Doctoral Thesis

Studies on Learning-Based Methods for Controllers of Multi-Agent Systems

マルチエージェントシステムのコントローラの学習型手法に関する研究

D214412 Han Ziyao

Graduate School of Advanced Science and Engineering
Hiroshima University
September 2024

Abstract

This thesis focuses on developing collective behaviors for Multi-Agent Systems (MASs). An MAS is a system composed of multiple autonomous agents interacting with each other and the environment without relying on a centralized control structure. Although agents in MASs can have pre-designed behaviors, complex environments often make it difficult or even impossible to develop appropriate agent behaviors in advance. Learning-based methods offer an alternative approach, allowing agents to learn new behaviors online to improve the performance of the entire MAS steadily. The two main learning-based methods used in MAS are Evolutionary Robotics (ER) and Deep Reinforcement Learning (DRL). This thesis presents contributions to the field of MASs from the following two aspects.

Firstly, this thesis discusses how the evolutionary robotics approach can be applied to develop controllers with image inputs. The ability of agents to perceive and interpret their environment is crucial for effective interaction. Among various sensory inputs, camera inputs play a pivotal role, providing rich visual information that can be critical for object recognition and coordination among agents. However, evolutionary algorithms have limitations when used with high-dimensional inputs. Therefore, a deep neuroevolution method is proposed to generate collective behaviors. The results of computer simulations show that the proposed method outperforms Deep Q-Learning, a typical DRL approach, and demonstrates higher flexibility, scalability, and fault tolerance.

Secondly, this thesis investigates how the DRL approach can be applied to generating collective behaviors for MASs. Although DRL has achieved many successes in static environment tasks, it suffers from the sparse reward problem, where it is difficult for the agent to obtain rewards during exploration, resulting in slow learning speeds and poor performance. An intuitive solution is reward shaping, where denser rewards are designed for various agent behaviors. However, in some complex scenarios, it is practically infeasible to design rewards for specific behaviors of agents due to the vast state and action space. An alternative solution is Hierarchical Reinforcement Learning (HRL), where the original task is decomposed into multiple sub-tasks. Sub-controllers are first trained to complete corresponding sub-tasks. The high-level controller is then trained to activate different trained sub-controllers to accomplish the original task. However, when the training environment changes, it is hard to guarantee that pre-trained sub-controllers can still stably complete the corresponding sub-tasks. This thesis proposes novel methods integrating DRL with Imitation Learning (IL) and HRL to overcome the sparse reward problem. The collective behaviors are evaluated in a beach volleyball game and a key-to-door transport task. The results show that the proposed methods overcome the sparse reward problem and outperform conventional DRL and HRL methods.

Contents

1 Introduction **1**

1.1 Background and Motivation . . . 1

1.2 Thesis Objectives . . . 2

1.3 Structure of the Thesis . . . 4

2 Review of Multi-Agent Systems and Design Methods **7**

2.1 Multi-Agent Systems and Swarm Robotics Systems . . . 7

2.1.1 Multi-Agent Systems . . . 7

2.1.2 Swarm Robotics Systems . . . 9

2.1.3 Design Methods of MASs and SRSs . . . 12

2.2 Learning-Based Methods . . . 13

2.2.1 Deep Learning . . . 13

2.2.2 Reinforcement Learning . . . 20

2.2.3 Imitation learning . . . 28

2.2.4 Evolutionary Robotics . . . 30

3 Generating Collective Transport of a Swarm Robotic System by Deep Neuroevolution **35**

3.1 Introduction . . . 35

3.2 Related Works . . . 36

3.3 Methods and Experiment Settings . . . 37

3.3.1 Deep Neuroevolution . . . 37

3.3.2 Environment Settings . . . 37

3.3.3 Fitness/Reward Settings . . . 40

3.3.4 Network Structure . . . 40

3.3.5 Hyper-parameter Settings . . . 41

3.4 Results . . . 41

3.5 Discussion . . . 42

3.5.1 Flexibility Experiment . . . 43

3.5.2 Scalability Experiment . . . 45

3.6 Conclusions 47

4 Developing Multi-Agent Adversarial Environment Using Reinforcement Learning and Imitation Learning 49

4.1 Introduction 49
4.2 Research Methodology 50
4.2.1 Proximal Policy Optimization(PPO) 51
4.2.2 Generative Adversarial Imitation Learning (GAIL) 51
4.2.3 Attention Mechanism 52
4.2.4 Beach Volleyball Game 53
4.2.5 Self-Play 53
4.3 Environmental Settings 54
4.3.1 Robot Settings 54
4.3.2 Environment 54
4.3.3 Self-play Settings 56
4.3.4 Elo Rating System 57
4.4 Simulation 1: Overcoming The Sparse Reward Problem with The Imitation Reinforcement Learning Method 58
4.4.1 Reward Settings 58
4.4.2 Results 60
4.5 Simulation 2: Generating Competitive Behavior in An Adversarial Environment with The Attention Mechanism 66
4.5.1 Results 66
4.6 Conclusions 72

5 Collective Transport Behavior in a Robotic Swarm with Hierarchical Imitation Learning 73

5.1 Introduction 73
5.2 Related Works 74
5.3 Methods 75
5.3.1 Hierarchical Reinforcement Learning 75
5.3.2 Hierarchical Imitation Learning 76
5.4 Experiment 77
5.4.1 The Key-to-door Transport Task 77
5.4.2 Robot Specifications 78
5.4.3 Reward Settings 78
5.4.4 Implementation of the Conventional PPO, HRL, and HIL Methods 79
5.4.5 Experimental Settings 81
5.5 Results and Discussion 82

5.6 Conclusions 86

6 Conclusions **91**

A Publications Presented in the Thesis **103**

B List of Publications **105**

Acknowledgements **107**

Chapter 1

Introduction

1.1 Background and Motivation

Artificial intelligence (AI) [1] is a science and technology dedicated to imbuing computer systems with human intelligence. With the surge in computing power, the exponential growth in data volume, and the rapid advancement of algorithms, AI has become one of the most captivating topics in today's technology landscape [2]. The definition of AI encompasses many aspects, including the ability to simulate intelligent human behaviors such as learning, reasoning, problem-solving, language understanding, and perception [3]. In AI, a robot can be described as an intelligent agent, as AI and Robotics share fundamental concepts such as perception, cognition, action, and learning. The integration of Artificial Intelligence (AI) and Robotics represents a crucial intersection of cutting-edge technologies, reshaping the landscape of automation, autonomy, and intelligence [4] [5]. In recent years, the integration of AI and Robotics has accelerated with the rise of intelligent agents capable of perception, cognition, and decision-making.

However, limitations in the capabilities of a single agent have become apparent, especially in systems that require multi-agent cooperation. For this reason, design methods that utilize multiple agents to cooperate with each other to complete a given task have become a hot topic, referred to as multi-agent systems (MASs). An MAS [6] is a computing system composed of multiple agents that can independently perceive the environment, perform reasoning and decision-making, and interact with other agents through communication and collaboration. Multi-agent systems are designed to solve complex problems that require collaboration or competition among multiple participants, such as resource allocation, task allocation, coordination, and conflict resolution [7]. Breakthroughs in deep learning, reinforcement learning, and evolutionary robotics have empowered robots with human-like capabilities, enabling them to interact with the environment and adapt to dynamic conditions.

Moreover, in MASs, a nature-inspired robotic system called swarm robotics systems (SRSs) [8] has attracted the attention of many researchers due to its characteristics. The fundamental idea is to develop large-scale multi-agent systems that perform tasks efficiently, robustly, and adaptively. Swarm robotic systems draw inspiration from social animals, such as ant colonies, bird flocks, and fish schools, to develop robotics systems that exhibit swarm intelligence features. These features include scalability, flexibility, and fault tolerance [9]. Scalability is the ability to accomplish a given task using different numbers of robots. Flexibility is the ability of an SR system to overcome changes in tasks. Fault tolerance is the ability to complete a task even when robots are lost.

Although agents in MASs can have pre-designed behaviors, complex environments often make it difficult or even impossible to develop appropriate agent behaviors in advance. Furthermore, hardwired behavior might become unsuitable in changing environments. Learning-based methods are an alternative approach in which agents leverage machine learning algorithms to discover and predict environmental changes and adapt to unforeseen circumstances. The two main learning-based methods used in MAS are Evolutionary Robotics (ER) and Deep Reinforcement Learning (DRL) [10]. However, developing MAS and SRS controllers using learning-based methods is still tricky in non-stationary environments. For example, evolutionary algorithms can require significant computational resources, especially for large populations, complex fitness evaluations, or high-dimensional inputs. Similarly, in DRL approaches, input redundancy often results when an agent is provided with too much information, resulting in slow learning and poor performance. Moreover, DRL suffers from the sparse reward problem in some complex tasks. The sparse reward problem refers to cases in which it is difficult for the agent to obtain rewards during exploration. Given the above, several design approaches based on ER and DRL are proposed in this thesis to improve the performance of the MASs and SRSs in dynamic environments.

1.2 Thesis Objectives

This thesis aims to improve the performance of MASs when ER and DRL are adopted as the design method. In this thesis, contributions are presented to the fields of learning-based methods for MASs in the following three aspects:

The first objective of this thesis is to generate collective behaviors in a robotic swarm with the ER approach. In multi-agent systems, perceiving the environment is essential for effective interaction and decision-making between agents. Visual input is particularly critical among various sensory inputs, providing rich visual information essential for navigation, object recognition, and coordination between agents. However, evolutionary algorithms have limitations when used with high-dimensional inputs. For this reason,

this thesis proposes a deep neuroevolution method that leverages convolutional neural networks to generate collective behaviors for swarm robotic systems with raw camera inputs. The collective behaviors are evaluated in a collective transport task. The results show that the proposed method outperforms Deep Q-Learning in most cases, showcasing flexibility, scalability, and fault tolerance.

The second objective of this thesis is to generate competitive and collective behavior for a MAS with DRL approaches. An intuitive solution to the sparse reward problem is to provide the agent with denser rewards by designing dense reward functions for specific states or agent's behavior. However, in dynamic scenarios, it is infeasible to design rewards for specific behaviors of agents due to the vast state and action spaces. Moreover, doing so often results in the agent performing particular behaviors to obtain more dense rewards rather than completing the given task. Imitation learning (IL) is another approach to solving the sparse reward problem. In IL, agents can obtain the information they require from expert demonstrations, such as determining what action to take in a given state. In this case, agents can learn what to do by imitating expert demonstrations, even in a sparse reward environment. However, expert demonstrations tend to be generated by human players rather than spontaneously generated by agents, which is undesirable in decentralized MASs. Therefore, this thesis proposes a combination of DRL and IL methods. In the proposed method, the controller trained by DRL using dense rewards serves as an expert and uses imitation learning to guide agents to generate behaviors under sparse rewards. Additionally, the attention mechanism, inspired by cognitive attention, is integrated with DRL to address input redundancy. The competitive and collective behaviors are evaluated in a beach volleyball game. The results of computer simulations showed that the controllers trained in the proposed method overcame the sparse reward problem and performed better than those trained in the conventional DRL method.

The third objective of this thesis is to address the sparse reward problem in sequential decision-making tasks for a SRS. Sequential decision-making is widely used in reinforcement learning to describe problems in which agents must complete tasks in a particular order. When the reward function is designed to provide rewards for task completion, it is difficult for agents to obtain rewards by randomly exploring the environment. Hierarchical reinforcement learning (HRL) is a typical solution, in which the complex task is decomposed into several simple subtasks. In HRL, the high-level controller is trained to output different subcontrollers, which are pre-trained to complete the subtasks, to complete the original complex task. However, there might be a limit to this flexibility; when the environment in which the high-level controller was trained is changed, it is difficult to guarantee that the pre-trained subcontrollers can still stably complete the corresponding subtasks in the new environment. This thesis proposes the hierarchical imitation learning (HIL) method, which combines hierarchical and imitation learning, to train the controller

of a robotic swarm to overcome the sparse reward problem. The collective behaviors are evaluated in a key-to-door transport task. The results of computer simulations showed that the proposed method successfully solved the sparse reward problem compared to conventional DRL. Moreover, when the training environment was changed, the controller trained by the proposed method achieved more stable performance compared to the conventional HRL approach.

1.3 Structure of the Thesis

The overall structure of this thesis is depicted in Figure 1.1. All experiments in this thesis were conducted through computer simulations. In this section, we provide a summary of each chapter.

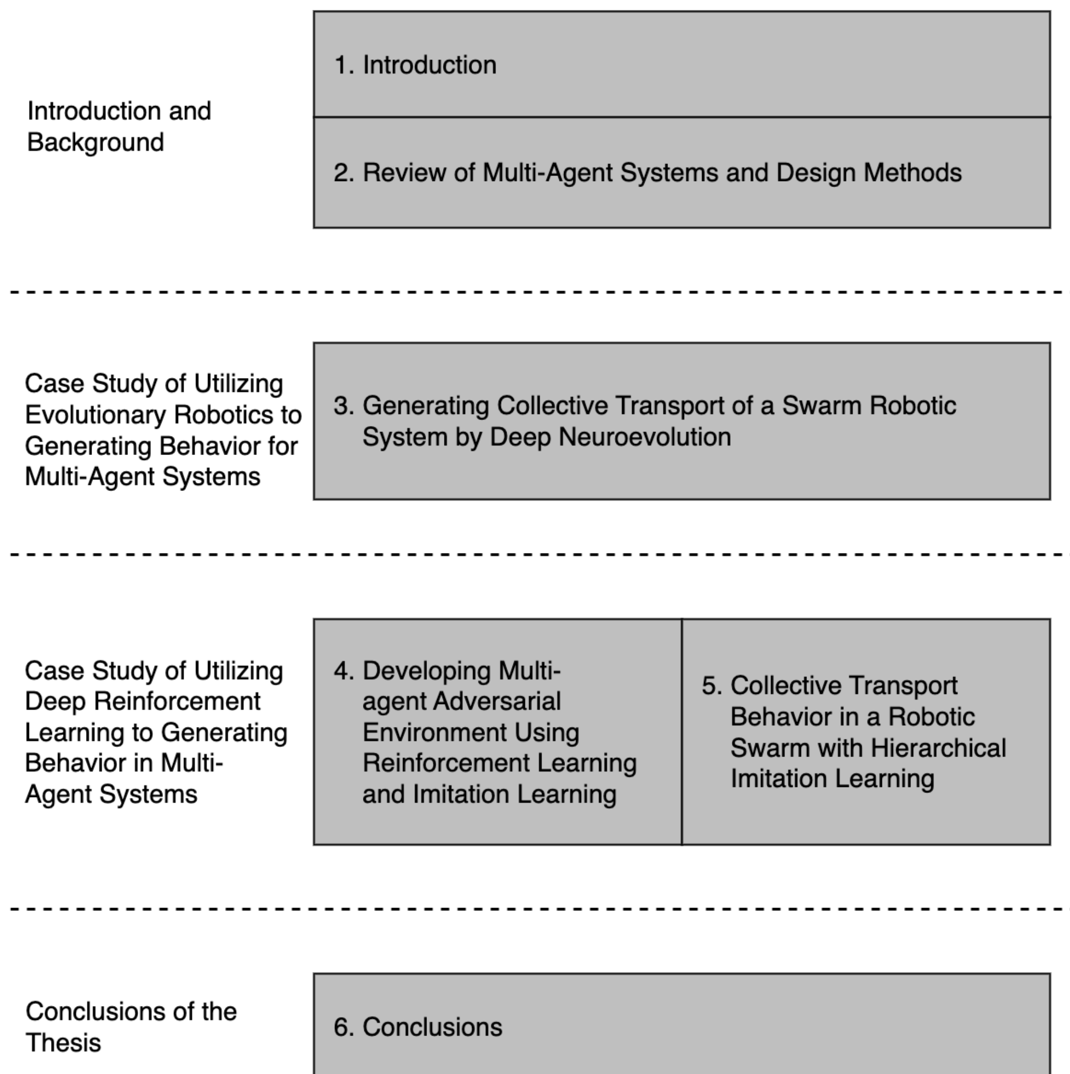


Figure 1.1: Overview of the thesis structure

This thesis is comprised of five chapters:

- **Chapter 2** describes multi-agent systems and design methods. The first section reviews multi-agent systems and swarm robotics systems and introduces two design methods, rule-based systems and learning-based methods. The second section provides introductions to deep learning, reinforcement learning, imitation learning, and evolutionary robotics, covering basic topics and recent trends.
- **Chapter 3** describes how evolutionary robotics is utilized to generate collective transport behaviors in a swarm robotic system with raw camera inputs.
- **Chapter 4** aims to generate competitive and collective behaviors for a multi-agent system using deep reinforcement learning approaches. The proposed method integrates the deep reinforcement learning and imitation learning to overcome the sparse reward problem.
- **Chapter 5** proposes a novel training method for a robotic swarm to achieve sequential decision-making tasks. The proposed method combines hierarchical reinforcement and imitation learning to accomplish a key-to-door task.
- **Chapter 6** summarizes this thesis.

Chapter 2

Review of Multi-Agent Systems and Design Methods

In this chapter, we describe multi-agent systems and design methods. Section 2.1 provides a review of multi-agent systems and swarm robotics systems. Two design methods, rule-based systems and learning-based methods, are introduced in this section. Section 2.2 offers introductions to deep learning, reinforcement learning, imitation learning, and evolutionary robotics, covering basic topics and recent trends.

2.1 Multi-Agent Systems and Swarm Robotics Systems

2.1.1 Multi-Agent Systems

An agent [11] is a computational entity that is capable of perceiving its environment, processing information, making decisions, and taking actions to achieve specific goals or objectives. In the context of MASs, an agent is an autonomous entity that operates within a larger system, interacting with other agents and potentially with its environment. Agents can be software-based entities, robots, humans, or even organizations. They possess their own internal state, knowledge, and reasoning mechanisms, allowing them to act autonomously and adaptively in dynamic and uncertain environments. Agents may exhibit different levels of autonomy, ranging from reactive agents that respond directly to stimuli in their environment to deliberative agents that engage in reasoning and planning to achieve more complex goals. The concept of agents is foundational to the study and development of MAS, enabling the modeling, analysis, and design of complex distributed systems comprised of multiple interacting entities.

A Multi-Agent System (MAS) [6] is a computational system composed of multiple

autonomous agents, each possessing its own capabilities, knowledge, goals, and behaviors. These agents interact with each other and potentially with their environment to achieve individual or collective objectives. The interactions among agents may involve communication, cooperation, coordination, negotiation, and competition. MASs draw inspiration from various disciplines, including artificial intelligence, distributed computing, game theory, and social sciences. The key characteristics of MASs include agent autonomy, emergent behavior, agent heterogeneity and homogeneity, coordination and cooperation, centralization and decentralization.

- **Agent Autonomy**: Agents in MASs possess a degree of autonomy, meaning they can perceive their environment, make decisions, and take actions independently. This autonomy allows agents to operate flexibly and adaptively in dynamic and uncertain environments.

- **Emergent Behavior**: One of MAS's key characteristics is emergent behavior, where complex global patterns or behaviors arise from the interactions of simple local rules followed by individual agents. Emergent behavior often leads to self-organization, adaptation, and the emergence of collective intelligence in the system.

- **Agent Heterogeneity and Homogeneity**: Agents in MASs can exhibit heterogeneity or homogeneity in terms of their capabilities, knowledge, goals, and behaviors. The heterogeneity enables agents to specialize in different tasks, collaborate effectively, and complement each other's strengths and weaknesses. On the other hand, agent homogeneity can simplify the design, analysis, and management of MASs by reducing complexity and enabling more straightforward communication and coordination among agents.

- **Coordination and Cooperation**: MASs often involve coordination and cooperation among agents to achieve shared objectives. This may require negotiation, consensus-building, task allocation, and conflict resolution mechanisms to ensure that agents can work together harmoniously towards common goals.

- **Centralization and Decentralization**: MAS architectures can vary in terms of centralization, ranging from fully decentralized systems where each agent operates independently to centralized systems where a central authority orchestrates the actions of all agents. Hybrid architectures that combine elements of decentralization and centralization are also common.

MASs find applications across various domains, such as robotics [12], intelligent transportation systems [13], distributed sensor networks [14], e-commerce [15], and social networks [16], due to their ability to model and solve complex problems involving multiple interacting entities.

- **Robotics**: MASs are widely used in robotics for tasks such as exploration [17], search and rescue [18], and coordination of robot teams [19]. Agents in MAS-based robotic systems collaborate to achieve common objectives, navigate through environments, and perform tasks efficiently.

- **Intelligent Transportation Systems (ITS)**: MASs play a crucial role in ITS for traffic management [20], congestion control [21], route optimization [22], and autonomous vehicle coordination [23]. Agents representing vehicles, traffic signals, and infrastructure interact to optimize traffic flow, reduce congestion, and enhance safety.

- **Distributed Sensor Networks**: MASs are deployed in distributed sensor networks for tasks such as environmental monitoring [24], surveillance [25], and disaster management [26]. Agents representing sensors collaborate to collect, process, and share data, enabling real-time monitoring and response.

- **E-Commerce**: In e-commerce, MASs are utilized for recommendation systems [27], dynamic pricing [28], supply chain management [29], and auction platforms [30]. Agents representing customers, sellers, and intermediaries interact to personalize recommendations, optimize pricing strategies, and streamline logistics.

- **Social Networks**: MAS-based approaches are employed in social network analysis, sentiment analysis [31], community detection [32], and influence propagation [33]. Agents model user interactions, analyze content, and identify trends to understand social dynamics, detect anomalies, and predict user behavior.

MAS provides a framework for modeling, analyzing, and designing complex systems comprised of multiple interacting entities, enabling the study and development of intelligent, adaptive, and scalable distributed systems.

2.1.2 Swarm Robotics Systems

Swarm robotics [8] is a rapidly growing field that has gained the attention of researchers and engineers from various areas. Swarm Robotics Systems (SRSs) [9] draw inspiration from the collective behaviors observed in social insects. These systems involve the design and control of many simple robots that collaborate to accomplish collective tasks beyond the capabilities of a single robot. A robotic swarm functions in a distributed and self-organizing manner, which means that there is no central robot directing the others, and the robots are unaware of global information. SR systems are designed to exhibit three properties: fault tolerance, scalability, and flexibility.

- **Scalability**: Scalability refers to the ability of a system to maintain its performance and efficiency as the number of components or size of the problem it is addressing increases. In swarm robotic systems, scalability is essential, as it enables the system to adapt to varying problem sizes and complexities. Swarm robotic systems exhibit excellent scalability due to their decentralized control and local interactions. The absence of a central controller reduces the communication overhead and computational complexity, allowing the system to grow without significantly impacting its performance. The local interactions between robots and their environment enable them to adapt their behavior according to the task requirements and the size of the swarm. This ability to self-organize and adapt ensures that swarm robotic systems can efficiently tackle problems of different scales, from small-scale missions with a few robots to large-scale operations involving hundreds or thousands of robots.

- **Flexibility**: Flexibility refers to the ability of a system to adapt its behavior to handle different tasks or changing environmental conditions. In swarm robotic systems, flexibility is crucial for ensuring that the swarm can efficiently accomplish a variety of tasks in diverse and dynamic environments. The flexibility of swarm robotic systems arises from their self-organization and adaptability, which enable them to adjust their behavior in response to changes in the environment or task requirements. Individual robots in a swarm can autonomously adjust their actions based on local information and interactions, allowing the swarm to adapt its global behavior to the prevailing conditions. Moreover, the decentralized control in swarm robotic systems supports the development of modular and reusable algorithms that can be easily adapted to different tasks, further enhancing their flexibility.

- **Fault tolerance**: Fault tolerance refers to the ability of a system to continue functioning effectively despite the failure or malfunction of one or more of its components. In the context of swarm robotic systems, fault tolerance is a critical property, as it ensures that the swarm can continue performing its tasks even if individual robots encounter issues or fail completely. The decentralized nature of swarm robotic systems contributes significantly to their fault tolerance. Since there is no central controller, the failure of a single robot does not compromise the entire system. Instead, the remaining robots in the swarm can adapt their behavior and reorganize themselves to compensate for the loss, thereby maintaining the overall performance of the swarm. Additionally, the redundancy in swarm robotic systems, where multiple robots can perform the same task, further enhances their fault tolerance by ensuring that no single point of failure exists.

These characteristics allow SRSs to exhibit Swarm Intelligence (SI) [34]. SI refers to the collective behaviors of decentralized self-organizing systems that result from lo-

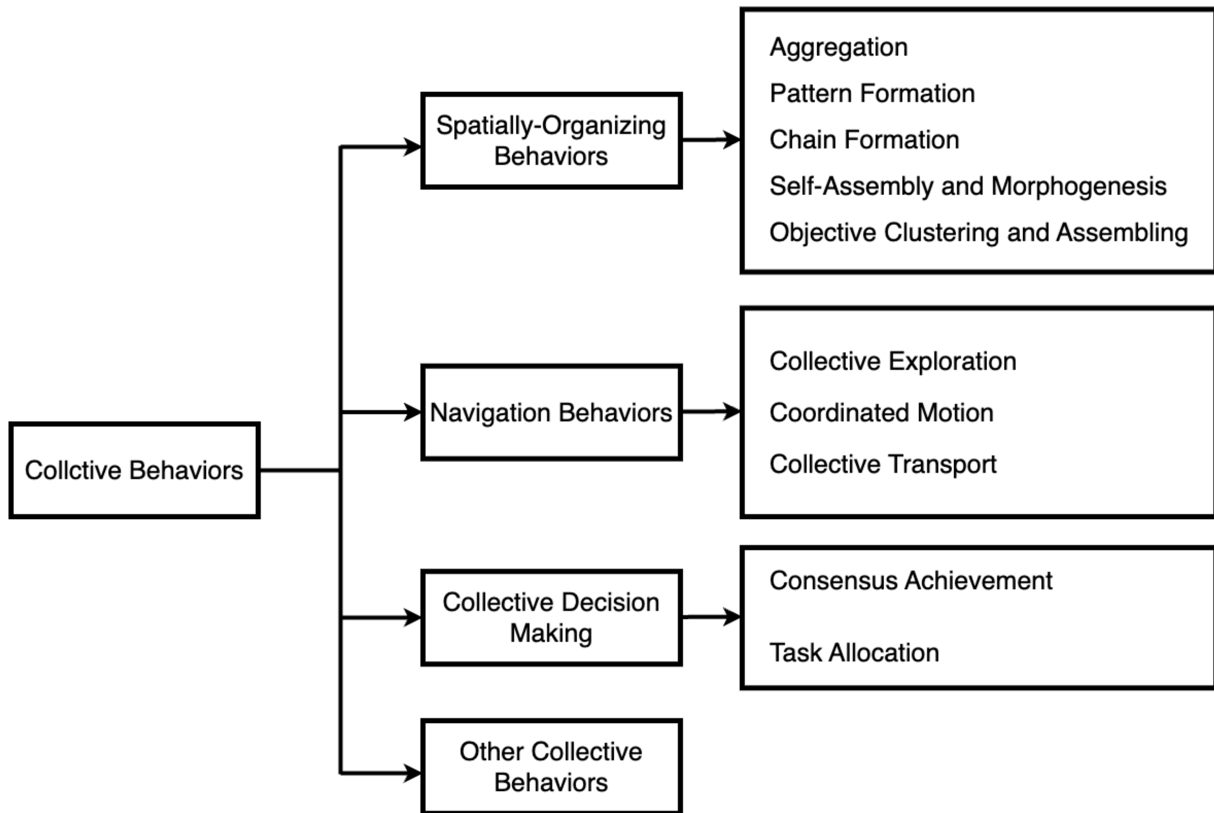


Figure 2.1: Collective behaviors in swarm robotics systems

cal interactions between individual agents and their environment. This behavior can be observed in various biological systems such as ant colonies, bird flocks, and fish schools, and has inspired the development of swarm robotic systems. Studying and understanding collective behaviors is essential for designing and implementing efficient and reliable algorithms for swarm robotic systems.

There are several types of collective behaviors observed in swarm robotic systems, many of which are inspired by biological systems such as ant colonies, bird flocks, and fish schools, as shown in Fig. 2.1. Here we give a brief review of some expected behaviors as follows:

- The coordination of robots in physical space to achieve specific tasks or objectives, known as spatially-organizing behaviors, holds significant potential in various applications, including environmental monitoring, construction, and logistics. For instance, robots can facilitate efficient transportation of goods in a warehouse by coordinating their movements to avoid collisions and optimize their paths [35].

- Navigation behaviors, which involve coordinated and efficient movement of robots in their environment, play a vital role in numerous swarm robotic systems applications.

For example, robots can aid in searching for survivors in unfamiliar zones [36], surveillance of dynamic areas [37], and collective transport of heavy objects [38].

- Collective decision-making is a fundamental feature of swarm robotic systems that enables groups of robots to make decisions collaboratively and achieve common objectives, such as consensus achievement and task allocation [39].

2.1.3 Design Methods of MASs and SRSs

Designing controllers for MASs and SRSs involves developing algorithms and protocols that govern the behavior of individual agents and their interactions with each other and the environment [10].

Rule-Based Systems

Rule-based systems use predefined rules to dictate agents' behavior. These rules are typically derived from domain-specific knowledge and encoded into structures such as finite state machines (FSM) [40], behavior trees [41], or production systems [42]. These rules determine how agents respond to various stimuli or changes in their environment, facilitating straightforward decision-making processes. In FSM, an agent transitions between states based on predefined conditions or events. Each state represents a specific behavior or action. For instance, a robot navigating a maze might use FSM where states include 'Move Forward', 'Turn Left', 'Turn Right', and 'Stop'. Transitions between these states are triggered by sensory inputs, such as detecting a wall or an open path.

Rule-based systems are relatively easy to design and implement. Their clear and explicit nature allows for straightforward debugging and maintenance. Given their simplicity, rule-based systems can be quickly implemented without the need for complex algorithms or extensive computational resources. They are suitable for applications where the environment and requirements are well-understood and relatively static.

However, rule-based systems struggle to adapt to dynamic or unforeseen environmental changes. Since the rules are predefined, any situation not accounted for by the existing rules can lead to suboptimal or erroneous behaviors. Moreover, as the complexity of the environment or task increases, the number of rules and their interactions can grow exponentially, making the system cumbersome and difficult to manage. This complexity can also lead to difficulties in maintaining and updating the system as new rules are added.

Learning-Based Methods

Learning-based methods involve applying machine learning techniques to enable agents to learn and adapt their behaviors through interactions with the environment and other agents [6]. These methods allow agents to improve performance over time without being explicitly programmed for every possible scenario and to identify optimal or near-optimal solutions for complex problems that are difficult to address using manual design approaches. Some popular learning-based methods are:

- **Evolutionary Algorithms (EAs)**: EAs are a class of optimization algorithms inspired by the principles of natural evolution, such as selection, mutation, and crossover. In the context of MASs and SRSs, EAs evolve and optimize the behaviors, strategies, and parameters of individual agents or the entire system. These algorithms iteratively improve a population of candidate solutions to find optimal or near-optimal solutions for complex problems. However, EAs can be computationally expensive and may require many generations to converge to optimal solutions.
- **Reinforcement Learning (RL)**: RL is a type of machine learning in which agents learn to make decisions by performing actions in an environment to maximize cumulative rewards. In multi-agent and swarm robotics systems, RL involves multiple agents learning and adapting their strategies through interactions with the environment and other agents. Moreover, deep reinforcement learning (DRL) combines RL with deep learning to handle high-dimensional states and action spaces.

2.2 Learning-Based Methods

2.2.1 Deep Learning

Deep learning, a subfield of machine learning, has become an essential aspect of artificial intelligence in recent years. It is primarily concerned with algorithms that enable computers to learn from and make predictions or decisions based on data through the use of artificial neural networks (ANNs). ANNs are computational models inspired by the human brain's structure and functionality, comprising interconnected nodes or neurons that process and transmit information.

The early history of deep learning can be traced back to the 1940s, with the introduction of the McCulloch-Pitts neuron model, a simplified representation of a biological neuron. This model laid the groundwork for the development of the perceptron in the late 1950s by Frank Rosenblatt. The perceptron, a single-layer neural network, was the first algorithm designed to classify linearly separable patterns. However, the perceptron's

limitations in solving problems that were not linearly separable led to a decline in interest in neural networks during the 1960s and 1970s.

The resurgence of interest in neural networks began in the 1980s, with the introduction of the backpropagation algorithm by Rumelhart, Hinton, and Williams. The backpropagation algorithm allowed for the efficient training of multi-layer neural networks by minimizing the error between predicted and actual outputs through gradient descent. This discovery led to the development of more complex neural network architectures and various learning techniques.

Deep learning gained significant momentum in the 2000s, driven by the availability of larger datasets, advances in computational power, and improvements in training algorithms. In 2006, Geoffrey Hinton [43] and his colleagues introduced the concept of deep belief networks (DBNs), which demonstrated the potential for training deep architectures by utilizing unsupervised pre-training followed by supervised fine-tuning. This breakthrough marked the beginning of the deep learning revolution.

In the following years, several specialized neural network architectures were developed to address specific tasks. Convolutional Neural Networks (CNNs), first introduced by Yann LeCun in the 1990s [44], became the standard for image recognition tasks due to their ability to exploit the spatial structure of images through the use of convolution and pooling operations. Recurrent Neural Networks (RNNs), designed to handle sequence data, were enhanced with Long Short-Term Memory (LSTM) cells to alleviate the vanishing gradient problem, which limited the ability to learn long-range dependencies in sequences.

More recently, the Transformer architecture, introduced by Vaswani et al. in 2017 [45], has revolutionized natural language processing (NLP) by employing a self-attention mechanism to capture complex dependencies in textual data. Transformers have become the foundation for state-of-the-art NLP models such as BERT[46], GPT[47], and T5[48], which have achieved unprecedented performance in a wide range of NLP tasks.

Deep learning has made significant strides in a variety of application domains, including computer vision, natural language processing, speech recognition, and reinforcement learning. The ability of deep learning models to automatically learn hierarchical representations of data has led to breakthroughs in numerous fields, such as autonomous vehicles, medical diagnosis, finance, and many more.

Despite its remarkable achievements, deep learning still faces several challenges, including interpretability, robustness against adversarial attacks, data privacy, and resource efficiency. Addressing these challenges is crucial for the continued growth and adoption of deep learning in real-world applications.

Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computing systems inspired by the biological neural networks that constitute the human brain. They are designed to recognize patterns and perform complex tasks, such as image and speech recognition, natural language processing, and more. An ANN is composed of interconnected nodes or neurons organized into layers: input, hidden, and output layers. Each connection between neurons has an associated weight, which determines the strength of the connection.

A neuron receives input from other neurons or external data, processes it, and produces an output. Mathematically, the output of a neuron is the result of applying an activation function to a weighted sum of its inputs, as shown in the following equation:

$$y_i = f\left(\sum_{j=1}^{n} w_{ij} x_j + b_i\right), \tag{2.1}$$

where y_i is the output of the i-th neuron, x_j is the input from the j-th neuron, w_{ij} is the weight of the connection between the i-th and the j-th neurons, b_i is the bias term for the i-th neuron, and f is the activation function.

Common activation functions include the sigmoid function, the hyperbolic tangent (tanh) function, and the Rectified Linear Unit (ReLU) function. The sigmoid function is given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2.2}$$

the hyperbolic tangent function by:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \tag{2.3}$$

and the Rectified Linear Unit function by:

$$\mathrm{ReLU}(x) = \max(0, x). \tag{2.4}$$

The process of computing the output of an ANN given its input is known as feedforward. During feedforward, the input data is passed through the input layer, then successively through the hidden layers, and finally through the output layer, producing the final output. The activation function is applied at each neuron in the hidden and output layers.

Training an ANN involves adjusting the weights and biases to minimize a loss function that measures the difference between the predicted output and the actual output. This is done using the backpropagation algorithm, which calculates the gradient of the loss function with respect to each weight and bias by applying the chain rule of calculus. The

weights and biases are then updated using an optimization algorithm, such as gradient descent, to minimize the loss function.

The gradient of the loss function with respect to the output of the i-th neuron in the output layer, $\frac{\partial L}{\partial y_i}$, can be computed directly from the loss function, which depends on the problem at hand. For example, for a regression problem with mean squared error as the loss function, the gradient is:

$$\frac{\partial L}{\partial y_i} = 2(y_i - t_i), \tag{2.5}$$

where t_i is the target output of the i-th neuron.

The gradient of the loss function with respect to the output of a neuron in a hidden layer, $\frac{\partial L}{\partial h_i}$, can be computed using the chain rule:

$$\frac{\partial L}{\partial h_i} = \sum_{k=1}^{m} \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial h_i}, \tag{2.6}$$

where h_k is the output of the k-th neuron in the next layer, and m is the total number of neurons in the next layer.

The gradient of the loss function with respect to the weights and biases can be computed similarly:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial w_{ij}}, \tag{2.7}$$

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial b_i}. \tag{2.8}$$

Once the gradients have been computed, the weights and biases can be updated using an optimization algorithm, such as gradient descent. The basic gradient descent update rule is:

$$w_{ij} = w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}}, \tag{2.9}$$

$$b_i = b_i - \alpha \frac{\partial L}{\partial b_i}, \tag{2.10}$$

where α is the learning rate, a hyperparameter that controls the step size of the weight and bias updates.

In summary, artificial neural networks are powerful models that can learn to represent complex functions by adjusting their weights and biases during training. The backpropagation algorithm is a key technique for computing the gradients of the loss function with respect to the weights and biases, enabling the optimization of the network using gradient descent or other optimization algorithms.

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for processing grid-like data, such as images, where local spatial dependencies are crucial. CNNs have been proven to be highly effective in various computer vision tasks, including image classification, object detection, and semantic segmentation. The main building blocks of CNNs are convolutional layers and pooling layers, which help to learn and encode local features while reducing the dimensionality of the input.

Convolutional Layer: The primary function of a convolutional layer is to learn and capture local patterns in the input data by sliding a set of learnable filters (or kernels) across the input. Each filter is responsible for detecting a specific pattern, such as an edge or a texture, in the input data. The output of the convolutional layer is a feature map, which is generated by computing the dot product between the filter and the local input region (patch) at each spatial location. Mathematically, the convolution operation can be represented as:

$$F_{out}(x,y) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} F_{in}(x+m, y+n) K(m,n) \tag{2.11}$$

where F_{out} is the output feature map, F_{in} is the input feature map, K is the filter, and k is the size of the filter.

In practice, convolutional layers often use multiple filters, which are applied to the input simultaneously, generating multiple feature maps. Furthermore, to introduce non-linearity in the model, an activation function, such as the Rectified Linear Unit (ReLU), is applied element-wise to the output feature maps.

Pooling Layer: The primary goal of a pooling layer is to reduce the spatial dimensions of the feature maps while retaining important information. Pooling layers help to make the learned features more robust to small translations and distortions in the input data. Two commonly used pooling operations are max pooling and average pooling. Max pooling selects the maximum value within a local input region, while average pooling computes the average value. The pooling operation can be defined as:

$$P_{out}(x,y) = \text{pool}\left(\{F_{\text{in}}\,(m,n) \mid m = x \cdot s_x, \ldots, x \cdot s_x + p_x - 1, n = y \cdot s_y, \ldots, y \cdot s_y + p_y - 1\}\right) \tag{2.12}$$

where P_{out} is the output after pooling, F_{in} is the input feature map, *pool* is the pooling function (e.g., max or average), s_x and s_y are the strides in the x and y directions, and p_x and p_y are the pooling sizes in the x and y directions.

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed to handle sequential data. In contrast to feedforward neural networks, RNNs maintain an internal hidden state that allows them to capture temporal dependencies in the input data. RNNs have shown great success in a variety of tasks, such as natural language processing, time series prediction, and speech recognition.

One of the main challenges in training RNNs is the vanishing and exploding gradient problem. This issue arises when gradients become too small or too large, making it difficult for the network to learn long-range dependencies. To address this problem, two types of gating mechanisms were introduced: Long Short-Term Memory (LSTM) cells [49] and Gated Recurrent Unit (GRU) cells [50].

LSTM cells were proposed by Hochreiter and Schmidhuber in 1997 [49] to overcome the vanishing gradient problem. An LSTM cell is composed of four main components: an input gate, a forget gate, an output gate, and a cell state. The cell state is responsible for maintaining long-range dependencies, while the gates control the flow of information in and out of the cell. The equations governing an LSTM cell are:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{2.13}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{2.14}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{2.15}$$

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{2.16}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{2.17}$$

$$h_t = o_t \odot \tanh(C_t) \tag{2.18}$$

where f_t, i_t, and o_t are the forget, input, and output gates, respectively, x_t is the input at time step t, h_t is the hidden state at time step t, C_t is the cell state at time step t, and W, U, and b are the weight matrices and bias vectors for each gate. The following figure shows the architecture of LSTM:

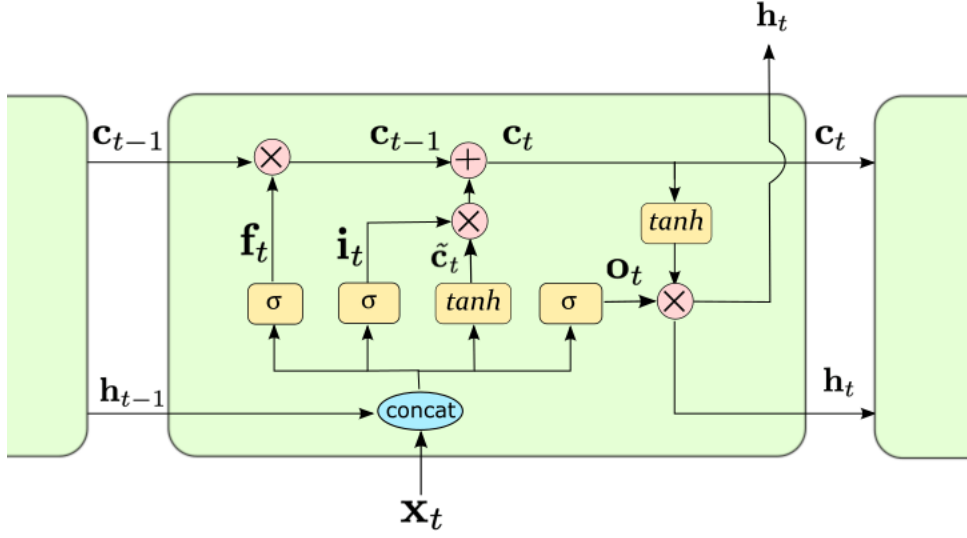


Figure 2.2: A schematic representation of an LSTM cell.

GRU cells were introduced by Cho [50] as a simplified alternative to LSTM cells. GRUs have two gates, a reset gate and an update gate, which control the flow of information in and out of the cell. The equations governing a GRU cell are:

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{2.19}$$

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{2.20}$$

$$\tilde{h}t = \tanh(W_h x_t + U_h(r_t \odot ht - 1) + b_h) \tag{2.21}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{2.22}$$

where r_t and z_t are the reset and update gates, respectively, and $\tilde{h}t$ is the candidate activation. The final hidden state h_t is a combination of the previous hidden state $ht - 1$ and the candidate activation $\tilde{h}_t$, controlled by the update gate z_t. Fig. 2.3 shows the architecture of GRU.

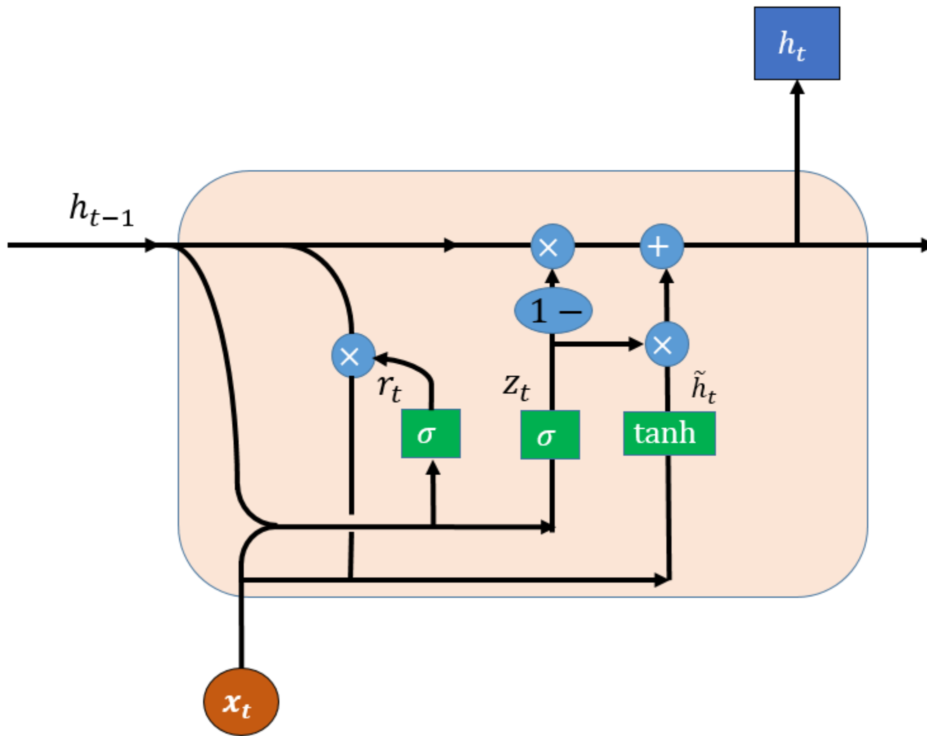


Figure 2.3: A schematic representation of a GRU cell.

LSTM and GRU cells have shown to be effective in capturing long-term dependencies in sequential data and have become a popular choice in various applications, such as machine translation and speech recognition.

RNNs have shown remarkable performance in various tasks involving sequential data, such as natural language processing, time series prediction, and speech recognition.

In natural language processing, RNNs have been used for various tasks such as language modeling, sentiment analysis, and machine translation. For example, in machine translation, an RNN can be trained to take a sentence in one language and output a corresponding sentence in another language.

In time series prediction, RNNs can be used to predict future values of a time series based on its past values. This is useful in applications such as stock price prediction and weather forecasting.

In speech recognition, RNNs have been used to recognize speech and convert it into text. RNNs have also been used in speech synthesis, where they can be used to generate realistic-sounding speech from text.

2.2.2 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning that deals with learning optimal decision-making strategies through interaction with an environment [51]. It is

based on the concept of learning from trial and error, wherein an agent takes actions in an environment to achieve a specific goal. The agent learns from the feedback received in the form of rewards or penalties, which helps it develop a policy for selecting the best action in a given state. The primary objective of RL is to maximize the cumulative reward received over time. Few recent studies have attempted to use deep reinforcement learning or novel reinforcement learning algorithms to design a controller for a swarm robotic systems [52][53][54].

The traditional reinforcement learning algorithms, such as Q-learning and SARSA, have proven to be effective in solving relatively small-scale problems. However, when faced with high-dimensional state and action spaces, these methods often face difficulties due to the curse of dimensionality. This limitation led researchers to explore the potential of combining RL with deep learning techniques, giving birth to the field of deep reinforcement learning.

Deep learning, a subfield of machine learning, has been incredibly successful in recent years due to its capability to learn abstract and hierarchical representations from large amounts of data. Deep learning techniques mainly involve the use of artificial neural networks with multiple layers, allowing them to learn complex patterns and representations. Some popular deep learning architectures include Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Long Short-Term Memory (LSTM) networks.

The integration of deep learning techniques into reinforcement learning allows the learning agent to extract high-level features from raw input data, such as images or text, and use these features to make better decisions. This ability to learn from raw data is particularly advantageous when dealing with complex, high-dimensional environments. The pioneering work of DeepMind on Deep Q-Networks (DQN) demonstrated the immense potential of DRL, as their DQN agent managed to achieve human-level performance on a variety of Atari games using only raw pixel data as input.

Since the advent of DQN, researchers have developed various DRL algorithms, addressing different aspects of RL, such as value-based, policy-based, and model-based methods. These algorithms have been further enhanced by incorporating advanced exploration-exploitation strategies, transfer learning, and multi-task learning approaches, leading to improved performance and generalization across diverse tasks and domains.

Despite the remarkable success of deep reinforcement learning, several challenges remain to be addressed. These challenges include interpretability and explainability, safety and robustness, scalability and generalization, and ethical considerations associated with the deployment of DRL systems in real-world applications.

Markov Decision Processes (MDPs)

A Markov Decision Process (MDP) is a mathematical model used to describe decision-making problems in reinforcement learning. It provides a framework for modeling the agent-environment interaction. An MDP can be defined as a tuple (S, A, P, R, γ), where:

- S is a finite set of states in the environment.
- A is a finite set of actions that the agent can perform.
- P is the state transition probability function, $P(s'|s, a)$, representing the probability of transitioning from state s to state s' when taking action a.
- R is the reward function, $R(s, a, s')$, denoting the immediate reward received after transitioning from state s to state s' by taking action a.
- $\gamma \in [0, 1]$ is the discount factor, which determines the importance of future rewards.

MDPs assume that the environment is fully observable and satisfies the Markov property, meaning the current state encapsulates all relevant information, and the future state is conditionally independent of the past states given the present state. The overview of MDP is shown in Fig. 2.4.

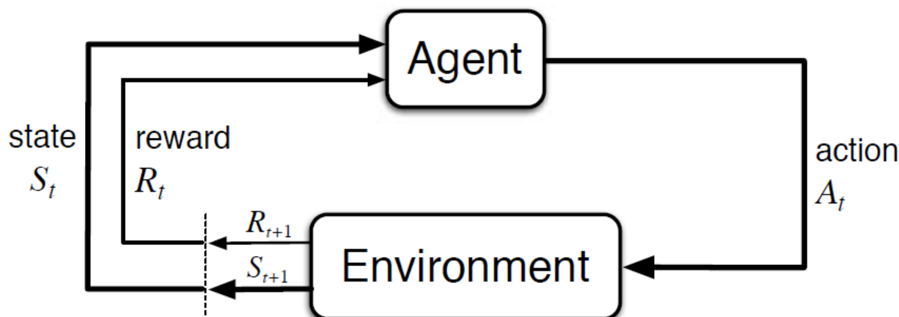


Figure 2.4: Overview of MDP.

Bellman Equations

Bellman equations are recursive relationships that describe the structure of optimal policies and value functions in MDPs. They provide a foundation for reinforcement learning algorithms. There are two primary types of Bellman equations:

1. **Bellman Expectation Equation**: This equation is used for evaluating a given policy π. The state-value function $V^{\pi}(s)$ and action-value function $Q^{\pi}(s, a)$ under policy π are defined as:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')] \tag{2.23}$$

$$Q^{\pi}(s,a) = \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma \sum_{a' \in A} \pi(a'|s') Q^{\pi}(s',a')] \tag{2.24}$$

2. **Bellman Optimality Equation**: This equation is used for finding the optimal policy π^*. The optimal state-value function $V^(s)$ and optimal action-value function $Q^*(s,a)$ are defined as:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')] \tag{2.25}$$

$$Q^(s,a) = \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma \max_{a' \in A} Q^(s',a')] \tag{2.26}$$

These Bellman equations establish the relationships between the value functions of the current state and its successor states, enabling the development of RL algorithms that leverage these properties to learn optimal policies.

Policy and Value Functions

A policy, denoted as π, is a mapping from states to actions that defines the agent's behavior. In other words, a policy describes the probability of taking a specific action a in a given state s, represented as $\pi(a|s)$. The goal of reinforcement learning is to find the optimal policy π^*, which maximizes the expected cumulative reward.

Value functions are essential components in reinforcement learning, as they estimate the expected cumulative rewards under a given policy. There are two types of value functions:

1. **State-Value Function**: Denoted as $V^{\pi}(s)$, the state-value function represents the expected cumulative reward when starting from state s and following policy π. The optimal state-value function $V^*(s)$ is the maximum value achievable by any policy in state s.

2. **Action-Value Function**: Denoted as $Q^{\pi}(s,a)$, the action-value function represents the expected cumulative reward when starting from state s, taking action a, and then following policy π. The optimal action-value function $Q^*(s,a)$ is the maximum value achievable by any policy after taking action a in state s.

Temporal-Difference (TD) Learning

Temporal-Difference (TD) learning is a central concept in reinforcement learning that combines the ideas of dynamic programming and Monte Carlo methods. It allows agents to learn directly from raw experiences without requiring a model of the environment. TD learning methods update value function estimates based on the difference between the current estimate and a new estimate obtained from the latest experience, known as the TD error.

The generic TD update rule for state-value functions can be expressed as:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.27}$$

where α is the learning rate, r_{t+1} is the reward received at time $t+1$, and γ is the discount factor.

Q-Learning and SARSA

Q-learning and SARSA are popular model-free RL algorithms that learn the action-value function $Q(s, a)$.

Q-learning is an off-policy algorithm that learns the optimal action-value function by iteratively updating its estimates based on the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{2.28}$$

Q-learning learns the optimal policy, even when following an exploratory or suboptimal behavior policy.

SARSA (State-Action-Reward-State-Action) is an on-policy algorithm that learns the action-value function for the current behavior policy. SARSA updates its action-value estimates based on the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.29}$$

Unlike Q-learning, SARSA takes into account the agent's current policy when updating the action-value function, which makes it sensitive to the agent's exploration strategy. The agent learns to act optimally with respect to its current policy, rather than learning the optimal policy directly.

In summary, this section has provided an overview of the fundamental concepts and algorithms in reinforcement learning, including Markov Decision Processes, Bellman equations, policy and value functions, Temporal-Difference learning, and popular model-free RL algorithms like Q-learning and SARSA. These foundational elements serve as a basis for understanding deep reinforcement learning, which combines reinforcement learning with deep learning techniques to tackle complex decision-making problems.

Value-Based Deep Reinforcement Learning

In value-based deep reinforcement learning (Deep RL) algorithms, the core idea is to approximate the optimal action-value function using deep neural networks. By combining deep learning techniques with reinforcement learning, it becomes possible to learn complex representations and solve challenging problems with high-dimensional state spaces. In this section, we discuss two primary algorithms in value-based deep RL: Deep Q-Networks (DQN) and its extensions, Double DQN and Dueling DQN.

The Deep Q-Network (DQN) algorithm, introduced by Mnih et al. [55], revolutionized the field of reinforcement learning by successfully training deep neural networks to learn the optimal action-value function in high-dimensional environments, such as Atari games. DQN builds upon the traditional Q-learning algorithm and uses a deep neural network as a function approximator to estimate the action-value function $Q(s,a)$. The objective is to minimize the following loss function:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta^{-}\right) - Q(s,a;\theta)\right)^2\right] \tag{2.30}$$

Here, D denotes the experience replay buffer, which stores tuples of the form (s, a, r, s'), representing state, action, reward, and next state. $U(D)$ is a uniform random sampling from the buffer. The target network, parameterized by θ^{-}, is used to compute the target values, while the online network, parameterized by θ, is updated through backpropagation. The target network's parameters are periodically updated with the online network's parameters, ensuring stable learning.

DQN introduces two key techniques to stabilize training and improve learning efficiency:

- **Experience Replay:** DQN uses a replay buffer to store the agent's experiences, which are sampled uniformly at random during training. This process helps break the correlations between consecutive samples and smooths the learning process.

- **Target Network:** DQN employs a separate target network with a fixed set of parameters to compute the target Q-values for the loss function. The target network's parameters are updated periodically, reducing the oscillations in the learning process and stabilizing training.

The Double DQN (DDQN) algorithm, proposed by van Hasselt et al. [56], is an extension of the DQN algorithm that addresses the overestimation bias in the Q-value estimation. The bias is introduced because the max operator in the target computation uses the same values for both the selection and evaluation of actions. DDQN mitigates this issue by using the online network to select the best action while using the target network to evaluate the action's value:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma Q\left(s', \arg\max_{a'} Q\left(s', a'; \theta\right); \theta^{-}\right) - Q(s,a;\theta)\right)^2\right] \tag{2.31}$$

By decoupling action selection from action evaluation, Double DQN significantly reduces the overestimation bias, leading to more accurate value estimation and better performance.

The Dueling DQN architecture, introduced by Wang et al. [57], is another extension of the DQN algorithm that improves the efficiency of the value estimation process. The key insight of the Dueling DQN is to separate the estimation of the state value function $V(s)$ and the advantage function $A(s,a)$, allowing the network to learn which states are valuable without having to learn the effect of each action in each state. This separation is particularly beneficial in cases where the actions do not significantly affect the environment.

The Dueling DQN architecture consists of two streams of fully connected layers, which share a common convolutional feature extraction backbone. One stream computes the state value function, $V(s;\theta,\alpha)$, while the other computes the advantage function, $A(s,a;\theta,\beta)$. The two streams are then combined to obtain the action-value function:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\alpha) + \left(A(s,a;\theta,\beta) - \frac{1}{|A|}\sum_{a'} A\left(s, a'; \theta, \beta\right)\right) \tag{2.32}$$

The subtraction term inside the parentheses is a baseline that ensures the advantage function has a zero value when averaged over all actions, which allows the network to better distinguish the relative importance of each action. The Dueling DQN architecture can be combined with other DQN extensions, such as Double DQN, to further improve performance.

In summary, value-based deep RL algorithms, such as DQN and its extensions, have been instrumental in advancing the field of reinforcement learning. These algorithms have demonstrated impressive performance in various challenging tasks, including high-dimensional environments like Atari games. By leveraging deep neural networks to approximate the action-value function, DQN and its variants have successfully tackled problems previously thought to be intractable for traditional reinforcement learning methods.

Policy-Based Deep Reinforcement Learning

Policy-based methods directly learn a parameterized policy that can select actions without the need for an intermediate value function. In this section, we will discuss two popular policy-based deep reinforcement learning algorithms: the REINFORCE algorithm and Proximal Policy Optimization (PPO).

The REINFORCE algorithm, also known as the Monte Carlo Policy Gradient method, is a foundational algorithm in policy-based reinforcement learning. It was introduced by Williams in 1992 [58], and it is based on the idea of using the gradient of the expected reward to directly optimize the policy. The objective function in the REINFORCE algorithm is given by:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau)], \tag{2.33}$$

where θ represents the parameters of the policy, τ denotes a trajectory, and $R(\tau)$ is the total reward of the trajectory. The expectation is taken over the trajectories generated by the policy $p_\theta(\tau)$.

The gradient of the objective function with respect to the policy parameters can be written as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]. \tag{2.34}$$

The REINFORCE algorithm uses this gradient to update the policy parameters:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau), \tag{2.35}$$

where α is the learning rate.

However, the vanilla REINFORCE algorithm suffers from high variance in the gradient estimates, which can lead to unstable learning. A common approach to reduce variance is to subtract a baseline $b(s_t)$ from the total reward:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(R(\tau) - b(s_t)) \right]. \tag{2.36}$$

The baseline can be chosen as the state-value function $V(s_t)$, which can be learned separately by a value function approximator.

Proximal Policy Optimization (PPO) is an advanced policy optimization algorithm that was introduced by Schulman et al. in 2017 [59]. PPO addresses the challenges of sample inefficiency and sensitivity to hyperparameters in earlier policy gradient methods, such as the Trust Region Policy Optimization (TRPO) [60].

The central idea of PPO is to restrict the policy updates to a trust region around the current policy to ensure stability and avoid large, harmful updates. PPO achieves this by introducing a surrogate objective function, which is a clipped version of the objective function used in policy gradient methods. The surrogate objective function for PPO is defined as:

$$L(\theta) = \mathbb{E}_t \left[\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A^{\pi_{\theta_{\text{old}}}}(s_t, a_t), , \text{clip} \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_{\text{old}}}}(s_t, a_t) \right) \right], \tag{2.37}$$

where $\pi_\theta(a_t|s_t)$ is the probability of action a_t under the current policy, $\pi_{\theta_{\text{old}}}(a_t|s_t)$ is the probability of action a_t under the previous policy, $A^{\pi_{\theta_{\text{old}}}}(s_t, a_t)$ is the advantage function, and ϵ is a hyperparameter that determines the trust region size.

The PPO algorithm proceeds in the following steps:

1. Collect trajectories using the current policy $\pi_\theta(a_t|s_t)$.
2. Estimate the advantage function $A^{\pi_{\theta_{\text{old}}}}(s_t, a_t)$ for each state-action pair in the trajectories.
3. Update the policy parameters θ by maximizing the surrogate objective function $L(\theta)$ using gradient ascent.
4. Repeat steps 1-3 for a predefined number of iterations or until a stopping criterion is met.

PPO can be combined with different neural network architectures, such as feed-forward, convolutional, or recurrent networks, depending on the problem domain. It has demonstrated impressive results in a wide range of applications, including robotics, continuous control tasks, and game playing.

In summary, policy-based deep reinforcement learning methods, such as REINFORCE and PPO, offer an alternative approach to directly optimizing the policy without the need for an intermediate value function. They have been successful in various applications and provide a foundation for further developments in deep reinforcement learning.

2.2.3 Imitation learning

Unlike supervised learning, which relies on labeled data, RL relies on the design of a reward function. However, designing an appropriate reward function is challenging, especially in real-world scenarios where rewards are sparse or not readily available. This can lead to difficulties in training policies that meet actual needs. Imitation Learning (IL) [61] offers an alternative approach by allowing agents to learn directly from expert demonstrations without relying on the reward signal from the environment. IL uses expert state-action data to train policies that mimic expert behavior. At the moment, IL approaches used in academia can be broadly categorized into two main categories: behavior cloning and inverse reinforcement learning.

Behavior Cloning (BC)

Behavior cloning (BC) [62] is a machine learning method that involves training a model to mimic the behavior of an expert or a set of experts. The primary objective is to gather expert demonstrations into a dataset and then train a model to predict the expert's behavior based on the environment as it is at that moment. The training involves collecting a dataset of expert demonstrations of pairs of states and corresponding actions. A supervised learning model is then trained on the dataset to anticipate the expert's behaviors based on the present state. After training, the model can be used in the environment to emulate the expert's actions. BC can learn a decent strategy rapidly when there is substantial training data. In many real-world situations, BC can be used as a technique for policy pre-training because it is easy to implement. When the environment is too complex, BC enables the strategy to quickly reach higher levels by imitating the behavioral data of expert agents, rather than inefficiently interacting with the environment to explore actions.

However, if the training data is not comprehensive enough or if there are inaccuracies in the expert demonstrations, the learned policy may not generalize well to unseen states or situations. As a result, even small errors in the learned policy can accumulate over time, leading to compounding errors. These compounding errors can be particularly problematic in dynamic or complex environments where the agent needs to adapt and make decisions in novel situations. They can result in the agent deviating significantly from the expert's behavior and making suboptimal or unsafe decisions.

Inverse Reinforcement Learning (IRL)

Inverse Reinforcement Learning (IRL) [63] stands out as a unique approach in machine learning. Its primary aim is to deduce the underlying reward function that an agent optimizes through its behavior. Unlike traditional reinforcement learning, where the reward function is explicitly defined, IRL takes a different route. It seeks to infer this function from observed actions, thereby unveiling the agent's implicit goals or preferences. This process paves the way for the development of agents that can learn complex behaviors without the need for manual specification of a reward function, which is often a challenging task in real-world scenarios.

The core idea behind IRL is that an agent's behavior is guided by its goals or preferences, which can be inferred from its actions. In a typical IRL setup, the agent's actions are assumed to be driven by a policy that maximizes a specific reward function. By observing these actions, we can work backward to infer the reward function, providing insights into the agent's underlying motivations and objectives.

The training process of IRL can be described in the following three steps:

- **Data Collection**:Collecting state-action pairs from expert demonstrations involves recording the sequence (trajectory) of states and corresponding actions performed by experts while completing a task in the environment. For instance, in autonomous driving applications, this entails capturing data such as vehicle speed, steering wheel angle, braking, acceleration, and more, from the human driver's operation.

- **Reward Function Inference**: Use IRL algorithms to estimate reward functions that explain expert behavior. This step involves identifying a reward structure that, when maximized, results in expert-like actions. According to its method of solving the reward function, the IRL algorithm can be divided into methods based on maximum margin, such as apprentice learning [64], maximum margin planning methods [65], structured classification [66], neural inverse reinforcement learning [67], and methods based on probabilistic models, such as maximum entropy IRL [68], relative entropy IRL [69], deep inverse reinforcement learning [70].

- **Policy Learning**: Apply reinforcement learning techniques to learn a policy that maximizes the inferred reward function. The goal is to train the agent to perform the task consistently with the expert's strategy.

While IRL provides a powerful framework for understanding and replicating complex behaviors, it still has some limitations. IRL relies heavily on high-quality, representative expert demonstration data. Poor or limited data can lead to inaccurate reward inference and suboptimal policies. Additionally, reward function inference can be computationally intensive in complex environments with high-dimensional state and action spaces.

2.2.4 Evolutionary Robotics

Evolutionary robotics is an interdisciplinary field that integrates principles from robotics, artificial intelligence, and evolutionary biology to create and optimize control systems for autonomous robots. Since its inception in the early 1990s, evolutionary robotics has gained significant attention due to its potential for producing robust, adaptive, and scalable robotic systems [71][72][73].

In evolutionary robotics, an evolutionary algorithm is used to create and optimize controllers. Evolutionary algorithms are a family of optimization methods that mimic the process of natural evolution, such as genetic algorithms, genetic programming, and evolutionary strategies. The main components of an evolutionary algorithm are a population of candidate solutions (controllers), a fitness function to evaluate the quality of these solutions, and a set of evolutionary operators (e.g., mutation and crossover) to generate new solutions. The evolutionary process starts with an initial population of randomly generated controllers. The fitness of each controller is evaluated based on its performance

in the given task or environment. The best-performing controllers are then selected, and new controllers are generated using evolutionary operators. This process is repeated over multiple generations, leading to the evolution of controllers that can perform better in the given task or environment.

A controller is often represented as an artificial neural network (ANN) in evolutionary robotics. The weights of the ANN are optimized by an evolutionary algorithm. Several collective behaviors are developed using this approach, for example, aggregation [74][75], flocking [76], path formation [77][78], and cooperative transport [39][79][80].

Evolutionary Computation

Evolutionary computation (EC) is a branch of artificial intelligence that encompasses a family of optimization algorithms inspired by the process of natural evolution [81][82]. The fundamental principle behind EC is to apply Darwinian concepts of selection, variation, and inheritance to solve complex problems in various domains. Since their inception in the 1960s, evolutionary algorithms (EAs) have shown remarkable capability in solving complex optimization problems, particularly those with non-linear, multimodal, and high-dimensional characteristics.

The primary motivation for developing EAs stems from the limitations of traditional optimization techniques, such as gradient-based methods. These traditional methods are sensitive to the choice of initial solutions, prone to getting trapped in local optima, and often unable to handle discrete or combinatorial search spaces. In contrast, EAs are population-based methods that explore the search space by maintaining and evolving a set of candidate solutions. This global search approach allows EAs to escape local optima, adapt to changing environments, and provide robust optimization solutions.

The field of evolutionary computation has grown significantly over the past few decades, with numerous algorithms and variants being proposed, each with its own strengths and weaknesses.

Genetic Algorithms

Genetic algorithms (GAs) are one of the most widely studied and applied evolutionary algorithms [83][84]. They are inspired by the process of natural selection and aim to evolve a population of candidate solutions to optimize a given objective function. The main components of a GA are representation, fitness function, selection, crossover, and mutation. The process of GA is shown in Fig. 2.5.

- **Representation**: In GAs, candidate solutions, also known as individuals or chromosomes, are typically represented as fixed-length binary strings or real-valued vectors. Each element in the string or vector is referred to as a gene. The choice of

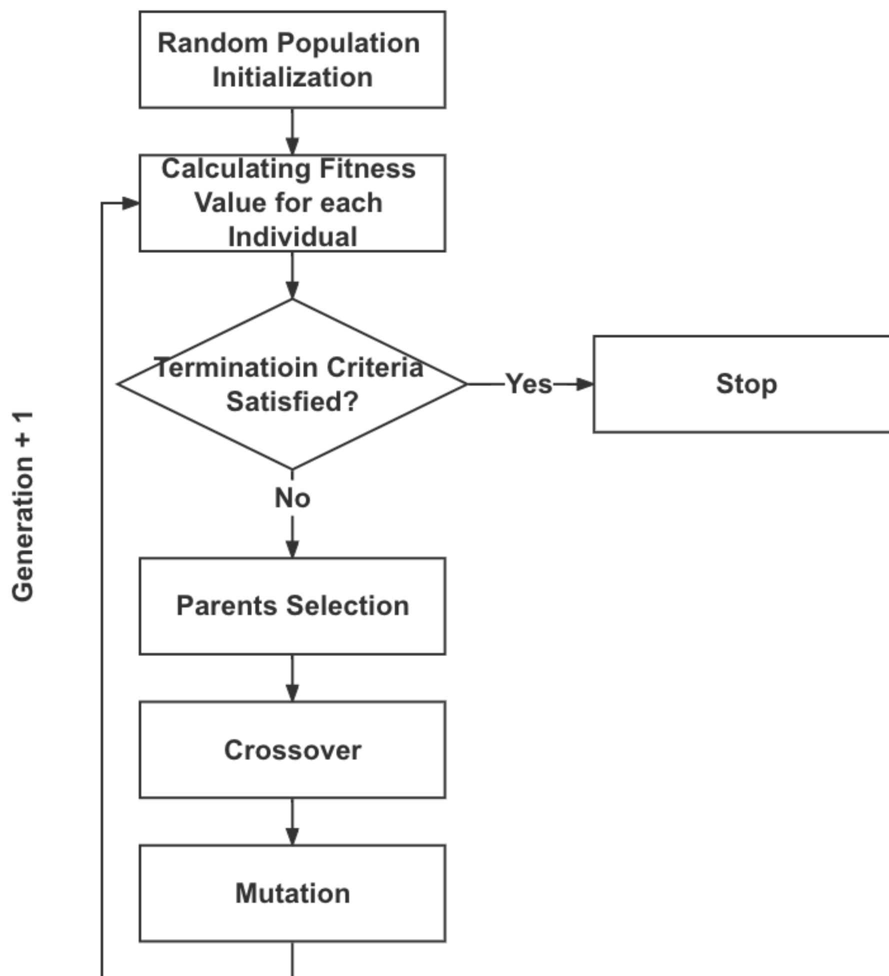


Figure 2.5: Overview of GA.

representation depends on the nature of the problem and can significantly impact the algorithm's performance.

- **Fitness Function**: The fitness function evaluates the quality of each candidate solution and assigns a fitness value, which reflects how well the solution satisfies the optimization objective. In most cases, the fitness function is derived from the problem-specific objective function.

- **Selection**: Selection is the process of choosing individuals from the current population to participate in the creation of the next generation. The selection process is biased towards individuals with higher fitness values, mimicking the concept of survival of the fittest. Common selection methods include roulette wheel selection, tournament selection, and rank-based selection.

- **Crossover**: Crossover, also known as recombination, is the process of combining

the genetic material of two parent individuals to produce one or more offspring. The goal of crossover is to promote the exploration of new regions in the search space by creating new candidate solutions. Common crossover operators include one-point crossover, multi-point crossover, and uniform crossover.

- **Mutation**: Mutation introduces small random perturbations into the offspring's genetic material, providing the necessary diversity and preventing premature convergence to local optima. Mutation operators depend on the representation of the candidate solutions. Examples of mutation operators include bit-flip mutation for binary strings and Gaussian mutation for real-valued vectors.

Genetic Programming

Genetic programming (GP) is an extension of genetic algorithms that evolves computer programs or symbolic expressions to solve a given problem [76][85]. In GP, candidate solutions are typically represented as tree structures, where the nodes represent functions or operations, and the leaves represent terminal symbols or input variables. The main components of a GP are similar to those of a GA but are adapted to handle the tree-based representation.

Evolutionary Strategies

Evolutionary strategies (ES) are a class of evolutionary algorithms primarily designed for continuous optimization problems [86][87]. They are characterized by the use of self-adaptive mutation parameters, which evolve alongside the candidate solutions. ES algorithms use a combination of mutation and recombination operators, and their main difference from GAs lies in the selection and adaptation mechanisms.

Chapter 3

Generating Collective Transport of a Swarm Robotic System by Deep Neuroevolution

3.1 Introduction

A Swarm Robotic System (SRS) is composed of relatively simple robots, which could exhibit behaviors based on the interactions among robots and between robots and the environment [88]. A swarm robotic system exhibits three properties; fault-tolerance for operating despite the loss of robots or disturbances in the environment, flexibility for generating modular solutions for different tasks, and scalability for operating with a wide range of group sizes.

The design methods of the swarm robotic systems are classified into behavior-based design methods and automatic design methods. Behavior-based design methods are guided only by the designer's intuition and experience in designing controllers to obtain the desired collective behavior. The automatic design methods transform the design problem into an optimization problem. Automatic design methods are divided into Reinforcement Learning (RL) and Evolutionary Robotics (ER). Typically, the controller of the swarm is represented by a neural network, and the parameters of the neural network are optimized using an evolutionary algorithm or a reinforcement learning algorithm.

The Deep Neural Network (DNN) has attracted attention in recent years. Especially, DNNs with convolutional layers have made remarkable progress in image recognition. Convolutional neural networks can extract features of images automatically.

With raw images, the developers of the controllers can reduce the efforts in designing the inputs of the neural networks. To use raw images, one of the methods is the Deep Reinforcement Learning (DRL) algorithm, which combines deep learning and reinforcement

learning. However, DRLs are very sensitive to the reward settings. Developers need to carefully design the reward and conduct experiments until the agent acts as expected.

Deep Neuroevolution(DNE) is a method to update the parameters of the deep neural networks with evolutionary algorithms[89]. Gradient-based algorithms, like DRL, might trap in local optima due to the multi-modal characteristic of the loss function. DNE can solve this problem by evaluating several individuals at the same time and choosing individuals with higher fitness scores. This chapter applies Deep Neuroevolution (DNE) and Deep Q-Learning (DQN) algorithms on collective transport tasks by a swarm robotic system with computer simulations.

The chapter is organized in the following way. Related works will be introduced in section 2. In section 3, we will explain the methods used in this chapter. In section 4, the details of the experiments will be introduced. We will show the results of the experiments in section 5. Section 6 will conclude this chapter.

3.2 Related Works

Most of the current literature on swarm robotic systems uses behavior-based design methods. Some minority research uses evolutionary robotics (ER) to design the SRS controller automatically. Over the past decade, most research in ER for swarm robotic systems has emphasized the use of neural networks. In the evolutionary robotics approach, the controller is conventionally represented by artificial neural networks, in which synaptic weights are optimized with an evolutionary algorithm. An evolutionary algorithm evaluates and optimizes controllers based on a predefined fitness function that indicates the performance of the robots. Evolutionary robotics approaches have succeeded in generating a wide range of robotic swarm behaviors, such as aggregation [75] [74], flocking [90], path formation [78], collective construction [91], and cooperative transport [79]. However, for most of them, raw camera images are not used due to the high dimensionality of the state space.

Several attempts have been made to combine deep neural networks with evolutionary algorithms. Existing literature [89] and [92] suggest that evolutionary algorithms are competitive approaches for training deep convolutional neural networks. However, none of them considered the multi-agent task, which is more challenging than the single-agent task due to the dynamics of the environment.

Another methodology for designing the robotic swarm controller is deep reinforcement learning algorithms. Deep reinforcement learning (DRL) combines deep convolutional neural networks with reinforcement learning algorithms. DRL algorithms have achieved many milestones on Go [93], Atari 2600 video games [55], StarCraft II [94], etc. However, DRL is sensitive to the design of the reward system. Especially for a swarm robotic

system, developers need to carefully design the reward settings to generate collective behavior, which requires lots of time.

3.3 Methods and Experiment Settings

This section introduces the details of the method and the experiments, including the specifications of the agents and the experiment environments. In this chapter, we use two different reward/fitness settings and compare the performance of DNE and DQN to test the robustness of the algorithms.

3.3.1 Deep Neuroevolution

Deep Neuroevolution (DNE) is a learning method that combines evolutionary algorithm and Deep Convolutional Neural Network [89]. Felipe et al. [89] tested DNE on Atari 2600 games with raw images and found that DNE could perform similarly or better on some games than DQN. In DNE, each generation has N individuals represented as a convolutional neural network whose parameters are θ. Each individual is evaluated with the fitness function $F(\theta_i)$ at every generation. Individuals will be ranked based on their fitness scores. Top T individuals will be chosen to become the next generation's parents. The child is reproduced by mutating a randomly chosen parent. A mutation operation is applied by adding Gaussian noise to the parameters of the neural network, as shown in Eq. 3.1.

$$\theta' = \theta + \sigma\epsilon \text{ where } \epsilon \sim N(0,1) \tag{3.1}$$

Where σ is a hyper-parameter determined by fine-tuning, the best individual from the last generation will be reserved for the next generation during training, called *elitism*. The crossover is not used in this study due to its instability. This process will be repeated until some termination conditions are satisfied. In this study, the experiment will be terminated when it reaches the maximum number of generations, or the fitness score can no longer improve.

3.3.2 Environment Settings

This study compares DNE and DQN in a collective transport task for a swarm robotic system. The robots need to collectively transport food scattered in the field back to the nest in the collective transport task. It is worth noting that the food is too heavy to be moved by one robot alone. So, the robots need to learn to move the food towards the nest cooperatively.

The experiments are conducted in computer simulations using the 3D physics engine called Unity3D. The DNE and DQN algorithms are implemented using Chainer, a deep learning framework for Python.

The experiment environment is a square arena, as shown in Fig. 3.1, which is surrounded by four walls of 50m in length. A 10m diameter nest area is located in the middle of the arena. There are four foods and 36 robots in the environment.

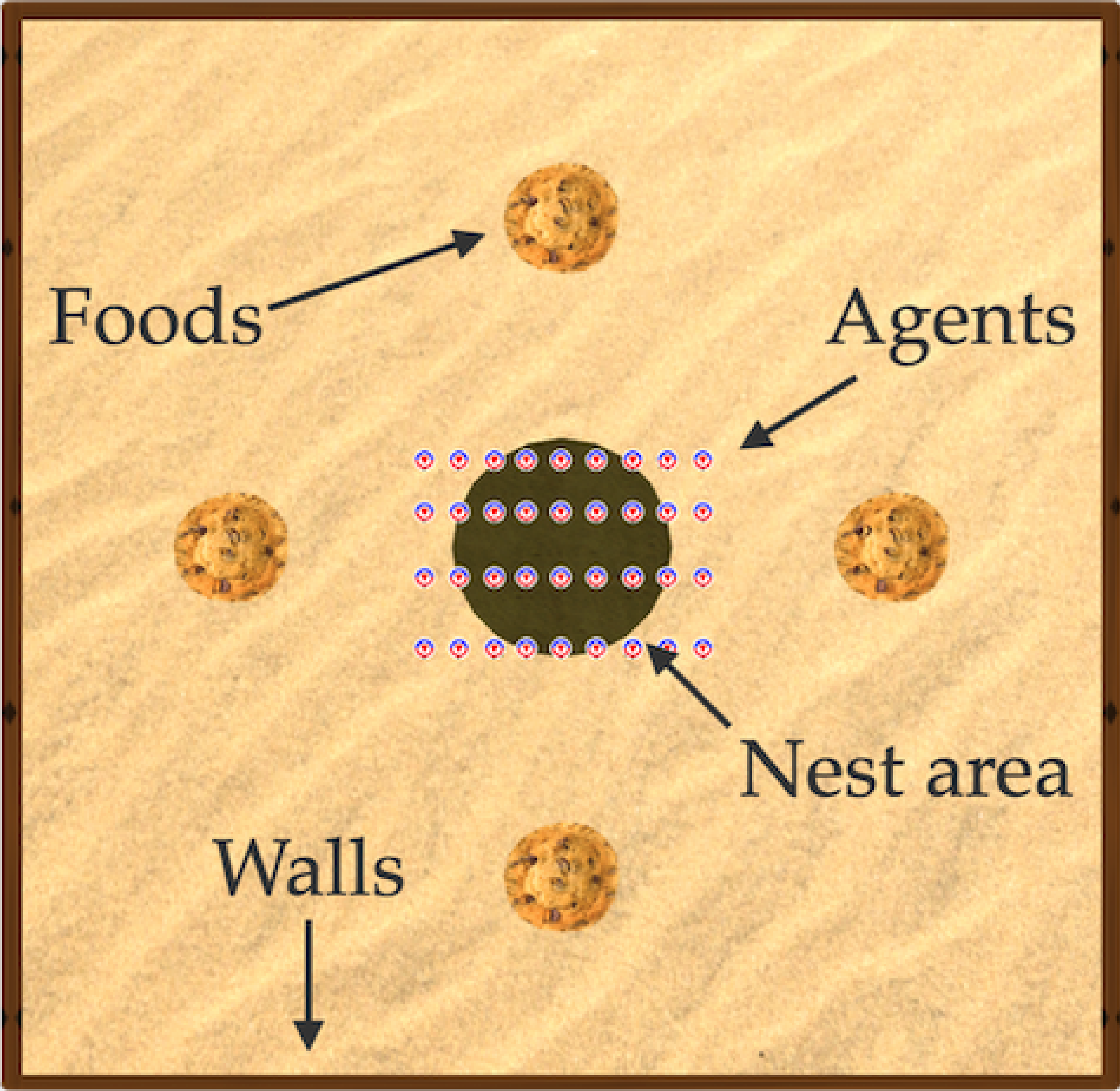


Figure 3.1: Top view of the environment 1

The design of the agent is shown in Fig. 3.2. Both the height and diameter of the agent are 1.0m. Each agent has two wheels. Each agent is equipped with eight infrared sensors, which can detect objects within 1 meter. We put an RGB camera mounted on

top of the agent. The resolution of the RGB camera is 128 × 128 pixels with 90° field of view. The agent has an electric compass that shows the direction of the agent. The agents can move forward and rotate left and right.

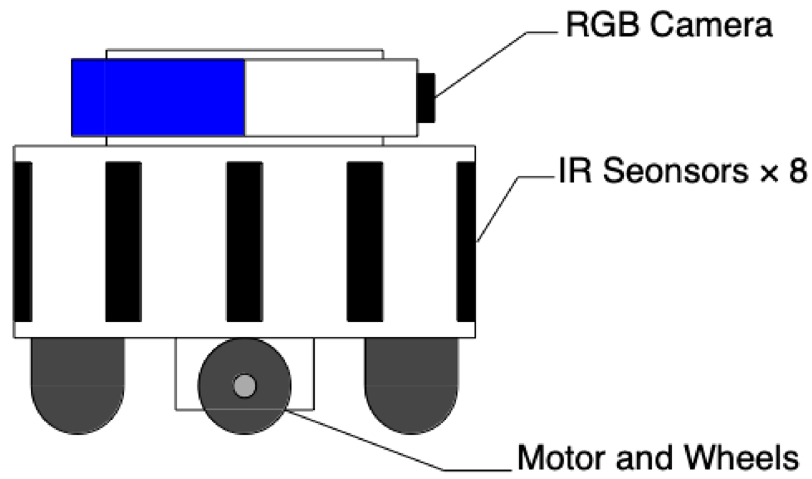


Figure 3.2: Settings of the agent

The food's diameter is set to 5.0m. To ensure that foods will not block the view of robots, we set the height of the food to 0.7m, which is slightly lower than the agent's height.

At the beginning of each episode, robots will be generated in the nest area, as shown in Fig. 3.1. Four foods will be placed around the nest.

Reward and fitness play a similar role in DQN and DNE. We use two kinds of reward and fitness settings to compare the performances of DNE and DQN. Agents will get rewards at each time step in DQN. In DNE, fitness is the sum of cumulative rewards from all robots during an episode.

3.3.3 Fitness/Reward Settings

To test the robustness of both algorithms, we design two different reward/fitness settings. The first reward settings are based on the distance between the foods and the nest. The details are shown in Table 3.1. The second reward setting is based on the image, as shown in Table 3.2.

Table 3.1: Distance-based rewards

Reward	Conditioin
0.2	if agent touches food
2	if agent pushes food to nest direction
-0.1	at each time step

Table 3.2: Image-based rewards

Reward	Conditioin
1.5	if foods pixels take more than 45% in the images
2.5	if foods pixels take more than 45% in the images and nest pixels in view

3.3.4 Network Structure

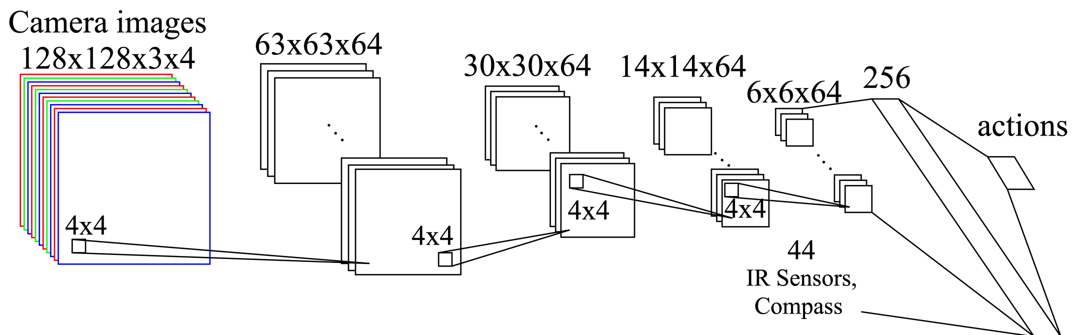


Figure 3.3: Structure of the neural network

The architecture of DNN is shown in Fig. 3.3. The network inputs are 128 $\times$ 128 sizes RGB images of four timesteps and sensor readings. The total number of nodes from the image input is 196608 nodes (128 $\times$ 128 $\times$ 3 RGB colors $\times$ 4 timesteps). The neural network contains four convolutional and two linear layers. The kernel size of the convolutional layers is 4 $\times$ 4, the stride is set to 2, and the number of channels is 64 for each layer. We use hyperbolic tangent as the activation function of the convolutional layers. After four convolutional layers, the outputs of the last convolution layer are concatenated with the other sensor values and fed into two fully connected layers. The sensor values are normalized into [0, 1]. The ReLU activation function is used in the fully-connected layers. The outputs of the neural network correspond to the Q values for each action.

3.3.5 Hyper-parameter Settings

In DRL, the mini-batch size during learning is 32, and RMSpropGraves is used as the optimizer. The robot selects its action based on the ϵ-greedy method. In the first episode, ϵ is set to 1, which means the robot selects the action randomly to collect the experience. After the first episode, the action is selected with $\epsilon = 0.1$, which means agents have a 0.1 probability of selecting the action randomly. Otherwise, the action is selected based on the Q values.

In DNE, we set the population size of individuals as N = 24, T is set to 2, and σ is set to 0.02, the time steps in an episode is 2000, which is the same with DRL.

3.4 Results

Fig. 3.4 shows the distance between foods and the nest during the DNE and DRL training process under the distance-based reward setting (Table 3.1) of 4 trials. Here, the distance is the mean value of four foods. In Fig. 3.4, the shaded area indicates the standard deviation. When the distance is around 4, the food is successfully pushed into the nest area.

According to Fig. 3.4, for the DNE, the average distance converges to 4 at the end of the training process, indicating that all foods are successfully pushed into the nest. On the other hand, for the DRL, the average distance drops sharply and converges to 12 at the early stage, suggesting that only a few foods are pushed into the nest area.

Fig. 3.5 shows the results of the second reward setting, which is based on the image's pixels. From the figure, we can observe that the average distance of DNE decreases gradually during the training process and converges to 4 at the end of training. As for DRL, the average distance drops quickly in the first 100 generations, reaching a stable state at around 200 generations. However, in the remaining training process, the average

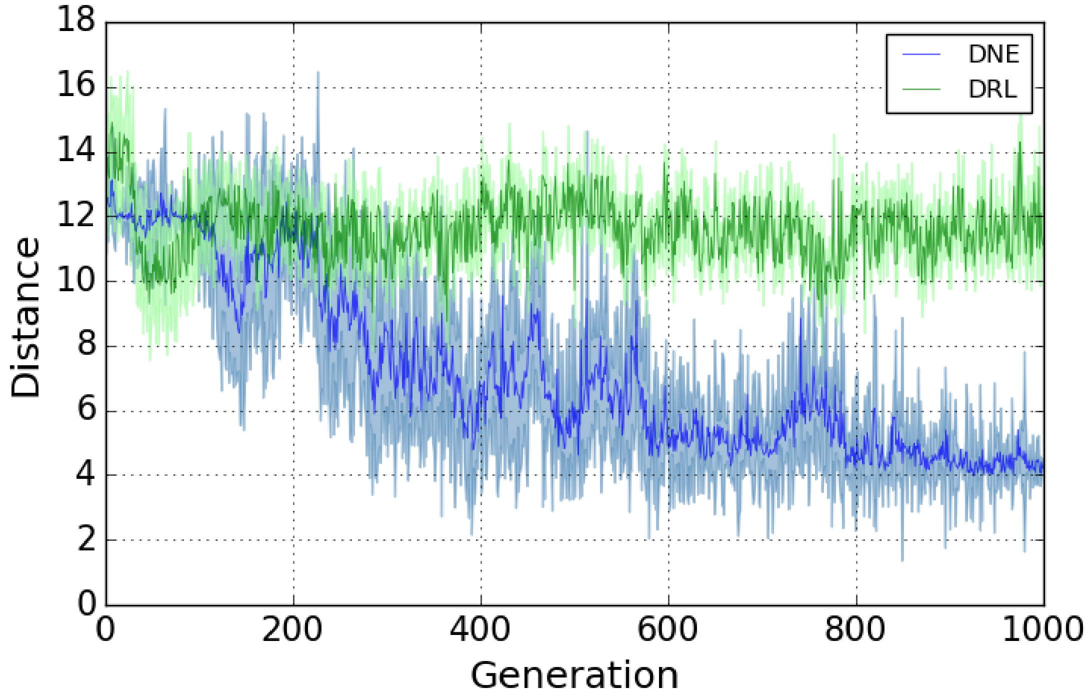


Figure 3.4: Average distance between four foods and nest area at the end of an episode during training under distance-based rewards.

distance of DRL increases compared to the first 200 generations. Consequently, both algorithms achieve similar performance by the end of training, indicating that both are capable of pushing all the foods into the nest area.

By comparing Fig. 3.4 and Fig. 3.5, DNE exhibits a similar tendency in the training process. The distance decreases gradually and converges to a stable state under two different reward settings. As for DRL, the curve shows a difference under different reward settings, suggesting that DRL is highly sensitive to the design of the reward system. Therefore, developers need to carefully design the reward when using DRL for swarm robotic systems.

3.5 Discussion

In this section, flexibility and scalability experiments are conducted to assess the performance of DNE and DQN. The flexibility experiment involves the addition of various obstacles to the environment. To test the scalability of the trained controllers, we vary the number of robots in the scalability experiment.

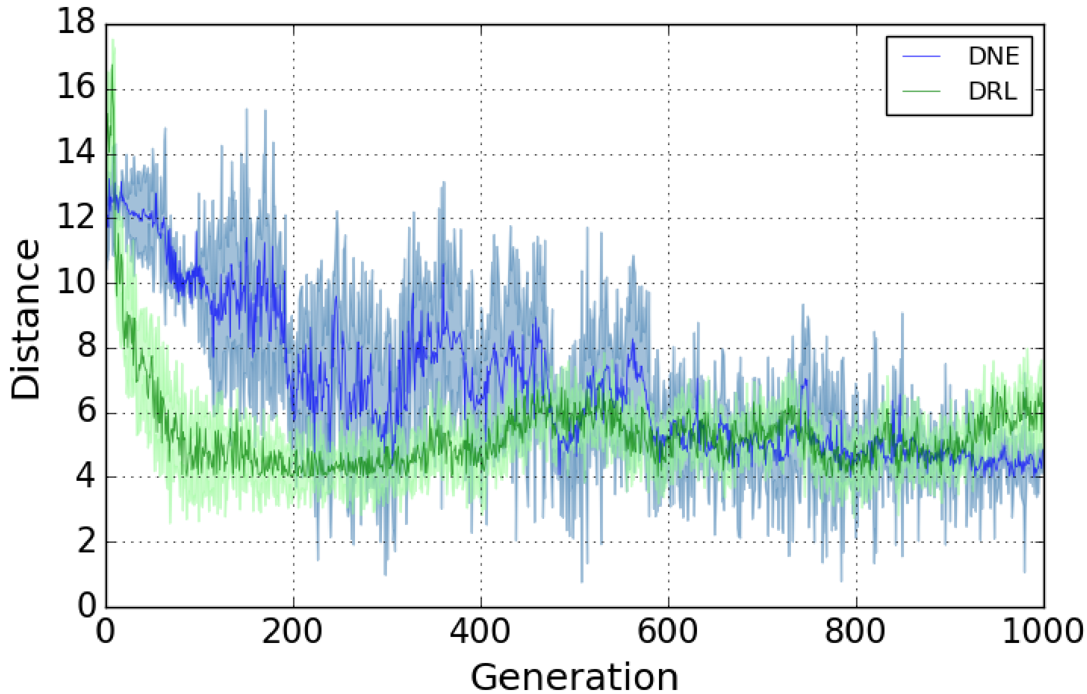


Figure 3.5: Average distance between four foods and nest area at the end of an episode during training under image-based rewards.

3.5.1 Flexibility Experiment

In the flexibility experiment, several obstacles are introduced into the environment, as depicted in Fig. 3.6. Specifically, three obstacles are placed between the foods and the nest. In this setup, the robots must navigate around these obstacles while transporting the food items. The image-based reward is utilized, consistent with the previous experiment, as DRL struggles to accomplish the task with a distance-based reward. It's important to highlight that there are no obstacles near Food 3, allowing for a comparison of performance under conditions with and without obstacles.

Fig. 3.7 illustrates the outcomes in the environment with obstacles. As depicted, DRL fails to collect all four foods within a single episode. Specifically, the figure indicates that DRL manages to push the foods over a limited range in the initial 110 generations, with no significant progress observed thereafter. Conversely, akin to the previous experiment, the average distance covered by DNE gradually decreases throughout the training period with obstacles. Ultimately, in the final stage, DNE demonstrates the capability to transport the foods while navigating around various obstacles.

The average distance traveled by each food is depicted in Fig. 3.8. In this task, DRL failed to learn how to transport Food 1 and Food 4. It took approximately 900

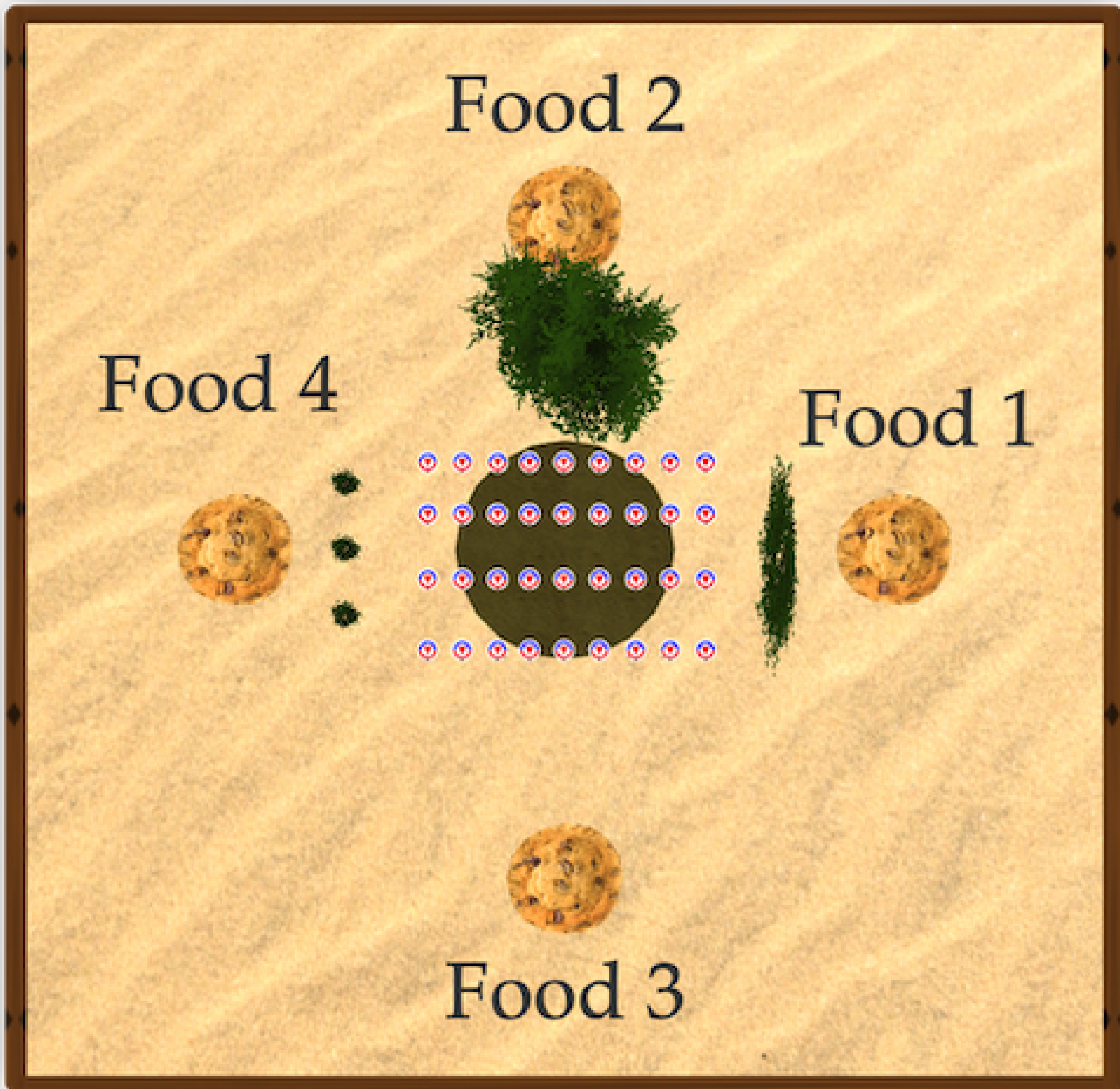


Figure 3.6: Environment with obstacles

generations for DNE to evolve and successfully collect foods amidst various obstacles. This experiment highlights DNE's capability to avoid local optima, where DRL tends to get easily trapped.

By comparing Fig. 3.7 and Fig. 3.8, where image-based reward is used, we observe that DNE exhibits a stable curve, similar to the experiment without obstacles. However, for DRL, performance deteriorates in the presence of obstacles, suggesting that DNE demonstrates better flexibility than DQN.

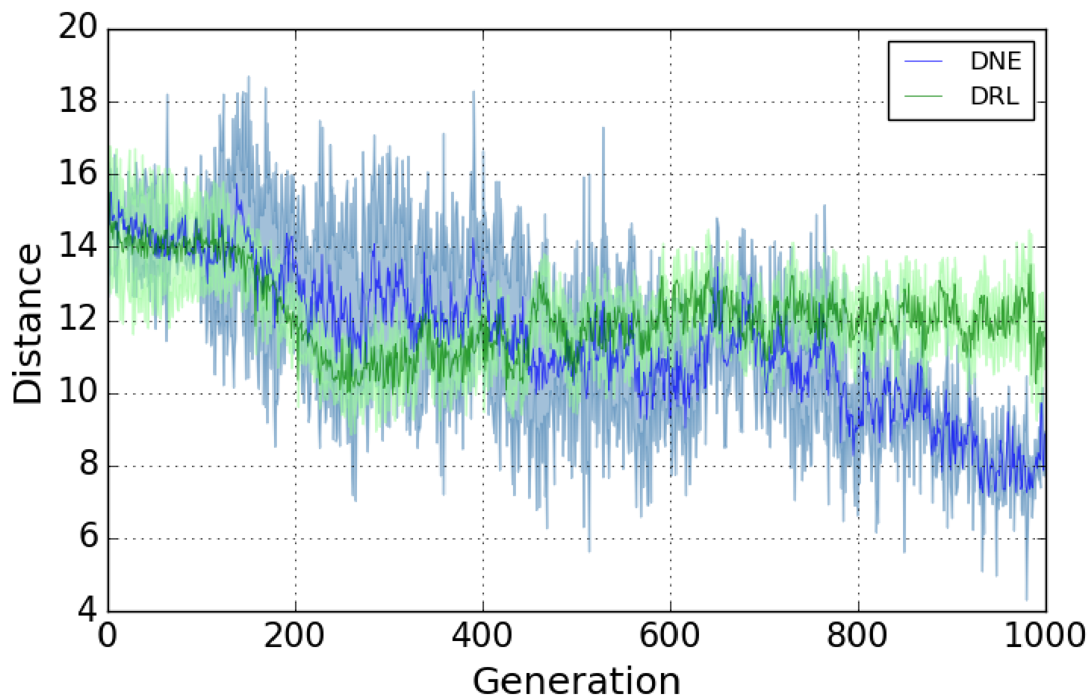


Figure 3.7: Average distance between four foods and nest area at the end of an episode during training with obstacles.

3.5.2 Scalability Experiment

To test the scalability of the controllers trained by DNE and DQN, we test the performance with 6, 12, 24, 36, and 48 robots in the environment with obstacles using image-based reward settings.

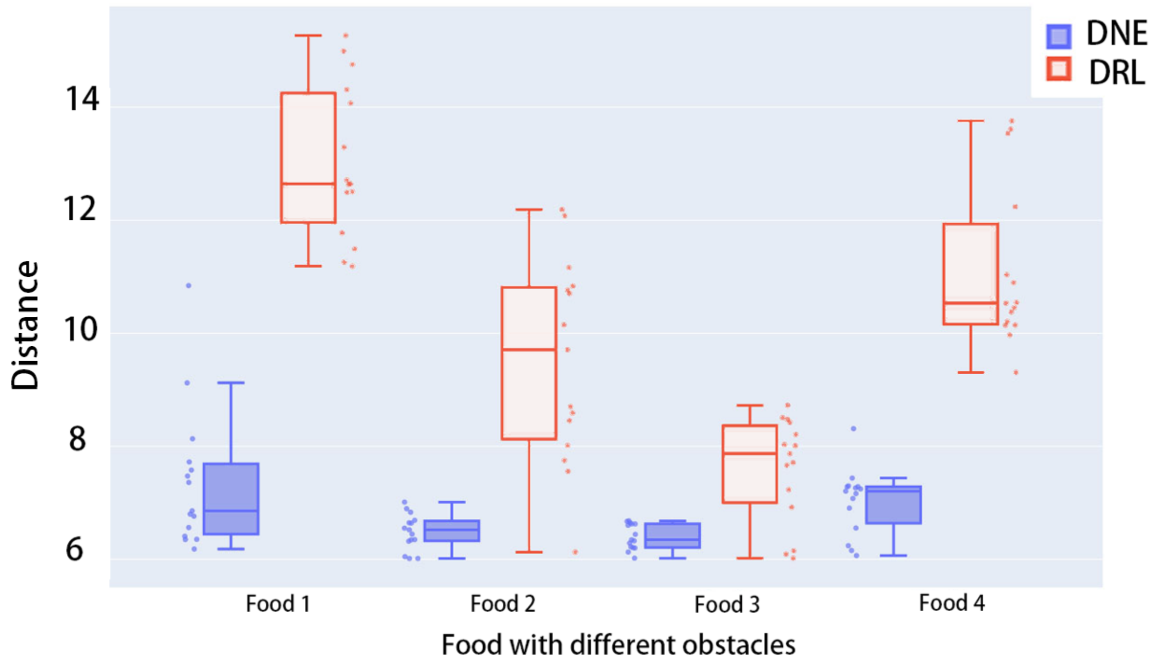


Figure 3.8: The distances between four foods and nest area with different obstacles.

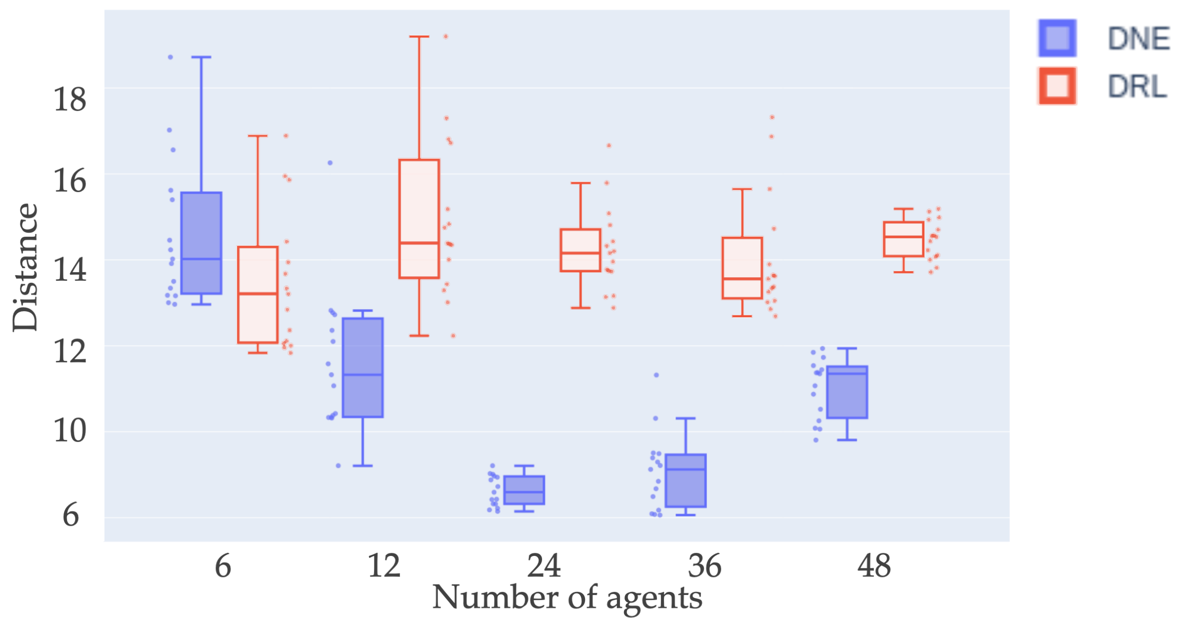


Figure 3.9: The distances between four foods and nest area with a various number of robots.

In this experiment, we compare the performance of a robotic swarm with varying quantities of robots to examine its scalability. The environment and the trained controllers remain consistent with the experiment involving obstacles. Fig. 3.9 illustrates the average distance between foods and the nest for different numbers of robots. Initially, when the number of robots is relatively small, both DNE and DRL struggle to accomplish the task of transporting all the foods, with DRL showing slightly better performance than DNE. However, as the number of robots increases, except for 48 robots, the distance covered by DNE decreases significantly, indicating successful collection of all the foods, while DRL fails to exhibit a similar decrease in distance. Nevertheless, when the number of robots reaches 48, the performance of both DNE and DRL declines. This decline can be attributed to the reduced space available for each robot to operate effectively. With more robots in the environment, collisions between robots become more frequent, hindering the transportation of the food.

3.6 Conclusions

This chapter focused on training a swarm robotic system to complete the collective transport task using image inputs, where robots are tasked with transporting food to the nest area. Deep Neuroevolution (DNE) and Deep Q-Learning (DQN) were employed to train the controllers for the robotic swarm. The robustness of the algorithms was tested using two types of reward/fitness settings, and experiments were conducted to assess the flexibility and scalability of the trained controllers.

The results of computer simulations indicate that DNE outperforms DQN under different settings in most cases. These findings suggest that DNE is less sensitive to changes in reward/fitness settings, thereby reducing costs and labor required to achieve the desired behavior. However, it's worth noting that training controllers for a robotic swarm using DNE demands significantly more computational resources compared to DQN. As a result, future research efforts may focus on developing new algorithms that combine evolutionary algorithms and deep reinforcement learning algorithms to reduce training time. This avenue could lead to more efficient and scalable training approaches for swarm robotic systems.

Chapter 4

Developing Multi-Agent Adversarial Environment Using Reinforcement Learning and Imitation Learning

4.1 Introduction

A multi-agent system (MAS) [6] is composed of multiple autonomous agents that interact with each other and the environment to achieve a common goal. Multi-agent systems have been widely adopted in many application domains because of the advantages offered, including robotics, intelligent transportation systems, and distributed control systems [95]. Although the agents in a MAS are capable of having pre-designed behaviors, they often have to learn new behaviors online in order to steadily improve the performance of the whole MAS. The complexity of the environment typically makes it difficult or even impossible to develop appropriate agent behaviors in advance. Additionally, behavior that is hardwired might become unsuitable in a dynamic or changing environment.

Deep Reinforcement Learning (DRL) is a promising approach in designing a controller for a MAS by interacting with its environment [51]. Recently, DRL has made remarkable achievements in various domains, such as robotic controls [96] and game playing [97]. In contrast to traditional control strategies, DRL does not rely on prior knowledge of environmental model parameters and offers strong adaptability and autonomous control capabilities [?]. In the DRL approach, the agent takes actions based on inputs following the policy in each state of the environment, which results in a reward and a state change. The goal of DRL is to learn an optimal policy that maximizes long-term cumulative rewards. However, a problem in DRL is that the rewards are sparse in some complex environments, which makes it difficult to obtain positive rewards during the exploration [98]. Notably, representing a task goal with a sparse reward function is often natural [99].

It is extremely difficult for DRL algorithms to connect a long series of actions to a distant future reward. Therefore, DRL algorithms struggle with the sparse reward problem. Furthermore, in order to generate competitive and cooperative behavior in an adversarial environment, it is necessary to provide the agent with a sufficient variety of information, such as information about teammates and opponents in the field of view, and to equip the agent with more sensors. However, input redundancy often results when an agent is provided with too much information. The existence of a large number of redundant features in the state space will increase the training time and resource consumption of the learning algorithm.

An intuitive solution to the sparse reward problem is to provide the agent with denser rewards by designing dense reward functions for specific states or agent behaviors. However, in some complex scenarios, it is practically infeasible to design rewards for specific behaviors of agents due to the vast state and action spaces. Moreover, this approach often leads to the agent prioritizing specific behaviors to obtain denser rewards over completing the given task. An alternative approach to overcome the sparse reward problem is imitation learning (IL) [61]. In IL, agents can obtain the information they need directly from the demonstration trajectory samples, like what action they should take in a certain state. In this case, agents can learn what to do by imitating expert demonstrations even if they are in a sparse reward environment.

On the other hand, one approach to address input redundancy is the attention mechanism, which is inspired by cognitive attention. In the realm of RL, we can also leverage this mechanism, which assigns a weight to each input for an agent. These weights represent the agent's focus on different pieces of environmental information.

In this chapter, we propose a combination of DRL and IL training methods to overcome the sparse reward problem. Additionally, we implement an attention mechanism to tackle the challenge of information redundancy. In our proposed approach, the controller trained using PPO, a DRL algorithm, and dense reward settings serves as the expert. Subsequently, PPO-GAIL, a combination of DRL and IL, trains the controller of a MAS with expert demonstrations and sparse reward settings. This study designs a beach volleyball adversarial environment where two groups of agents compete, requiring cooperation among teammates for defense or attack. The results demonstrate that controllers trained using our proposed method overcome the sparse reward problem and outperform those trained using conventional RL methods.

4.2 Research Methodology

In this section, we describe how imitation learning and the attention mechanism can be applied with deep reinforcement learning.

4.2.1 Proximal Policy Optimization(PPO)

Recently, DRL methods have been successfully applied in many scenarios, such as the Go [93], Atari-2600 games [97] , and Deepmind's StarCraft II [100]. There are many types of RL algorithms, such as Deep Q-Learning (DQN) [55], which combines a deep neural network with Q-learning to solve the problem of high-dimensional state and action spaces; Soft actor-critic (SAC) [101], which encourages the agent to explore as much as possible while completing the task by introducing entropy; and Proximal Policy Optimization (PPO) [59].

This study uses PPO as the RL method. PPO increases the utilization of the training data with the implementation of the clip function in the objective function. It also has the advantages of stable training process, high performance and scalability [102]. PPO is an actor-critic method. The actor interacts with the environment and learns a better policy with the policy gradient under the guidance of the critic. The critic outputs an advantage function through the data collected by the interaction between the actor and the environment. This advantage function will judge if an action is good or bad in the current stage according to the received reward. A typical form of the advantage function is shown in Eq. 4.1,

$$\hat{A}_t = R_t + \gamma V\left(s_{t+1}\right) - V\left(s_t\right) \tag{4.1}$$

where R_t stands for the reward the agent received at the timestep t, γ is the discount rate, and $V\left(s_t\right)$ is the state value function that evaluates how much future rewards the agent could receive after visiting the state s_t. The advantage function helps the actor optimize the agent's policy. The loss function used for updating the actor network is shown in Eq. 4.2,

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \operatorname{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{4.2}$$

where $r_t(\theta)$ is the ratio of new and old strategies: $r_t(\theta) = \frac{\pi_\theta((a_t|s_t))}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, $\hat{A}_t$ is the advantage function and ϵ is the hyperparameter. PPO makes multiple batches of epoch updates to increase the efficiency, and the function $\operatorname{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)$ limits $r_t(\theta)$, the ratio of new and old strategies, within $(1-\epsilon)$ and $(1+\epsilon)$ to prevent performance degradation caused by parameter updates.

4.2.2 Generative Adversarial Imitation Learning (GAIL)

Imitation learning involves agents learning manipulation by observing the expert's demonstration [103]. As agents trained by IL can directly learn from the good samples instead of learning from scratch, so the learning efficiency can be improved. IL algorithms have

been applied in many fields, including human-computer interaction and machine vision [104].

This study applies GAIL as the imitation learning method. GAIL imitates the occupancy measure of the expert's policy and tries to make the occupancy measure of all state-action pairs consistent with the experts. To achieve this goal, the strategy needs to interact with the environment to collect information. This is different from behavior cloning, which does not need to interact with the environment. GAIL contains a generator (G) and a discriminator (D), The generator will output actions corresponding to the given state. The input for the discriminator is the state-action pair, and the discriminator will output a number between 0 and 1 to represent the probability that this state-action pair is from the agent's policy rather than the expert's. The loss function used for updating the discriminator is shown in Eq. 4.3

$$L(\theta) = -\mathbb{E}_{\rho_\pi}\left[\log D_\phi(s,a)\right] - \mathbb{E}_{\rho_E}\left[\log\left(1 - D_\phi(s,a)\right)\right] \tag{4.3}$$

where $D_\phi(s,a)$ is the output of the discriminator network, ϕ is the weight for the D network, ρ_π is the occupancy measure of the agent and ρ_E is the occupancy measure of the expert.

4.2.3 Attention Mechanism

The attention mechanism operates on the principle of assigning distinct weights to various positions or segments of the input data, enabling the model to selectively concentrate on crucial information within the input [45].

In the context of a multi-agent cooperative adversarial environment, the attention mechanism serves to bolster collaboration and adaptability among agents [105]. Within a multi-agent system, each agent receives input information from the environment, encompassing details such as the agent's surrounding state, actions taken by other agents, and information relevant to the ongoing task. Each agent partitions its input into three components: Query, Key, and Value. These components, learned from the model, signify the content to be emphasized, the content used for weight computation, and the actual values, respectively.

This study leverages the correlation between Query and Key to calculate the weight associated with each Key. The process involves determining the similarity between Query and Key, followed by normalization of the computed similarities. The resultant weights indicate which aspects of the input demand the model's attention during processing. Ultimately, these calculated weights are applied to the corresponding Value, resulting in a weighted summation that serves as the model's final output.

4.2.4 Beach Volleyball Game

MASs can be described as cooperative or competitive, depending on their objectives. In MASs, cooperation is a crucial behavioral pattern that can be achieved through coordination and collaborative efforts to accomplish shared objectives. On the other hand, competitive scenarios in MAS involve situations where multiple autonomous agents competitively interact with each other, often aiming to outperform each other to achieve individual objectives. In this study, a 3D beach volleyball environment is designed as the subject of the experiment. Similar to real-world beach volleyball, the environment consists of two teams, each composed of two agents. The two teams of agents need to compete with each other, and agents on the same team need to cooperate to defend or attack. In this 3D environment, according to the landing point and speed of the ball, the agent needs to move to a proper position and jump at the right time to hit the ball at different angles. Therefore, sparse reward settings (for example, gaining a reward when defeating the opponent) can hardly help agents learn effective behaviors with conventional reinforcement learning training methods. Furthermore, it is extremely difficult to design dense reward functions based on the specific behaviors of the agent due to the complex state and action spaces.

4.2.5 Self-Play

Self-play is a technique used to enhance the performance of multi-agent systems (MAS). This technique allows agents to learn and improve by competing or cooperating with copies of themselves. Over recent decades, self-play has proven effective in various domains, particularly in developing artificial intelligence for games. In self-play, agents take charge of their own learning and advancement. They engage in iterative self-interactions, identifying weaknesses and optimizing their decision-making processes. The key idea is that by consistently confronting a mirror image of their abilities and tactics, agents can empower themselves to become more efficient and effective. This approach is particularly valuable in situations where designing explicit training protocols or reward structures is challenging, as it enables agents to generate their training data through self-interaction, eliminating the need for external feedback.

Competitive self-play is a specific application of self-play in which agents compete against each other to improve their strategies and performance. This approach is efficient when adversarial interactions are vital to mastering the task. By competing with themselves, agents can simulate a wide range of competitive scenarios, which helps them develop robust strategies that can generalize to real-world adversaries. The hyperparameters settings of competitive self-play are introduced below:

- **Save Steps**: The number of training steps between snapshots of the current policy.

- **Team Change**: The number of training steps between switching the learning team.
- **Swap Steps**: The number of training steps between swapping the opponents policy with a different snapshot.
- **Play Against Latest Model Ratio**: The probability an agent will play against the latest opponent policy.
- **Window**: The size of the sliding window of past snapshots from which the agent's opponents are sampled.

4.3 Environmental Settings

In this study, we conducted a beach volleyball game in computer simulations using the Unity 3D game engine. The simulation is executed in a computer platform with Ubuntu 18.04 operating system equipped with Nvidia GeForce RTX 3070 GPU, AMD Ryzen 9 3950x CPU and 128 GB memory.

4.3.1 Robot Settings

The design of the agents is depicted in Fig. 4.1. There are two types of agents, the blue agent, and the purple agent. They share the same settings, except that the blue agent belongs to team blue and the purple agent belongs to team purple. Robots have four kinds of movements corresponding to forward movement, side movement, rotating, and jumping. Each agent is 1 meter high and weighs 20 kilograms. The maximum jump height for an agent is 2 meters, and the maximum running speed is 1.5 meters per second. The Robot's maximum jump height is intentionally set below the height of the net to prevent it from jumping into the opponent's court.

The agent's observations are captured using a vector sensor, which includes 11 vector observations. These observations encompass the agent's rotation, velocity, the ball's velocity, the direction from the agent to the ball, and a state value termed as 'touched.' Initially, 'touched' is set to 0. When the agent makes contact with the ball, 'touched' changes to 1. Additionally, every time the ball crosses the net, 'touched' is reset to 0.

4.3.2 Environment

The top view of the environment is shown in Fig. 4.2. There are two teams in the environment, the blue and purple teams. Each team contains two players. The environment is a rectangular area surrounded by borders. The playground is 20 meters long and 10 meters wide. A 3-meter high net is in the middle of the ground. Two teams of agents

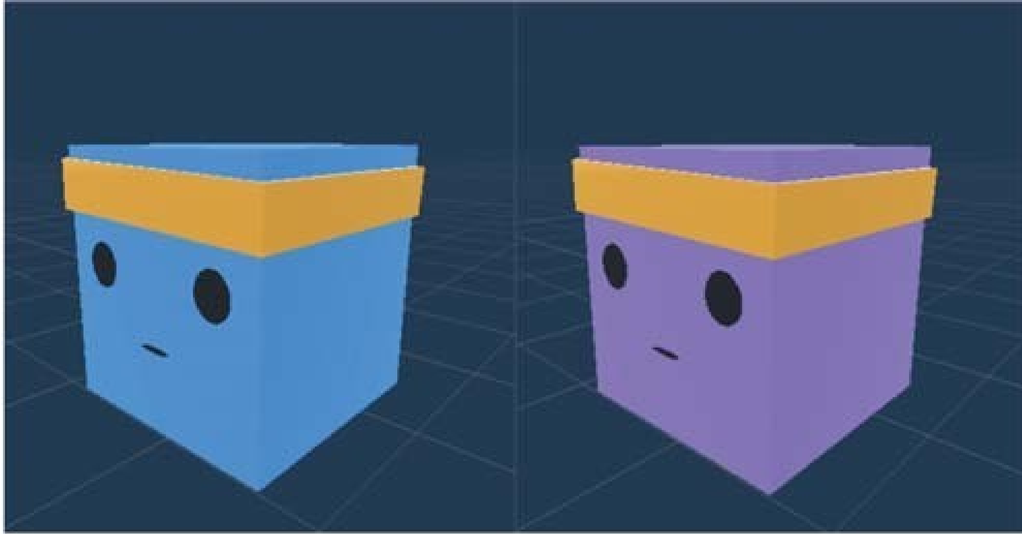


Figure 4.1: Design of the blue and purple agent

engage in a beach volleyball game within the environment. A volleyball will drop in a random position at the beginning of each episode. Robots need to move and hit the ball and let it hit their opponent's ground to win while preventing it from hitting their own ground, or they will lose.

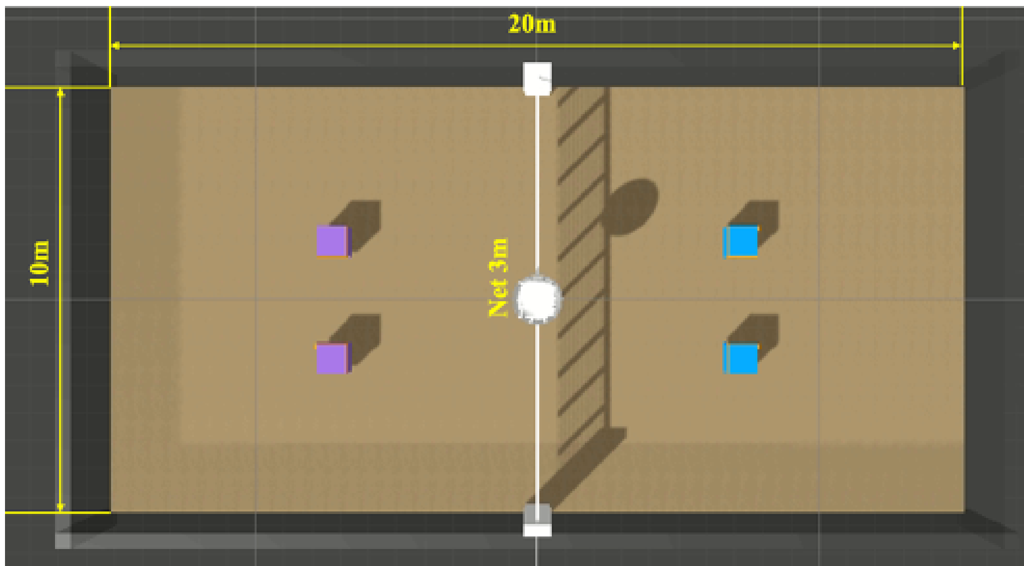


Figure 4.2: Top view of the environment

Moreover, this study introduces the double-hit rule to the environment. The double-hit rule is an essential regulation in volleyball, which occurs when a player touches the ball twice in succession without it crossing the net. A diagram illustrating this rule is provided in Fig. 4.3. If an agent violates the double-hit rule, its team will lose the game.

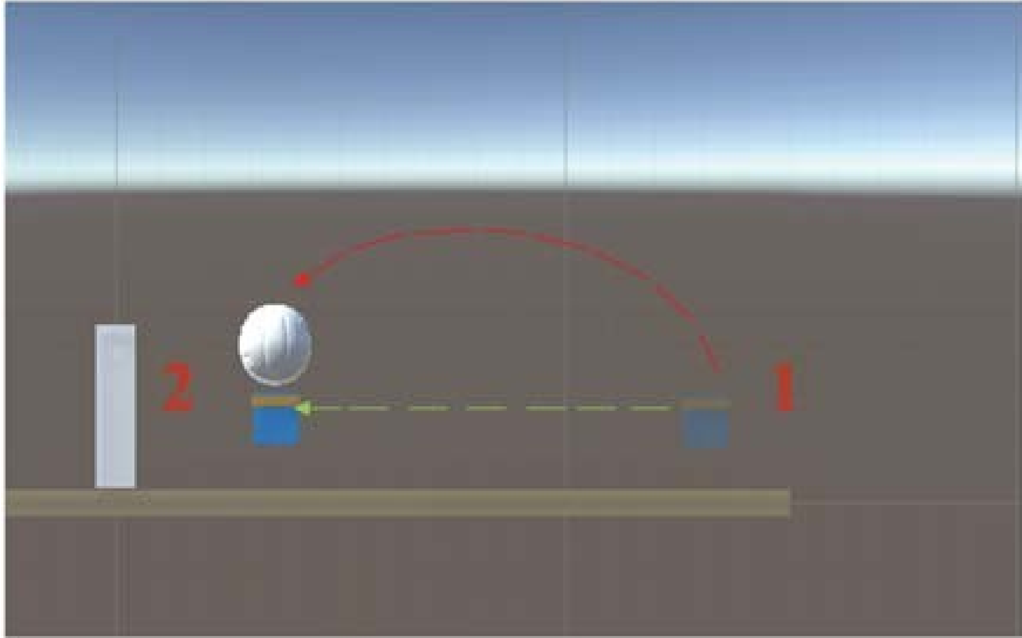


Figure 4.3: The diagram of the double hit rule

4.3.3 Self-play Settings

In this chapter, all experiments share a same self-play setting, which is shown in Table 4.1.

Table 4.1: Self-play settings

Hyperparameter	Value
Save steps	20000
Team change	100000
Swap steps	10000
Play against the latest model	0.5
Window	10

Every 10,000 training steps, a snapshot of the current network is saved. The maximum number of saved snapshots is 10. Once ten snapshots are reached, the oldest ones are discarded when new snapshots are taken. The teams swap every 100,000 steps. Additionally, every 10,000 steps, the adversary network selects one from the saved snapshots. Among them, the probability of selecting the latest snapshot is 50%.

4.3.4 Elo Rating System

In order to verify the performance of imitation reinforcement learning, the Elo rating system has been implemented to evaluate controllers. The Elo rating system was originally developed for rating chess players. Nowadays, it is widely used for ranking players of many other games[**?**]. In the Elo system, assuming that players A and B have current ratings of R_A and R_B, the expected win rate of A against B, according to the logistic distribution, is represented as E_A, as shown in Eq. 4.4.

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}} \tag{4.4}$$

If player A's actual score S_A in a match (win=1, draw=0.5, loss=0) is different from the expected win rate E_A, the rating of player A will be adjusted according to Eq. 4.5.

$$R'_A = R_A + (S_A - E_A) \tag{4.5}$$

Typically, in this study, the initial Elo score of each agent is set as 1200.

4.4 Simulation 1: Overcoming The Sparse Reward Problem with The Imitation Reinforcement Learning Method

In this study, the imitation reinforcement learning training method is employed to train the controllers of a multi-agent system to address the sparse reward problem. Initially, experts are trained using reinforcement learning with dense reward settings. Subsequently, the expert's demonstrations are utilized to train the controller using imitation reinforcement learning with sparse reward settings. Specifically, the Proximal Policy Optimization with Generative Adversarial Imitation Learning (PPO-GAIL), a combination of PPO and GAIL, serves as the imitation reinforcement learning algorithm.

In PPO-GAIL, the discriminator network in GAIL is trained to distinguish whether the state-action pair originates from the training controller or the expert's demonstration. The output of the discriminator network serves as an additional reward signal, which, combined with the sparse reward signals in the environment, forms the advantage function in PPO. The neural network settings of the actor, critic, and discriminator networks are illustrated in Fig. 4.4, Fig. 4.5, and Fig. 4.6.

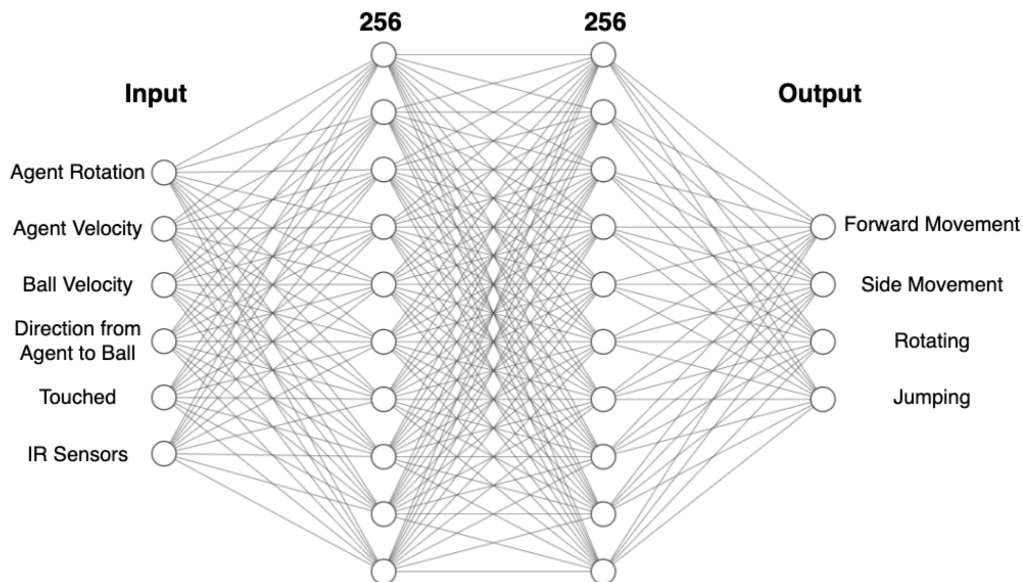


Figure 4.4: The network structure of the actor network

4.4.1 Reward Settings

In this study, two types of reward settings are designed: sparse reward settings and dense reward settings. In the sparse reward settings, as shown in Eq. 4.6, the agent receives a

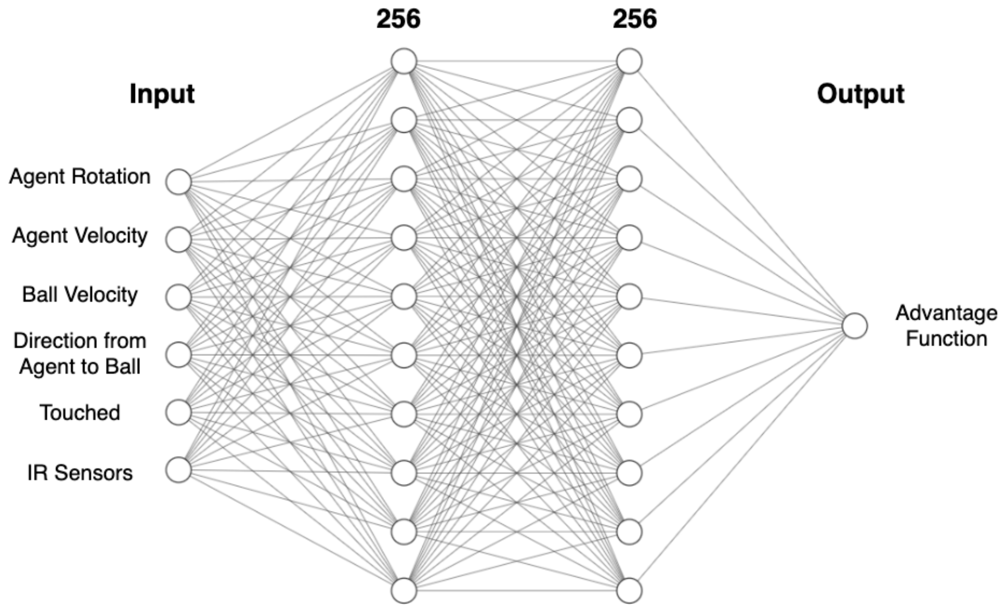


Figure 4.5: The network structure of the critic network

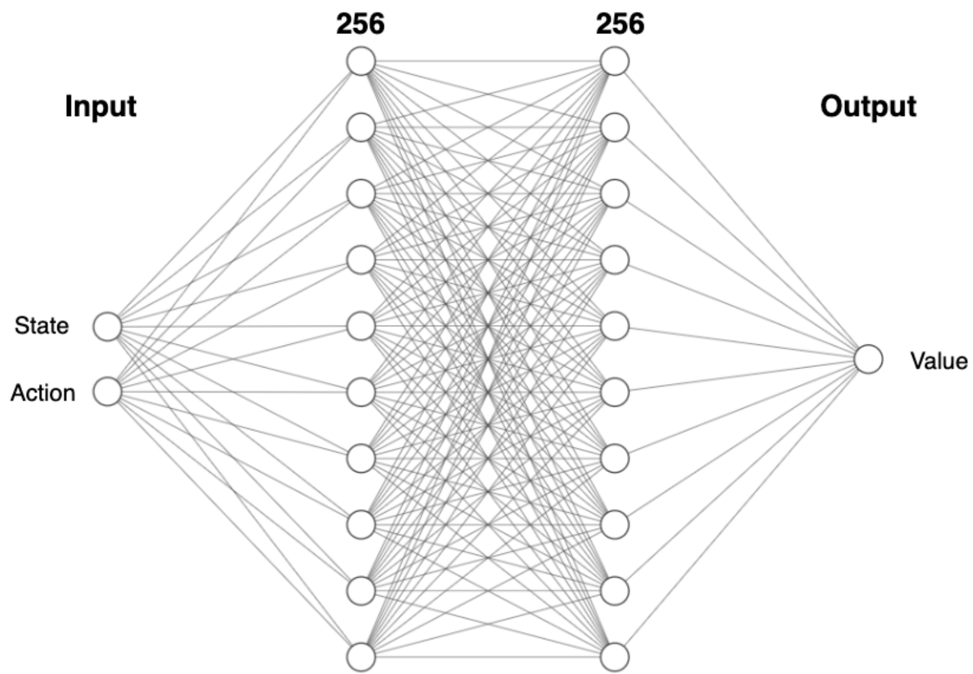


Figure 4.6: The network structure of the discriminator network

reward only when its team wins or loses the game. Consequently, agents encounter the sparse reward problem and may struggle to receive positive rewards during the initial exploration.

$$R_{Sparse} = \begin{cases} 1, & \text{if its team wins the game,} \\ -1, & \text{if its team loses the game} \end{cases} \tag{4.6}$$

On the other hand, as mentioned previously, designing rewards based on the states of agents is infeasible. In this study, dense rewards are designed based on the state of the environment, as shown in Eq. 4.7. The agent receives a positive reward when its team successfully hits the ball over the net and into the opponent' s half.

$$R_{Dense} = 1, \text{ if its team hits the ball across the net} \tag{4.7}$$

4.4.2 Results

In this study, three experiments are conducted to evaluate the performance of the conventional PPO and the PPO-GAIL training methods, as shown in Table 4.2. In the PPO-GAIL training method, the expert refers to the controller generated in Experiment 2, where the agents are trained with dense reward settings. The hyperparameter settings for these experiments are provided in Table 4.3.

Table 4.2: Experiment settings

Experiment	Training Method	Reward Settings
Experiment 1	Conventional PPO	Sparse
Experiment 2	Conventional PPO	Dense
Experiment 3	PPO-GAIL	Sparse

Table 4.3: Hyperparameter settings

Hyperparameter	Value
Hidden units	256
Number of layers	2
Batch size	2048
Buffer size	20480
Learning rate	0.0002
Gamma	0.96
Epsilon	0.15

Fig. 4.7 depicts the episode length of the three experiments during the learning process. In Experiment 1, the episode length converges to 20 steps, indicating that the agents are

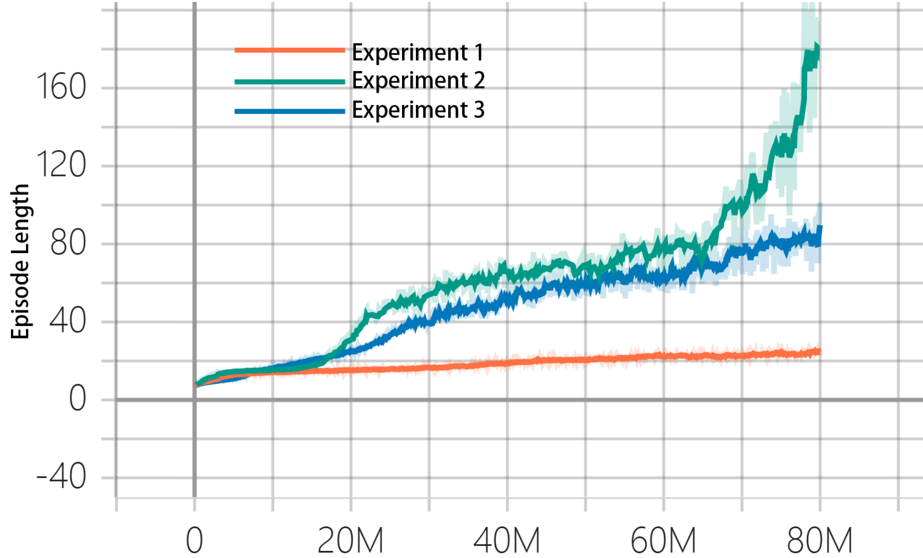


Figure 4.7: The results of the episode length

unable to hit the ball throughout the learning process. Experiment 2 achieves an episode length of 180 steps at the end of training, while Experiment 3 reaches 60 steps during the final stage of training. A comparison of the results between Experiment 1 and Experiment 2 reveals that agents trained with conventional PPO fail to learn how to play volleyball due to the sparse reward problem. Despite the longer gameplay duration of agents trained in Experiment 2 compared to those in Experiment 3, this does not imply that PPO with dense reward settings performs better than the combination of GAIL and PPO in this competitive environment.

Fig. 4.8 illustrates the Elo scores during the training process. Agents begin with an Elo score of 1200, and those trained with PPO and sparse reward settings consistently remain in the low segment, indicating a lack of learned strategies for playing volleyball. Conversely, agents trained with PPO and dense reward settings progressively achieve scores around 1450. Agents trained with the PPO-GAIL method reach the highest score of 1600 during the final stage of the training process.

Furthermore, to evaluate the performance of trained controllers, 500 competitions are conducted between the controllers generated in Experiment 2 and Experiment 3. The competition results are presented in Table 4.4, indicating that our strategy outperforms the control group with a winning rate of 94.4% compared to the control group's 5.6%.

The agents trained with PPO and dense rewards settings generate behaviors to maximize the cumulative reward. They stand near the middle of the field and hit the ball to get it over the net as much as possible. In order to get more rewards, they are more willing to hit the ball that is easier for the opponent to catch than to defeat the opponent. To some extent, the dense reward setting helps the agents learn to hit the ball, but it also

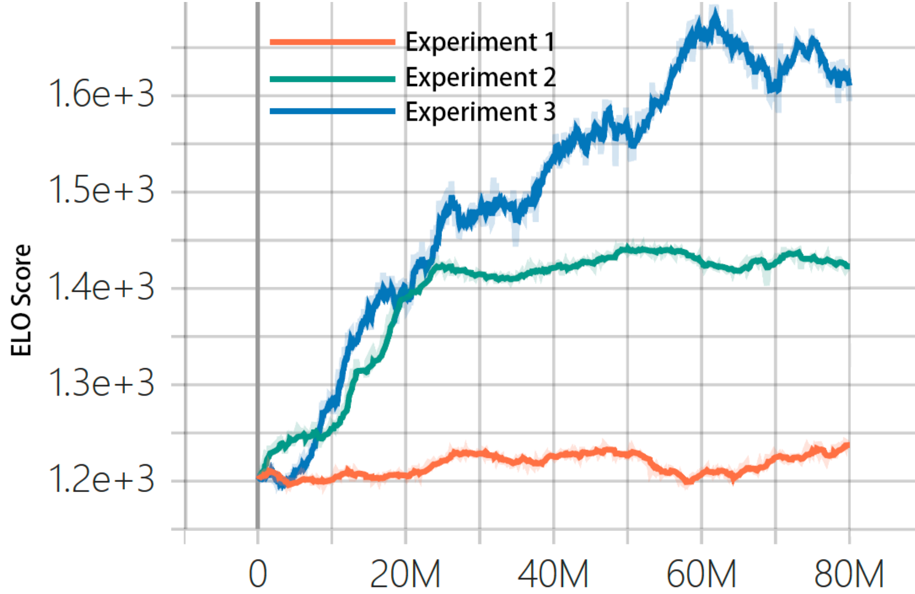


Figure 4.8: Elo training results

Table 4.4: Wins and winning rate between two trained controllers

Controller	Wins	Wining Rate
GAIL-PPO+Sparse Reward	473	94.4 %
PPO+Dense Reward	27	5.6 %

prevents them from effectively beating their opponents.

On the other hand, agents trained using the PPO-GAIL method exhibit more effective behaviors aimed at winning the game. This study incorporated the double hit rule into the experiment, where an agent cannot hit the ball again if it fails to clear the net with its initial hit. Consequently, cooperation between teammates becomes crucial to execute successful attacks. Fig. 4.9 illustrates the cooperative behavior, wherein two agents on the same team hit the ball over the net with two touches. This behavior is defined as passing behavior, with the first agent to hit the ball considered the passer and the second agent as the catcher.

The occurrences of cooperative behavior are recorded during the learning period, with each instance of the ball crossing the net marking a round of the game. One hundred rounds of the experiment were conducted, and Table 4.5 shows the times of cooperative behaviors in the initial stage, middle stage, and final stage.presents the frequency of cooperative behaviors observed at different stages: initial, middle, and final.

To visualize the distribution of cooperative behaviors, heat maps were utilized in this

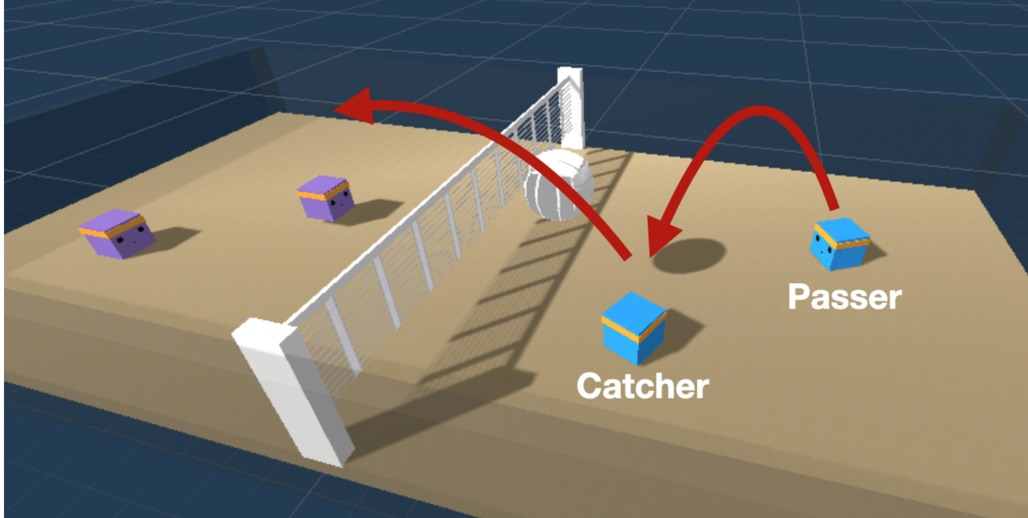


Figure 4.9: Cooperative behaviors

Table 4.5: Times of cooperative behaviors during the learning process

Training Period	Times of Cooperative Behaviors (100 Rounds)
Initial Stage	0
Middle Stage	12
Final Stage	37

study. The hitting positions of the passer are recorded and depicted in Fig. 4.10 for each passing behavior, while Fig. 4.11 shows the locations where the catcher receives the ball and hits it over the net. These figures reveal that during passing behaviors, the ball is often hit from the back to the front of one's half. This is attributed to the challenge faced by the back-hitting agent (passer) in returning the ball over the net with a single touch, necessitating the involvement of another agent (catcher) positioned in the front to receive and return the ball. The agents demonstrate cooperative passing behavior particularly when the opponent hits the ball into their half.

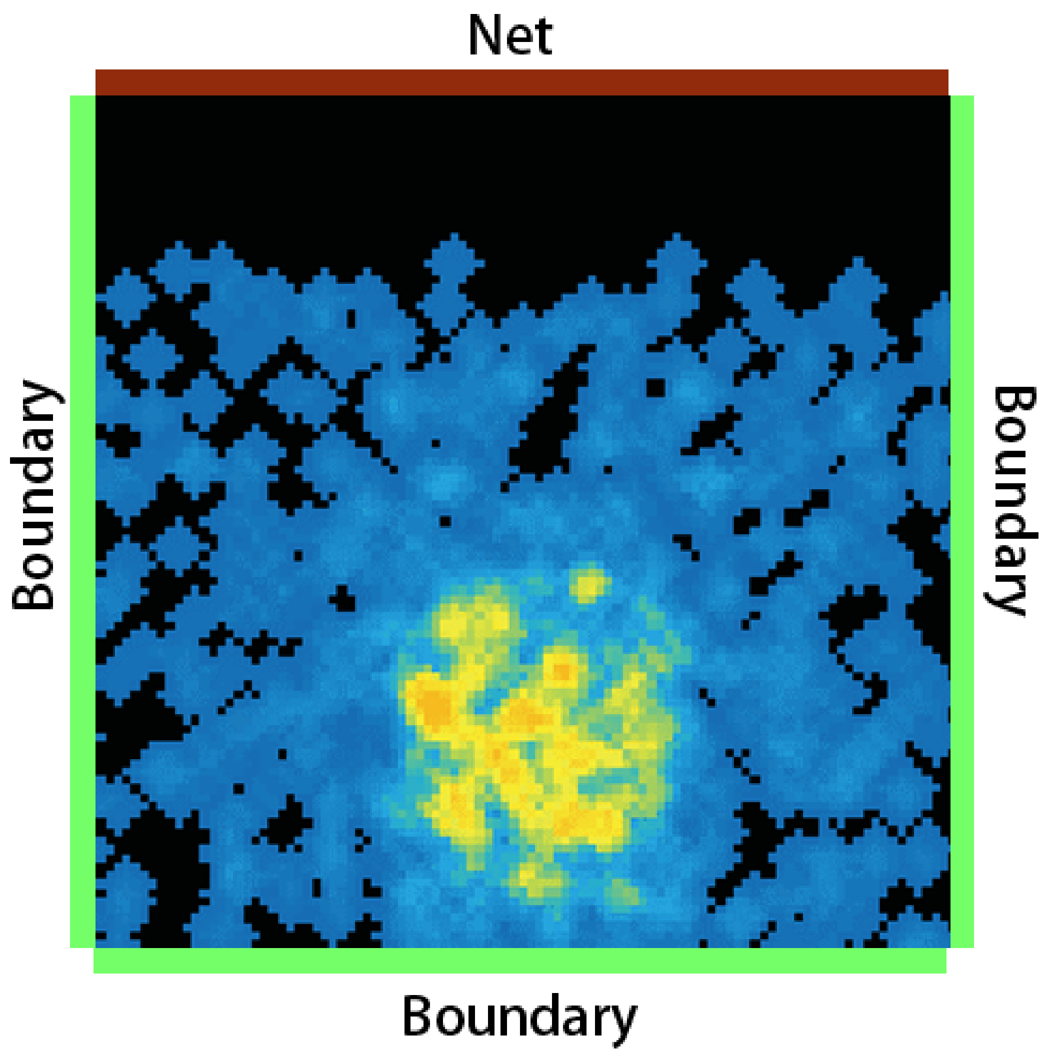


Figure 4.10: The heat map of the passer

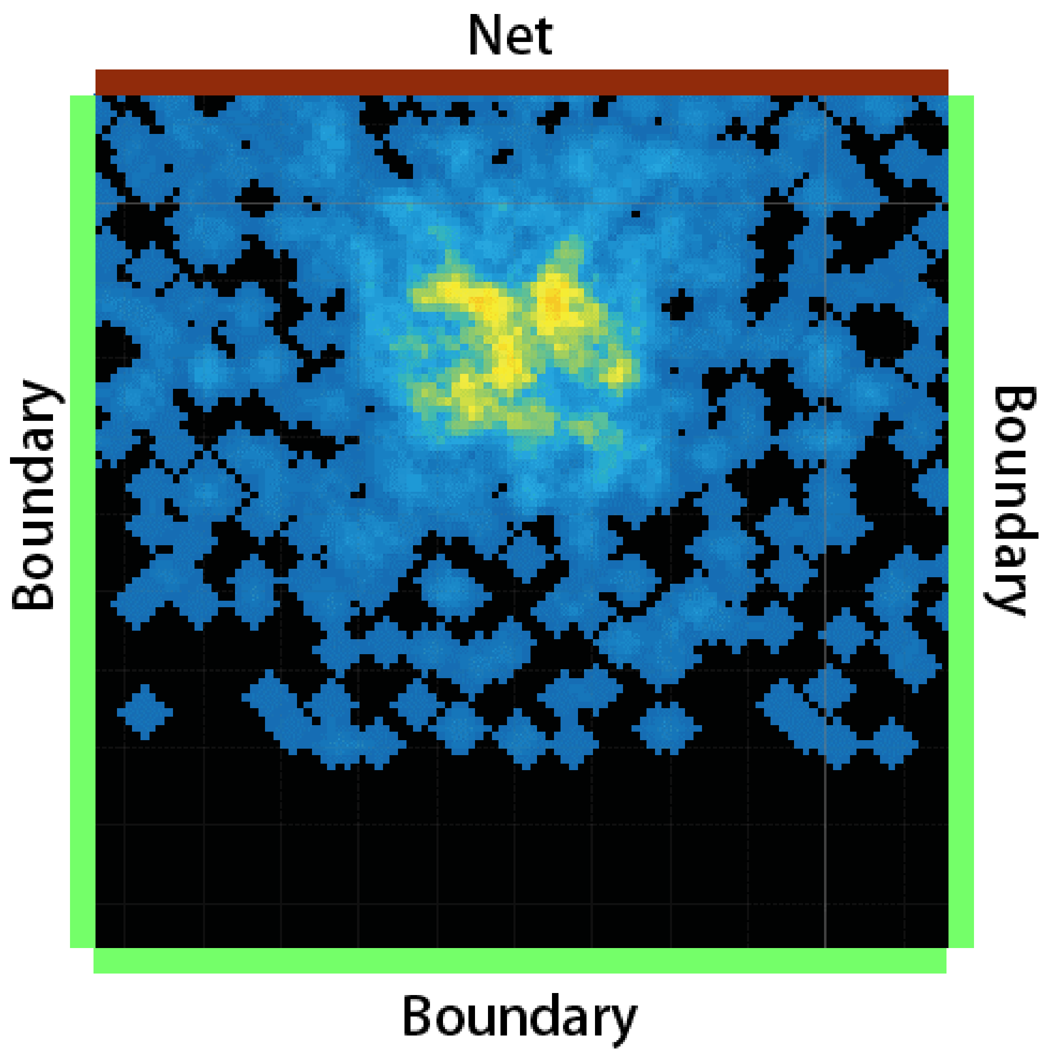


Figure 4.11: Heat map of the catcher

4.5 Simulation 2: Generating Competitive Behavior in An Adversarial Environment with The Attention Mechanism

In this section, we incorporated IR sensors as additional inputs with the expectation that agents would exhibit improved cooperative and adversarial behaviors. These IR sensors enable the agents to identify various elements in the environment, including the net, walls, teammates, and opponents. Several experiments were conducted based on the number of front IR sensors equipped on the agents and whether an attention mechanism was employed. The experiment settings are detailed in Table 4.6. The neural network settings for the PPO-GAIL method remained consistent with those used in Simulation 1. However, for the PPO-GAIL with attention mechanism method, an attention layer was inserted between the input layer and the hidden layer for all networks, as illustrated in Fig. 4.12

Table 4.6: Experiment Settings

Experiment	Training Method	IR Sensor Amount
Experiment 1	PPO-GAIL	6
Experiment 2	PPO-GAIL	20
Experiment 3	PPO-GAIL	100
Experiment 4	PPO-GAIL with Attention Mechanism	6
Experiment 5		20
Experiment 6		100

4.5.1 Results

In the first three experiments, PPO-GAIL was employed for training, and the corresponding Elo scores are illustrated in Fig. 4.13. A decrease in Elo scores was observed when simply increasing the number of infrared sensors, suggesting that the additional inputs provided to the agents were not effectively processed.

In the last 3 experiments, an attention mechanism was incorporated during the training of agents equipped with 6, 20, and 100 IR sensors. The Elo training results of Experiments 4, 5, and 6 are shown in Fig. 4.14. The results indicate that, with the introduction of the attention mechanism, improvements were observed in the Elo scores of agents equipped with 20 and 100 IR sensors. However, the performance of the agent equipped with 6 IR sensors, despite showing increased learning speed compared to before the attention

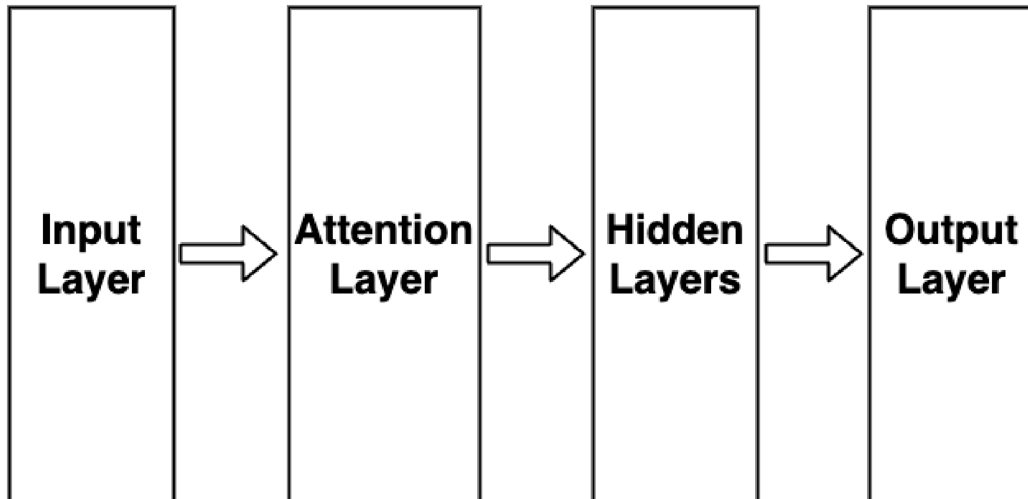


Figure 4.12: The implementation of the attention mechanism

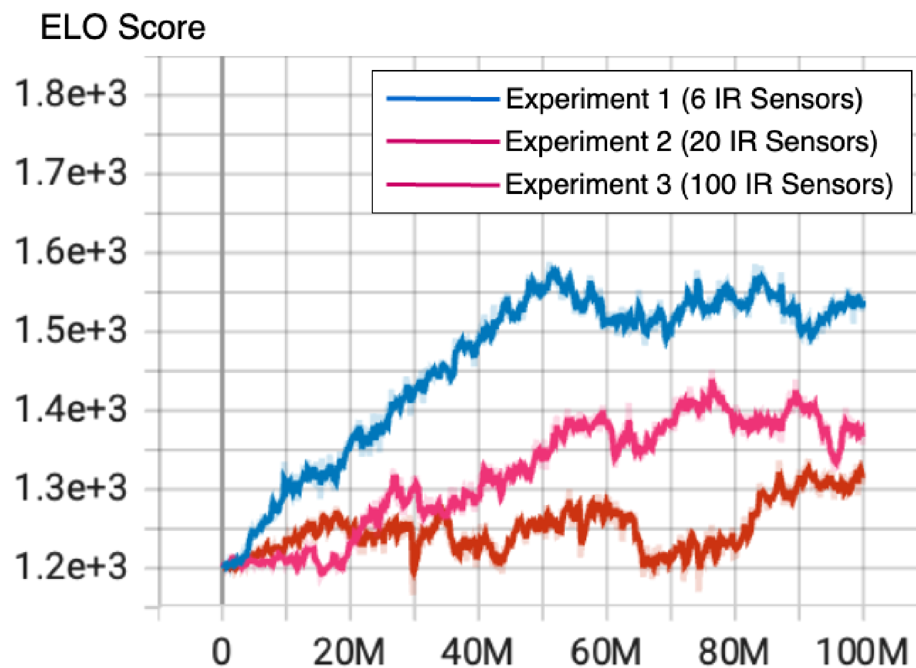


Figure 4.13: Elo training results of Experiments 1, 2, and 3 (without attention mechanism)

mechanism was added, did not exhibit an increase in Elo scores.

Moreover, adversarial matches and analysis were conducted using 100 IR sensors (ATT) (Blue Team) and 100 IR sensors (Purple Team), as illustrated in Fig. 4.15. This study conducted two hundred competitions between 100 IR sensors(ATT) and 100 IR sensors. The results of the competition are shown in Table. 4.7.

From the Elo scores and winning rates, it is evident that the addition of an attention mechanism to the 100 IR sensors has indeed resulted in improved adversarial performance when competing against agents with the same number of sensors.

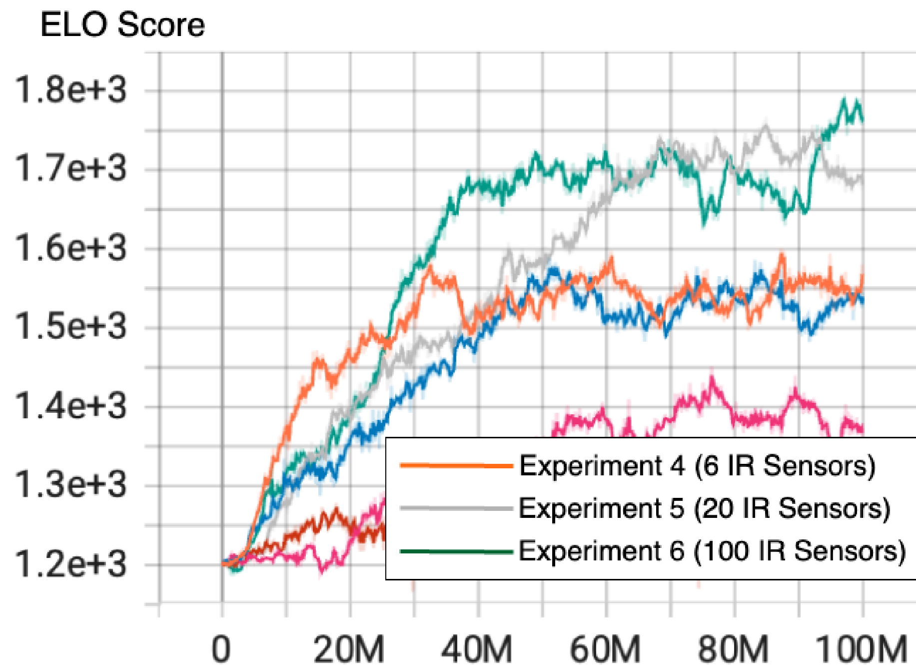


Figure 4.14: Elo training results of Experiments 4, 5, and 6 (with attention mechanism)

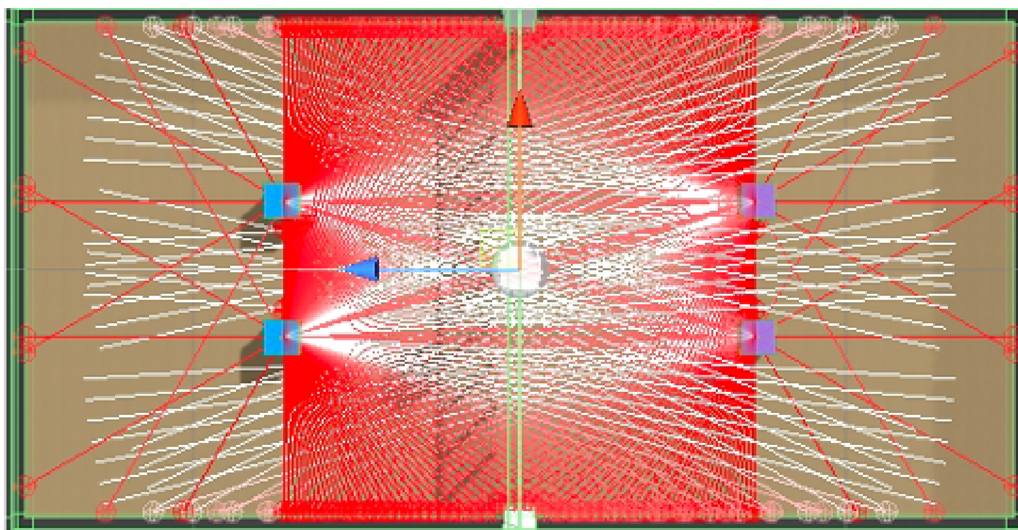


Figure 4.15: Adversarial matches

Table 4.7: Results of the competitions

	100 IR sensor(ATT)	100 IR sensor
Wins (in 200)	167	33
Winning rate	83.5%	16.5%

In this study, the offensive behaviors of the four agents in the 200 matches were documented by defining an agent hitting the ball over the net as one attack behavior.On the Blue Team (100 IR sensors ATT), Blue1 is positioned on the left, and Blue2 on the right. Similarly, on the Purple Team (100 IR sensors), Purple1 is on the left, and Purple2 is on the right. The results of attacks are presented in Table. 4.8.

Table 4.8: Results of the attacks

Team	Agent	Number of attacks
Blue (100 IR sensor ATT)	Blue 1	102
	Blue 2	225
Purple (100 IR sensor)	Purple 1	51
	Purple 2	52

From Table. 4.8, it is evident that the agents with an attention mechanism, both individually and as a team, exhibit more attack behavior. This provides them with increased opportunities to secure winning rewards.

The average distance (D) between the ball and two agents in 500 instances of loss was recorded.The instances and calculation method are illustrated in the fig. 4.16.

Based on the data from these 500 instances, a box plot can be generated, as depicted in Fig. 4.17. It is evident that when the team with 100 IR sensors experiences losses, they tend to maintain a considerable distance from the ball. This suggests that agents from the team with the attention mechanism (100 IR sensors ATT) often prefer hitting the ball to a location far from their opponents.

The distance between each agent and its teammate in 500 instances of agent hits was recorded. In a two-player collaborative adversarial game, whether it’s volleyball, tennis, badminton, or similar, having players positioned too closely is not an optimal choice. A close proximity between two players implies a reduced ability to cover a wider range of balls coming from the opponents. The box plot results are shown in Fig. 4.18.

From Fig. 4.18, it can be observed that the team utilizing the attention mechanism (100 IR sensors ATT) maintains a consistent distance with its teammate when receiving the ball.This indicates that the two agents have learned to ensure a greater range of catching the ball by keeping their distance.

On the other hand, the team without the attention mechanism (100 IR sensors) does not maintain a distance with their teammate; due to the lack of an attention mechanism, they are unable to process additional inputs such as the position of their teammates and opponents. This makes them tend to move together and unable to spread out their stance to get a greater range of hits.

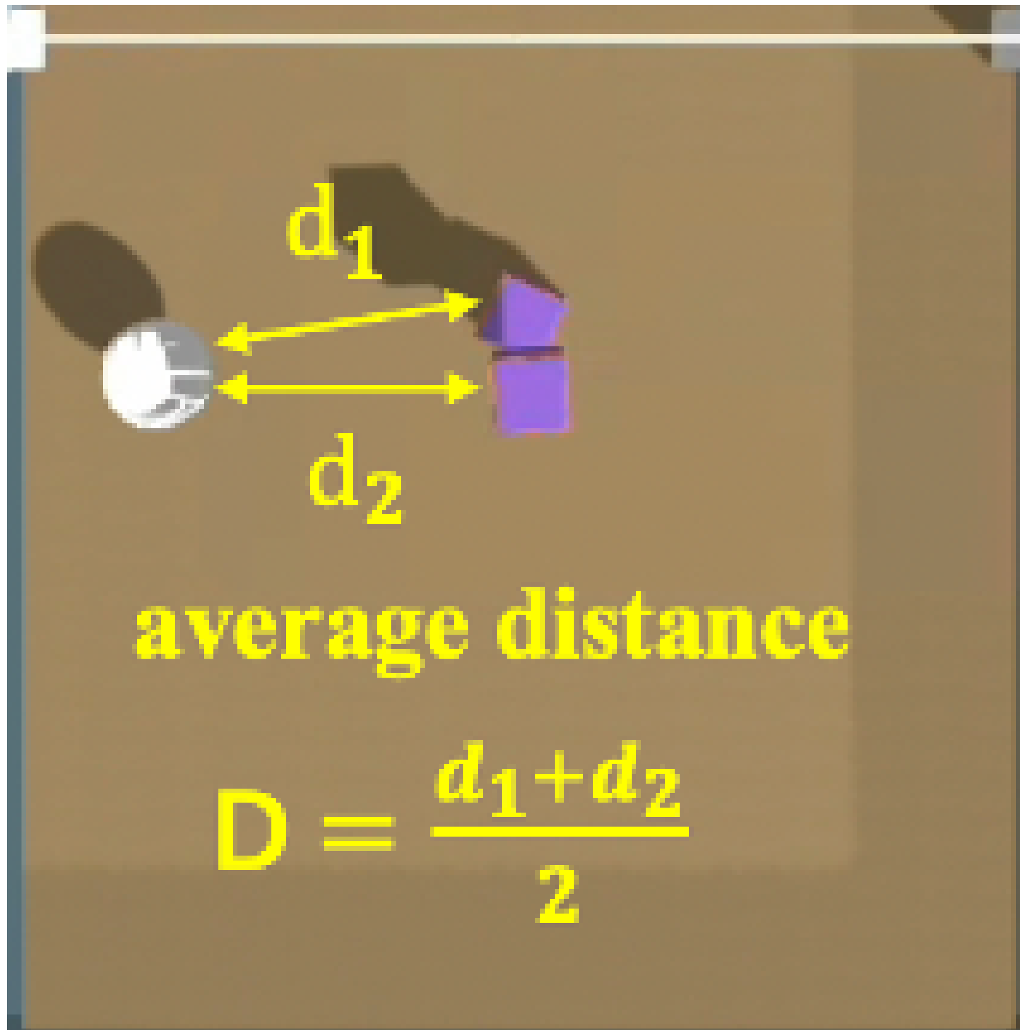


Figure 4.16: The instance of loss

Moreover, the lack of improvement in the performance of the agent with 6 IR sensors suggests that the information provided by only 6 IR sensors is not sufficient, and the attention mechanism did not gain enough valuable information to enhance the scores. On the other hand, the improvement in Elo scores for agents with 20 and 100 IR sensors after the addition of the attention mechanism indicates that the additional information input is not meaningless. It suggests that a simple PPO neural network might struggle to effectively process this information, leading to suboptimal Elo scores.

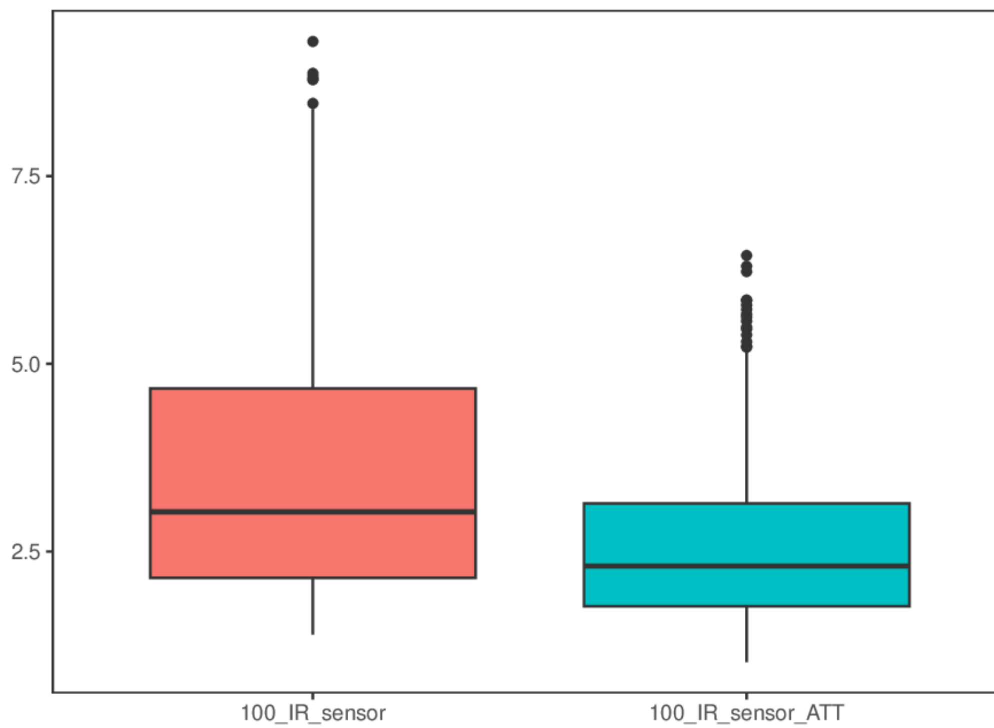


Figure 4.17: Average distance between two agents during losses

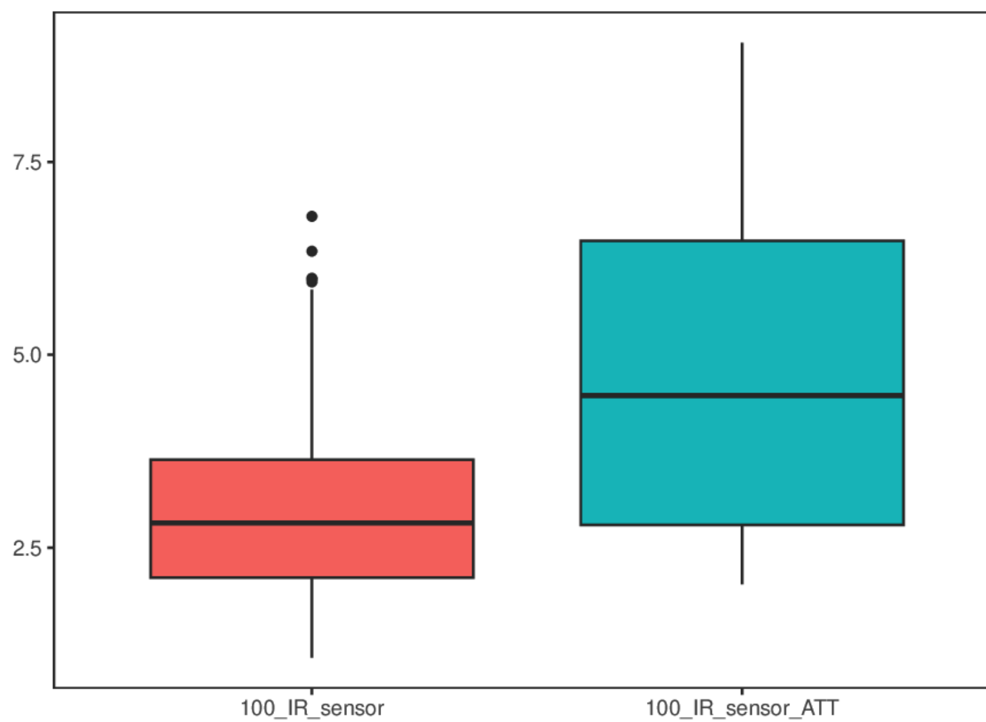


Figure 4.18: The distance between two agents when receiving the ball

4.6 Conclusions

In this study, we employed a combination of imitation learning and reinforcement learning to address the sparse reward problem encountered by a multi-agent system during beach volleyball gameplay. Following training, the PPO-GAIL, integrating both RL and IL, successfully mitigated the sparse reward problem by counteracting the adverse effects of dense reward settings. Controllers trained using our approach achieved higher scores in the Elo rating system compared to the control group, which solely relied on reinforcement learning. Furthermore, agents controlled by our strategy demonstrated superior performance, winning 473 out of 500 competitions against the control group.

Additionally, an attention mechanism was incorporated to alleviate the impact of additional IR sensors on the agents' learning strategies. The results indicated that agents trained with the combination of attention mechanisms attained higher Elo scores than those trained without attention mechanisms. Moreover, they exhibited improved performance in both adversarial and cooperative behaviors.

Chapter 5

Collective Transport Behavior in a Robotic Swarm with Hierarchical Imitation Learning

5.1 Introduction

Swarm robotics (SR) [8] is a field of inspired by the collective behaviors observed in social insects, involving the design and control of multiple simple robots collaborating to achieve tasks beyond the capabilities of individual units. A robotic swarm functions in a distributed and self-organizing manner, which means that there is no central robot directing the others, and the robots are unaware of global information [9]. SR systems are designed to exhibit three properties: fault tolerance, scalability, and flexibility [88]. Fault tolerance is the ability to complete a task even when robots are lost. Scalability is the ability to accomplish a given task by using different numbers of robots. Flexibility is the ability of an SR system to overcome changes in tasks. These characteristics provide robotic swarms with a wide range of possible applications, including searching and transportation.

A major research direction in SR involves designing suitable controllers for swarm robots. Automatic design [106] has been recognized as a promising approach for developing robotic swarm control strategies. In the automatic design method, the design problem is transformed into an optimization problem, and then an optimization algorithm is utilized to automatically generate the required control strategy. One of the main subdomains of automatic design methods is reinforcement learning (RL) [51]. This approach involves an agent taking actions in an environment to achieve a goal and receiving feedback in the form of rewards or penalties. The goal of RL is to learn an optimal policy that maximizes long-term cumulative rewards. Therefore, the performance of RL depends heavily on the design of the reward functions. However, RL suffers from the sparse reward problem in

some complex tasks [107]. The sparse reward problem refers to cases in which it is difficult for the agent to obtain rewards during exploration, resulting in slow learning speed and poor performance.

Intuitive solutions include reward shaping [108], in which denser rewards are designed for various agent behaviors, and hierarchical reinforcement learning (HRL) [109] performed by decomposing complex tasks into several simple subtasks. However, reward shaping requires the designer to have prior knowledge, and it is difficult to ensure that different rewards do not affect each other. Moreover, it is easy for agents to exhibit specific behaviors to obtain denser rewards instead of completing tasks. In contrast, in the conventional HRL approach, subcontrollers are first trained to complete the subtasks. The high-level controller is then trained to output different subcontrollers to complete the original complex task. However, when the training environment changes, the subcontroller often needs to be retrained to ensure that it can complete the corresponding subtask in the new environment, which significantly limits the versatility of this method.

Imitation learning (IL) [61] is another approach for solving the sparse reward problem. In IL, agents can obtain the information they require from expert demonstrations, such as determining what action to take in a given state. In this case, agents can learn what to do by imitating expert demonstrations, even in a sparse reward environment. However, expert demonstrations tend to be generated by human players rather than being spontaneously generated by agents, which is undesirable in swarm robotics.

In this chapter, we propose the hierarchical imitation learning (HIL) method, which combines hierarchical and imitation learning. This method is designed to train the controller of a robotic swarm to overcome the sparse reward problem. We present a key-to-door transport task where agents must locate a button to open the door and push food to the goal area outside the door. Computer simulations were conducted to evaluate the effectiveness of the proposed method. The results demonstrate that the HIL method successfully addresses the sparse reward problem compared to conventional reinforcement learning approaches. Additionally, when subjected to changes in the training environment, controllers trained using the proposed method exhibit more stable performance than those trained using conventional hierarchical learning approaches.

5.2 Related Works

Most available studies on automatic design in swarm robotics originate from evolutionary robotics, in which control policies are optimized using artificial evolution. A wide range of tasks, such as aggregation [109], flocking [110], collective transport [79], and work allocation [39], have been effectively addressed by evolutionary robotics. However, artificial evolution suffers from a prohibitively high computational cost when the dimensionality of

the parameters to be optimized is large.

Another approach in automatic design is deep reinforcement learning (DRL) [52], where the policy is described as deep neural networks optimized using gradient descent methods, which require much fewer computational resources than evolutionary robotics approaches [92]. In recent years, based on the development and application of deep neural networks, deep reinforcement learning (DRL) [111], a combination of deep learning and reinforcement learning, has seen considerable success in many applications, such as Go, Deepmind's StarCraft II, and Atari-2600 games.

Sequential decision making is a term widely used in reinforcement learning to describe problems in which agents must complete a series of tasks in a specific order. Such scenarios occur in a range of real-world tasks [112]. When the reward function is designed to provide rewards for task completion, it is difficult for agents to obtain rewards by randomly exploring the environment. Moreover, agents struggle to connect their current actions with future rewards. For example, in [55], although DRL achieved better results than human players in many games, like Breakout and Pong, it performed poorly in some games with complex environments, such as Montezuma's Revenge. In this game, players must avoid various traps and enemies, decrypt puzzles and find the keys for each layer to explore a huge pyramid composed of multiple layers.

5.3 Methods

In this section, the conventional HRL and proposed HIL approaches are first described. Then, the DRL and IL algorithms utilized in this study are introduced.

5.3.1 Hierarchical Reinforcement Learning

Hierarchical learning refers to a paradigm in machine learning and artificial intelligence in which complex tasks are broken down into a hierarchy of simpler subtasks. This approach was inspired by the idea that many real-world problems involve a natural decomposition into multiple levels of abstraction, allowing for more efficient learning and decision-making. Hierarchical learning can be applied in various fields, such as reinforcement learning [109].

In the hierarchical reinforcement learning method, subcontrollers are trained to perform different subtasks with distinct reward functions. A high-level controller is then trained to activate different subcontrollers to accomplish the complex task. The output of the high-level controller is the switching signal for each subcontroller. In other words, the high-level controller completes the original task by outputting the behaviors of the different trained subcontrollers. However, there may be limitations to this flexibility; when the environment in which the high-level controller was trained changes, it is challenging

to guarantee that the trained subcontrollers can still reliably complete the corresponding subtasks in the new environment.

Furthermore, because the output is an activation signal, it is challenging for the high-level controller to generate new collective behaviors by activating different trained subcontrollers, significantly limiting the adaptability of this method.

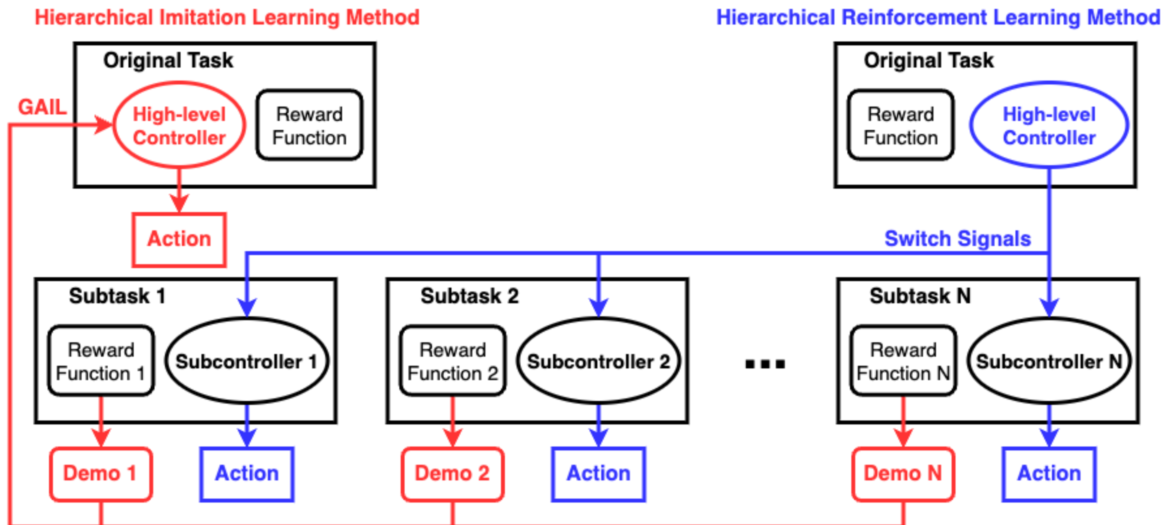


Figure 5.1: Paradigms of the HRL and HIL methods. The same settings in the two methods are marked in black. The differences are marked in red and blue, where red represents the HIL method and blue represents the HRL method.

5.3.2 Hierarchical Imitation Learning

In this study, we utilized the hierarchical imitation learning (HIL) approach, which combines hierarchical and imitation learning, to overcome the sparse reward problem. The paradigms of the HRL and HIL methods are illustrated in Fig. 5.1. Similar to hierarchical learning, the original task is first decomposed into multiple subtasks using the proposed method. Different reward functions are then used to train the subcontrollers to complete the corresponding subtasks. The difference is that the proposed method uses the imitation learning algorithm to train the high-level controller. The trained subcontrollers act as experts recording demonstrations. Finally, the demonstrations of all subcontrollers are gathered together to use the imitation learning algorithm to train the high-level controller. The high-level controller directly outputs actions instead of the activation signals of the subcontrollers. Even if the training environment changes, under the guidance of the subcontroller demonstrations and the reward function, the proposed method can still train the high-level controller to complete the task in the new environment.

5.4 Experiment

The experiments were executed on a computer using the Ubuntu 20.04.6 LTS operating system equipped with an Nvidia GeForce RTX 3070 Ti GPU, an AMD Ryzen 9 3950x CPU, and 128 GB of memory.

5.4.1 The Key-to-door Transport Task

In this study, a key-to-door transport task was designed as the experimental subject. The robots must carry food to the target area; however, a door controlled by a button separates them from the target room. Therefore, the robots must find the button in the room and press it to open the door before carrying food to the goal area. Computer simulations were conducted using Unity3D. An overview of the experimental environment is presented in Fig. 5.2. The button and door are located at the fixed positions. Notably, the button requires more than four robots to press, and the food is heavy and requires more than three robots to move it. This means that the robotic swarm must generate collective behavior to accomplish the task.

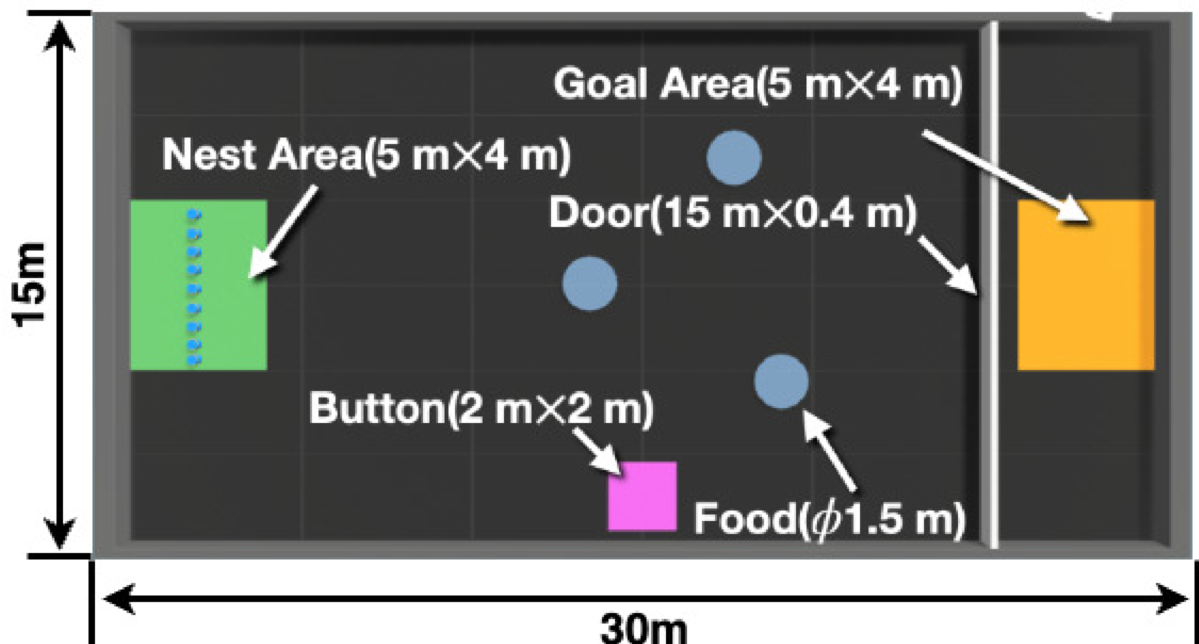


Figure 5.2: Top view of the environment

Fig. 5.3 illustrates the procedure of the designed task. At the start of each episode, nine robots were initialized in the nest area, and three food items were randomly generated. Subsequently, the robots explored the environment to locate the button. The door opened when more than four robots stood on the button. Finally, the robots clustered together to push all food items to the goal area.

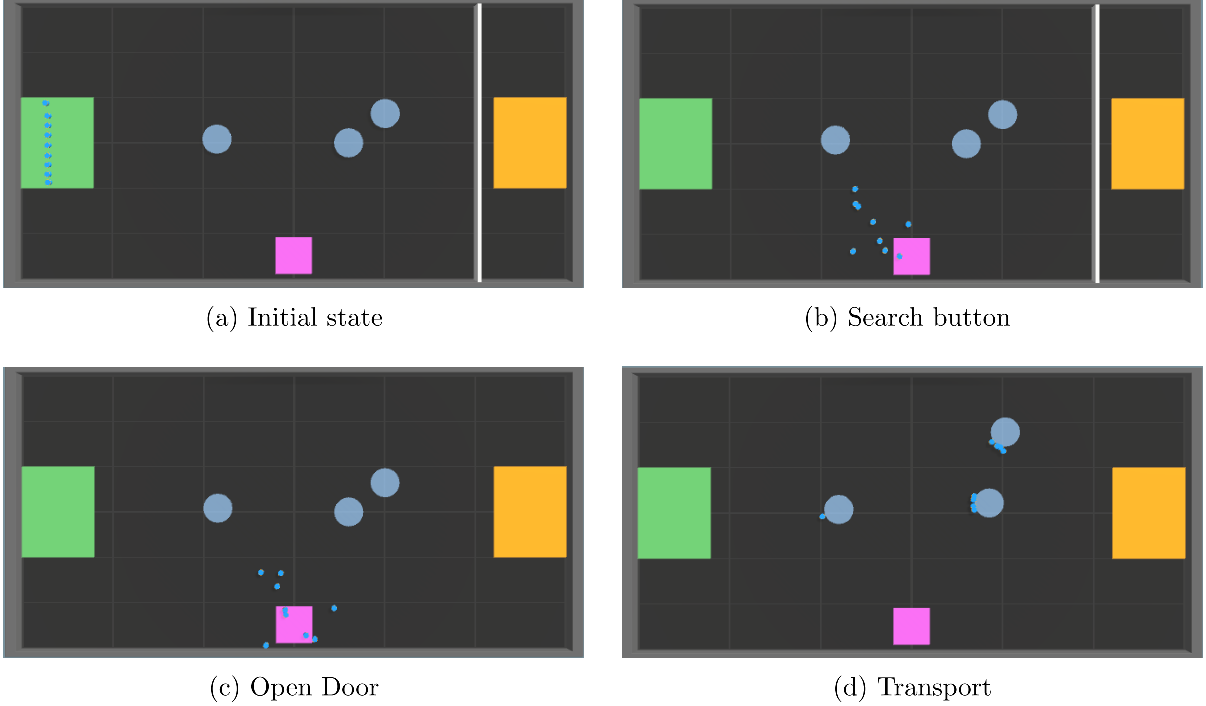

(a) Initial state
(b) Search button
(c) Open Door
(d) Transport

Figure 5.3: Procedure of key-to-door task

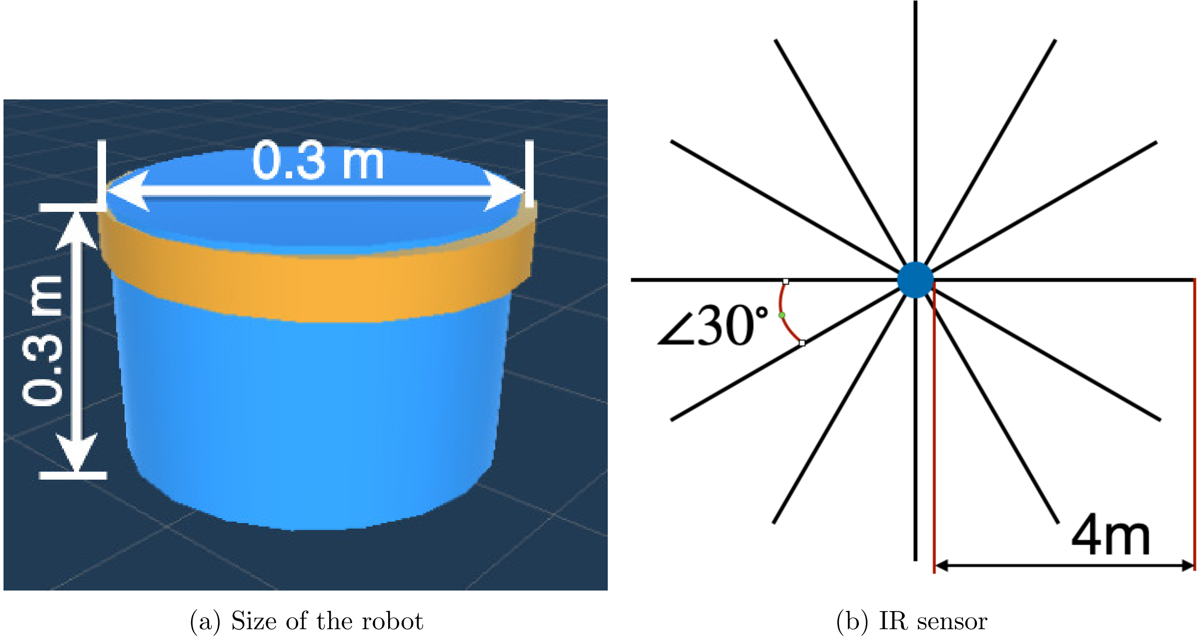
5.4.2 Robot Specifications

The design of the robots is depicted in Fig. 5.4. Each robot has a diameter and height of 0.3 m. The robot's observation is represented by vector inputs, which include three components: the velocity of the robot, direction information from the electronic compass, and input from distance sensors. The electronic compass provides the rotation angle of the robot, while 12 distance sensors, with a range of 4 m and an interval of 30° (shown in Fig. 5.4b), detect the distance between the robot and surrounding objects. The robot is capable of executing seven types of discrete actions: moving forward, moving backward, moving left, moving right, turning clockwise, turning counterclockwise, and remaining still.

5.4.3 Reward Settings

In this study, three types of reward functions were designed. The sparse reward setting was conducted according to R_{sparse}, shown in Eq. 5.1:

$$R_{sparse} = \begin{cases} 2.5, & \text{if the door is open or any food} \\ & \text{is carried to the goal area,} \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$



(a) Size of the robot

(b) IR sensor

Figure 5.4: Design of the robot

The robot receives a rewards of 2.5 if it achieves any subtask. Dense reward settings were also implemented. $R_{opendoor}$, shown in Eq. 5.2, was designed for the door-opening subtask:

$$R_{opendoor} = \begin{cases} 1, & \text{if the robot touches the button,} \\ 0, & \text{otherwise.} \end{cases} \tag{5.2}$$

Because the button requires more than four robots to press it together, the robots are encouraged to cluster on the button by receiving a reward of 1 whenever they touch it. Furthermore, the reward designed for the transport subtask was applied in $R_{transport}$, as expressed in Eq. 5.3:

$$R_{transport} = \begin{cases} \boldsymbol{v_F} \cdot \boldsymbol{d_{FtoG}}, & \text{if the touching food is moving,} \\ 0, & \text{otherwise.} \end{cases} \tag{5.3}$$

where $\boldsymbol{v_F}$ is a vector representing the velocity of the food, and $\boldsymbol{d_{FtoG}}$ is a unit vector representing the direction from the food to the goal area. The robots were rewarded if they pushed the food in the direction of the goal area.

5.4.4 Implementation of the Conventional PPO, HRL, and HIL Methods

In the hierarchical approach, the original task is decomposed into two subtasks: the door-opening task and transport task, shown in Fig. 5.5. The sub-controllers were trained

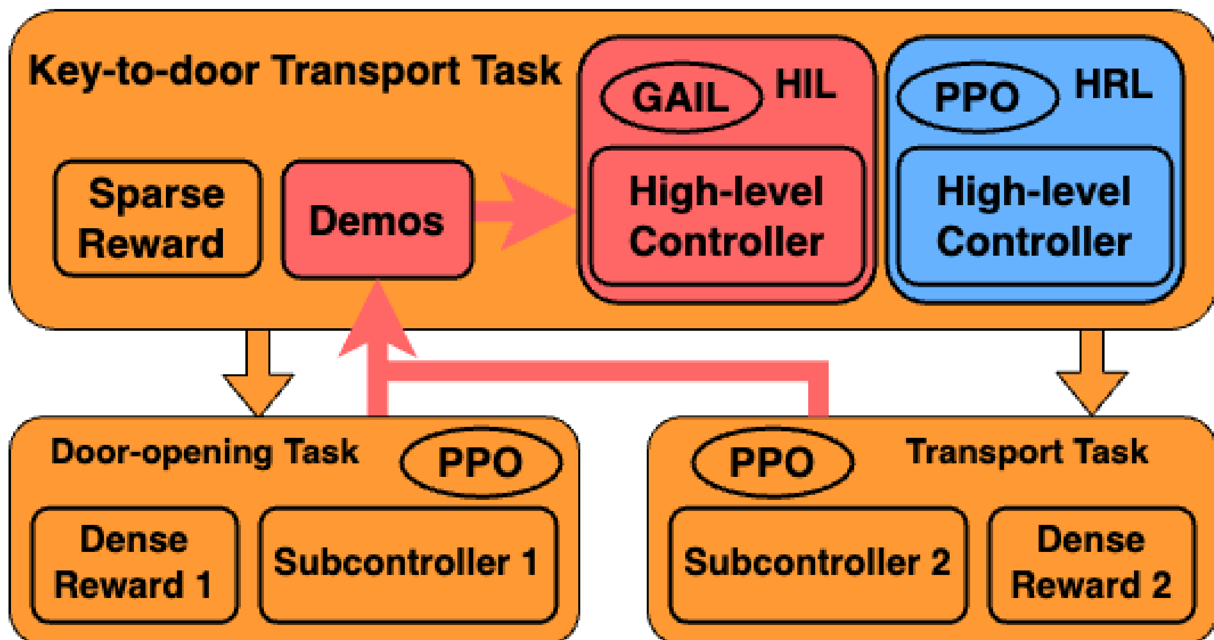


Figure 5.5: Implementation of the HRL and HIL methods

using PPO and dense rewards.

In the door-opening task, shown in Fig. 5.6, the robotic swarm was trained to cluster on the button to open the door, and the episode will be terminated if the door is open.

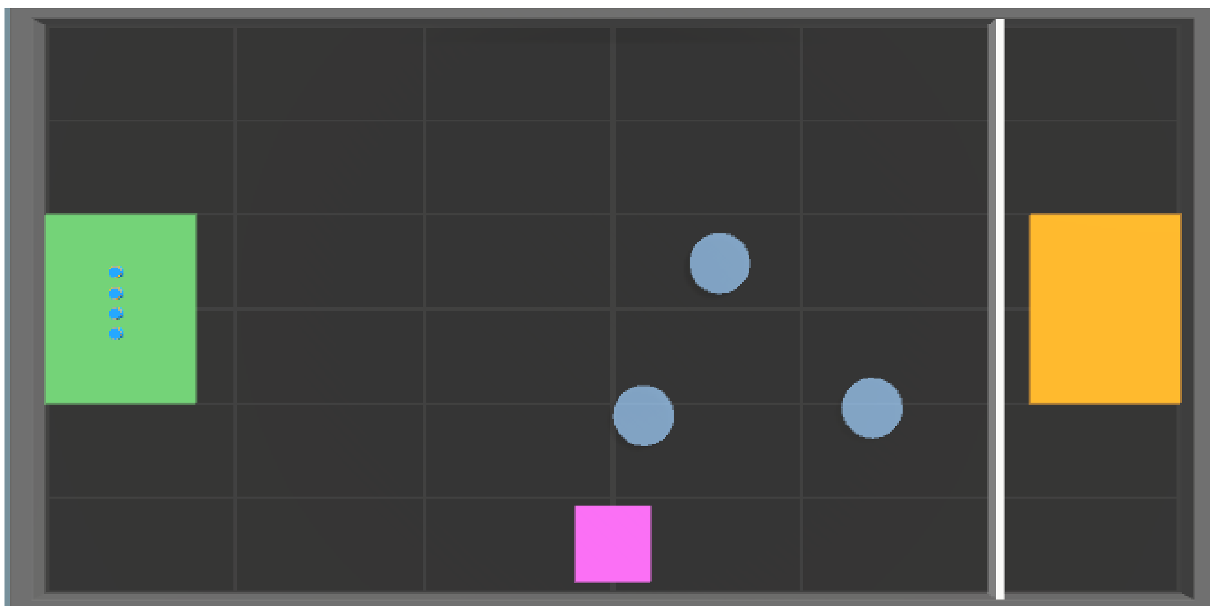


Figure 5.6: Door-opening task

In the transport task, shown in Fig. 5.7, the robotic swarm was trained to push all food to the goal area when the door was open.

In this study, the conventional PPO, HRL, and HIL methods were implemented to

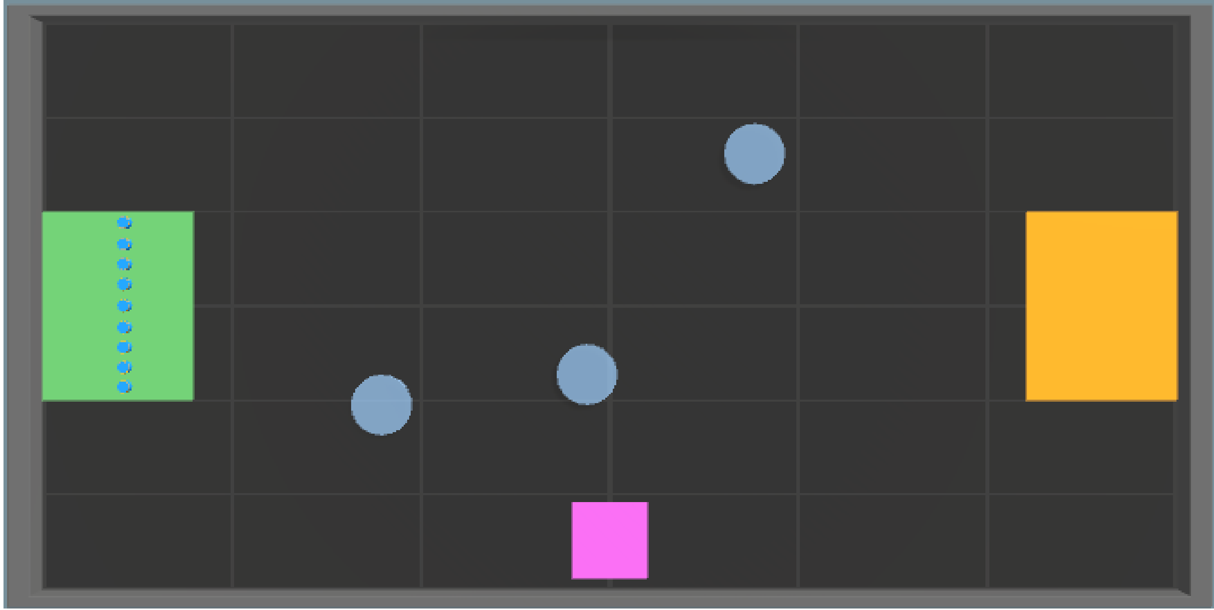


Figure 5.7: Transport task

train the controllers of a robotic swarm to perform a key-to-door transport task. Because the robots in the SR system were homogenous, all robots shared the same training modules for each training method.

- **The Conventional PPO Method**: In the conventional PPO approach, the controller is trained using PPO to directly accomplish the key-to-door task.
- **The HRL Method**: In the HRL method, the subcontrollers trained for the previous two subtasks were deployed on the robots. The high-level controller was trained using PPO and sparse rewards to output switching signals to activate different subcontrollers.
- **The HIL Method**: In the proposed HIL method, subcontrollers completing the subtasks were recorded as expert demonstrations composed of state-action pairs. The high-level controller was trained using GAIL and sparse rewards to output actions to accomplish the original task.

5.4.5 Experimental Settings

Several experiments were conducted in this study, and their settings are listed in Table 5.1. In Experiments 1 and 2, the conventional PPO was applied with sparse and dense reward settings, respectively, to train a robotic swarm for accomplishing the key-to-door transport task. Experiments 3 and 4 implemented the HRL and HIL methods. Initially, two subcontrollers were trained with PPO and the corresponding dense rewards to achieve

the two subtasks. In Experiment 3, PPO and sparse rewards were used to train the high-level controller to activate the trained subcontrollers. In contrast, in Experiment 4, the process of the trained subcontrollers completing the subtasks was recorded as demos, and together with the sparse reward, the result was applied to GAIL to train the high-level controller to output actions. It's important to note that in the first four experiments, the position of the button remained fixed, irrespective of the original task or the subtasks. Experiments 5 and 6 were conducted to address changes in the training environment. In these experiments, the HRL and HIL methods were employed, and variations in the button position settings led to changes in the environment during training. While the button position remained fixed during the training of the subcontrollers, it appeared in random positions when training the high-level controller.

Table 5.1: Experimental Settings

Experiment	Training Method	Reward Settings	Button Position
Experiment 1	Conventional PPO	R_{sparse}	Fixed
Experiment 2	Conventional PPO	$R_{opendoor}+R_{transport}$	Fixed
Experiment 3	HRL method	R_{sparse}	Fixed
Experiment 4	HIL method	R_{sparse}	Fixed
Experiment 5	HRL method	R_{sparse}	Fixed for subcontrollers and
Experiment 6	HIL method	R_{sparse}	random for high-level controller

The hyperparameters used in this study are listed in Table 5.2. Each episode was terminated when the robotic swarm either completed the task or when the time reached the maximum episode length of 4000 time steps. Training ceased once the maximum number of time steps (90 million) was reached.

5.5 Results and Discussion

Each experiment consisted of 90,000,000 steps. The average training durations for the conventional PPO, conventional HRL, and proposed HIL methods were 12, 15, and 20 hours, respectively. Episode length was recorded to evaluate the training process of the controllers, as shown in Fig. 5.9. In Figs. 5.9a and Fig. 5.9b, the episode length remains at 4000 steps, indicating that the controller applying the conventional PPO method cannot complete the given task under sparse or dense reward settings. Moreover, simulations were conducted in which the robots deployed the controllers trained in Experiments 1 and 2. Fig. 5.8 shows the trajectories of the robots in one episode. The green line shows that robots trained with conventional PPO and sparse rewards tended to move along walls

Table 5.2: Hyperparameter Settings

Hyperparameter	
Maximum episode length	4000
Maximum step count	90000000
Hidden units	256
Number of layers	2
Batch size	128
Buffer size	2048
Learning rate	0.0003
Epsilon	0.2
Gamma	0.95

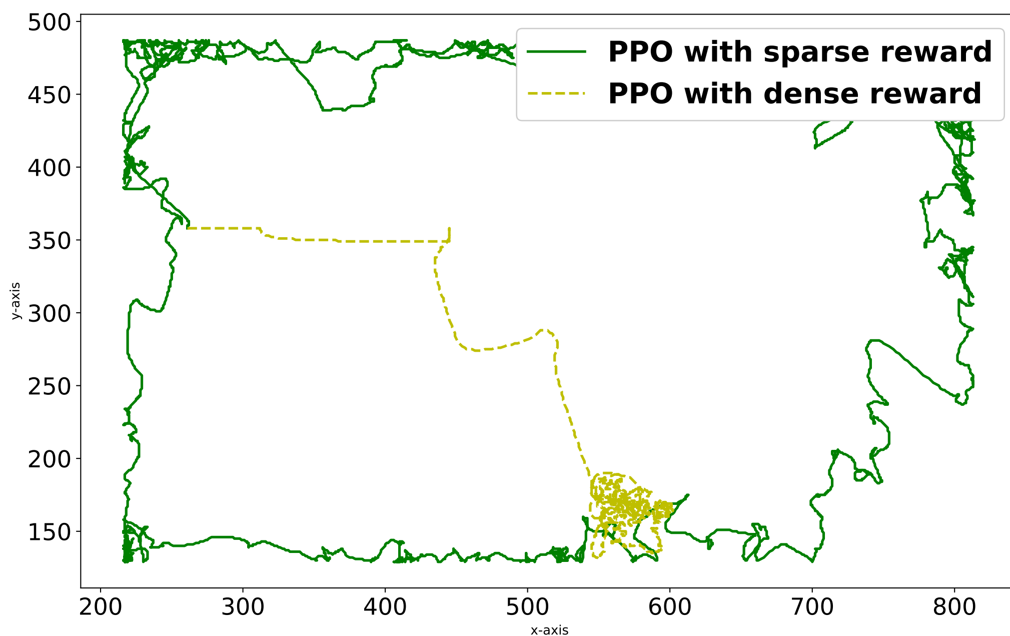
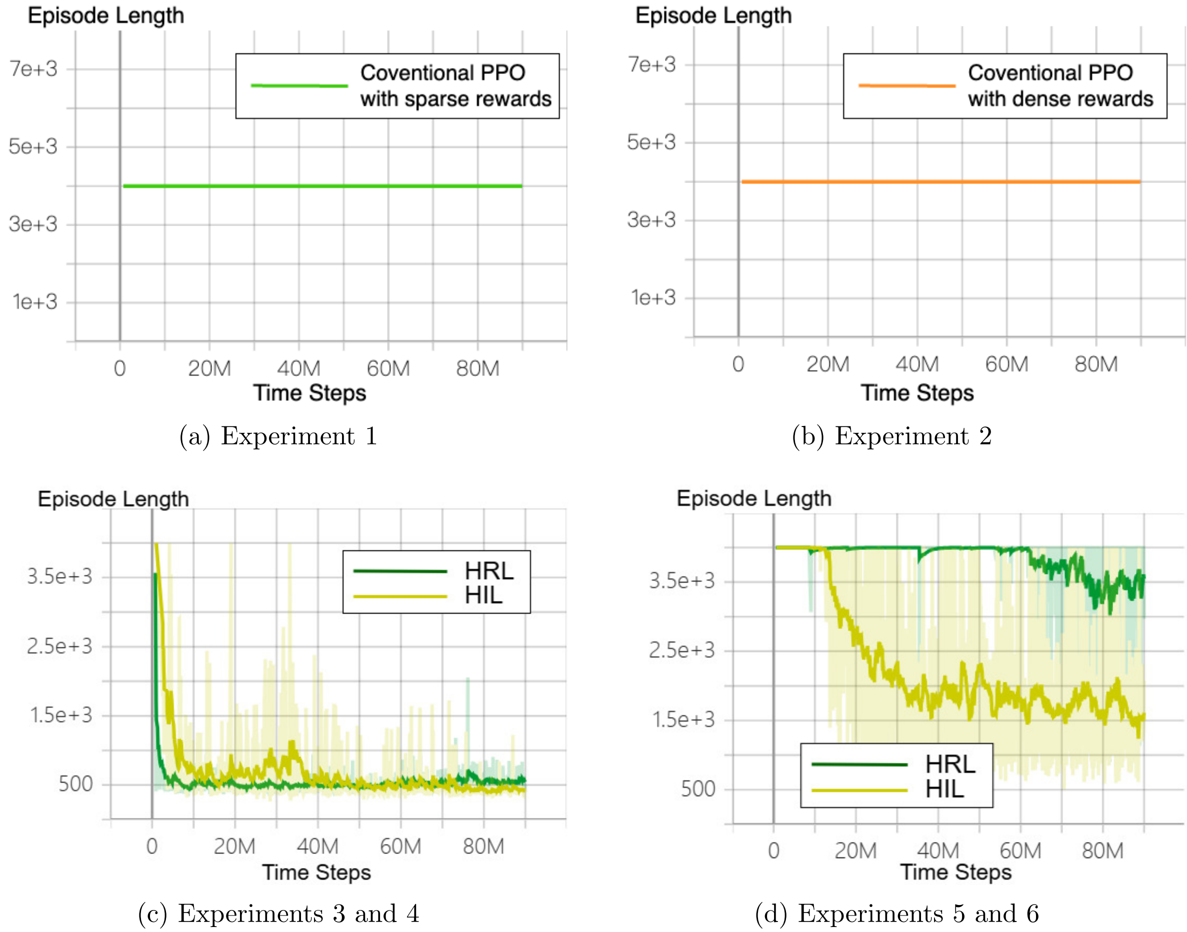


Figure 5.8: Trajectories of a trained robot in Experiments 1 and 2 (contains the data of one episode)

and cluster in corners because the rewards were too sparse to obtain. However, the yellow line shows that robots trained with conventional PPO and dense rewards clustered on the button near the coordinates (550, 150) and did not push the food because the dense reward of touching the button was most easily obtained.

As shown in Fig. 5.9c, the episode length decreased during the training process, con-



(a) Experiment 1

(b) Experiment 2

(c) Experiments 3 and 4

(d) Experiments 5 and 6

Figure 5.9: Results of episode length during the training process

verging to approximately 500 steps at the end of the training, indicating that the HRL and HIL methods successfully trained the robotic swarm to achieve the given task with the sparse reward setting. Additionally, the trajectories of the robots trained in Experiments 3 and 4 were collected for each episode, as shown in Fig. 5.10. The yellow and blue lines show that the robot trained using the HRL and HIL methods first reached the button position near the coordinates (550, 150). The robot then searched for food in the field and delivered it to the target area near the coordinates (850, 300).

Fig. 5.9d shows the training process of Experiments 5 and 6, in which the environment changed when training the high-level controllers. The yellow line indicates that training using the conventional HRL method was extremely slow when the training environment changed. The episode length decreased by approximately 60 million steps in the middle stages of training and finally converged at approximately 3500 steps. Moreover, the blue line shows that the controller trained with the proposed HIL method learned to accomplish the task at approximately 15 million steps, and the episode length converged at approximately 1500 steps.

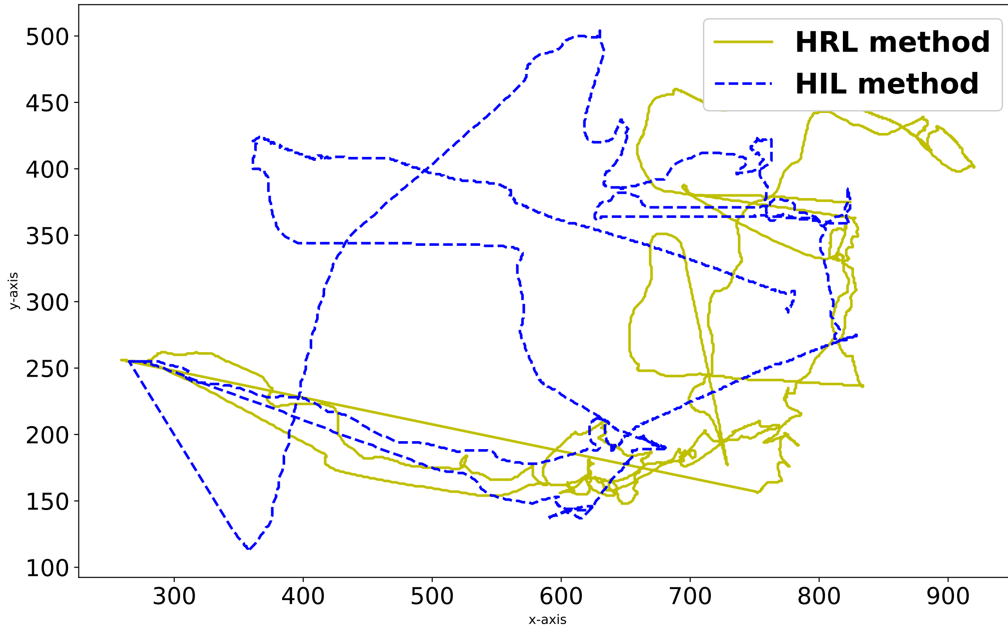


Figure 5.10: Trajectories of a trained robot in Experiments 3 and 4 (contains the data of one episode)

Furthermore, the trajectories of the trained robots in Experiments 5 and 6 were collected for one episode in which the button appeared in a random position, as shown in Fig. 5.11. In Fig. 5.11, the yellow line shows that the robot trained with the HRL method still searched for the button at the coordinates (550, 150) by activating the door-opening subcontroller. However, if the button did not appear nearby, the robot began to move around, became stuck in a corner of the scene, and was ultimately unable to complete the task. The blue line shows that when the training environment changed, the robot trained with the proposed HIL method learned new behaviors to complete the task. Initially, the robot did not move directly to the button's location in the subtask. Instead, it went straight to the scene to find the button and finally pushed the food to the target area to complete the task. The results show that the controller trained using the proposed HIL method performed better than the HRL method in overcoming changes in the training environment.

Additionally, the scalability and flexibility of the controller trained using the proposed HIL method were tested. The test controller was trained in Experiment 4, in which nine robots pressed the fixed button and pushed the cylindrical food items to the goal area. The scalability was tested by conducting simulations with different numbers of trained robots. The simulations were run for 100 trials for each group size. The results for the

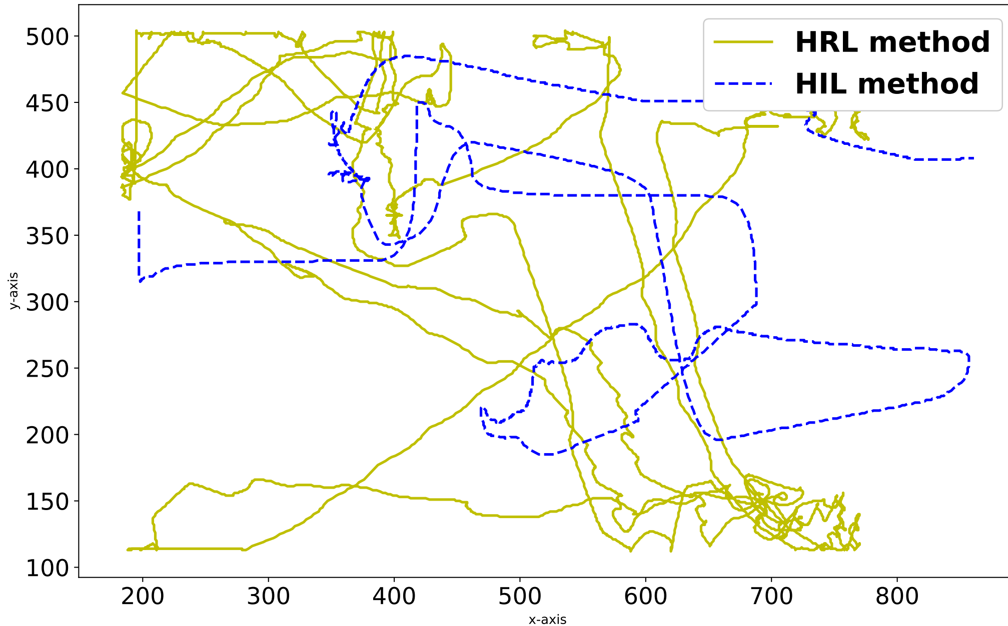


Figure 5.11: Trajectories of a trained robot in Experiments 5 and 6 (contains the data of one episode)

episode lengths are shown in Fig. 5.12. The trained controller could complete the task with similar performance when using 4, 6, 9, and 18 robots. When the group size reached 36 robots, the performance became unstable.

The flexibility was tested by executing simulations with different food shapes, as shown in Fig. 5.13, and different numbers of food items, that is, 2, 3, 4, 6, and 8.

The results of 100 trials for each shape and amount of food are shown in Fig. 5.14. The results in Fig. 5.14a show that the trained controller performed similarly for different food shapes. In Fig. 5.14b, the episode length increased proportionally with an increase in the amount of food. When the number of food items exceeded six, the performance of the trained controller became unstable because of the overcrowded scene.

5.6 Conclusions

This study proposes the hierarchical imitation learning method to achieve the key-to-door transport task with sparse rewards. In the proposed method, the original task is partitioned into subtasks and subcontrollers are trained to solve each subtask. The high-level controller is then trained using sparse rewards and GAIL, in which the subcontrollers collect the demos used by GAIL. The proposed method was compared with PPO, the

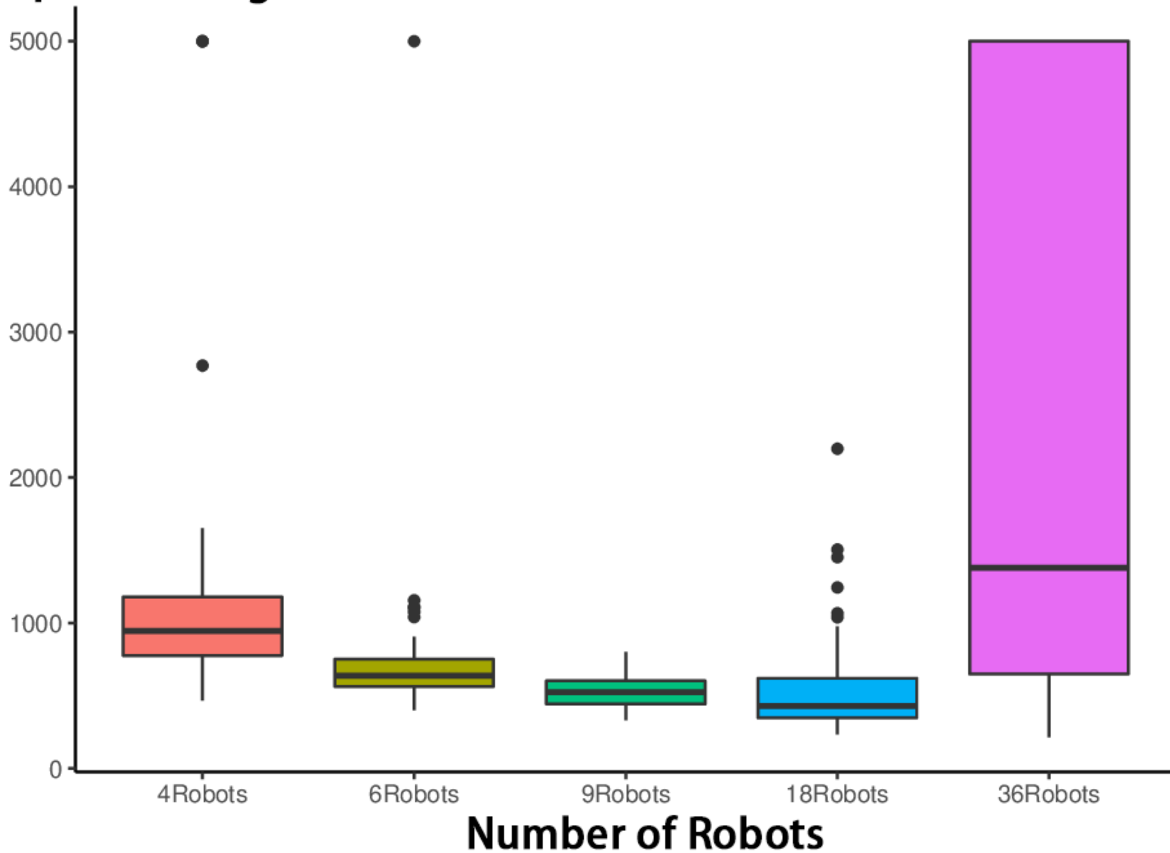
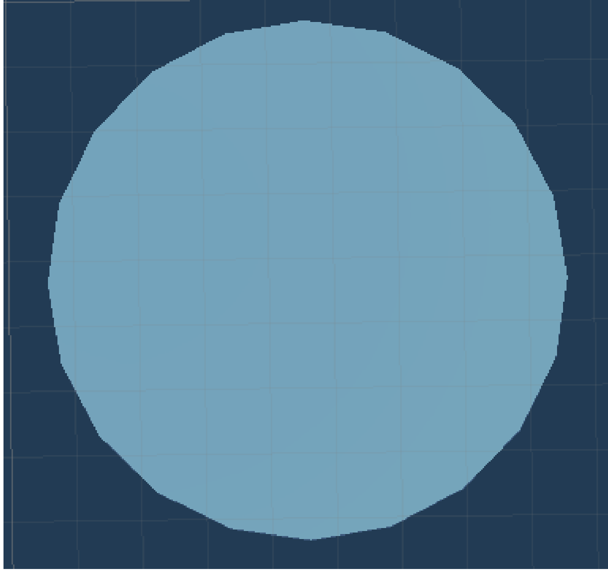


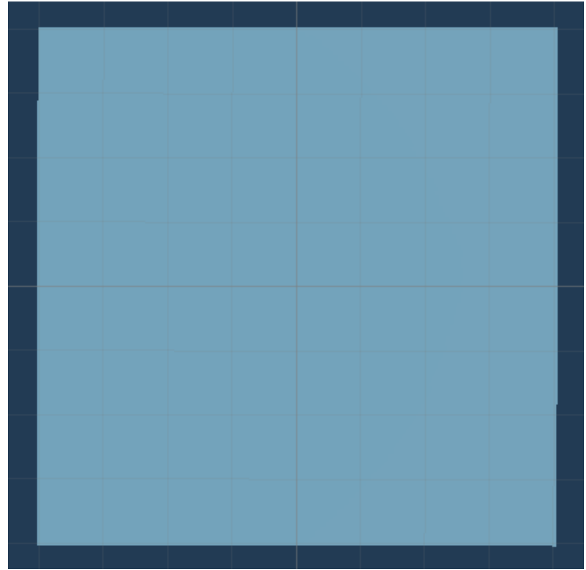
Figure 5.12: Results of the scalability test

conventional RL algorithm. The results showed that the proposed approach can effectively train a controller for a robotic swarm in a key-to-door task with sparse rewards.

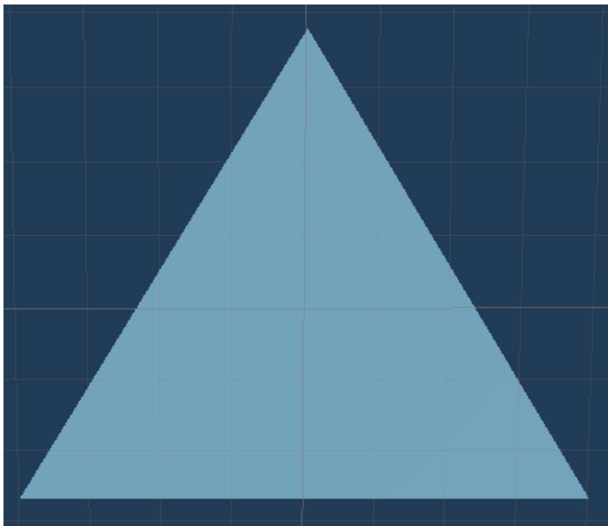
Moreover, the ability of the proposed method to overcome changes during training was tested by changing the button position in the training environment. The results showed that compared with the HRL method, in which the high-level controller can only activate subcontrollers, this method, in which the high-level controller directly outputs actions, can better adapt to changes in the environment during the training process. Additionally, the scalability and flexibility of the controller trained in the proposed method were tested using different numbers of robots, food shapes, and numbers of food items. The results showed that the controller trained using the proposed HIL method exhibited good scalability and flexibility.



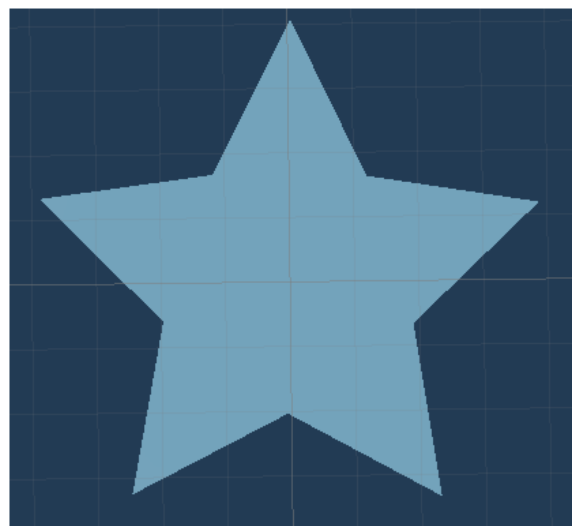
(a) Cylinder



(b) Cube

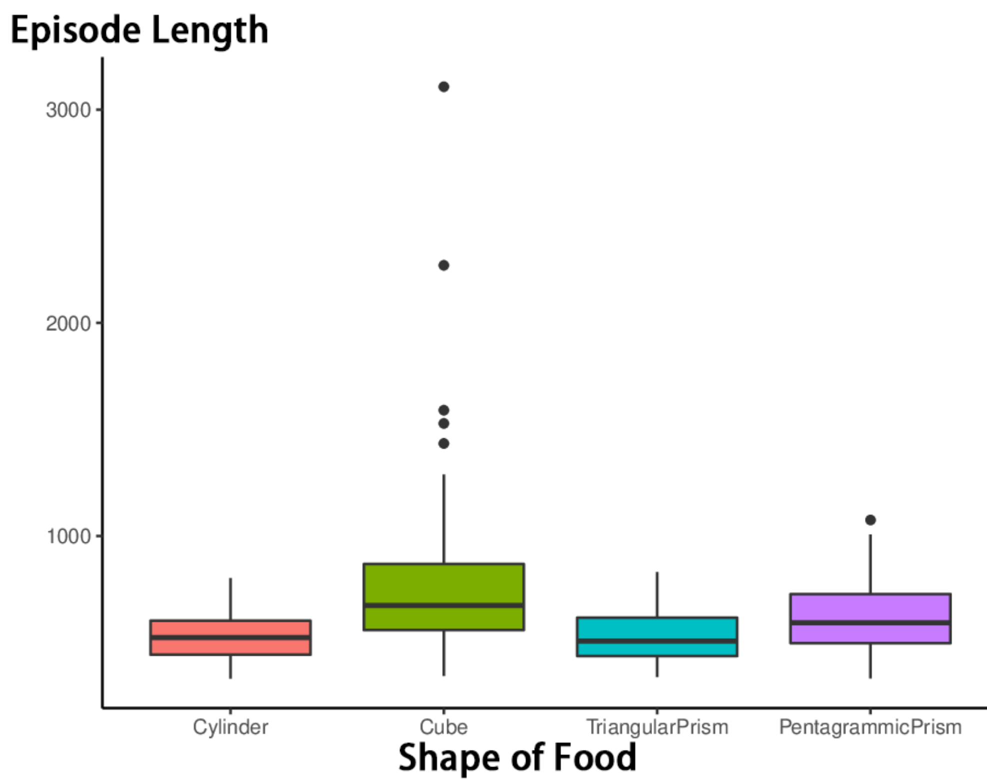


(c) Triangular Prism

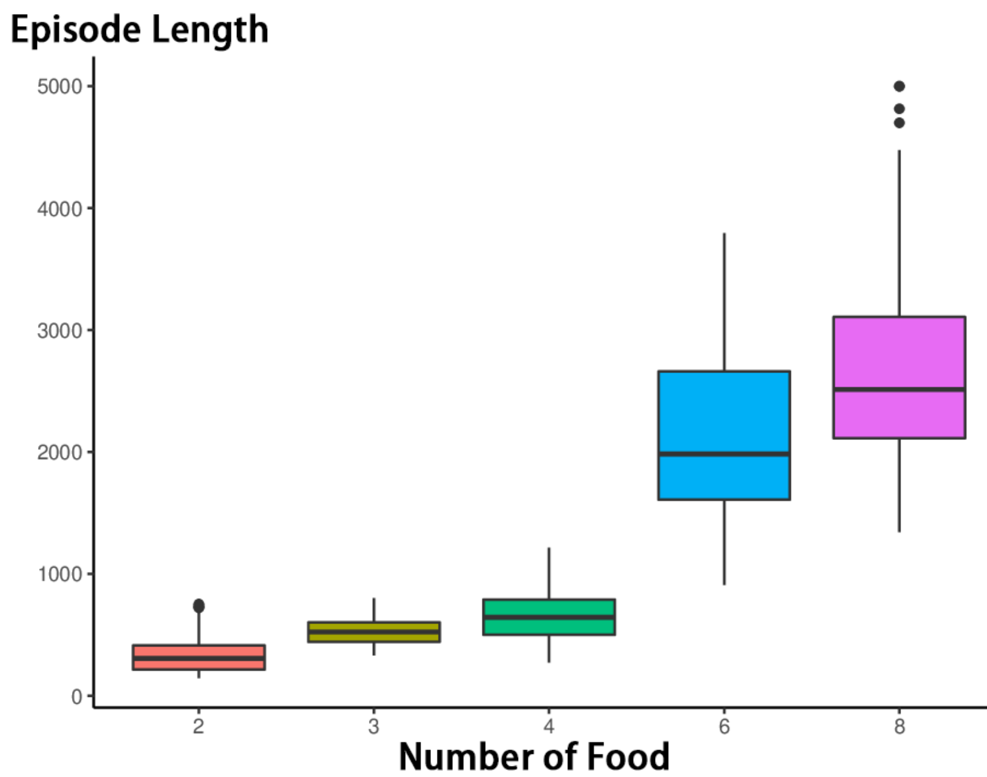


(d) Pentagrammic Prism

Figure 5.13: Food shapes



(a) Three different food shapes (3 food items were present in the trial)



(b) Different number of food items (cylindrical)

Figure 5.14: Results of the flexibility test

Chapter 6

Conclusions

This thesis explores the development of controllers for multi-agent systems, with a focus on learning-based methods such as evolutionary robotics and deep reinforcement learning. The contribution of this thesis to the field of multi-agent systems can be viewed from three perspectives.

Firstly, this thesis investigates the applicability of evolutionary robotics and deep reinforcement learning for developing controllers for a swarm robotic system with image inputs. Evolutionary algorithms are typically challenged by high-dimensional search spaces, whereas convolutional neural networks are capable of handling image inputs with fewer parameters. To this end, Chapter 3 proposes the use of the deep neuroevolution method that leverages convolutional neural networks to generate collective behaviors with raw camera inputs. The results demonstrate that this method performs better than Deep Q-Learning in most cases, while also exhibiting flexibility, scalability, and fault tolerance.

Secondly, this thesis investigates how the DRL approach can be applied to generating competitive and collective behaviors for MASs. Although DRL has achieved many successes in static environment tasks, it suffers from the sparse reward problem, in which it is difficult for the agent to obtain rewards during exploration, resulting in slow learning speed and poor performance. An intuitive solution is reward shaping, in which denser rewards are designed for various agent behaviors. However, in some complex scenarios, it is practically infeasible to design rewards for specific behaviors of agents due to the vast state and action space. In Chapter 4, the proposed method uses imitation learning to employ the controller trained by reward shaping as an expert to help the agent generate behaviors under sparse rewards. Additionally, the attention mechanism, inspired by cognitive attention, is integrated with DRL to address input redundancy. The results of computer simulations show that the controllers trained in the proposed method overcome the sparse reward problem and performed better than those trained in the conventional DRL method.

Thirdly, this thesis investigates how to improve the flexibility of hierarchical rein-

forcement learning. An alternative solution to the sparse reward problem is Hierarchical reinforcement learning (HRL), in which the original task is decomposed into multiple subtasks. Subcontrollers are first trained to complete corresponding subtasks. The high-level controller is then trained to activate different trained subcontrollers to accomplish the original task. However, when the training environment is changed, it is hard to guarantee that pre-trained subcontrollers can still stably complete the corresponding subtasks. In Chapter 5, a novel method integrating DRL with Imitation Learning (IL) and HRL is proposed to overcome the sparse reward problem. In the proposed method, the original task is partitioned into subtasks and subcontrollers are trained to solve each subtask. The high-level controller is then trained using sparse rewards and GAIL, in which the subcontrollers collect the demos used by GAIL. The collective behaviors are evaluated in a beach volleyball game and a key-to-door transport task. The proposed method is compared with PPO, the conventional RL algorithm. The results show that the proposed approach can effectively train a controller for a robotic swarm in a key-to-door task with sparse rewards. Moreover, the ability of the proposed method to overcome changes during training is tested by changing the button position in the training environment. The results show that compared with the HRL method, in which the high-level controller can only activate subcontrollers, this method, in which the high-level controller directly outputs actions, can better adapt to changes in the environment during the training process. Additionally, the scalability and flexibility of the controller trained in the proposed method are tested using different numbers of robots, food shapes, and numbers of food items. The results show that the controller trained using the proposed HIL method exhibited good scalability and flexibility.

Overall, the proposed methods successfully generate collective behaviors for multi-agent systems and provide promising solutions to input redundancy and sparse reward problems that conventional learning-based methods suffer from.

References

[1] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[2] Nils J Nilsson. *The quest for artificial intelligence.* Cambridge University Press, 2009.

[3] John McCarthy. What is artificial intelligence? 2007.

[4] Kanna Rajan and Alessandro Saffiotti. Towards a science of integrated ai and robotics, 2017.

[5] Robin R Murphy. *Introduction to AI robotics.* MIT press, 2019.

[6] Jacques Ferber and Gerhard Weiss. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-wesley Reading, 1999.

[7] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations.* Cambridge University Press, 2008.

[8] Marco Dorigo, Mauro Birattari, and Manuele Brambilla. Swarm robotics. *Scholarpedia*, 9(1):1463, 2014.

[9] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[10] Ali Dorri, Salil S Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *Ieee Access*, 6:28573–28593, 2018.

[11] Nicholas R Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1:7–38, 1998.

[12] Jun Ota. Multi-agent robot systems as distributed autonomous systems. *Advanced engineering informatics*, 20(1):59–70, 2006.

[13] Sergey Satunin and Eduard Babkin. A multi-agent approach to intelligent transportation systems modeling with combinatorial auctions. *Expert Systems with Applications*, 41(15):6622–6633, 2014.

[14] Victor Lesser, Charles L Ortiz, and Milind Tambe. *Distributed sensor networks: A multiagent perspective*, volume 9. Springer Science & Business Media, 2003.

[15] Joel SE Teo, Eiichi Taniguchi, and Ali Gul Qureshi. Evaluating city logistics measure in e-commerce with multiagent systems. *Procedia-Social and Behavioral Sciences*, 39:349–359, 2012.

[16] Josep M Pujol, Ramon Sangüesa, and Jordi Delgado. Extracting reputation in multi agent systems by means of social network topology. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 467–474, 2002.

[17] Hian Lee Kwa, Jabez Leong Kit, and Roland Bouffanais. Balancing collective exploration and exploitation in multi-agent and multi-robot systems: A review. *Frontiers in Robotics and AI*, 8:771520, 2022.

[18] Daniel S Drew. Multi-agent systems for search and rescue applications. *Current Robotics Reports*, 2:189–200, 2021.

[19] Jin Dai, Alessandro Benini, Hai Lin, Panos J Antsaklis, Matthew J Rutherford, and Kimon P Valavanis. Learning-based formal synthesis of cooperative multi-agent systems with an application to robotic coordination. In *2016 24th Mediterranean Conference on Control and Automation (MED)*, pages 1008–1013. IEEE, 2016.

[20] Bo Chen, Harry H Cheng, and Joe Palen. Integrating mobile agent technology with multi-agent systems for distributed traffic detection and management systems. *Transportation Research Part C: Emerging Technologies*, 17(1):1–10, 2009.

[21] Junjie Hu, Arshad Saleem, Shi You, Lars Nordström, Morten Lind, and Jacob Østergaard. A multi-agent system for distribution grid congestion management with electric vehicles. *Engineering Applications of Artificial Intelligence*, 38:45–58, 2015.

[22] Mortaza Zolfpour-Arokhlo, Ali Selamat, and Siti Zaiton Mohd Hashim. Route planning model of multi-agent system for a supply chain management. *Expert Systems with Applications*, 40(5):1505–1518, 2013.

[23] Chao Yu, Xin Wang, Xin Xu, Minjie Zhang, Hongwei Ge, Jiankang Ren, Liang Sun, Bingcai Chen, and Guozhen Tan. Distributed multiagent coordinated learning

for autonomous driving in highways based on dynamic coordination graphs. *Ieee transactions on intelligent transportation systems*, 21(2):735–748, 2019.

[24] Ioannis N Athanasiadis and Pericles A Mitkas. An agent-based intelligent environmental monitoring system. *Management of Environmental Quality: An International Journal*, 15(3):238–249, 2004.

[25] Sheng-Li Du, Xi-Ming Sun, Ming Cao, and Wei Wang. Pursuing an evader through cooperative relaying in multi-agent surveillance networks. *Automatica*, 83:155–161, 2017.

[26] Daniel Massaguer, Vidhya Balasubramanian, Sharad Mehrotra, and Nalini Venkatasubramanian. Multi-agent simulation of disaster response. In *ATDM workshop in AAMAS*, volume 2006, 2006.

[27] Agostino Forestiero. Multi-agent recommendation system in internet of things. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 772–775. IEEE, 2017.

[28] Ville Könönen. Dynamic pricing based on asymmetric multiagent reinforcement learning. *International journal of intelligent systems*, 21(1):73–98, 2006.

[29] S Srinivasan, Dheeraj Kumar, and Vivek Jaglan. Multi-agent system supply chain management in steel pipe manufacturing. *IJCSI International Journal of Computer Science Issues*, 7(4):30–34, 2010.

[30] Valentin Robu, Han Noot, Han La Poutré, and Willem-Jan Van Schijndel. A multi-agent platform for auction-based allocation of loads in transportation logistics. *Expert Systems with Applications*, 38(4):3483–3491, 2011.

[31] Zied Kechaou, Mohamed Ben Ammar, and Adel M Alimi. A multi-agent based system for sentiment analysis of user-generated content. *International Journal on Artificial Intelligence Tools*, 22(02):1350004, 2013.

[32] Remy Cazabet and Frederic Amblard. Simulate to detect: a multi-agent system for community detection. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, volume 2, pages 402–408. IEEE, 2011.

[33] Weihua Li, Quan Bai, and Minjie Zhang. A multi-agent system for modelling preference-based complex influence diffusion in social networks. *The Computer Journal*, 62(3):430–447, 2019.

[34] Eric Bonabeau, Directeur de Recherches Du Fnrs Marco, Marco Dorigo, Guy Théraulaz, Guy Theraulaz, et al. *Swarm intelligence: from natural to artificial systems*. Number 1. Oxford university press, 1999.

[35] Heiko Hamann, Yara Khaluf, Jean Botev, Mohammad Divband Soorati, Eliseo Ferrante, Oliver Kosak, Jean-Marc Montanier, Sanaz Mostaghim, Richard Redpath, Jon Timmis, et al. Hybrid societies: challenges and perspectives in the design of collective behavior in self-organizing systems. *Frontiers in Robotics and AI*, 3:14, 2016.

[36] KN McGuire, Christophe De Wagter, Karl Tuyls, HJ Kappen, and Guido CHE de Croon. Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment. *Science Robotics*, 4(35):eaaw9710, 2019.

[37] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. Swarm robotic behaviors and current applications. *Frontiers in Robotics and AI*, 7:36, 2020.

[38] Luneque Silva Junior and Nadia Nedjah. Efficient strategy for collective navigation control in swarm robotics. *Procedia Computer Science*, 80:814–823, 2016.

[39] Yufei Wei, Motoaki Hiraga, Kazuhiro Ohkura, and Zlatan Car. Autonomous task allocation by artificial evolution for robotic swarms in complex tasks. *Artificial Life and Robotics*, 24:127–134, 2019.

[40] Thomas Sütterlin, Simone Huber, Hartmut Dickhaus, and Niels Grabe. Modeling multi-cellular behavior in epidermal tissue homeostasis via finite state machines in multi-agent systems. *Bioinformatics*, 25(16):2057–2063, 2009.

[41] Ramiro A Agis, Sebastian Gottifredi, and Alejandro J García. An event-driven behavior trees extension to facilitate non-player multi-agent coordination in video games. *Expert Systems with Applications*, 155:113457, 2020.

[42] Maria Caridi and Sergio Cavalieri. Multi-agent systems in production planning and control: an overview. *Production Planning & Control*, 15(2):106–118, 2004.

[43] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[44] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[46] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[47] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[48] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[49] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[50] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[51] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[52] Maximilian Hüttenrauch, Sosic Adrian, Gerhard Neumann, et al. Deep reinforcement learning for swarm systems. *Journal of Machine Learning Research*, 20(54):1–31, 2019.

[53] Yufei Wei, Xiaotong Nie, Motoaki Hiraga, Kazuhiro Ohkura, and Zlatan Car. Developing end-to-end control policies for robotic swarms using deep q-learning. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 23(5):920–927, 2019.

[54] Toshiyuki Yasuda and Kazuhiro Ohkura. Sharing experience for behavior generation of real swarm robot systems using deep reinforcement learning. *Journal of Robotics and Mechatronics*, 31(4):520–525, 2019.

[55] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[56] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[57] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[58] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.

[59] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[60] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[61] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[62] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

[63] Andrew Y Ng, Stuart Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000.

[64] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[65] Nathan D Ratliff, J Andrew Bagnell, and Martin A Zinkevich. Maximum margin planning. In *Proceedings of the 23rd international conference on Machine learning*, pages 729–736, 2006.

[66] Edouard Klein, Matthieu Geist, Bilal Piot, and Olivier Pietquin. Inverse reinforcement learning through structured classification. *Advances in neural information processing systems*, 25, 2012.

[67] Chen Xia and Abdelkader El Kamel. Neural inverse reinforcement learning in autonomous navigation. *Robotics and Autonomous Systems*, 84:1–14, 2016.

[68] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, Anind K Dey, et al. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

[69] Abdeslam Boularias, Jens Kober, and Jan Peters. Relative entropy inverse reinforcement learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 182–189. JMLR Workshop and Conference Proceedings, 2011.

[70] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. Deep inverse reinforcement learning. *CoRR, abs/1507.04888*, 2015.

[71] Iain D Couzin. Collective cognition in animal groups. *Trends in cognitive sciences*, 13(1):36–43, 2009.

[72] Inman Harvey[1], Philip Husbands[1], and Dave Cliff. Issues in evolutionary robotics. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, volume 2, page 364. MIT press, 1993.

[73] Dario Floreano, Francesco Mondada, et al. Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot. *From animals to animats*, 3:421–430, 1994.

[74] Onur Soysal, Erkin Bahçeci, and Erol Şahin. Aggregation in swarm robotic systems: Evolution and probabilistic control. *Turkish Journal of Electrical Engineering and Computer Sciences*, 15(2):199–225, 2007.

[75] Vito Trianni, Roderich Groß, Thomas H Labella, Erol Şahin, and Marco Dorigo. Evolving aggregation behaviors in a swarm of robots. In *Advances in Artificial Life: 7th European Conference, ECAL 2003, Dortmund, Germany, September 14-17, 2003. Proceedings 7*, pages 865–874. Springer, 2003.

[76] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications.* Morgan Kaufmann Publishers Inc., 1998.

[77] Motoaki Hiraga, Yufei Wei, Toshiyuki Yasuda, and Kazuhiro Ohkura. Evolving autonomous specialization in congested path formation task of robotic swarms. *Artificial Life and Robotics*, 23:547–554, 2018.

[78] Valerio Sperati, Vito Trianni, and Stefano Nolfi. Self-organised path formation in a swarm of robots. *Swarm Intelligence*, 5:97–119, 2011.

[79] Roderich Groß and Marco Dorigo. Towards group transport by swarms of robots. *International Journal of Bio-Inspired Computation*, 1(1-2):1–13, 2009.

[80] Muhanad H Mohammed Alkilabi, Aparajit Narayan, and Elio Tuci. Cooperative object transport with a swarm of e-puck robots: robustness and scalability of evolved collective strategies. *Swarm intelligence*, 11:185–209, 2017.

[81] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015.

[82] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[83] David E Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989(102):36, 1989.

[84] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[85] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4:87–112, 1994.

[86] Hans-Paul Schwefel. *Numerical optimization of computer models*. John Wiley & Sons, Inc., 1981.

[87] Hans-Paul Paul Schwefel. *Evolution and optimum seeking: the sixth generation*. John Wiley & Sons, Inc., 1993.

[88] Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In *Swarm Robotics: SAB 2004 International Workshop, Santa Monica, CA, USA, July 17, 2004, Revised Selected Papers 1*, pages 10–20. Springer, 2005.

[89] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[90] Gianluca Baldassarre, Stefano Nolfi, and Domenico Parisi. Evolving mobile robots able to display collective behaviors. *Artificial life*, 9(3):255–267, 2003.

[91] Geoff S Nitschke, Martijn C Schut, and AE Eiben. Evolving behavioral specialization in robot teams to solve a collective construction task. *Swarm and Evolutionary Computation*, 2:25–38, 2012.

[92] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[93] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[94] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[95] Zhang Jin, Wang Hui, and Li Ping. Towards the applications of multi-agent techniques in intelligent transportation systems. In *Proceedings of the 2003 IEEE International Conference on Intelligent Transportation Systems*, volume 2, pages 1750–1754. IEEE, 2003.

[96] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[97] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[98] Pawel Ladosz, Lilian Weng, Minwoo Kim, and Hyondong Oh. Exploration in deep reinforcement learning: A survey. *Information Fusion*, 85:1–22, 2022.

[99] Mel Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*, 2017.

[100] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[101] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[102] Junwei Zhang, Zhenghao Zhang, Shuai Han, and Shuai Lü. Proximal policy optimization via enhanced exploration efficiency. *Information Sciences*, 609:750–765, 2022.

[103] Jiang Hua, Liangcai Zeng, Gongfa Li, and Zhaojie Ju. Learning for a robot: Deep reinforcement learning, imitation learning, transfer learning. *Sensors*, 21(4):1278, 2021.

[104] Pierre Sermanet, Kelvin Xu, and Sergey Levine. Unsupervised perceptual rewards for imitation learning. *arXiv preprint arXiv:1612.06699*, 2016.

[105] Gianni Brauwers and Flavius Frasincar. A general survey on attention mechanisms in deep learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(4):3279–3298, 2021.

[106] Levent Bayındır. A review of swarm robotics tasks. *Neurocomputing*, 172:292–321, 2016.

[107] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing solving sparse reward tasks from scratch. In *International Conference on Machine Learning*, pages 4344–4353. PMLR, 2018.

[108] Adam Daniel Laud. *Theory and application of reward shaping in reinforcement learning*. University of Illinois at Urbana-Champaign, 2004.

[109] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003.

[110] Vito Trianni and Manuel López-Ibáñez. Advantages of task-specific multi-objective optimisation in evolutionary robotics. *PloS one*, 10(8):e0136406, 2015.

[111] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[112] Diederik M Roijers, Peter Vamplew, Shimon Whiteson, and Richard Dazeley. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, 48:67–113, 2013.

Appendix A

Publications Presented in the Thesis

This appendix provides a list of publications that are presented in the thesis. This appendix only shows a list of work published in academic journals and international conferences. The full list of publications is in Appendix B.

Chapter 3

- Yupeng Liang, Boyin Jin, Ziyao Han, and Kazuhiro Ohkura, "Collective Transport with Swarm Robotic System Using Deep Neuroevolution", Proceedings of SICE Annual Conference 2021, pp. 580–583 (2021)

Chapter 4

- Ziyao Han, Yupeng Liang, and Kazuhiro Ohkura, "Developing multi-agent adversarial environment using reinforcement learning and imitation learning", Artificial Life and Robotics, Vol. 28, No. 4, pp. 703–709 (2023)
- Ziyao Han, Fan Yi and Kazuhiro Ohkura, "Generating Competitive Behavior in Adversarial Environment Using Reinforcement Learning", Proceedings of the AROB-ISBC-SWARM 2024, pp. 441–445 (2024)

Chapter 5

- Ziyao Han, Fan Yi, and Kazuhiro Ohkura, "Collective Transport Behavior in a Robotic Swarm with Hierarchical Imitation Learning", Journal of Robotics and Mechatronics, Vol.36, No. 3, pp. 538–545 (2024)

Appendix B

List of Publications

Journal Publications

- Ziyao Han, Yupeng Liang, and Kazuhiro Ohkura, "Developing multi-agent adversarial environment using reinforcement learning and imitation learning", Artificial Life and Robotics, , Vol. 28, No. 4, pp. 703–709 (2023)
- Ziyao Han, Fan Yi, and Kazuhiro Ohkura, "Collective Transport Behavior in a Robotic Swarm with Hierarchical Imitation Learning", Journal of Robotics and Mechatronics, , Vol.36, No. 3, pp. 538–545 (2024)
- Xiaotong Nie, Yupeng Liang, Ziyao Han and Kazuhiro Ohkura, "Generating collective wall-jumping behavior for a robotic swarm with self-teaching automatic curriculum learning", Artificial Life and Robotics, Vol. 28, No. 1, pp. 67–75 (2023)
- Yupeng Liang, Ziyao Han, Xiaotong Nie and Kazuhiro Ohkura , "Improving generative adversarial network with multiple generators by evolutionary algorithms", Artificial Life and Robotics, Vol. 27, No. 4, pp. 761–769 (2022)
- Boyin Jin, Yupeng Liang, Ziyao Han, Motoaki Hiraga, and Kazuhiro Ohkura, "A Hierarchical Training Method of Generating Collective Foraging Behavior for a Robotic Swarm", Artificial Life and Robotics, Vol. 27, No. 1, pp. 137–141 (2022)
- Boyin Jin, Yupeng Liang, Ziyao Han, and Kazuhiro Ohkura, "Generating Collective Foraging Behavior for Robotic Swarm Using Deep Reinforcement Learning", Artificial Life and Robotics, Vol. 25, No. 4, pp. 588–595 (2020)

International Conferences

- Ziyao Han, Fan Yi and Kazuhiro Ohkura, ”Generating Competitive Behavior in Adversarial Environment Using Reinforcement Learning”, Proceedings of AROB-ISBC-SWARM 2024, pp. 441–445(2024)
- Yupeng Liang, Boyin Jin, Ziyao Han, and Kazuhiro Ohkura, ”Collective Transport with Swarm Robotic System Using Deep Neuroevolution”, Proceedings of SICE Annual Conference 2021, pp. 580–583 (2021)
- Boyin Jin, Yupeng Liang, Ziyao Han, and Kazuhiro Ohkura, ” Hierarchical Training Method Implementation on Generating Collective Foraging Behavior for Robotic Swarm”, Proceedings of the 26th International Symposium on Artificial Life and Robotics, pp. 56–61 (2021)
- Yupeng Liang, Boyin Jin, Ziyao Han, and Kazuhiro Ohkura, ”Generating Control Policy for a Flying Swarm with Deep Reinforcement Learning”, Proceedings of the 25th International Symposium on Artificial Life and Robotics, pp. 550–554 (2020)
- Boyin Jin, Yupeng Liang, Ziyao Han, and Kazuhiro Ohkura, ”Generating Collective Foraging Behavior for Robotic Swarm Using Deep Reinforcement Learning”, Proceedings of the 3rd International Symposium on Swarm Behavior and Bio-Inspired Robotics, WeP1a-3, pp. 16–22 (2019)

Domestic Conferences

- Ziyao Han, Boyin Jin, Yupeng Liang, and Kazuhiro Ohkura, ”Collective Transport Behavior in a Robotic Swarm with One Team Reward Signal”, 第28回計測自動制御学会中国支部学術講演会論文集, 3B-5, pp. 69–70 (11, 2019) 岡山大学 2019年11月23日
- Ziyao Han, Boyin Jin, Yupeng Liang, and Kazuhiro Ohkura, ”Collective Transport Behavior in a Robotic Swarm based on Double Deep Q-Network Algorithm”, 第27回計測自動制御学会中国支部学術講演会論文集, 1E-4, pp. 61–62 (12, 2018) 県立広島大学 2018年12月1日
- Yanxu Zhao, Boyin Jin, Ziyao Han, and Kazuhiro Ohkura, “Solving Cooperative Foraging Problem by a Robotic Swarm with Several DRL Methods”, 第28回計測自動制御学会中国支部学術講演会論文集, 1B-4, pp. 15–16 (11, 2019) 岡山大学 2019年11月23日

Acknowledgements

Without the support of many people, writing this doctoral thesis would not have been possible. Firstly I thank my supervisor, Prof. Kazuhiro Ohkura, for his patient supervising, understanding and tolerance to my faults, encouraging me in my trough, and giving me advice on my research and daily life.

I would like to thank my doctoral thesis committee Prof. Keiji Yamada, Prof. Takeshi Iwamoto, and Prof. Yoshiyuki Matsumura, for revising the thesis and insightful comments.

Special thank to my colleagues Yupeng Liang, Boyin Jin, Fan Yi, Xiaotong Nie and many other people who helped me a lot in the laboratory. And also thanks to past and current members of Machine Intelligence and Systems A Laboratory (formerly Manufacturing Systems A Laboratory).

Last, profound thanks to my family, my parents Lei Han and Min Fan for their understanding, support, encouragement, and love, without which I would not come this far.

2024.7
Ziyao Han