+$09] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009.

[SHM$^+$16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

[SHS$^+$17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[SKZK17] Adrian Sosic, Wasiur R. KhudaBukhsh, Abdelhak M. Zoubir, and Heinz Koeppl. Inverse reinforcement learning in swarm systems. In Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee, editors, *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pages 1413–1421. ACM, 2017.

[SMSM99] Richard S. Sutton, David A. McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 1057–1063. The MIT Press, 1999.

[SV00] Peter Stone and Manuela M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Auton. Robots*, 8(3):345–383, 2000.

[SZWL23] Yunchong Song, Chenghu Zhou, Xinbing Wang, and Zhouhan Lin. Ordered GNN: ordering message passing to deal with heterophily and over-smoothing. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[TBG+21] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021.

[TTK+23] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023.

[VBC+19] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019.

[VCC+18] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Mach. Learn.*, 8:279–292, 1992.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[YHC+18] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '18. ACM, July 2018.

[ZCH+20] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[ZYC+18] Lianmin Zheng, Jiacheng Yang, Han Cai, Ming Zhou, Weinan Zhang, Jun Wang, and Yong Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[ZYCK20] Yuchen Zhou, F. Richard Yu, Jian Chen, and Yonghong Kuo. Cyber-physical-social systems: A state-of-the-art survey, challenges and opportunities. *IEEE Commun. Surv. Tutorials*, 22(1):389–425, 2020.

Acknowledgements

I would like to express my gratitude to everyone who has supported me throughout this project and, more broadly, throughout my entire academic journey. My deepest thanks go first and foremost to my parents, who, even unintentionally, show me the beauty in things every day and serve as irreplaceable life role models. I want to thank my girlfriend for listening to me and exploring my restless mind, bringing calm and peace of mind. I am also immensely grateful to my lifelong friends, who brighten my days and remind me that there will always be room to stay young at heart. Lastly, to my fellow classmates, with whom I have shared both joys and fears, this is just a "see you later." I hope our paths cross again and that we can build something great together.