Warsaw University of Technology

FACULTY OF
POWER AND AERONAUTICAL ENGINEERING

Institute of Aeronautics and Applied Mechanics

Master's diploma thesis

in the field of study of Automatic Control and Robotics
and specialisation in Robotics

A Deep Reinforcement Learning Algorithm for Swarm Robotics

Clayton Frederick Souza Leite
student record book number 295508

thesis supervisor
Prof. D.Sc. Teresa Zielińska
thesis co-supervisor
Prof. Marco Baglietto

Warsaw, 2018

Abstract

Inspired by the latest advances in deep reinforcement learning that have shown impressive and unprecedented results in artificial intelligence tasks and aware of the lack of a general deep reinforcement learning algorithm in swarm robotics, we propose modifications to the most recent works by Google DeepMind, adapting them to the specifications of swarm robotics and providing the flexibility of using them to any task within swarm robotics with very slight changes.

To the best of our knowledge, this work represents the first effort towards the development of a deep learning algorithm in swarm robotics that provides a framework to produce any behavior without the need of a human expert to hand-engineer controllers based on the task, the sensory inputs of the robots, the environment states and the robots' kinematics. Instead, the algorithm proposed here allows the robots to learn the controllers themselves by means of a sequence of interactions with the environment.

We validate the proposed algorithm in computer simulations with four different tasks in swarm robotics: aggregation, dispersion, chain formation and square lattice formation. Then we perform several tests from which we conclude that the performance of the algorithm, irrespective of the task, live up to the expectations of deep learning by exhibiting in general very satisfactory results, even though the training was carried out on a portable computer for a relatively short time.

Keywords: deep reinforcement learning, swarm robotics, deep learning.

Streszczenie

Praca jest inspirowana najnowszymi osiągnięciami w zakresie głębokiego uczenia (deep learning). Stwierdzono iż te narzędzia nie były wykorzystane w robotyce roju i zaproponowano modyfikację najnowszej metody zespołu Google DeepMind. Pozwala ona na zastosowanie głębokiego uczenia ze wspomaganiem (deep reinforcement learning) w wielu zadaniach z zakresu robotyki roju (swarm robotics).

Dogłębny przegląd literatury, pozwala stwierdzić, że jest to pierwsza próba utworzenia algorytmu ba tyle ogólnego, iż pozwala on na wygenerowanie dowolnego zachowania roju bez konieczności podawania szczegółów, w tym m.in. kinematyki robotów. Zaproponowany algorytm znajduje odpowiednie parametry ruchu dzięki interakcji robota z otoczeniem.

Zbadaliśmy działanie algorytmu w symulacji komputerowej. Algorytm był użyty w czterech zadaniach robotyki roju, mianowicie w grupowaniu, rozpraszaniu, formowaniu łańcucha oraz tworzeniu formacji na bazie kwadratu. Zostały również wykonane testy, które potwierdzają, że algorytm, bez względu na zadanie, sprostuje oczekiwaniom i dostarcza satysfakcjonujących wyników, pomimo faktu, ze do uczenia się wykorzystywano przenośny komputer a czas uczenia był krótki.

Słowa kluczowe: głębokie uczenie ze wspomaganiem. robotyka roju, głębokie uczenie.

Sommario

Ispirato dagli ultimi progressi in apprendimento profondo per rinforzo che hanno dimostrato risultati impressionanti e senza precedenti nei compiti d'intelligenza artificiale e consapevole della mancanza di un algoritmo generale di apprendimento profondo per rinforzo nella robotica degli sciami, proponiamo modifiche ai lavori più recenti di Google DeepMind, adattandoli alle specifiche della robotica degli sciami e fornendo la flessibilità di usarli per qualsiasi compito nella robotica degli sciami con lievi modifiche.

Per quanto ne sappiamo, questo lavoro rappresenta il primo sforzo verso lo sviluppo di un algoritmo di apprendimento profondo nella robotica degli sciami che fornisce uno strumento per produrre qualsiasi comportamento senza la necessità di un esperto umano progettando a mano i controllori basato sul compito, le entrate sensoriali dei robot, gli stati dell'ambiente e la cinematica dei robot. Invece, l'algoritmo qui proposto permette che i robot apprendano i controllori da soli mediante una sequenza d'interazioni con l'ambiente.

Convalidiamo l'algoritmo proposto in simulazioni al computer con quattro diversi compiti nella robotica degli sciami: aggregazione, dispersione, formazione di catena e formazione di motivi quadrati. Quindi eseguiamo diversi test dai quali concludiamo che la prestazione dell'algoritmo, indipendentemente dal compito, soddisfano le aspettative di apprendimento profondo esibendo in generale risultati molto soddisfacenti, anche se il apprendimento è stato eseguito su un computer portatile relativamente per poco tempo.

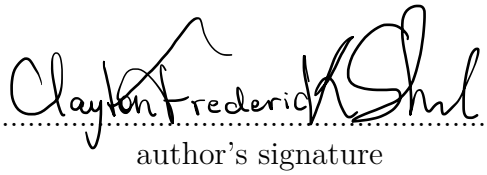
Parole chiave: apprendimento per rinforzo, robotica degli sciami, apprendimento profondo.

Statement of the author of the thesis

Being aware of my legal responsibility, I certify that this diploma:

- has been written by me alone and does not contain any content obtained in a manner inconsistent with the applicable rules,
- had not been previously subject to the procedures for obtaining professional title or degree at the university

Furthermore I declare that this version of the diploma thesis is identical with the electronic version attached.

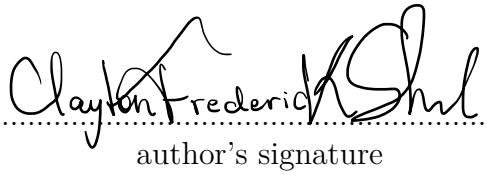
24 August 2018	
date	author's signature

Statement

I agree to make my diploma thesis available to people, who may be interested in it. Access to the thesis will be possible in the premises of the faculty library. The thesis availability acceptance does not imply the acceptance of making the copy of it in whole or in parts. Regardless of the lack of agreement the thesis can be viewed by:

- the authorities of Warsaw University of Technology
- members of The Polish Accreditation Committee
- state officers and other persons entitled, under the relevant laws in force in the Polish Republic, to free access to materials protected by international copyright laws.

Lack of consent does not preclude the control of the thesis by anti-plagiarism system.

24 August 2018	
date	author's signature

Contents

List of Figures

List of Symbols

Roman symbols

a	action
a_0	initial action
a_t	action taken at time t
a_k	action taken at time step k
$a_t^{(i)}$	action taken at time t by agent i
B	minibatch size
$\text{card}(S)$	cardinality of a set S
C	frequency, in time steps, at which the target network has its weights set equal to the policy network
d_r	diameter of the ring-shaped body of the ePuck robot
$d_i(t, j)$	distance between agents i and j at time t
$d_{i,k}(t)$	distance between agent i and its k-th nearest robot at time t
D	replay memory
E	square combination
$G_i(E, t)$	square discrepancy function for agent i at time instant t for the square combination E
$G_i(t)$	discrepancy function for square lattice formation task and for agent i at time t
$G_T(t)$	total discrepancy function for square lattice formation task at time t
$G_i(t_1, t_2)$	discrepancy function for square lattice formation task due and for agent i at time t_2 considering that from t_1 to t_2 only agent i has moved
$H_i(t)$	discrepancy function for dispersion task and for agent i at time t
$H_T(t)$	total discrepancy function for dispersion task at time t

$H_i(t_1, t_2)$	discrepancy function for dispersion task and for agent i at time t_2 considering that from t_1 to t_2 only agent i has moved
$J_i(t)$	discrepancy function for chain formation task and for agent i at time t
$J_T(t)$	total discrepancy function for chain formation task at time t
$J_i(t_1, t_2)$	discrepancy function for chain formation task and for agent i at time t_2 considering that from t_1 to t_2 only agent i has moved
$I_i(t)$	discrepancy function for aggregation task and for agent i at time t
$I_T(t)$	total discrepancy function for aggregation task at time t
$I_i(t_1, t_2)$	discrepancy function for aggregation task and for agent i at time t_2 considering that from t_1 to t_2 only agent i has moved
k	time step
L	loss function
M	number of episodes
n	number of robots/agents
n_S	number of squares
N	capacity of replay memory (maximum number of transitions)
p_j	priority of transition j
P	probability in exploration and exploitation strategy
P_j	probability of sampling transition j from replay memory
Q	state-action value function (or Q-function)
Q^*	optimal state-action value function (or optimal Q-function)
Q_k	state-action value function (or Q-function) at time step k
Q_π	state-action value function (or Q-function) for policy π
$\hat{Q}$	state-action value function (or Q-function) approximated by target's neural network
s	state
s_0	initial state
s_t	environment state at time t
s_k	environment state at time step k
S	number of transitions stored in replay memory
r_t	reward received at time t
r_k	reward received at time step k

$r_t^{(i)}$	reward received at time t by agent i
$r_i^G(t)$	reward function for agent i at time t for the square lattice formation task
$r_i^H(t)$	reward function for agent i at time t for the dispersion task
$r_i^J(t)$	reward function for agent i at time t for the chain formation task
R, R_d, R_c	upscale factors for discrepancy functions (arbitrary constants)
t	time
t_{lim}	maximum elapsed time in the execution of an episode
T	terminal state
T_B	Boltzmann temperature
U_C	discounted reward factor due to a collision
U_M	discounted reward factor due to movement of the agent
v	state value function
$v*$	optimal state value function
v_π	state value function for policy π
w_j	importance-sampling weight for transition j
W	number of stages
x_i	x coordinate of agent i
$x_{i,j}$	x coordinate of the j-th closest agent to agent i
y_i	y coordinate of agent i
$y_{i,j}$	y coordinate of the j-th closest agent to agent i
z_j	target for transition j used for training the policy network

Greek symbols

α	exponent parameter in prioritized replay memory
α_k	learning rate at step k
β	exponent parameter in prioritized replay memory
δ_j	temporal difference error for transition j
γ	discount rate
ϵ	probability in ϵ-greedy exploration and exploitation strategy
θ	weights of policy/online network
$\hat{\theta}$	weights of target network

ξ	parameter in prioritized replay memory
$\lambda_t^{(i)}$	sequence of observations, actions taken and communications by agent i at time t
π	policy function
σ	upscale reward function
τ	amount of (milli)seconds in a time step
ϕ_t, ψ_t	preprocessed state s_t at time t
$\phi_t^{(i)}$	preprocessed state s_t by agent i
$\psi_t^{(i)}$	preprocessed state s_t due to communication to agent i
ω	angular speed

Other symbols

$\mathcal{A}$	finite set of actions
$\mathcal{A}_i$	finite set of actions of agent i
$\mathcal{E}(\mathcal{S}_A)$	set of all square combinations in which the agents in $\mathcal{S}_A$ can organize themselves
$\mathcal{P}$	state transition probability function
$\mathcal{R}$	reward function
$\mathcal{R}_i$	reward function of agent i
$\mathbb{R}$	set of real numbers
$\mathcal{S}$	finite set of states
$\mathcal{S}_A$	set of all agents

List of Tables

1

Introduction

1.1 What is swarm robotics?

In nature, the survival of certain insects, such as ants, relies on their abilities in working in a group by synchronizing the efforts of every individual to achieve a common goal. This collective behavior of social insects grants them the ability to perform relatively complex tasks, even though such insects are noticeably simple and possess very limited reasoning capabilities and knowledge of their surroundings.

For example, in ants, the process of scouring an area for food sources happens initially in a chaotic way, through the use of random walking. However, once the food is found, ants start to communicate and inform each other about its location, with the use of chemical substances. When this happens, order emerges and the efficiency of the foraging task is greatly improved. Self-organization in ants also appears in the case when the food source is relatively large and heavy for a single individual to carry. They join forces by grasping the item at different points and by performing a collectively coordinated movement to bring the resource to their nest.

Such collective behavior observed in nature has fascinated and inspired researchers in

devising new approaches to the field of artificial intelligence, originating then a discipline known as swarm intelligence. Swarm intelligence concerns the design of intelligent multi-agent systems motivated by the study of social insects [6] and has been widely used in robotics, where it receives the term *swarm robotics*.

1.2 Characteristics of swarm robot systems

As described in [60], in swarm robotics, a swarm system is desired to have three main properties: robustness, flexibility, and scalability. Robustness is the characteristic of a swarm robot system in which every single robot unit is redundant for the operation the swarm performs, in the sense that the loss of a unit does not impede the execution and completion of the task. Naturally, a necessary (but not sufficient) condition to robustness is that no robot in the swarm coordinates the operation. Flexibility is the ability of the swarm of robots to adapt its behavior according to different environments and tasks. Finally, scalability refers to the ability of the swarm system in successfully performing and completing a certain task regardless of the exact number of robots that compose the swarm, that is, whether the swarm system is composed by tens, hundreds or thousands of units.

1.3 Classification of swarm robot systems

Studies in swarm robotics can be classified into different categories according to their main focus, as cited by [76]. The first category concerns the behavior of the robots with respect to themselves. For instance, in a surveillance application, it is desirable that the robots of the swarm disperse themselves over the environment to cover a vast area. The second category refers to the behavior of the robots with respect to entities in the environment. As an example, in a disaster situation, a swarm of robots could help in the search for human victims. More sophisticated applications often include a hybrid of the two categories, such as in the cooperative transportation of an object through the environment.

A more common classification divides the studies of swarm robotics according to the desired task of the robots:

- **Flocking.** Some species of birds are known to gather together and travel to a common

destination in the search for better food availability. This collective migratory behavior is called flocking and can be used in swarm robotics as a navigation mechanism [2].

Figure 1.1: Giant murmuration of starlings traveling across the twilight sky. Source: http://www.dailymail.co.uk/news/article-2143051/Winged-whirlwind-Starlings-dramatic-tornado-formation-thousands-gather-setting-sun.html. Access date: 21 January 2018

- **Aggregation.** Aggregation is a common behavior present in a wide variety of species: from bacteria to mammals [60], and it is a precondition for other collective behaviors [67]. It consists in gathering many individuals in a common spot, and it is a precursor task to tasks such as collective movement, self-assembly, and pattern formation [54] since these tasks require the robots to be initially gathered closely together.

- **Dispersion.** The opposite of aggregation is dispersion (also known as segregation), i.e., individuals initially close to each other are required to spread out in order to maximize the covered area while maintaining connectivity within the swarm [71]. Dispersion is commonly used in surveillance tasks where the robots of the swarm are required to spread over the environment to maximize the total area they can watch at once. It is also a precursor behavior for foraging.

- **Foraging.** In this task, the members of the swarm search for items scattered in an environment and return them to a common location called *nest*. This behavior is commonly found in ants when they search for food sources. Applications of foraging include toxic waste clean-up, search, and rescue and collection of terrain sample [1].

- **Pattern formation.** This task corresponds to the formation of a geometrical pattern

by the swarms. In nature, this behavior is seen in some birds as they form a V-shape when flocking either to allow a better communication and visual contact or by improving aerodynamic efficiency [21]. Possible uses for pattern formation are in surveillance, search and rescue, and area exploration applications [20].

Figure 1.2: V-shape pattern formation in birds. Scientists believe that birds exploit aerodynamics in their favor to conserve energy and be able to travel long distances. Source: http://www.isciencetimes.com/articles/6691/20140117/birds-v-formation-aerodynamics-energy-distances-migration.htm. Access date: 21 January 2018

- **Self-assembly.** In the task of self-assembly, the swarm of robots needs to form physical connections between themselves so that a certain robotic structure is obtained. This behavior is found in ants as they form chain-like structures by attaching themselves to each other in order to form bridges or rafts to traverse difficult terrains [45]. Self-assembly is a necessary precursor behavior to connected movement.

Figure 1.3: A chain-like structure of ants in the form of a bridge. Ants can self-assemble into chain-like structures similar to bridges to traverse a gap. Source: https://phys.org/news/2015-11-ants-bridges-bodies-video.html. Access date: 17 January 2018.

- **Chain formation.** A precursor behavior to self-assembly is chain formation since prior to assembling themselves in chain-like structures, the agents are required to position themselves in a chain formation. This behavior can also be used for purposes of navigation, as it can be observed when foraging ants connect their nest with foraging areas by forming a chain of individuals [7].

- **Object clustering.** This problem is very similar to foraging. The difference is that the items are not required to be grouped together in a predefined location, i.e., they can be placed together anywhere [37]. A variation of this task happens when the items are of different types and they need to be sorted in addition to being clustered. An example of application is garbage collection.

- **Connected movement.** This problem is similar to flocking, however here the robots are physically connected to each other and must change the morphology of the connection to avoid obstacles. Connected movement can be useful in providing stability to the robot swarm as it advances on rough terrains or to traverse gaps that a single robot would fall into [2].

- **Cooperative transport.** Ants are known to combine forces and coordinate their movements when carrying a relatively huge prey to their nest. This behavior is known in swarm robotics as cooperative transport. An application is in unifying efforts to transport heavy or large objects.

1.4 What is deep reinforcement learning?

Reinforcement learning provides a way of programming agents to perform a certain task without needing to specify how the task is to be achieved [32], that is, with reinforcement learning the agents themselves learn the behavior necessary for the completion of the task through a sequence of trial-and-error interactions with an environment. When reinforcement learning is done with deep neural networks, it is named deep reinforcement learning. Reinforcement learning and its deep learning variant are discussed in Chapter 4.

1.5 Organization of the thesis

The second chapter gives the literature review of swarm robotics and deep reinforcement learning. Based on such literature review, the motivations for this research are detailed in the third chapter, along with the objective of the research and the plan of solution describing all steps to be taken to attain the objective. Chapter 4 introduces the background behind deep reinforcement learning and is essential to the understanding of Chapter 5. Chapter 5 presents the contribution of this thesis: a deep reinforcement learning algorithm to swarm robotics, and in special to the dispersion, pattern formation, aggregation and chain formation tasks. Chapter 6 consists of a series of experiments that take place in computer simulations to prove the validity of the contribution of this work. Finally, the last chapter is devoted to final comments concerning the thesis and proposals for future work.

2

Literature Review

2.1 Swarm robotics

Researchers in swarm robotics have mostly focused on proof-of-concept studies with multi-robot systems performing a variety of tasks in simulator or laboratory environments [60], i.e., swarm robotics is a relatively new field, and it is still far from any everyday life application [78]. The literature review of the common tasks researched in swarm robotics for the last two decades is given below.

2.1.1 Flocking

In 1987, Reynolds [56] proposed a distributed behavioral model for the aggregated motion of particles in computer animation. The model permitted the particles to exhibit the flocking motion with the possibility of avoiding obstacles as seen in nature and has been adopted in many applications such as computer animation, robotics, spacecraft and UAVs [78]. Reynolds model consisted in modeling the motion of the particles with two opposite behaviors: attraction and repulsion. Particles far from each other should be attracted one by the other, while particles located too close to each other should experience a repulsion. The attraction

and repulsion of the particles - not only the repulsion of the particles one with respect to the other but also the repulsion of the particles with respect to obstacles - were modeled using a virtual force proportional to the inverse of the squared distance between the particles, which is a Newtonian gravity model.

Hettiarachchi and Spears [25] compared the Newtonian gravity force with the Lennard-Jones force for modeling the flocking behavior of robots with obstacle avoidance characteristics. The authors obtained superior results when using the Lennard-Jones force law[1], as it provided a more fluid motion of the robots when maneuvering through obstacles.

Regarding the direction of the motion, Nasseri and Asadpour [53] studied the flocking problem when the flock is formed by informed agents, i.e., agents that have the knowledge of the goal location, and are responsible for guiding the flock, and naive agents, i.e., agents whose task is only to follow the flock. The authors concluded that the flock is able to reach the goal location, even in cases when the flock is composed in majority by naive agents and that the time taken for the flock to reach the goal increases as the number of informed agents decreases.

Not only virtual force laws were used to the problem of flocking. Morihiro *et al.* [50] introduced reinforcement learning with the use of the Q-learning technique in computer simulations to model the flocking behavior with and without an approaching predator. Dada and Ramlan [12] modeled the flocking task as an optimization problem, and solved it with the use of a hybrid optimization algorithm consisting of the primal-dual interior point method combined with particle swarm optimization (PSO).

2.1.2 Aggregation

In swarm robotics, the two common methods to approach the problem of aggregation are probabilistic finite state machines (PFSM), artificial evolution [7]. An example of the application of PFSM in aggregation was studied by Soysal and Şahin [65]. Soysal, Bahçeci, and Şahin [64] used neural networks combined with the genetic algorithm to provide an artificial evolution approach to the problem. The authors found that the evolutionary approach

[1]The Lennard-Jones force law is a mathematical model that approximates the attraction or repulsion of two neutral atoms or molecules. At long distances, the molecules experience an attractive force and, at short distances, a repulsive force. This provides a natural balance between the molecules [25].

outperformed the PFSM method studied in [65]. Trianni *et al.* [68] used an evolutionary algorithm for the evolution of a neural network controller in order to model a simple but effective clustering behavior.

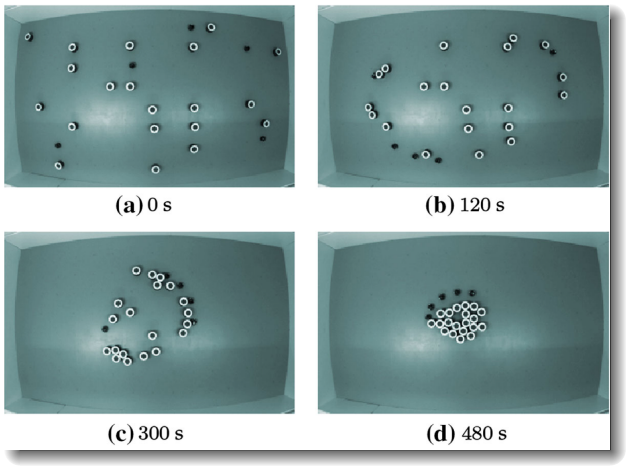


Figure 2.1: Aggregation task performed by robots. Source: Lopes *et al.* (2016) [38]

A different approach to probabilistic methods and evolutionary algorithms is studied in the work of Garnier *et al.* [18]. In this work, the authors implemented a controller based on a behavioral model studied in experiments with cockroaches, and the results obtained showed a good resemblance with biological observations.

2.1.3 Dispersion

McLurkin and Smith [44] proposed five mathematical models to approach the dispersion problem in a group of physical robots with only local communication between robots. In their work, the robots were able to disperse themselves in large and complex indoor environments with different characteristics, such as stationary dispersion and oscillating dispersion, depending on which algorithm was used to control them.

Morlok and Gini [52] studied four different algorithms responsible for the dispersion of a swarm initially clustered in an unknown environment without active intra-robot communication. The experiments were done in virtual environments and the results showed that the algorithm that had access to the information regarding the position of neighbor robots performed better.

Ugur, Turgut, and Şahin [71] designed an algorithm similar to potential-field approaches. Each robot of the swarm used the intensity readings obtained from its wireless sensor to estimate the distance to other robots. The algorithm proposed had the goal of moving the robots in such a way that the distance estimation between them would be maximized while minimizing the number of disconnected robots in the swarm. The tests were performed in a physics simulator. A similar work is done in [39], however, with a less realistic wireless communication model as compared to [71].

Figure 2.2: Dispersion task performed by robots. Source: Don Miner (2007) [16]

2.1.4 Foraging

For the task of foraging, Vaughan *et al.* [73] proposed an algorithm inspired by the trail following behavior in ants and the waggle dance in honey bees[2]. In their paper, simulated robots initially search through an environment by random walks and, as they move through the environment, they periodically record the estimation of their current position and heading. Once a robot reaches the goal, it shares the path it followed among the other robots through a wireless communication network. Each robot now processes this information transmitted and uses it to plan its motion in order to reach the goal, which is assumed to be a rich item source location. The results show that the method is robust in the presence of significant localization errors and hence applicable to real swarm systems.

In [72], the authors studied a similar algorithm where the shared information among robots is used to generate trails. This algorithm is expanded in [59] by considering more than one goal in the environment. However, in environments where there is more than one

[2]These insects are able to share knowledge about the location of rich food sources, either by the use of pheromones, in the case of the ants, or by a waggling dance, in the case of bees.

goal, trails that overlap each other are likely to appear. This translates into possible spatial interference between the robots as they meet when traveling to different goal locations, thus reducing the overall efficiency of the swarm system. The authors also proposed a solution to address the problem of such overlapping trails.

The approaches described above are based on the sharing of successful experiences of the robots through a wireless connection. Some authors have proposed to use the environment as a medium for the communication being robots. Such an idea is inspired by the trail pheromones in ants. When a worker ant finds a food source, it releases a chemical substance called pheromone when returning home [40], creating a trail that allows other colony members to locate the food source more easily, and then transport it to the nest.

2.1.5 Pattern formation

Spears *et al.* [63] made use of virtual attraction and repulsion forces to solve the task of pattern formation. In computer simulations, the method proposed by the authors was able to lead a swarm of particles from a random arrangement to a hexagonal lattice.

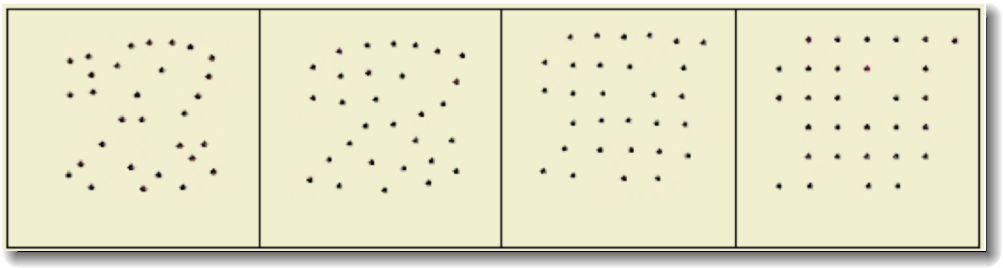


Figure 2.3: Square lattice (right) formed by a swarm of particles from a random initial position (left). Source: Martinson and Payton (2005) [41]

Also with the use of virtual forces, Martinson and Payton [41] obtained square lattices (Fig. 2.3) from random initial positions. The algorithm showed robustness to noise in sensor readings, and to the presence of obstacles in the area. Other approaches using virtual physics were studied in [4, 42, 62].

More complex shapes, such as the shapes of letters of the alphabet, were obtained with the algorithm of Guo, Meng and Jin [20]. Such algorithm - inspired by principles of gene regulation and cellular interactions in multi-cellular organisms - showed to be effective against

obstacles, and robust to sudden losses of robots. The experiments were not only conducted in computer simulations, but also with a real set of robots.

2.2 Deep reinforcement learning

The revolution in reinforcement learning with deep learning started in 2015 [3] with the development of an algorithm - named Deep Q-Network or DQN [49] - that could make an agent learn to play a wide range of Atari 2600 video games at a superhuman level. The DQN algorithm consisted of a deep neural network whose inputs were pixels obtained from screenshots of the games, and outputs were discrete actions to be taken by the learning agent. The network was trained with the Q-learning algorithm [75] in such a way that its outputs (actions) could maximize the score of the agent in the game.

Nair *et al.* [51] extended the DQN algorithm by parallelizing the process of learning with multiple machines, instead of a single machine. The authors deployed multiple agents running in parallel interacting with their own instances of the same environment, but jointly learning a common global network. The algorithm was named Gorila DQN (General Reinforcement Learning Architecture DQN), and could outperform DQN, however, relying on much more computational resources.

In 2016, another extension to the DQN algorithm - the Double DQN - was proposed in [23], addressing a common shortcoming in the DQN algorithm and providing a much better performance in several games compared to its precedent algorithm.

Inspired by recent findings in neuroscience related to how living beings learn new tasks, Schaul *et al.* [61] have further improved the Double DQN algorithm with a technique named prioritized experience replay. This algorithm was later improved with the dueling DQN[3] proposed in [74], especially for tasks with large action spaces.

The previous algorithms have considered only discrete action spaces. In parallel as improvements in algorithms to discrete action spaces were developed, Timothy *et al.* [69] adapted successful ideas of deep reinforcement learning to continuous action spaces with the Deep Deterministic Policy Gradient (DDPG) algorithm. The DDPG algorithm was tested in simulated environments in tasks such as controlling an inverted pendulum, driving a car

[3]The algorithm is named dueling DQN even though it is an improvement of the Double DQN algorithm.

in a racing game and synthesizing gait for a bipedal robot. Also in the continuous action space, Mnih *et al.* devised the Asynchronous Actor-Critic Agent (A3C) algorithm [48], using multiple agents interacting with their own copies of the same environment similarly as in the Gorila algorithm, however without the use of multiple machines. The A3C algorithm was compared with all the previous algorithms (except for the DDPG algorithm), outperforming them in average in a multitude of tasks.

In Late 2017, Hessel *et al.* [26] proposed various changes and additions to the Double DQN algorithm with prioritized experience replay, creating the Rainbow algorithm and delivering with it unprecedented performance.

In March 2018, Horgan *et al.* [28] published two algorithms: the Ape-X DQN based on the algorithm of Schaul *et al.* for discrete action spaces, and the Ape-X DPG which is a variant of the DDPG algorithm for continuous action spaces. The Ape-X framework also followed the general idea of using multiple agents interacting with their own instances of the same environment. Concerning the performance, only the Ape-X DQN algorithm was compared side-by-side with previous algorithms (except for the DDPG algorithm), delivering superior results to them across most of the 57 Atari 2600 games used in the test.

Deep reinforcement learning in swarm robotics has been studied in [29] with a variant of the DDPG algorithm. The task consisted of one target robot sending out messages, and various other search robots whose the goal was of locating the target robot based on the information provided by the messages. Strictly speaking, such a task is not cooperative, as the search robots could find the target without the need of cooperative interactions.

In [30], the task of dispersion was considered with the use of the DDPG algorithm. The authors have tried two different approaches: guided learning and non-guided learning. In guided learning, the learning algorithm had access to the global state of the environment, as opposed to the non-guided approach. It was found that only the guided learning approach was able to deliver positive results.

3

Objectives

3.1 Motivation

In this section, we will introduce the motivations behind the study of swarm robotics and the use of deep reinforcement learning in it.

3.1.1 Why swarm robotics?

As mentioned in [76], to perform certain complex tasks such as search and rescue, a system that operates with a single robot needs to have a sophisticated structure and control system, which translates into high design costs and an increased proneness to faults; as a consequence, the chance that the operation it performs will be compromised becomes higher. Also, it is expected that such a single robot will have a relatively high power consumption, which impacts its autonomy and can be crucial in applications where charging is disadvantageous or not possible at all.

Swarm robotics tries to address these shortcomings by using a large number of simple and cheap robot units and decentralizing the control of the operation, that is, no robot unit acts as a central unit that coordinates the operation. By parallelizing the execution of a

task with a large number of units and using decentralized control, faults to a unit does not compromise the task. This is the main advantage of swarm robotics over a single robot system. Also, swarm robotics can operate successfully with minimal prior knowledge of the world in dynamic and unpredictable environments or environments where the human intervention is difficult [71].

Nevertheless, it should be noted that the simplicity of the robots in a swarm system does not make the task of devising strategies to control the swarm system any easier, and the purpose of swarm robotics is not to substitute single robot systems, but to deliver better results in scenarios where a single robot cannot perform well enough.

According to [5], it is expected that over the next decades swarm robotics will provide novel approaches to applications such as farming, disaster management, deep sea inspection, surveillance and even space exploration. The authors also mention that the application of swarm intelligence to robots at scales in the order of millimeters or even micrometers could bring unprecedented advances to materials science and health technologies.

3.1.2 Why deep reinforcement learning in swarm robotics?

Even though deep reinforcement learning is a rather recent topic[1], it is already driving a revolution in artificial intelligence, as it allows agents to learn very complex tasks without the need to specify how the task is to be achieved, in an end-to-end fashion. Certainly one of the most impressive achievements of deep reinforcement learning is the AlphaGo [13], the first computer program to defeat a professional human Go player and a Go world champion. Among classical games, Go was long known as the most challenging one for artificial intelligence. As a matter of fact, traditional artificial intelligence methods don't stand a chance in Go.

In swarm robotics, deep reinforcement learning has been very scarcely explored, comparing to other artificial intelligence methods. As seen in the literature review, the majority of methods concerns the use of virtual physics and probabilistic finite state machines (PFSM). For this reason and for the fact that deep reinforcement learning is showing unprecedented results in artificial intelligence, it is certainly worth exploring it further in swarm robotics. And this is the aim of this thesis.

[1] Deep reinforcement learning has first appeared in 2015.

3.2 Objective of the research

The objective of this research is to apply the latest advances in deep reinforcement learning with slight modifications to swarm robotics, in particular, to the tasks of dispersion, pattern formation of square lattices, aggregation and chain formation. In order to fulfill this objective, the following steps should be completed:

1. Study the DQN, DDQN, prioritized experience replay and the Ape-X DQN algorithms in detail.

2. Based on these algorithms, propose a deep reinforcement learning algorithm for swarm robotics able to address any task. In special, demonstrate how to address the tasks of dispersion, pattern formation of a square lattice, aggregation and chain formation with the proposed algorithm.

3. Test the proposed algorithm in computer simulations for the tasks aforementioned.

4. Evaluate and comment on the results of the experiments.

4

Background

The goal of this chapter is to introduce the background behind deep reinforcement learning based on the literature. The concepts presented here are essential for the understanding of the next chapter, which consists of the contribution of this thesis. The text elaborated here is based on the following works: [8, 9, 23, 28, 49, 57, 58, 70]. The reader can refer to them for a more in-depth approach.

4.1 Reinforcement learning

Reinforcement learning consists of a topic in machine learning that provides a way of programming agents to perform a certain task without needing to specify how the task is to be achieved [32], that is, with reinforcement learning the agents themselves learn the behavior necessary for the completion of the task through a sequence of trial-and-error interactions with an environment.

Reinforcement learning can be divided into two cases regarding the number of agents: single-agent case and multi-agent case. We will first consider the single-agent case, and later expand it to the multi-agent case. Then we will present the latest advances in single-agent

deep reinforcement learning. Such advances are then modified and adapted in Chapter 5 as we will propose a general multi-agent deep reinforcement learning algorithm for swarm robotics.

4.2 Single-agent reinforcement learning

The single-agent case is illustrated in Figure 4.1. At the time step t, the agent observes the state of the environment s_t and - based on it - performs an action a_t. As a result, the environment transitions to a state s_{t+1} and the agent receives a reward r_{t+1}. The reward is a numeric value that evaluates the outcome of taking the action a_t at state s_t and transitioning to a state s_{t+1}. The goal of the agent is to learn a policy (i.e., a behavior) that maximizes the cumulative reward it receives in the long run.

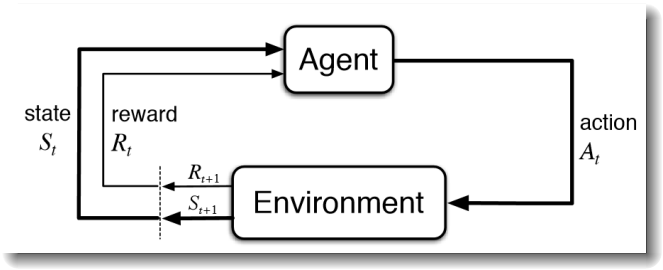


Figure 4.1: The standard reinforcement learning model. Source: Richard S. Sutton and Andrew G. Barto (2017) [58]

4.2.1 Markov decision processes

The problem of reinforcement learning for a single agent is formalized with the use of Markov decision processes (MDPs).

Definition 4.2.1. A Markov decision process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state transition probability function, and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function.

The state transition probability function $\mathcal{P}$ assigns a probability of reaching a state s' given that the agent performs an action a_t at the state s_t, while the reward function $\mathcal{R}$ assigns a real-valued number to the transition from state s_t to state s' as a result of action

a_t. For deterministic systems, $\mathcal{P}$ assumes either zero or one, and $\mathcal{R}$ is fully determined based on the current state s_t and action a_t.

Some MDPs have terminal states. A terminal state is a state which represents either the completion of the task or its failure. A task in which there is a clear terminal state - such as in playing chess - is called an episodic task, as opposed to a continuing task. Here we will be considering only episodic tasks and be referring to as *episode* the sequence of states and actions from the beginning of an episodic task to its end.

4.2.2 Policy

A policy is a function $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that maps a state s_t and an action a_t to a probability of performing a_t at s_t, and determines the behavior of the agent. For deterministic systems, π can be simplified to $\pi : \mathcal{S} \to \mathcal{A}$. The goal of reinforcement learning is to find a policy that maximizes the expected discounted return from every state s defined as:

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_k^\infty \gamma^k r_{k+1} | s_0 = s \right] \tag{4.1}$$

where $\gamma \in [0, 1)$ is called discount rate, r_{k+1} is the reward as instant k as the result of action a_k at state s_k (and state s_{k+1} in stochastic systems), and s_0 is the initial state.

The function (4.1) can be interpreted as how good it is to be at state s and follow the policy π; and for this reason, it is called state value function for policy π. The discount rate is chosen by the designer, and it exists to keep the math bounded and consider the uncertainty of future rewards. Choosing a discount rate closer to zero will guarantee a "myopic" behavior to the agent, i.e., a behavior that prefers immediate rewards rather than long-term rewards, while a discount rate closer to one guarantees a "far-sighted" behavior. It should be noted that, in episodic tasks, all rewards obtained after the terminal state are zero, thus transforming the sum in (4.1) into a finite sum.

An approach to maximize (4.1) is by working finding the optimal state-action value function (Q-function) [8]. The Q-function is defined as:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_k^\infty \gamma^k r_{k+1} | s_0 = s, a_o = a \right] \tag{4.2}$$

The Q-function (4.2) defines how good it is to be at state s, perform action a and follow the policy π thereafter. The optimal Q-function $Q^*(s,a)$ is defined as $Q^*(s,a) = \max_\pi Q_\pi(s,a)$.

4.2.3 Q-learning

The Q-learning algorithm [75] consists of finding $Q^*(s,a)$ by using iterative approximations. Starting with an arbitrary Q-function at state s_k, perform an action a_k, observe the reward r_{k+1} and the next state s_{k+1}. Then update the Q-function using the rule:

$$Q_{k+1}(s_k,a_k) = Q_k(s_k,a_k) + \alpha_k\Big[r_{k+1} + \gamma \max_{a'} Q_k(s_{k+1},a') - Q_k(s_k,a_k)\Big] \tag{4.3}$$

where Q_k is the Q-function at iterative step k, $\alpha_k \in (0,1]$ is the learning rate, and the term multiplying α_k (in square brackets) is denoted as temporal difference error.

Once one has found Q^*, the optimal policy - i.e., the policy that maximizes the expected discounted return (4.1) - can be obtained by:

$$\pi(s) = \text{argmax}_a Q^*(s,a) \tag{4.4}$$

Under certain conditions described in [75], Q_k can converge to Q^* as $k \to \infty$. One of the conditions is that all the state-action pairs are visited infinitely often. This is known as exploration.

4.2.4 Exploration and exploitation trade-off

One particularity of reinforcement learning is the exploration and exploitation trade-off. Since the goal of a reinforcement learning agent is to maximize the sum of rewards obtained, it has to prefer actions that it has found to produce the highest reward, that is, the agent has to exploit previous rewarding experiences. However, to discover the best actions to be taken, the agent has to constantly try actions that it has never performed before, that is, the agent has to explore new experiences. since these two behaviors (exploration and exploitation) are conflicting with each other, it is a crucial issue to balance them.

The simplest way of managing the exploration and exploitation trade-off is by using the

so-called ϵ-greedy strategy, which consists of alternating exploration and exploitation: the agent explores with probability 1 - ϵ and exploits with probability ϵ. When exploring, the agent randomly chooses an action among the available ones; when exploiting, the agent chooses an action a such that $a = \text{argmax}_a Q(s, a)$.

One important aspect that is not considered by the ϵ-greedy strategy is the fact that, as the learning proceeds, the model is possibly improving and thus exploration should become less and less needed. This aspect is taken into account in the Boltzmann distribution strategy [57]. For a given state s_t, the Boltzmann distribution assigns a positive probability to any possible action a available in s_t according to its state-action value $Q(s_t, a)$ and according to a parameter T_B called temperature [9]. For a state-action pair (s_t, a) the probability of taking a at s_t is given by:

$$P(s_t, a) = \frac{e^{Q(s_t,a)/T_B}}{\sum_i e^{Q(s_t,a_i)/T_B}} \tag{4.5}$$

where a_i is any action available at s_t, T_B is the temperature. As $T_B \to 0$, exploration vanishes giving place to purely greedy actions, as opposed to when $T_B \to \infty$, which signifies a purely exploratory selection of actions.

It should be noted that exploration and exploitation strategies are only applied during learning. The agent should always exploit when not learning.

4.3 Multi-agent reinforcement learning

Multi-agent reinforcement learning consists of distributed systems of independent agents that cooperate or compete to complete a certain task. Each agent needs to learn an optimal policy in the presence of other learning agents, that is, a stochastic environment.

4.3.1 Stochastic game

Although MDPs provide a solid mathematical formalization of single-agent cases, they do not offer the same for the multi-agent case [70]. The problem of multi-agent reinforcement learning is better formalized with the use of a generalization of Markov decision processes named stochastic game.

Definition 4.3.1. A stochastic game is a tuple $\langle \mathcal{S}, \mathcal{A}_1, \ldots, \mathcal{A}_n, \mathcal{P}, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}_i$ is a finite set of actions of an agent i with $\mathcal{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state transition probability function, $\mathcal{R}_i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a reward function for an agent i, and n is the number of agents.

In the multi-agent case, the state transitions depend on the joint action of all agents. Each agent learns its own policy $\pi_i : \mathcal{S} \times \mathcal{A}_i \to [0, 1]$ (or $\pi_i : \mathcal{S} \to \mathcal{A}_i$ if the policy is deterministic), its own state-value function $v_i : \mathcal{S} \to \mathbb{R}$ as defined in (4.1), as well as its own Q-function $Q_i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as in (4.2) that depends on the joint action of all agents.

In fully cooperative cases - which is the case of swarm robotics - the agents share the same reward function and have the goal of maximizing the common return.

4.3.2 Challenges in multi-agent reinforcement learning

Multi-agent reinforcement learning suffers from severe problems compared to the single-agent case. Here are the four main problems as mentioned in [8]:

- The increase in the number of agents in a reinforcement learning task leads to an exponential growth in the state-action space, resulting in an exponential increase in computational complexity.

- Defining reward functions to each agent is a challenging task due to the fact that such functions need to consider the joint effect of each agent in the completion of the task and not only the effect of the individual agent to which they are designed.

- A third problem arises when all agents - or at least some of them - are learning simultaneously. In such case, each agent's best policy changes over time as the other agents' policies change.

- Exploration in reinforcement learning is crucial for the efficiency of a reinforcement learning. However, for multi-agent cases, such exploration - if done in excess - can destabilize the agents, thus making the learning process much more difficult.

4.4 Deep reinforcement learning

As seen before, the use of deeply-layered artificial neural networks is very recent, yet it is responsible for impressive and unprecedented results in reinforcement learning. Deep neural networks can be used as a way of approximating the Q-function, and thus obtaining a policy.

Previously, we have defined that the goal of a reinforcement learning agent is to learn an optimal policy π that maximizes its received reward in the long run. Such policy determines its behavior - in a deterministic or stochastic way - given the observed state of the environment. Also, as discussed before, one common way of finding an optimal policy is by using iterative approximations as in (4.3) to find an optimal Q-function and thus obtain the optimal policy by (4.4).

When the state-action space is small enough, the Q-function can be found for all state-action pairs, and all its values can be simply stored in a lookup table. However, sometimes the state-action space is large enough to be unfeasible to have a lookup table with all the state-action pairs (s, a) and their corresponding values $Q(s, a)$ for two main reasons. First, the computational expense involved in working with high dimensional state-action spaces renders the task of learning a lookup Q-function table impractical, and second, the memory space required to store such lookup table can be considerably huge. These issues can be dealt with the use of function approximation techniques - in special with the use of neural networks.

4.5 Single-agent deep reinforcement learning

4.5.1 Deep Q-Network

The Deep Q-Network (DQN) [49] algorithm represents an impressive and efficient approach that overcomes previous known instability and divergence issues associated with the use of nonlinear function approximations methods for reinforcement learning.

To overcome these issues, first, the Deep Q-Network algorithm provides an agent with a biologically inspired mechanism named experience replay that stores experiences - that is, a transition consisting of the last observed state, the action taken, the reward obtained

and the next observed state - and randomize them to reduce correlations present when using sequences of experiences. Second, two separate networks are used: a policy or online network and an target network, both with the goal of approximating the Q-function. The policy network is trained towards target values $r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a')$ - where $\hat{Q}$ is the Q-function approximation by the target network - while the target network is never trained, instead it periodically gets its weights set to be equal to the policy network.

The authors apply the algorithm to play a wide range of Atari 2600 in an end-to-end reinforcement learning way - that is, directly mapping pixels to actions. The results reported show that the algorithm outperforms professional human game testers in 29 games. The detailed algorithm is presented in Algorithm 1 as a pseudo-code.

```
Initialize replay memory D to capacity N
Initialize policy state-action value network Q with random weights θ
Initialize target state-action value network Q̂ with weights θ̂ = θ
Define exploration and exploitation strategy
for episode = 1, M do
    Obtain initial state s_1 and preprocess φ_1 = φ(s_1)
    for t = 1, T do
        Select an action a_t based on the exploration and exploitation strategy
        Execute action a_t in the environment and observe reward r_t and state s_{t+1}
        Preprocess φ_{t+1} = φ(s_{t+1})
        Store transition (φ_t, a_t, r_t, φ_{t+1}) in D
        Sample a random minibatch of transitions (φ_t, a_t, r_t, φ_{t+1}) of size B from D
        for transition j = 1, B do
            if episode finishes at j + 1 then
                Set z_j = r_j
            else
                Set z_j = r_j + γ max_{a'} Q̂(φ_{j+1}, a')
        Perform optimization step to minimize loss function
        L = Σ_j^B (z_j − Q(φ_j, a_j))^2 / B with respect to network parameters θ
        Every C steps set the weights θ̂ = θ
```

Algorithm 1: Deep Q-Network algorithm as reported in [49], where M is the number of episodes, T is the terminal state, B is the size of the transitions minibatch and γ is the discount rate.

4.5.2 Double Deep Q-Network

Hado *et al.* [23] showed that the DQN algorithm suffered from common and signifi-

cant overestimations of action values that harmed the performance, and proposed a slight modification to the algorithm in order to prevent such shortcoming. The modified algorithm is known as the Double Deep Q-Network (DDQN) algorithm, and uses the formula $z_j = r_j + \gamma\ \hat{Q}(\phi_{j+1}, \text{argmax}_{a'} Q(\phi_{j+1}, a'))$, instead of $z_j = r_j + \gamma\ \max_{a'}\ \hat{Q}(\phi_{j+1}, a')$, for the calculation of the targets.

4.6 Prioritized experience replay

The prioritized experience replay represents a more effective use of the replay memory for learning [61] for deep reinforcement learning. Instead of random sampling mini-batches from the replay memory with uniform probability distribution - as in the DQN and DDQN algorithms - the prioritized experience replay strategy assigns probabilities of sampling a certain experience $\phi_j, a_j, r_j, \phi_{j+1}$ according to its temporal difference error $\delta_j = \hat{Q}(\phi_{j+1}, \text{argmax}_{a'} Q(\phi_{j+1}, a'))$. The probability of a certain transition j to be sampled from the replay memory is given by:

$$P_j = \frac{p_j^\alpha}{\sum_i p_i^\alpha} \tag{4.6}$$

where the exponent α determines how much prioritization is used, and p_j is the priority of transition j given in two possible forms:

$$p_j = |\delta_j| + \xi \tag{4.7}$$

$$p_j = \frac{1}{\text{rank}(j)} \tag{4.8}$$

where ξ is a positive small constant that prevents the transition j from being sampled even if its temporal difference error is zero, and $\text{rank}(j)$ is the rank of transition j when the replay memory is sorted according to $|\delta_j|$. Formulas (4.7) and (4.8) are the proportional prioritization and rank-based prioritization rules, respectively.

The authors also adopt the so-called importance-sampling weight $w_j \in [0, 1]$, which multiplies the loss function calculated for transition j and serves to scale down the magnitude of the gradient of high temporal difference error transitions.

$$w_j = \frac{\min_i P_i^\beta}{P_j^\beta} \tag{4.9}$$

where the exponent $\beta \in [0, 1]$ represents how much the magnitude of the gradient is scaled down for high δ_j transitions.

Initialize replay memory D to capacity N with exponents α and β, and $p_{MAX} = 1$
Initialize policy state-action value network Q with random weights θ
Initialize target state-action value network $\hat{Q}$ with random weights $\hat{\theta} = \theta$
Define exploration and exploitation strategy
Set $p_{MAX} = 1$
for *episode = 1, M* **do**
  Obtain initial state s_1 and preprocess $\phi_1 = \phi(s_1)$
  for *t = 1, T* **do**
    Select an action a_t based on the exploration and exploitation strategy
    Execute action a_t in the environment and observe reward r_t and state s_{t+1}
    Preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D with priority $p_t = p_{MAX}$
    for *transition j = 1, B* **do**
      Sample a single transition $(\phi_j, a_j, r_j, \phi_{j+1})$ from D with probability $P_j = p_j^\alpha / \sum_i p_i^\alpha$ for $i = 1, 2, ..., S$ where S is the number of transitions stored in D
      Compute importance-sampling weight $w_j = (\min_i P_i^\beta)/P_j^\beta$
      if *episode finishes at* $j + 1$ **then**
        Compute $z_j = r_j$
      else
        Compute $z_j = r_j + +\gamma\ \hat{Q}(\phi_{j+1}, \text{argmax}_{a'} Q(\phi_{j+1}, a'))$
      Compute $\delta_j = z_j - Q(s_j, a_j)$
      Update transition priority in D: $p_j \leftarrow |\delta_j| + \xi$ or $p_j \leftarrow 1/\text{rank}(j)$
      Set $p_{MAX} = \max_i p_i$
    Perform optimization step to minimize loss function $L = \sum_j^B w_j (z_j - Q(\phi_j, a_j))^2 / B$ with respect to network parameters θ
    Every C steps set the weights $\hat{\theta} = \theta$

Algorithm 2: Double Deep Q-Network algorithm with prioritized experience replay as reported in [61], where M is the number of episodes, T is the terminal state, B is the size of the transitions minibatch, γ is the discount rate, and α, β and ξ are the parameters of the prioritized experience replay method

4.6.1 Ape-X DQN

The Ape-X DQN algorithm is similar to the prioritized experience replay algorithm -

shown in Algorithm 2 - however, instead of one single agent, multiple instances of the same agent are used. Each agent interacts with its own instantiation of the environment, but share the same online and target networks, also sending experiences to the same replay memory. The agents run on multiple CPUs to generate data, while the training of the neural network happens on a single GPU.

Initialize replay memory D to capacity N and parameters α, β, ξ
Initialize policy state-action value network Q with random weights θ
Initialize target state-action value network $\hat{Q}$ with random weights $\hat{\theta} = \theta$
Define exploration and exploitation strategy
Set $p_{MAX} = 1$
for *episode = 1, M* **do**
  Obtain initial state s_1 and preprocess $\phi_1 = \phi(s_1)$ for all agents
  for *t = 1, T* **do**
    for *each agent* **do**
      Select an action a_t based on the exploration and exploitation strategy
      Execute action a_t in the agent's environment and observe reward r_t and state s_{t+1}
      Preprocess $\phi_{t+1} = \phi(s_{t+1})$
      Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in the agent's local buffer with priority $p_t = p_{MAX}$
      Periodically transfer transitions from the agent's local buffer to replay memory D
    for *transition j = 1, B* **do**
      Sample a single transition $(\phi_j, a_j, r_j, \phi_{j+1})$ from D with probability $P_j = p_j^\alpha / \sum_i p_i^\alpha$ for $i = 1, 2, ..., S$ where S is the number of transitions stored in D
      Compute importance-sampling weight $w_j = (\min_i P_i^\beta)/P_j^\beta$
      if *episode finishes at* $j + 1$ **then**
        Compute $z_j = r_j$
      else
        Compute $z_j = r_j + +\gamma \hat{Q}(\phi_{j+1}, \text{argmax}_{a'} Q(\phi_{j+1}, a'))$
      Compute $\delta_j = z_j - Q(s_j, a_j)$
      Update transition priority in D: $p_j \leftarrow |\delta_j| + \xi$ or $p_j \leftarrow 1/\text{rank}(j)$
      Set $p_{MAX} = \max_i p_i$
    Perform optimization step to minimize loss function $L = \sum_j^B w_j (z_j - Q(\phi_j, a_j))^2 / B$ with respect to network parameters θ
    Every C steps set the weights $\hat{\theta} = \theta$

Algorithm 3: Ape-X DQN algorithm as in [28], where M is the number of episodes, T is the terminal state, B is the size of the transitions minibatch, γ is the discount rate, and α, β and ξ are the parameters of the prioritized experience replay method.

5

Contribution

In this chapter, we present a modification to the Ape-X DQN algorithm that can be applied to swarm robotics. In theory, the algorithm proposed here can be applied to any task in swarm robotics by just tweaking the reward function for the respective task. In special, we show how to use the proposed algorithm to the tasks of dispersion and pattern formation of a square lattice.

5.1 Overview of the algorithm

The algorithm proposed in this work consists of multiple stages, each of which composed of eight components: the agents, a shared environment, an exploration and exploitation strategy, a shared online/policy network, a shared target network, a learner, and a prioritized replay memory.

The multi-stage learning is a simple concept. At a certain stage j, the neural networks are initialized with the weights of the target network of the previous stage $j-1$, unless the stage j is the initial stage, in which case the weights are initialized randomly. What differs from stage to stage is that each stage has a different number of robots.

As discussed in the previous chapter, exploration in large size state spaces leads to a much more difficult learning process, in the sense that the efficiency is greatly harmed. Therefore, concerning the number of robots, here we propose starting the multi-stage learning with few robots and gradually increasing this number from stage to stage at the same time as reducing exploration. In practice, reducing exploration means decreasing the Boltzmann temperature T_B in the Boltzmann distribution exploration and exploitation strategy, or reducing the parameter ϵ in the case of the ϵ-greedy strategy.

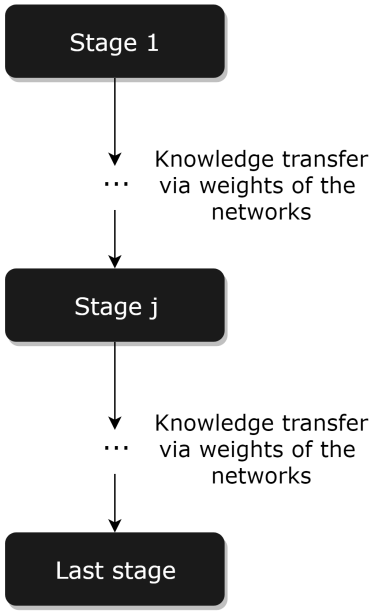


Figure 5.1: Multi-stage learning concept. The number of robots changes from stage to stage. Knowledge is transferred from a previous stage to the next stage by copying the weights from the target network of the previous stage to the policy and target networks of the next stage. Author's concept.

Initializing the training of the policy network of stage j with weights from stage $j-1$ is an approach known as finetuning. Finetuning is a common method in transfer learning, which is a topic that consists in studying approaches that allow leveraging previously acquired knowledge into new situations. In our case, when training with m robots, we use the knowledge (network weights) acquired when training with n robots. In finetuning, it is common to reduce the learning rate of the optimization algorithm to allow for better convergence.

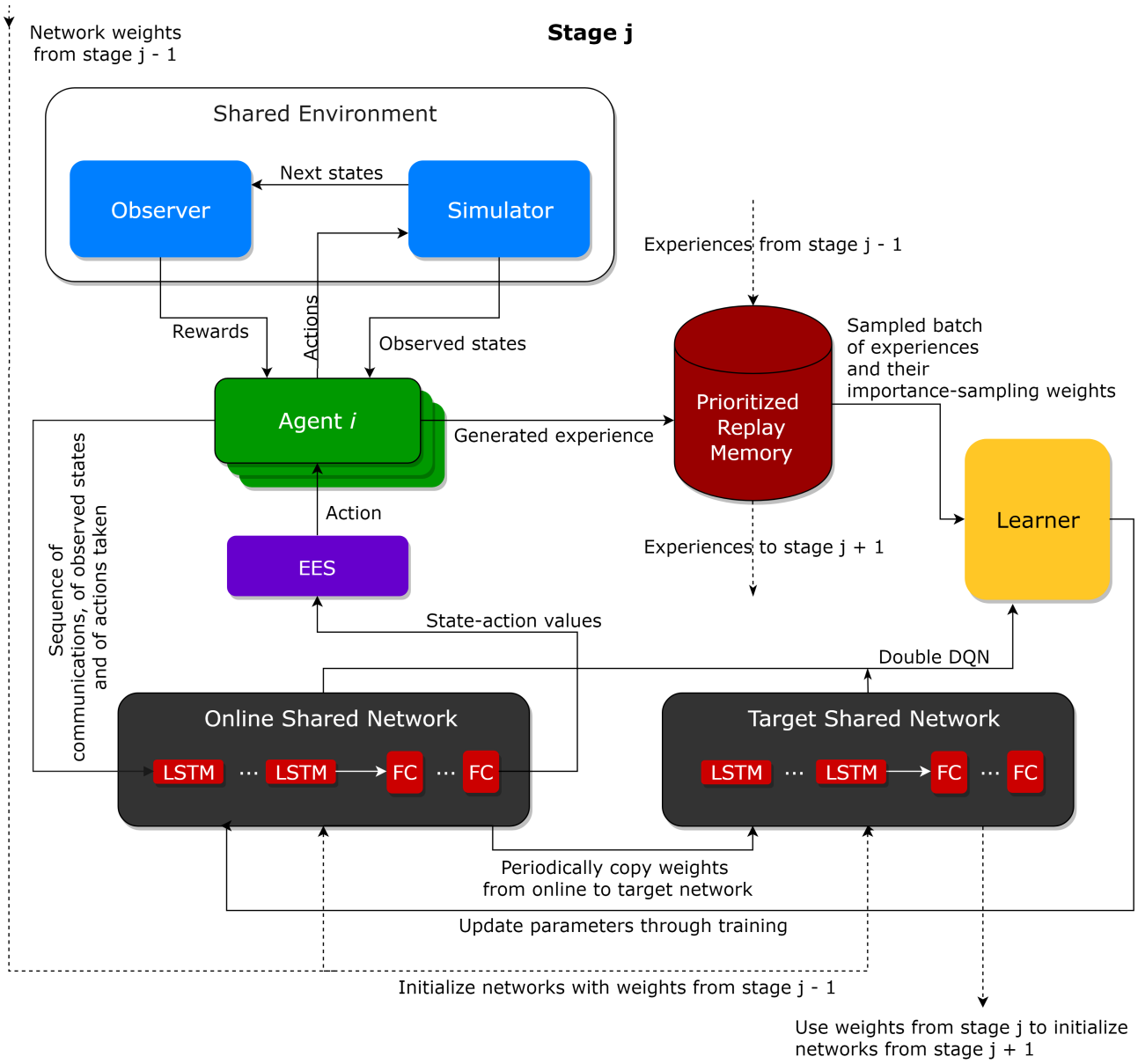


Figure 5.2: Proposed structure of the deep reinforcement learning algorithm for swarm robotics. Different from the Ape-X algorithm, the environment is shared among all agents. The neural networks are formed by a long short-term memory (LSTM) cells, as well as fully connected (FC) layers. EES stands for exploration and exploitation strategy. Author's concept.

The reason to use multiple stages, each of which having a different number of robots, is to meet the requirement of scalability. As we mentioned previously, the algorithms devised for swarm robotics cannot be restricted to the number of robots of the swarm. Statistically speaking, the data distribution of the sensory inputs is different for a different number of

robots in the swarm. Limiting the training to a specific data distribution related to a specific number of robots can lead to performance degradation during inference when the size of the swarm is not the same as that used during the training. Therefore, we argue that the training should occur with different numbers of robots. In the end, our goal is to make the robots learn a behavior that is not restricted to the size of the swarm.

Also, to meet the requirement of scalability, the prioritized replay memory should not be reset from stage to stage. Otherwise, it might happen that the robots unlearn how to behave with the sizes of the swarm from earlier stages - that is, catastrophic forgetting [34] can occur.

The architecture of a certain stage j is depicted in Figure 5.2 and is similar to that of the Ape-X DQN algorithm, except for the following: the neural networks are formed by long short-term memory (LSTM) cells and fully connected (FC) layers, the environment is shared among all agents, the agents are not distributed over different CPUs, and information is exchanged between the agent via communication.

It is important to remind that Figure 5.2 shows the structure of the algorithm only when the agents are learning the task. During inference - that is, after the learning process has finished - there isn't the need for the online shared network, the prioritized replay memory, the learner, or the exploration and exploitation policy. Also, the multi-stage concept is only using when training. In other words, during inference, first, the agent observes the environment with its sensors and exchange information with its neighbors. Then it uses a history of observations, actions taken and information obtained from communication as inputs to the target network (obtained from the last stage) to compute the state-action values for every action available. Finally, the agent performs the action whose state-action value is the highest.

In the following sections, we will discuss every component of the architecture in details.

5.2 The agents

In swarm robotics, agents (or robots, since we consider all robots to be agents) cooperate with each other to accomplish a certain task, therefore we will discard here competitive interactions between agents.

At an instant of time t, the environment can be described by its state s_t, which is composed of the states of all agents at that instant of time. Each agent perceives the environment through its sensors, however, agents are only able to observe part of the state of the environment. We say that $\phi_t^{(i)}$ is the observation made by an agent i at instant t, and it is a function of the state of the environment $\phi_t^{(i)} = \phi_t^{(i)}(s_t)$. Also, agent i receives information communicated by its neighbors, hence we denote as $\psi_t^{(i)} = \psi_t^{(i)}(s_t)$ the communicated information agent i receives at instant t, which is also a function of the state of the environment.

At an instant of time t, each agent i observes $\phi_t^{(i)}$ through its sensors, receives $\psi_t^{(i)}$ via communication, and performs an action $a_t^{(i)}$ in the environment - for instance, moving forward, opening its gripper, etc. The action to be taken at instant t is decided based on the following:

- k last observations $(\phi_t^{(i)}, \phi_{t-1}^{(i)}, \cdots, \phi_{t-k+1}^{(i)})$
- k last received information via communication $(\psi_t^{(i)}, \psi_{t-1}^{(i)}, \cdots, \psi_{t-k+1}^{(i)})$
- k last actions taken $(a_{t-1}^{(i)}, a_{t-2}^{(i)}, \cdots, a_{t-k}^{(i)})$

In short, each agent keeps in its memory a history of size k containing its last observations, communications and actions taken. Based on this history, an agent i can decide on which action $a_t^{(i)}$ to take. Naturally, during training, the action is also decided with an exploration and exploitation strategy (EES).

Therefore, first, the sequence of observations, communications, and actions taken passes through the online share neural network, which outputs state-action values for all actions available, serving as input to the exploration and exploitation strategy, which in turn outputs an action to be taken.

The reason to use a sequence of observations, instead of solely the last observation, lies on the fact that it placates the partial observability nature of the agents. However, a sequence of observations is not useful if the agent cannot remember which actions it has taken before, therefore it is also necessary to include a history of actions. Finally, communication between the agents is employed to further minimize the partial observability.

All agents take actions simultaneously and after this has been done, each agent receives a reward $r_t^{(i)}$ from the environment based on how much its action has collaborated to the

success of the task. Afterwards, each agent i saves in memory its last action taken $a_t^{(i)}$, makes a new observation $\phi_{t+1}^{(i)}$ of the state of the environment and receives further information $\psi_{t+1}^{(i)}$ via communication. As soon as this happens, each agent i stores its experience (also called transition) in the replay memory with maximum priority. The experience is a tuple defined in (5.1).

$$(\lambda_t^{(i)}, a_t^{(i)}, r_t^{(i)}, \lambda_{t+1}^{(i)}) \tag{5.1}$$

where,

$$\lambda_t^{(i)} = \Big((\phi_t^{(i)}, \phi_{t-1}^{(i)}, \cdots, \phi_{t-k+1}^{(i)}), (\psi_t^{(i)}, \psi_{t-1}^{(i)}, \cdots, \psi_{t-k+1}^{(i)}), (a_{t-1}^{(i)}, a_{t-2}^{(i)}, \cdots, a_{t-k}^{(i)})\Big) \tag{5.2}$$

$$\lambda_{t+1}^{(i)} = \Big((\phi_{t+1}^{(i)}, \phi_t^{(i)}, \cdots, \phi_{t-k+2}^{(i)}), (\psi_{t+1}^{(i)}, \psi_t^{(i)}, \cdots, \psi_{t-k+2}^{(i)}), (a_t^{(i)}, a_{t-1}^{(i)}, \cdots, a_{t-k+1}^{(i)})\Big) \tag{5.3}$$

It should be noted that, in this algorithm, the set of actions is a discrete set. For the tasks studied in this work, we propose the discretization in actions such as: move forward at speed v millimeters per second, move backward at speed v millimeters per second, rotate clockwise at angular speed ω radians per second, and rotate counterclockwise at speed ω radians per second. Also, the time instant t is discretized in steps of τ milliseconds. As for the communication, we propose that the information $\psi_t^{(i)}$ exchanged with an agent i is composed of the sensor readings of the k nearest neighbors to agent i, as well as their distances and relative orientations with respect to agent i.

5.3 The shared environment

The environment is composed of two parts: the observer and the simulator (in case the learning process occurs in a computer simulator).

The observer makes use of a reward function to judge the agents' actions. Such a reward function is different for different tasks. The observer can be easily implemented in a simulator. In real life, as proposed by [30], the observer can be implemented with a camera connected to a machine running computer vision algorithms that extract important information from the captured images and transform such information into rewards. Here we

consider the simulated case, therefore computer vision algorithms are not a requirement.

As mentioned before, in this thesis we will test the proposed deep reinforcement learning architecture for two-wheeled ground robots moving on a flat surface with four tasks: dispersion, aggregation, chain formation and square lattice formation.

Before mathematically presenting reward functions for the tasks, we consider the robustness requirement in swarm robotics. An algorithm for swarm robotics is robust when it leads the swarm to the successful completion of the task even if there is a loss in the swarm (for instance, a loss can happen when a certain robot discharges) during the execution of the task.

Here we argue that, in reinforcement learning, robustness can be tackled by a proper choice of reward function. The reward function used during training will dictate the behavior of the agents after the training; that is, after training, the agents will behave in the same way that led them to receive high rewards during training. In other words, the reward function instructs how to behave. Therefore, the key to satisfying the requirement of robustness is the design a reward function that can instruct the robots to behave towards the completion of the task even in the presence of failed robots. To do this, the reward functions presented shortly provide rewards to each agent individually based on whether or not its action has contributed to the successful completion of the task. This incentivizes each agent to do its best even when the other agents behave in the worst way.

5.3.1 The dispersion task

First, let's define the dispersion discrepancy function $H_i(t)$ for an agent i at time instant t, as it is commonly used in the literature.

$$H_i(t) = R(d_{i,1}(t) - l)^2 \tag{5.4}$$

where R is a positive arbitrary constant termed upscale factor, l is the desired distance to the nearest robot, and $d_{i,1}(t)$ is a function that returns the distance from robot i to its nearest robot at time instant t.

Considering that the agents start the dispersion task in positions close to each other, the goal of the agents is to maximize their distance to the nearest neighbor up until l, and

not exceed this value - that is, to minimize the total discrepancy given as (5.5). In the task of dispersion, minimizing (5.5) is important as it maximizes the total area covered by the robots.

$$H_T(t) = \sum_i H_i(t) \tag{5.5}$$

We now define $H_i(t_1, t_2)$ as the dispersion discrepancy function for an agent i at time instant t_2 considering that only agent i has moved from time instant t_1 to time instant t_2. Also, let $\sigma : \mathbb{R} \to \mathbb{R}$ be a bounded and monotonic decreasing function given as $\sigma(x) = \max(\sigma_{\text{MIN}}, -ax + \sigma_{\text{MAX}})$ with σ_{MIN}, a and σ_{MAX} being positive constants.

Finally, for the dispersion task, we introduce the reward function $r_i^H(t)$ at time instant t for an agent i:

$$r_i^H(t) = \begin{cases} \sigma(H_i(t)) \cdot (\text{sgn}(H_i(t-1) - H_i(t-1,t)) - U_M) & \text{if agent } i \text{ moved} \\ \sigma(H_i(t)) \cdot (\text{sgn}(H_i(t-1) - H_i(t-1,t))) & \text{otherwise} \end{cases} \tag{5.6}$$

where *sgn* is the sign function defined as $\text{sgn(x)} = |x|/x$, and U_M is a positive constant that punishes the agent for moving.

It should be noted that the reward function $r_i^H(t)$ simply evaluates whether or not agent i has contributed to minimizing the total discrepancy. A positive reward signifies that the agent has collaborated in decreasing the total discrepancy, while a negative reward denotes the opposite. Moreover, by definition, $\sigma(H_i(t))$ linearly decreases from σ_{MAX} to σ_{MIN} as the discrepancy $H_i(t)$ increases. Hence, the closer is the agent to an optimal discrepancy, the higher rewards it can get. This incentivizes the agent towards states of lower discrepancies since in reinforcement learning the agent learns a behavior that guarantees it the maximum reward in the long run. σ is bounded because otherwise, this would cause overflow issues as the discrepancy approaches zero.

Note that the calculation of the reward is done considering that only agent i has moved at the last instant of time. Therefore, the reward agent i receives depends solely on its own actions. It would not be fair to consider the movement of the other agents for the reward of an agent i, as this could result in agent i receiving a negative reward even in the case when

it has done its best to contribute to the minimization of the total discrepancy.

Also, the fact that the agent sees its reward being decreased by U_M (termed movement avoidance reward) when it moves is due to the fact that every movement consumes power, thus it is necessary to incentivize the robot to minimize the total discrepancy by moving as little as possible. The constant U_M should be chosen carefully, a high enough value could make the agents prefer to not move even if the total discrepancy has not reached anywhere near the optimal value.

To instruct the agents to avoid collisions, we propose to subtract the reward $r_i^H(t)$ by a positive constant U_C (termed collision avoidance reward), in case agent i has collided.

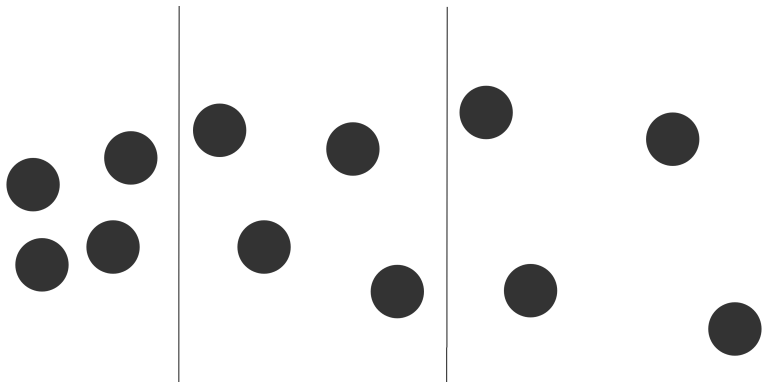


Figure 5.3: Expected dispersion behavior from left to right (top view) (transition from the initial positioning of the swarm to final positioning). Each black circle represents a robot. Author's concept.

5.3.2 The square lattices formation task

Similar to the dispersion task, we will devise a discrepancy function for the square formation task, and use it to define the reward function.

First, suppose the swarm is formed by $n > 4$ agents that are required to arrange themselves in n_S[1] squares of equal size by moving as little as possible. Now, let $\mathcal{S}_A$ be the set of all agents of which the swarm is composed, and $\mathcal{P}(\mathcal{S}_A)$ as its power set. Then, let's define $\mathcal{E}(\mathcal{S}_A)$ as:

$$\mathcal{E}(\mathcal{S}_A) = \{E \in \mathcal{P}(\mathcal{S}_A) \mid \text{card}(E) = 4\} \tag{5.7}$$

[1] n_s depends on the number of agents n. For example, if $n = 6$, then $n_s = 2$, since six robots can at maximum form two squares as in Figure 5.4.

Thus, $\mathcal{E}(\mathcal{S}_A)$ is the set of all possible combinations in which the agents can organize themselves to create squares, and $\text{card}(\mathcal{E}(\mathcal{S}_A)) = \dfrac{n!}{4!(n-4)!}$ (n is the number of agents in the swarm). Then, define the set function $G_i : \mathcal{E}(\mathcal{S}_A) \times \mathbb{R} \to \mathbb{R}$ as:

$$G_i(E,t) = \begin{cases} \min\limits_{\substack{j_1,j_2,j_3 \in E \\ j_1 \neq j_2 \neq j_3 \neq i}} \Big[2(d_i(t,j_1)-l)^2 + 2(d_i(t,j_2)-l)^2 + (d_i(t,j_3)-l\sqrt{2})^2\Big], \text{if } i \in E \\ 0, \text{otherwise} \end{cases} \tag{5.8}$$

where l is the desired length of the sides of the squares in the lattice and $d_i(t,j_1)$ is the distance from robot i to robot j_1, and similarly for $d_i(t,j_2)$ and $d_i(t,j_3)$. $G_i(E,t)$ is the square discrepancy function for agent i at time instant t for the combination E. The total square discrepancy function $G(E,t)$ for a square combination E at time instant t is defined as:

$$G(E,t) = \sum_{i \in E} G_i(E,t) \tag{5.9}$$

To find n_S best possible square combinations, we need to solve the following constrained optimization problem given in (5.10).

$$\begin{aligned} E^*_{1,t}, E^*_{2,t}, \cdots, E^*_{n_S,t} = \underset{E_1,E_2,\cdots,E_{n_S} \in \mathcal{E}(\mathcal{S}_A)}{\arg\min} & \Big[G(E_1,t) + G(E_2,t) + \cdots + G(E_{n_S},t)\Big] \\ & \text{subjected to } \text{card}(E_{1,t} \cup E_{2,t} \cup \cdots \cup E_{n_S,t}) = n_S \end{aligned} \tag{5.10}$$

The best possible square combinations are the combinations in which the agents can organize themselves to form the simplest (that is, the one to which the agents are closer to achieve) square lattice containing n_S smallest size squares. Notice that the constraint enforces that all agents participate in the formation of the square lattice.

Now, let $\mathcal{E}^*_t(\mathcal{S}_A)$ be the set containing all the best possible square combinations:

$$\mathcal{E}^*_t(\mathcal{S}_A) = \{E^*_{1,t}, E^*_{2,t}, \cdots, E^*_{n_S,t}\} \tag{5.11}$$

Finally, we define the square lattice discrepancy function $G_i(t)$ for an agent i at time

instant t.

$$G_i(t) = \frac{R}{\sum_{E^* \in \mathcal{E}_t^*(\mathcal{S}_A)} \text{card}(E^* \cap \{i\})} \sum_{\mathcal{E}_t^*(\mathcal{S}_A)} G_i(E, t) \tag{5.12}$$

where R is a positive arbitrary constant termed upscale factor.

The function $G_i(t)$ quantifies how much the agent i collaborates to the total discrepancy present in the square lattice formation at time instant t. Naturally, the goal of the agents is to minimize the total discrepancy. The total discrepancy at instant t is the sum of the discrepancies of all agents:

$$G_T(t) = \sum_i G_i(t) \tag{5.13}$$

Similarly, the reward function $r_i^G(t)$ for agent i in the square lattice formation task is given as:

$$r_i^G(t) = \begin{cases} \sigma(G_i(t)) \cdot (\text{sgn}(G_i(t-1) - G_i(t-1,t)) - U_M) & \text{if agent } i \text{ moved} \\ \sigma(G_i(t)) \cdot (\text{sgn}(G_i(t-1) - G_i(t-1,t))) & \text{otherwise} \end{cases} \tag{5.14}$$

where $G_i(t-1,t)$ is the square lattice formation discrepancy for agent i at time instant t considering that only agent i has moved from time instant $t-1$ to t. U_M and σ are used for the same reason as in the dispersion case. Also, the reward $r_i^G(t)$ is subtracted by U_C - as in the dispersion case - whenever agent i collides.

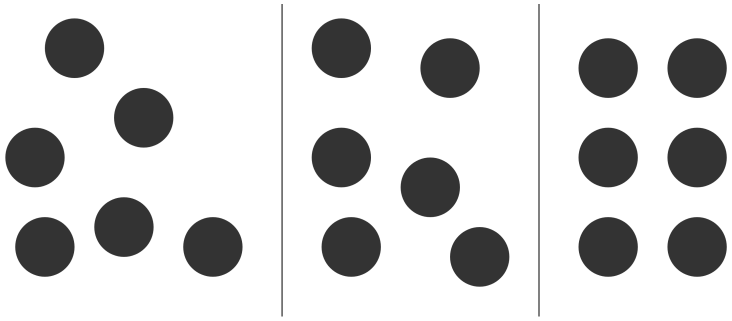


Figure 5.4: Expected square lattice behavior formation from left to right (top view) (transition from the initial positioning of the swarm to final positioning). Each black circle represents a robot. Even though the figure shows the formation of a rectangular shape, we call it a square lattice as it is formed by smaller squares. Author's concept.

5.3.3 The aggregation task

In the aggregation task, the agents of the swarm are required to gather together at a certain spot as close to each other as possible, while avoiding collisions. This will be taken into consideration in the following steps as we devise a discrepancy function for the aggregation task.

Again, let $\mathcal{S}_A$ be the set of all agents of which the swarm consists, and $d_i(t, j)$ a function that defines the distance between agents i and j at time instant t. Then, at time instant t, the furthest agent to agent i is given as:

$$d_{i,MAX}(t) = \max_{j \in \mathcal{S}_A} d_i(t, j) \tag{5.15}$$

We also define $n_i(t, x)$ as the number of agents that are at a distance less or equal to x with respect to agent i at time instant t:

$$n_i(t, x) = \text{card}(\{j \in \mathcal{S}_A \mid d_i(t, j) \leq x\}) \tag{5.16}$$

Now we imagine that each agent has a circular shape with radius l_A, where l_A is the radius of the smallest circumference parallel to the plane of motion that completely encloses the robot. Since the swarm is homogeneous, l_A is the same for all agents. Then, the smallest circumference that encloses agent i and all the agents at a distance less than x to agent i has a diameter of $\min(x, d_{i,MAX}(t)) + 2l_A$ at time instant t.

We define now the packing density of an agent i at time instant t for a distance x as:

$$\rho_i(t, x) = \frac{n_i(t, x)(2l_A)^2}{(\min(x, d_{i,MAX}(t)) + 2l_A)^2} \tag{5.17}$$

Notice that the packing density is a dimensionless quantity that measures how much compact is a certain group of agents. Therefore, our goal in the aggregation task is to maximize $\rho_i(t, x)$ (for all agents) towards the optimal packing density, which depends on the number of entities (in our case, such entities are the agents) and it is an open problem in mathematics for decades[2]. Here we will consider this optimal value to be 0.6 irrespective of the number of agents, and x to be a constant which represents the sensing range of agent i.

[2]Notice that formula (5.17) is a variant of the packing density formula for the circle packing problem.

Now, we define the aggregation discrepancy function $I_i(t)$ for an agent i at time instant t:

$$I_i(t) = R(\rho_i(t, x) - 0.6)^2 \tag{5.18}$$

where R is a positive arbitrary constant termed upscale factor.

Note that $(\rho_i(t, x) - 0.6)^2$ quantifies how distant is the density from the optimal one. Again, we look to minimize the total aggregation discrepancy function, given as:

$$I_T(t) = \sum_i I_i(t) \tag{5.19}$$

With this, we can finally define the reward function:

$$r_i^I(t) = \begin{cases} \sigma(I_i(t)) \cdot (\text{sgn}(I_i(t-1) - I_i(t-1, t)) - U_M) & \text{if agent } i \text{ moved} \\ \sigma(I_i(t)) \cdot (\text{sgn}(I_i(t-1) - I_i(t-1, t))) & \text{otherwise} \end{cases} \tag{5.20}$$

where $I_i(t-1, t)$ is the aggregation discrepancy for agent i at time instant t considering that only agent i has moved from time instant $t-1$ to t. U_M and σ are used for the same reason as in the previously described tasks. And again, the reward $r_i^I(t)$ is subtracted by U_C whenever agent i collides.

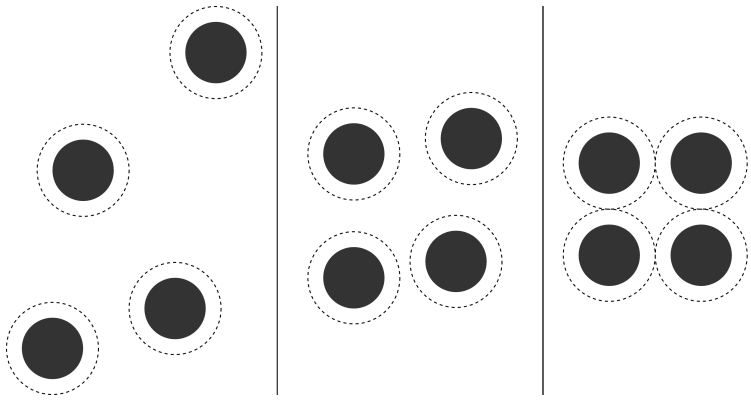


Figure 5.5: Expected aggregation behavior from left to right - packing density increases (top view) (transition from the initial positioning of the swarm to final positioning). Each black circle represents a robot, and the dashed circumference the boundary of a safe distance which the agents should learn to maintain from each other in order to avoid collisions. Author's concept.

5.3.4 The chain formation task

Consider three agents i, j and k. A necessary and sufficient condition for the agent i to be in a chain formation (collinear) with the agents j and k is given as follows:

$$\frac{(x_i - x_j)\cdot(x_i - x_k) + (y_i - y_j)\cdot(y_i - y_k)}{d_{i,j}\cdot d_{i,k}} = \begin{cases} -1 & \text{if } d_{i,j} + d_{i,k} = d_{j,k} \\ 1 & \text{otherwise} \end{cases} \tag{5.21}$$

$$\text{with } (x_i, y_i) \neq (x_j, y_j) \neq (x_k, y_k)$$

where (x_i, y_i), (x_j, y_j), (x_k, y_k) are the x and y coordinates of agents i, j and k, respectively. $d_{i,j}$ is the distance between the agents i and j. We assume that the agents lie on a flat surface, therefore we can neglect the z coordinate. Notice that the equation (5.21) gives the cosine between the vector with origin at agent i and end at agent j and the vector with origin at agent i and end at agent k. The cosine is *minus one* whenever agent i is located between agent j and k, and equal to *one* otherwise.

We use this condition to derive the chain formation discrepancy function $J_i(t)$ for an agent i at time instant t.

$$J_i(t) = R_c\left(\frac{(x_i - x_{i,1})\cdot(x_i - x_{i,2}) + (y_i - y_{i,1})\cdot(y_i - y_{i,2})}{d_{i,1}\cdot d_{i,2}} + a\right)^2 + R_d(d_{i,1} - l)^2 \tag{5.22}$$

where $(x_{i,j}, y_{i,j})$ are the x and y coordinates of the j-th closest neighbor to agent i, and $d_{i,j}$ is the distance between the agent i and its j-th closest neighbor. Naturally, all these quantities are time-dependent, however, for visual simplicity, we drop the suffix that denotes the dependence on time.

When $d_{i,1}$ and $d_{i,2}$ are both less than the distance between agent's i two closest agents, $a = 1$; otherwise $a = -1$. Furthermore, we impose that agent i is at a distance l to its nearest neighbor, this guarantees that the chain has a minimum length of $l\cdot(n-1)$, with n being the number of agents in the swarm. R_c and R_d are positive constants.

Such discrepancy function gives a measure of how far is an agent i from being collinear with its two nearest neighbors. Again, we look to minimize the total chain formation discrep-

ancy function given in (5.23). Notice that an optimal solution that minimizes (5.23) does not mean that all agents are collinear. In fact, we don't require collinearity of the entire swarm, as this is not necessary to form a chain.

$$J_T(t) = \sum_i J_i(t) \tag{5.23}$$

Similarly as for the other tasks, for the chain formation task, we define the reward function $r_i^J(t)$ at time instant t for an agent i:

$$r_i^J(t) = \begin{cases} \sigma(J_i(t)) \cdot (\text{sgn}(J_i(t-1) - J_i(t-1,t)) - U_M) & \text{if agent } i \text{ moved} \\ \sigma(J_i(t)) \cdot (\text{sgn}(J_i(t-1) - J_i(t-1,t))) & \text{otherwise} \end{cases} \tag{5.24}$$

where $J_i(t-1,t)$ is the chain formation discrepancy for agent i at time instant t considering that only agent i has moved from time instant $t-1$ to t. U_M and σ are used for the same reason as in the previously described tasks. And again, the reward $r_i^J(t)$ is subtracted by U_C whenever agent i collides.

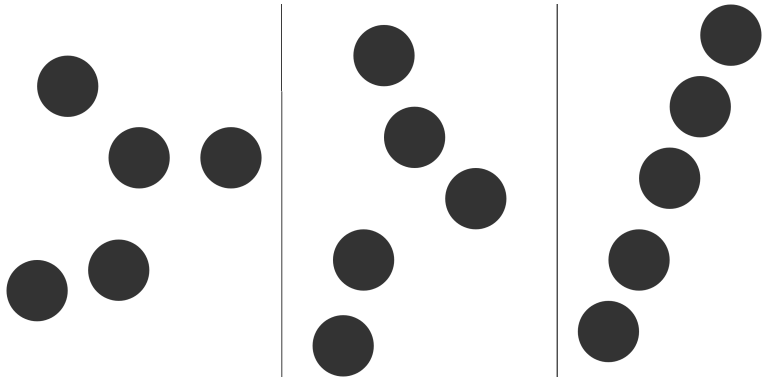


Figure 5.6: Expected chain formation behavior from left to right (top view) (transition from the initial positioning of the swarm to final positioning). Each black circle represents a robot. Author's concept.

5.3.5 Expansion to other tasks

For the four tasks described before, the only difference is in the discrepancy function. That is, the method devised in this work can be expanded to any other task solely by designing a discrepancy function for the task. Notice that all discrepancy functions follow the principle

of locality, that is, the discrepancy of an agent i for a certain task depends only on the agent itself and its immediate neighbors.

It is not compulsory that the rewards are calculated at every time step. In fact, in more complex tasks - such as connected transport - devising a reward function able to judge each agent action at every time step can be rather complicated. Instead, one can opt to sparse reward functions, that is, functions that reward each agent only after a series of time steps. Naturally, such sparse reward functions have a drawback: slower convergence rate.

5.3.6 Episodic tasks

Since the tasks considered here are episodic tasks, we propose four different possible ways for their end. The task is said to be completed only when the total discrepancy falls below a predefined minimum value. Conversely, the task is failed whenever one of the following situations happens: the total discrepancy rises above a predefined maximum value, the time of execution of the task exceeds a limit t_{lim}, or a collision occurs.

5.4 The exploration and exploitation strategy

The exploration and exploitation strategy can be, for instance, the ϵ-greedy strategy or the Boltzmann distribution. However, it is important to reduce the exploration, either by reducing the ϵ parameter, in the case of the ϵ-greedy method, or the Boltzmann temperature T_B, in the case of the Boltzmann distribution.

5.5 The shared networks

Following the DQN algorithm [49], two networks - the online or policy network and the target network - are used here, where only the online network is trained, while the target network has its weights periodically set to be equal to those of the online network. The robots are considered to be homogeneous, therefore the online and the target networks can be shared among all of them. In case the agents were heterogeneous, the networks could not be shared.

The architectures of both networks are identical to each other, and made of long short-term cells (LSTM) [27] and fully connected layers as in Figure 5.7. The input is the concatenation of a sequence of communications, of observed states and of actions taken. The output of the network is the output of the last fully connected layer, having the dimension equal to the number of actions and representing the state-action values for each of the actions.

The LSTM cells give the network the possibility of working with a timed sequence of inputs, instead of a sole input. Imagine the case when an agent perceives an object with one of its sensors at time instant $t-1$, but at time instant t the agent moves so that it can no longer sense the object. Naturally, the object never popped out of existence, instead, the agent is positioned in such a way that none of its sensors can detect the object. This is a clear case of partial observability. It is as if the agent had forgotten about the existence of the object. Using a timed sequence of inputs in the neural network provides the agent the possibility of deciding which action to perform based on information available at not only time instant t, but at previous instants $t-1$, $t-2$, etc. Therefore, the LSTM cells are indispensable for us.

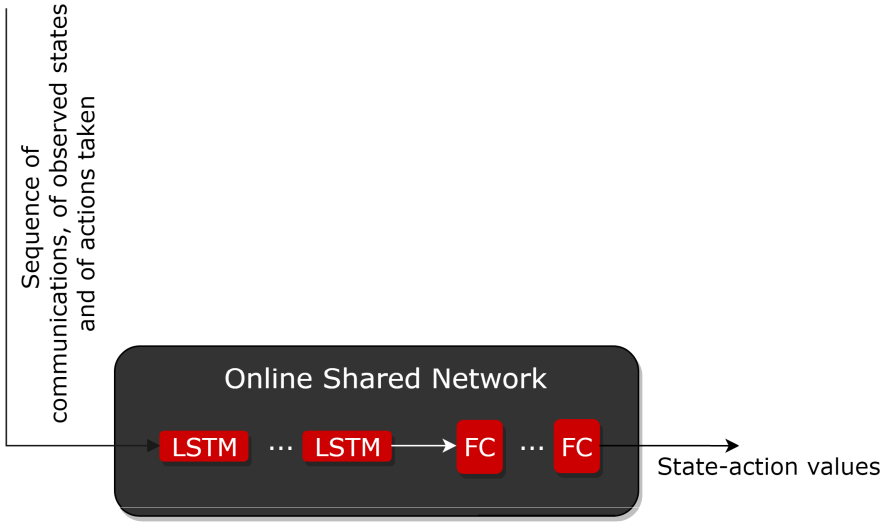


Figure 5.7: Proposed architecture of the neural networks. Author's concept.

5.6 The learner

The learner is responsible for training the online shared network. To do this, first, the

learner samples a minibatch with size B from the prioritized replay memory along with their importance-sampling weights. Since we are using a sequence of observations as input to the neural networks, it is necessary that the minibatch contains sequences of experiences, rather than sole experiences.

The next step is the use of the Double DQN algorithm with each sequence of experience j in the minibatch to obtain its corresponding target z_j. For each sequence of experience j, the learner updates the priority using either (4.7) or (4.8).

The learner then uses the targets z_j to update the weights of the online shared network by means of performing an optimization step to minimize the mean squared error loss function with importance-sampling weights (5.25) as proposed in [61]. The optimization algorithm can be freely chosen.

$$L = \frac{1}{B} \sum_{j}^{B} w_j (z_j - Q(\phi_j, a_j))^2 \tag{5.25}$$

where B is the minibatch size, Q is the Q-function approximation by the online share network, w_j is the importance sampling weight, z_j is the target, ϕ_j is the state, and a_j is the action. The subscript j is used to refer to transition j.

5.7 Prioritized replay memory

As it has been mentioned before, the prioritized replay memory works precisely as in the Ape-X DQN algorithm, however, the transitions are not discarded when progressing from stage to stage. The only case when transitions can be discarded is when the replay memory has reached its limit in size. In such case, before adding a new transition to the replay memory, we propose to delete the transition that has the least priority.

Finally, it should be noted that the transitions stored in the replay memory do not include any information on which agents have generated them; in fact, such information is irrelevant for the training.

5.8 Pseudo-code

for *stage = 1, W* **do**
- Define number of robots for the stage
- **if** *stage == 1* **then**
  - Initialize replay memory D to capacity N and parameters α, β, ξ
  - Initialize policy state-action value network Q with random weights θ
  - Initialize target state-action value network $\hat{Q}$ with random weights $\hat{\theta} = \theta$
- **else**
  - Use replay memory from previous stage
  - Copy weights from the target network of last executed stage
- Define exploration and exploitation strategy
- Set $p_{MAX} = 1$
- **for** *episode = 1, M* **do**
  - **for** *t = 1, T* **do**
    - **for** *each agent* **do**
      - **if** *t == 1* **then**
        - Make an observation $\phi_1 = \phi(s_1)$ of s_1
        - Communicate with nearest neighbors, obtaining $\psi_1 = \psi(s_1)$
        - Compose sequence of observations, actions and communications λ_1
      - Find state-action values $Q(\lambda_t, a)$ for all action a
      - Use the state-action values to select an action a_t based on EES
      - Execute action a_t in the environment and observe reward r_t
      - Make an observation $\phi_{t+1} = \phi(s_{t+1})$ of s_{t+1}
      - Communicate with nearest neighbors, obtaining $\psi_{t+1} = \psi(s_{t+1})$
      - Compose sequence of observations, actions and communications λ_{t+1}
      - Store transition $(\lambda_t, a_t, r_t, \lambda_{t+1})$ in the replay memory with priority $p_t = p_{MAX}$
    - **for** *transition j = 1, B* **do**
      - Sample a single transition $(\lambda_j, a_j, r_j, \lambda_{j+1})$ from D with probability $P_j = p_j^\alpha / \sum_i p_i^\alpha$ for $i = 1, 2, ..., S$ where S is the number of transitions stored in D
      - Compute importance-sampling weight $w_j = (\min_i P_i^\beta)/P_j^\beta$
      - **if** *episode finishes at j + 1* **then**
        - Compute $z_j = r_j$
      - **else**
        - Compute $z_j = r_j + +\gamma \hat{Q}(\phi_{j+1}, \text{argmax}_{a'} Q(\phi_{j+1}, a'))$
      - Compute $\delta_j = z_j - Q(s_j, a_j)$
      - Update transition priority in D: $p_j \leftarrow |\delta_j| + \xi$ or $p_j \leftarrow 1/\text{rank}(j)$
      - Set $p_{MAX} = \max_i p_i$
    - Perform optimization step to minimize loss function $L = \sum_j^B w_j (z_j - Q(\phi_j, a_j))^2 / B$ with respect to network parameters θ
    - Every C steps set the weights $\hat{\theta} = \theta$

⌊ Reduce learning rate of the optimization algorithm

Algorithm 4: The deep reinforcement learning algorithm proposed in this work.

6

Experiments and Results

6.1 The simulation setup

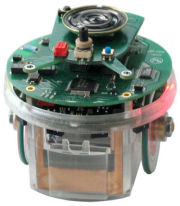


Figure 6.1: ePuck robot. Source: Mondada *et al.* [47].

The simulation was performed on the V-REP software, with the use of ePuck robot models [47]. The ePuck is a mini two-wheeled mobile robot developed by GCtronic and EPFL shaped as a ring of 70mm in diameter ($d_r = 70mm$). The wheels measures about 41mm in diameter, and are mounted 53mm apart from each other. Originally, the robot is equipped with eight infrared proximity sensors distributed (not uniformly, see Figure 6.2) over its circular shape, however, in the experiments, we simulate ultrasonic distance sensors, in place of infrared proximity sensors. In practice, such ultrasonic sensors would have to

work on different frequencies for each robot.

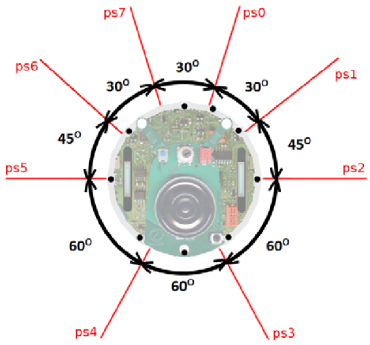


Figure 6.2: Infrared proximity sensors (ps0, ps1, ..., ps7) in the ePuck robot. Source: Tarquino and Nickele [66].

In the simulation, we define that, at every instant of time $\tau = 200ms$, the ePuck robot makes an observation. Each observation is comprised of eight readings, each of which corresponding to an ultrasonic sensor. Such readings are preprocessed to stay within the interval from zero to one. An ultrasonic sensor reading with value zero means that the sensor did not detect anything within its range, while a value of one indicates that the sensor detected something at the closest distance possible. The sensing range covers the interval from 1cm to 50cm.

State-action value	Meaning
q_1	No movement
q_2	Move forward at $40mm/s$
q_3	Move backward at $40mm/s$
q_4	Rotate counterclockwise at $1rad/s$
q_5	Rotate clockwise at $1rad/s$

Table 6.1: Action discretization for dispersion and pattern formation tasks. The choice of speeds is arbitrary, however reasonable.

Also, at every instant of time, the input to the neural network is composed of the following:

- the last four observations
- the last four received information via communication
- the last four actions taken

Since each observation is composed by eight readings, the last four observations contain $4 \times 8 = 32$ features. Also, as described in Section 5.2, the information an agent i receives at time instant t via communication is comprised of the observations made by its j nearest neighbors, as well as the distances and orientations to them. We consider here that an agent i communicates with its two nearest neighbors, therefore, the information it receives at every instant of time has $2 \times 4 \times (8 + 2) = 80$ features. Each of the four last actions taken is encoded using one-hot encoding, therefore, since we use five different actions, this sums up to $4 \times 5 = 20$ features. All in all, the input is composed of $32 + 80 + 20 = 132$ features that contains information from the last four time instants ($4 \times 200ms = 1s$).

Such input first passes through a sole long-short-term-memory (LSTM) cell with 50 hidden states. The last output of the LSTM cell then passes through a multi-layer perceptron with two hidden layers, each of which has 50 neurons. The output of the neural network is the output of the last layer of the multi-layer perceptron and consists of five state-action values, that is, a 5-tuple $(q_1, q_2, ..., q_5)$ defined in Table 6.1. These hyper-parameters were determined by trial and error with several preliminary experiments.

Also regarding the neural networks (the online and target networks), in the first stage, the weights are initialized according to Glorot and Bengio [19] (also known as Xavier initialization). As for the optimization algorithm, the Stochastic Gradient Descent is used and minibatch size B set to 2000. The training happens at every time step. In the double DQN rule application, the discount rate γ is set to 0.9 and the weights are copied from the policy network to target network at every 20 time steps.

For exploration and exploitation strategy, the Boltzmann distribution (4.5) is used. Each stage starts with an initial temperature of T_B that decays linearly at every episode from T_B in the first episode to zero in the last episode. The specific values for T_B depend on the stage and on the task and were determined by trial and error, as well as the upscale factor R and the distance l. Such values will be reported shortly.

For all tasks and for all episodes, the robots are initialized at random positions such that the maximum distance between two robots is less or equal to $3d_rn$, where $d_r = 7mm$ is the diameter of the ring-shaped body of the ePuck robot and n is the number of robots. An episode is completed whenever the discrepancy goes over $150n$, falls below *one* or the number of steps exceeds a limit that will be mentioned shortly. The movement discounting reward U_M

is set to 0.1 and the collision discounting reward U_C is equal to $five$ except for the aggregation task in which $U_C = 30$. The reward upscale function $\sigma(x) = \max(0.5, -0.03x + 5)$.

As for the prioritized replay memory, the capacity is set to 500000 transitions, $\alpha = 0.9$, $\epsilon = 0.001$ and β increases linearly from 0.1 (at the beginning of the stage) to one (at the end of the stage). This choice was inspired by the experiments performed in [61].

For reproducibility of results, the seed for all random operations is set to *zero*. Finally, the code[1] is written in Python 3.6 with TensorFlow 1.3.0 framework and makes use of CUDA programming for operations involving neural networks, such as training and inference. The GPU used is an NVIDIA GeForce GTX 1070 with 8GB RAM.

6.2 Training

During the training, we are interested in knowing if the agents are learning the task as the training proceeds. For this reason, during the training, we will observe four quantities:

- **Average joint reward per episode:** the average reward that the agents have obtained in an entire episode.
- **Average ratio total discrepancy/number of agents per episode:** the average total discrepancy in an episode divided by the number of agents.
- **Maximum ratio total discrepancy/number of agents per episode:** the maximum total discrepancy in an episode divided by the number of agents.
- **Minimum ratio total discrepancy/number of agents per episode:** the minimum total discrepancy in an episode divided by the number of agents.

Naturally, if the agents are learning the task, it is expected that the average joint reward increases as the training proceeds, while the quantities related to the discrepancy decrease. However, it is also expected that, exactly at the transition from stage to stage, the opposite happens, due to the fact that, at the transition, the robots are faced with an increased number of partners and must learn to cooperate with this increased number. This is what is observed in the Figures 6.3-6.6.

[1] The link to the code and the instructions for the use are given in the Appendix A.

6.2.1 Dispersion task

Dispersion task Upscale factor $R = 3000$, distance $l = 5d_r$ (350mm)					
Stage	No. of robots	No. of episodes	No. of steps/episode	Initial T_B (K)	Learning rate
I	3	100	200	4	1e-3
II	5	70	300	3	5e-4
III	7	50	400	3	2.5e-4
Total training time: 5.45h					

Table 6.2: Training details for the dispersion task.

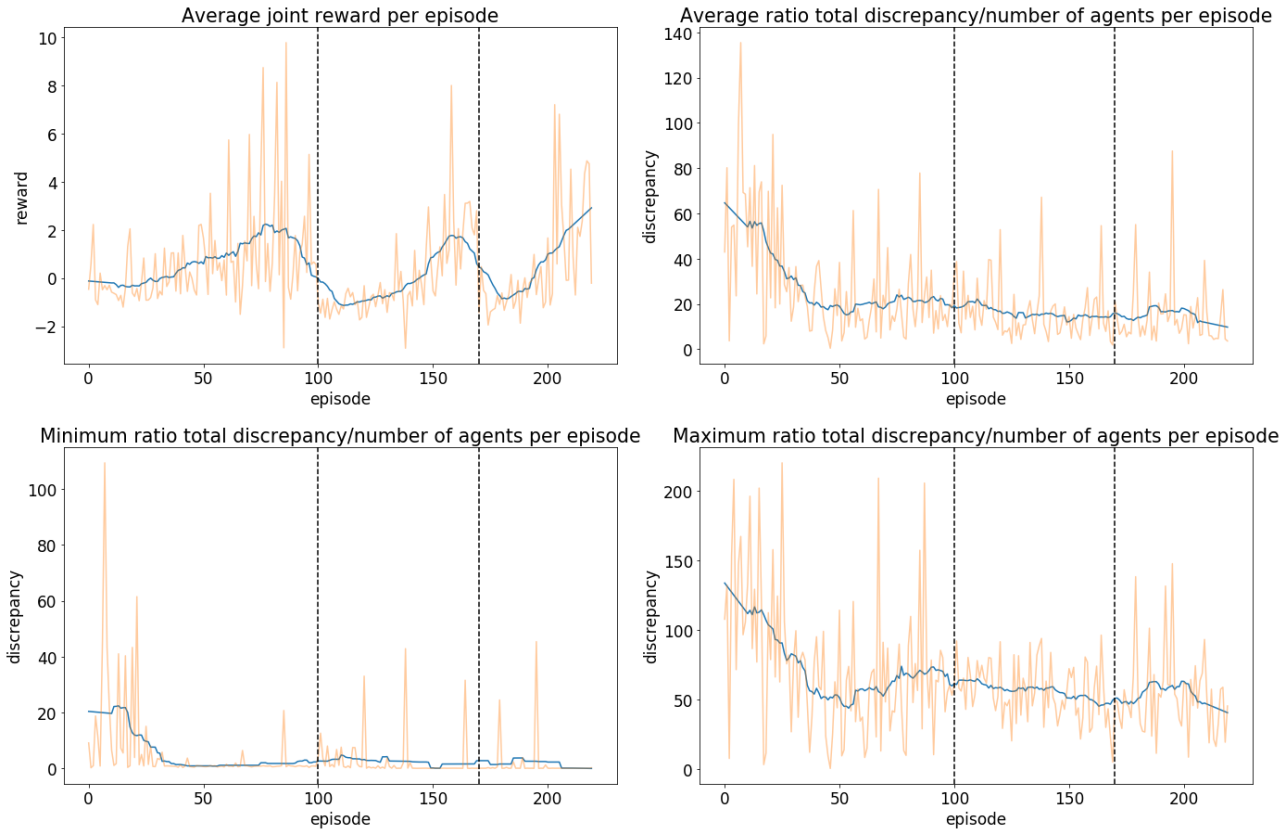


Figure 6.3: Results of the training for the dispersion task. Top left: average joint reward per episode. Top right: average ratio total discrepancy/number of agents per episode. Bottom left: minimum ratio total discrepancy/number of agents per episode. Bottom right: maximum ratio total discrepancy/number of agents per episode. In orange, the data is unfiltered, while in blue, the data is smoothened using a Savitzky-Golay filter of first-order and window size equal to 21. The dashed vertical lines divide the different stages. Author's concept.

6.2.2 Square lattice formation task

Square lattice formation task Upscale factor $R = 1000$, distance $l = 4d_r$ (280mm)					
Stage	No. of robots	No. of episodes	No. of steps/episode	Initial T_B (K)	Learning rate
I	4	150	200	4	1e-3
II	6	80	400	4	1e-3
Total training time: 10.66h					

Table 6.3: Training details for the square lattice formation task.

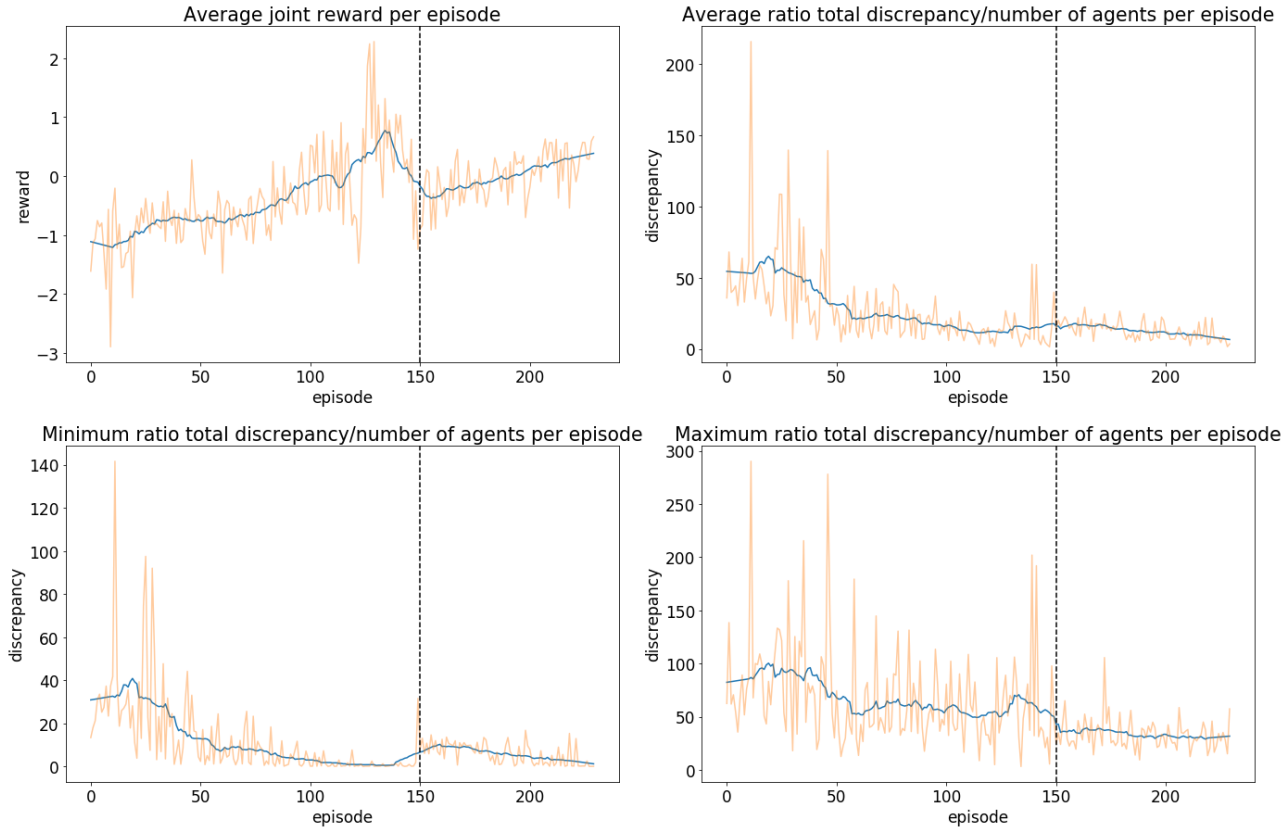


Figure 6.4: Results of the training for the square lattice formation task. Top left: average joint reward per episode. Top right: average ratio total discrepancy/number of agents per episode. Bottom left: minimum ratio total discrepancy/number of agents per episode. Bottom right: maximum ratio total discrepancy/number of agents per episode. In orange, the data is unfiltered, while in blue, the data is smoothened using a Savitzky-Golay filter of first-order and window size equal to 21. The dashed vertical lines divide the different stages. Author's concept.

6.2.3 Aggregation task

Aggregation task Upscale factor $R = 250$					
Stage	No. of robots	No. of episodes	No. of steps/episode	Initial T_B (K)	Learning rate
I	4	50	200	5	2e-3
II	6	50	250	3	1e-3
III	7	50	250	2	1e-3
Total training time: 3.62h					

Table 6.4: Training details for the aggregation task.

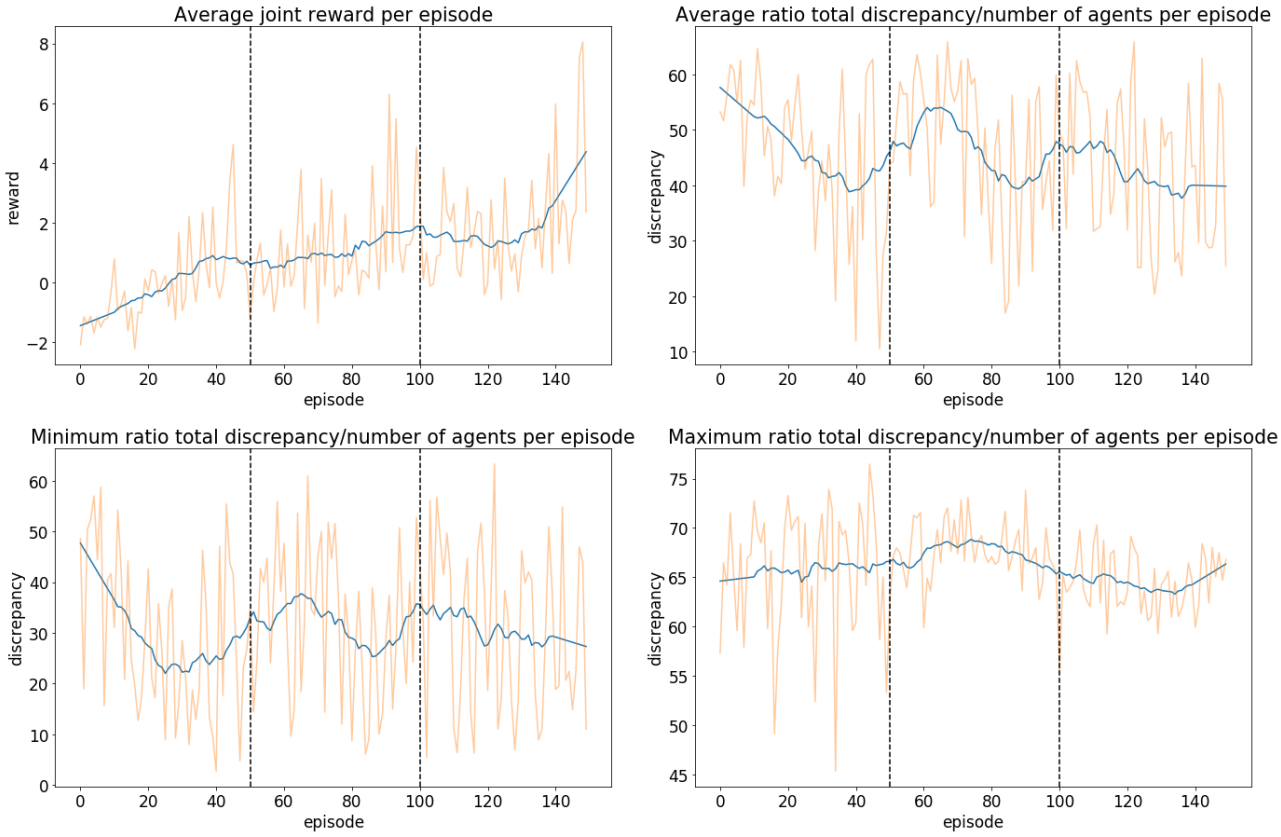


Figure 6.5: Results of the training for the aggregation task. Top left: average joint reward per episode. Top right: average ratio total discrepancy/number of agents per episode. Bottom left: minimum ratio total discrepancy/number of agents per episode. Bottom right: maximum ratio total discrepancy/number of agents per episode. In orange, the data is unfiltered, while in blue, the data is smoothened using a Savitzky-Golay filter of first-order and window size equal to 21. The dashed vertical lines divide the different stages. Author's concept.

6.2.4 Chain formation task

Chain formation task Upscale factors $R_c = 10$ $R_d = 1000$. Distance $l = 4d_r$ (280mm)					
Stage	No. of robots	No. of episodes	No. of steps/episode	Initial T_B (K)	Learning rate
I	3	50	250	5	1.5e-3
II	5	50	250	3	1e-3
III	7	50	400	2	8e-3
Total training time: 5.75h					

Table 6.5: Training details for the chain formation task.

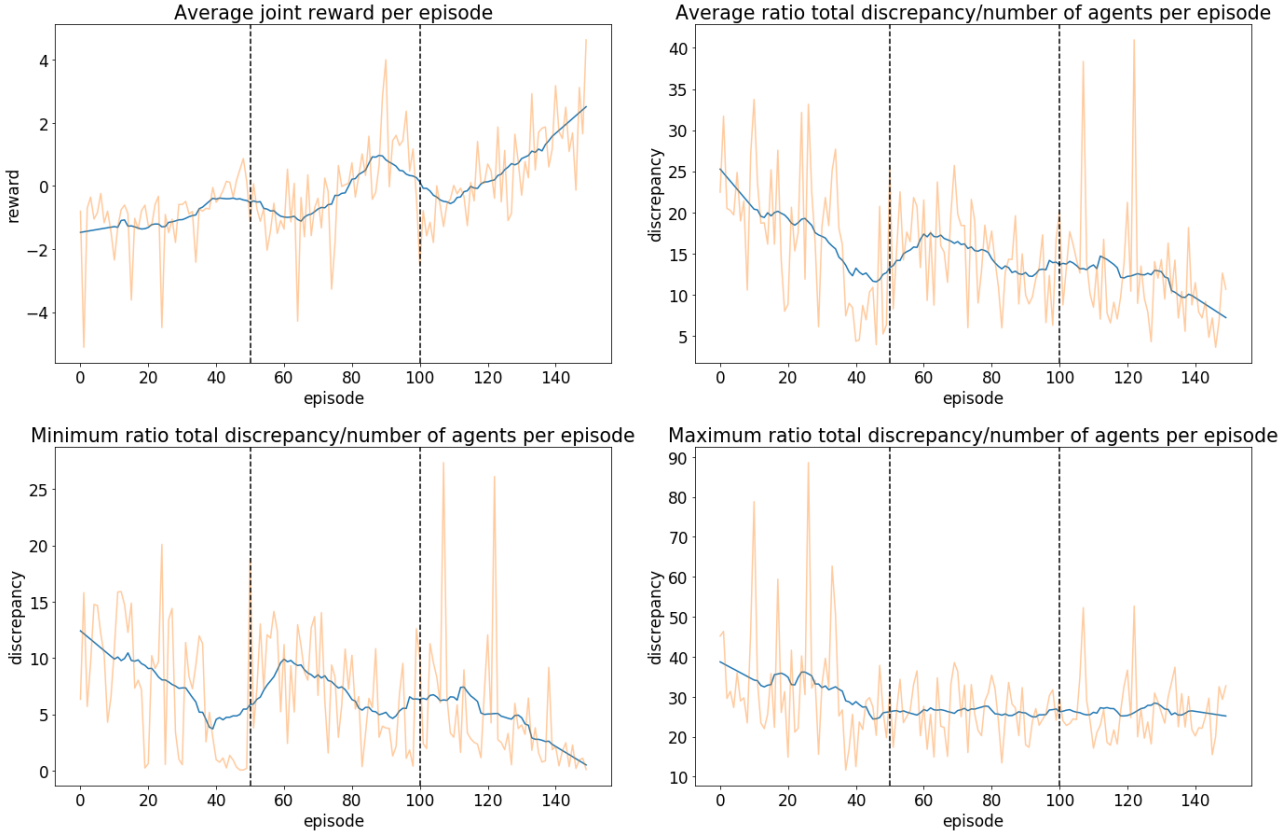


Figure 6.6: Results of the training for the chain formation task. Top left: average joint reward per episode. Top right: average ratio total discrepancy/number of agents per episode. Bottom left: minimum ratio total discrepancy/number of agents per episode. Bottom right: maximum ratio total discrepancy/number of agents per episode. In orange, the data is unfiltered, while in blue, the data is smoothened using a Savitzky-Golay filter of first-order and window size equal to 21. The dashed vertical lines divide the different stages. Author's concept.

6.3 Evaluation

6.3.1 Dispersion

To evaluate how the agents complete the task of dispersion after the training, we will perform 35 tests with different numbers of agents ranging from 3 to 9. In all tests, the agents start at random positions, just like as in the training phase. We shall observe the initial and final values of H_T/n, where n is the number of the agents in the test, as well as the number of collisions and the number of steps the swarm takes to complete the task. During the evaluation phase, the end of the task happens when all the agents of the swarm stop moving around and settle down at a particular position. Table 6.6 lists all the results obtained.

No. of agents n	**No. of tests**	**Average initial** H_T/n	**Average final** H_T/n	**No. of collisions**	**Average no. of steps to end**
3	5	79.40	0.02	0	47
4	5	76.29	0.49	0	70
5	5	79.75	0.14	0	81
6	5	74.16	1.50	1	85
7	5	81.21	4.64	0	75
8	5	78.76	7.85	1	123
9	5	75.10	9.64	1	132

Table 6.6: Summary of tests performed for the dispersion task.

From the results, as the number of agents increases, the swarm encounters more difficulties in completing the task, since the number of steps to the completion increases, as well as the final total discrepancy. Also, collisions are very rare but tend to appear as the number of agents increases. Figures 6.7 and 6.10 illustrate the behavior of the swarm in the execution of the dispersion task from the initial step to the final step. Of all 35 tests performed, there aren't cases with unsatisfactory results. Hence, overall, we can conclude that the performance is very good.

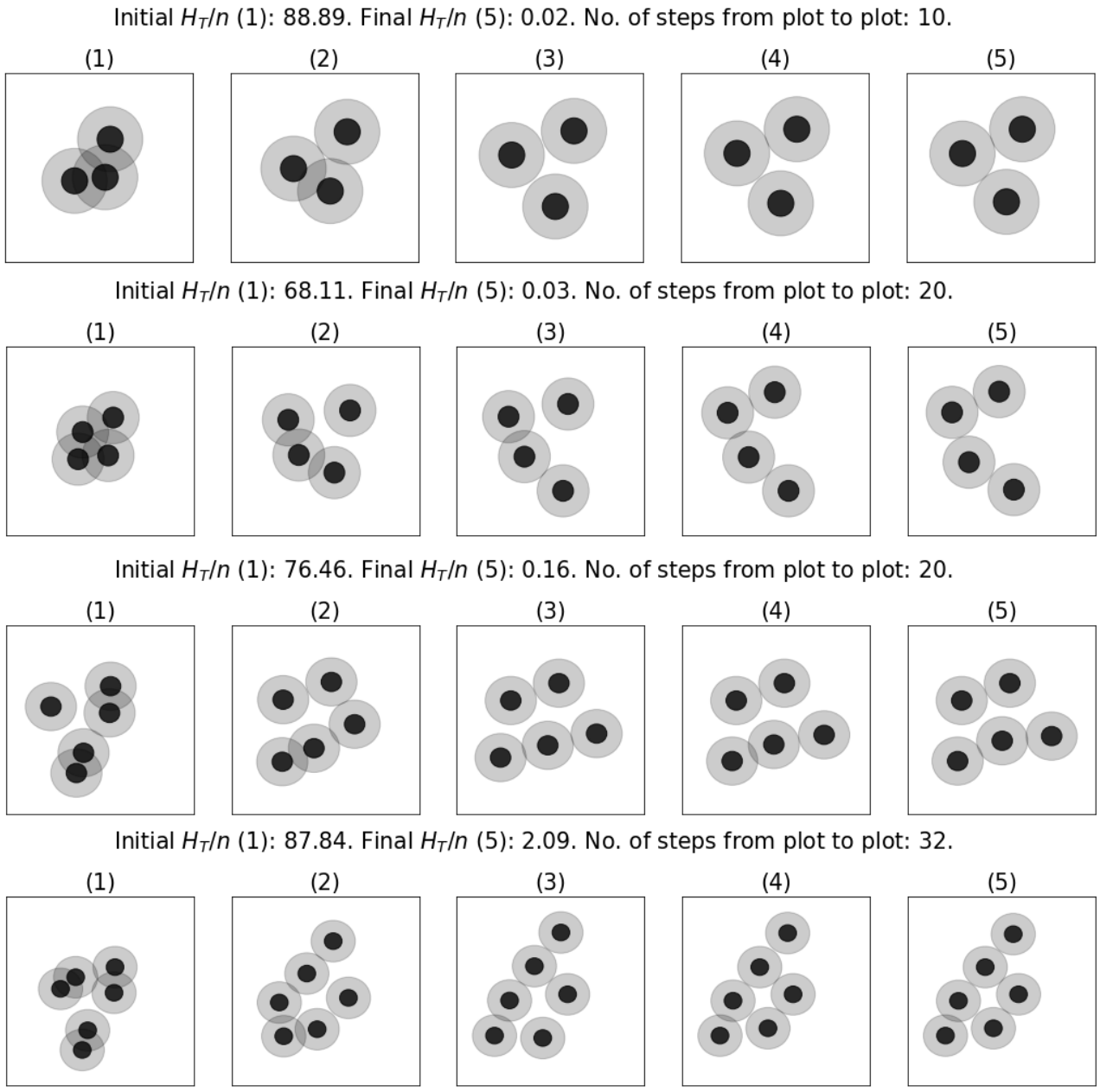


Figure 6.7: Examples of tests with agents performing the dispersion task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle enclosed with a grey circle that indicates the area that the robot can watch. Remember that the dispersion task can be used for surveillance with the goal of maximizing the area the agents can watch while maintaining the swarm connected. Therefore, we look at minimizing the overlapping areas of the gray circles. Author's concept.

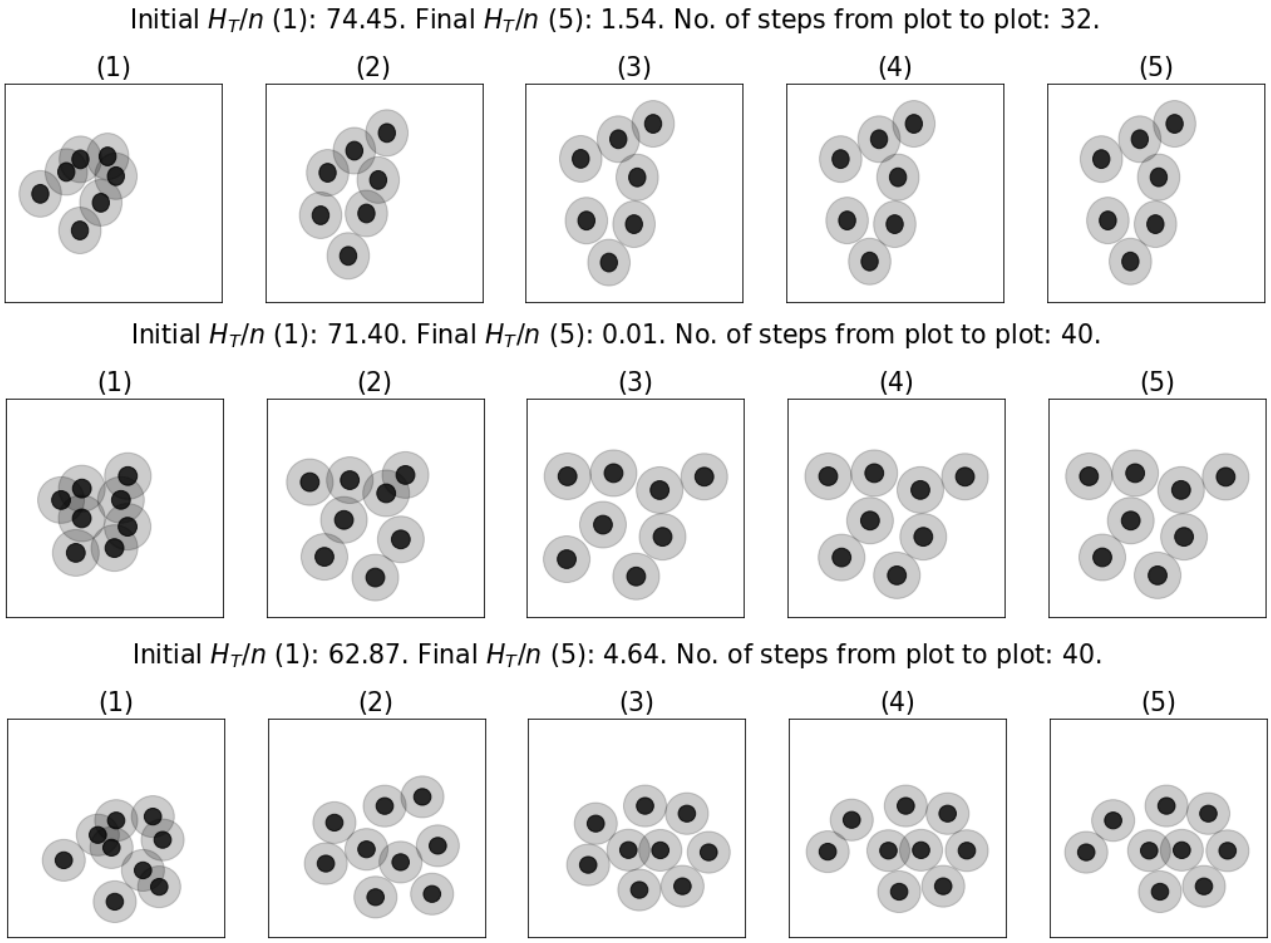


Figure 6.8: More examples of tests with agents performing the dispersion task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle enclosed with a grey circle that indicates the area that the robot can watch. Remember that the dispersion task can be used for surveillance with the goal of maximizing the area the agents can watch while maintaining the swarm connected. Therefore, we look at minimizing the overlapping areas of the gray circles. Author's concept.

6.3.2 Square lattice formation task

For the square lattice formation task, we evaluate the performance of the trained neural network in 35 tests detailed in Table 6.7. We remind that, in all tests, the agents start at random positions as in the training phase.

No. of agents n	**No. of tests**	**Average initial** G_T/n	**Average final** G_T/n	**No. of collisions**	**Average no. of steps to end**
4	10	38.68	0.80	0	88
6	10	36.99	5.12	0	255
8	10	37.29	7.33	0	490
9	5	42.43	7.25	1	740

Table 6.7: Summary of tests performed for the square lattice formation task.

In Figure 6.9, we notice the agents transition from random and unordered positions to regularly spaced configurations that exhibit square patterns[2]. The results are pretty satisfactory, given that there isn't any coordination in the swarm for the completion of the task. Also, we notice that skewed square patterns only appear when the number of agents is greater than 8. We attribute this skewness in the shape to the limitation imposed by the partial observability of the agents, since naturally, if the swarm is large enough up to the point where some agents cannot sense part of it, they cannot adjust their positions to reduce the skewness with respect to the further part of the swarm.

Among all 35 tests performed, we have noticed that two of them (which corresponds to roughly 5% of the cases) show sub-optimal solutions, that is, solutions in which square patterns can be hardly noticed. A clear explanation for this is not possible, since the agents are controlled by neural networks, and these often behave as black boxes. However, in the literature of deep learning [31], sub-optimal solutions are often addressed with more training. Therefore, we believe that more episodes in the training could lead to the extinction of such undesired cases.

[2]Notice that a square lattice does not necessarily mean that the overall shape formed by the robots is a square. Instead, it means that the robots are organized in shapes formed by square patterns.

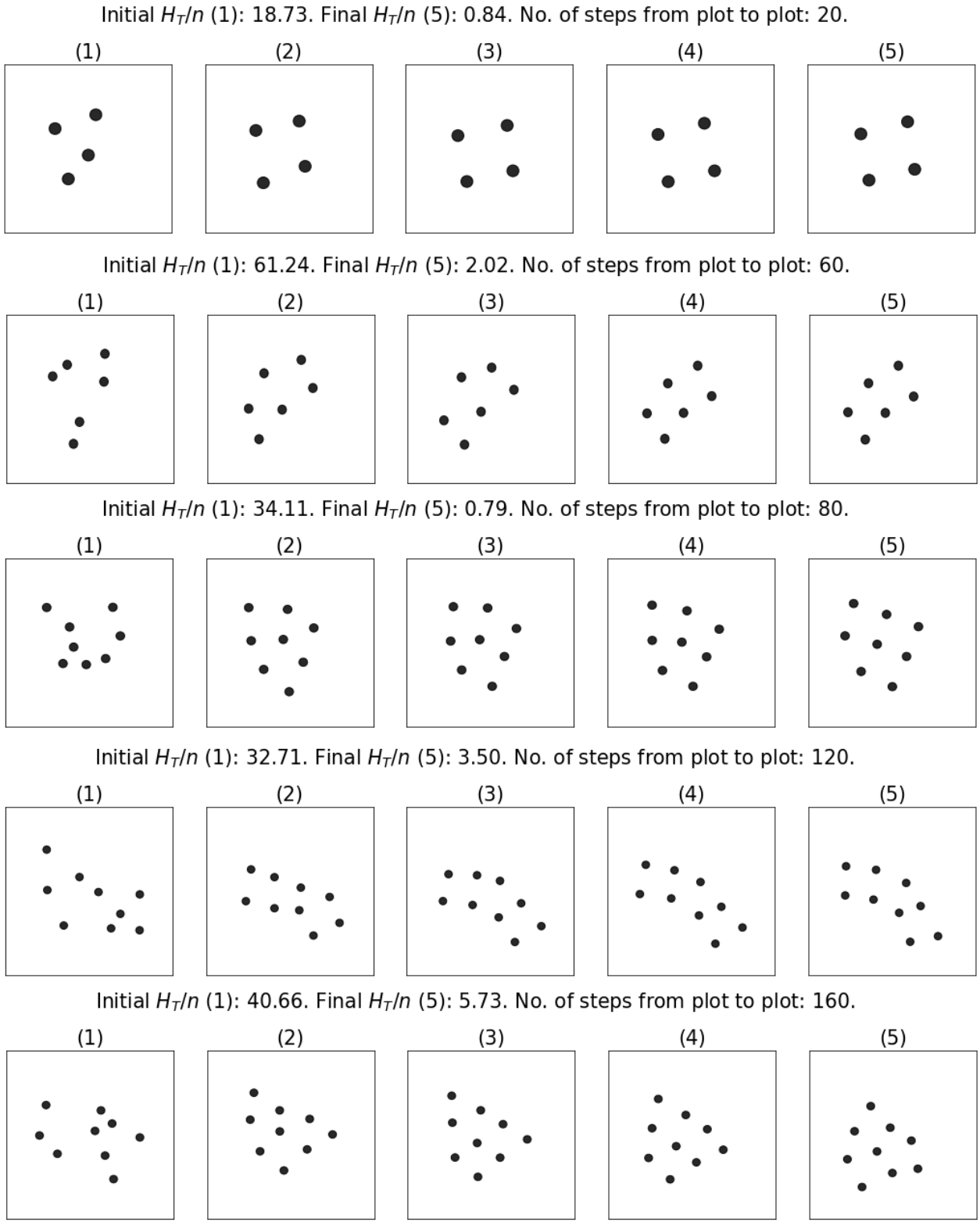


Figure 6.9: Examples of tests with agents performing the square lattice formation task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle. Author's concept.

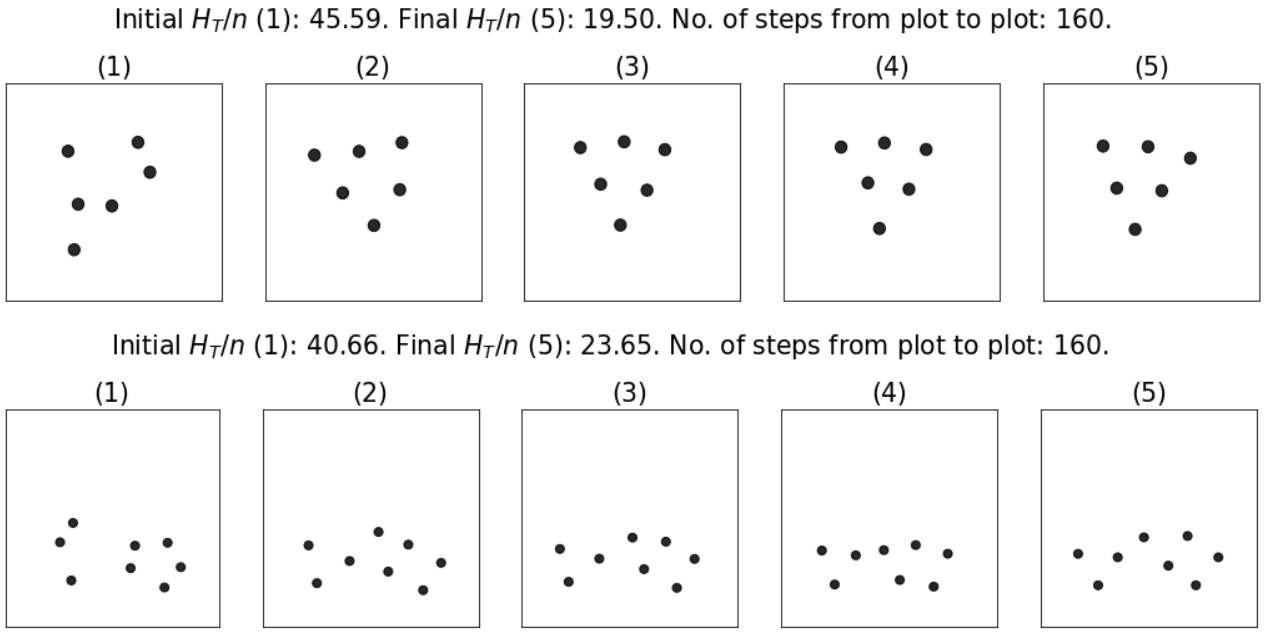


Figure 6.10: Few cases of unsatisfactory results in the square lattice formation task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle. Author's concept.

6.3.3 Aggregation task

Similarly, as we have done for the dispersion task, we will perform 35 tests with different numbers of agents ranging from 3 to 9. In all tests, the agents start at random positions, just like as in the training phase. Again, we observe the initial and final values of I_T/n (which describes the ratio of the total discrepancy by the number of agents in the swarm), as well as the number of collisions and how long it takes to the completion of the task. Table 6.8 lists all the results obtained.

No. of agents n	**No. of tests**	**Average initial** I_T/n	**Average final** I_T/n	**No. of collisions**	**Average no. of steps to end**
3	5	72.07	10.61	0	49
4	5	70.90	22.74	0	56
5	5	70.92	22.77	0	79
6	5	71.40	29.76	0	96
7	5	70.68	30.51	0	92
8	5	70.57	27.19	1	95
9	5	70.49	36.95	0	99

Table 6.8: Summary of tests performed for the aggregation task.

The average final I_T/n is quite above *zero*, however, this does not mean we have unsatisfactory results. We can make sense of these numbers by observing Figures 6.11 and 6.12, which illustrate the behavior of the swarm in the execution of the aggregation task from the initial step to the final step.

We judge that of all 35 tests, all results are pretty satisfactory, especially with respect to collisions. The aggregation task requires agents to gather as close as possible to each other, which significantly increases the chance of collisions. Still, only one collision was detected, and, more interestingly, the agents have learned to maintain a certain distance with respect to each other even though we didn't explicitly specify this in the discrepancy function. This cautious behavior of the agents happens because collisions generate very negative rewards and, since the goal of the agents is to produce a behavior that maximizes the reward obtained, avoiding getting too close to neighboring agents is definitely the way to avoid such negative rewards.

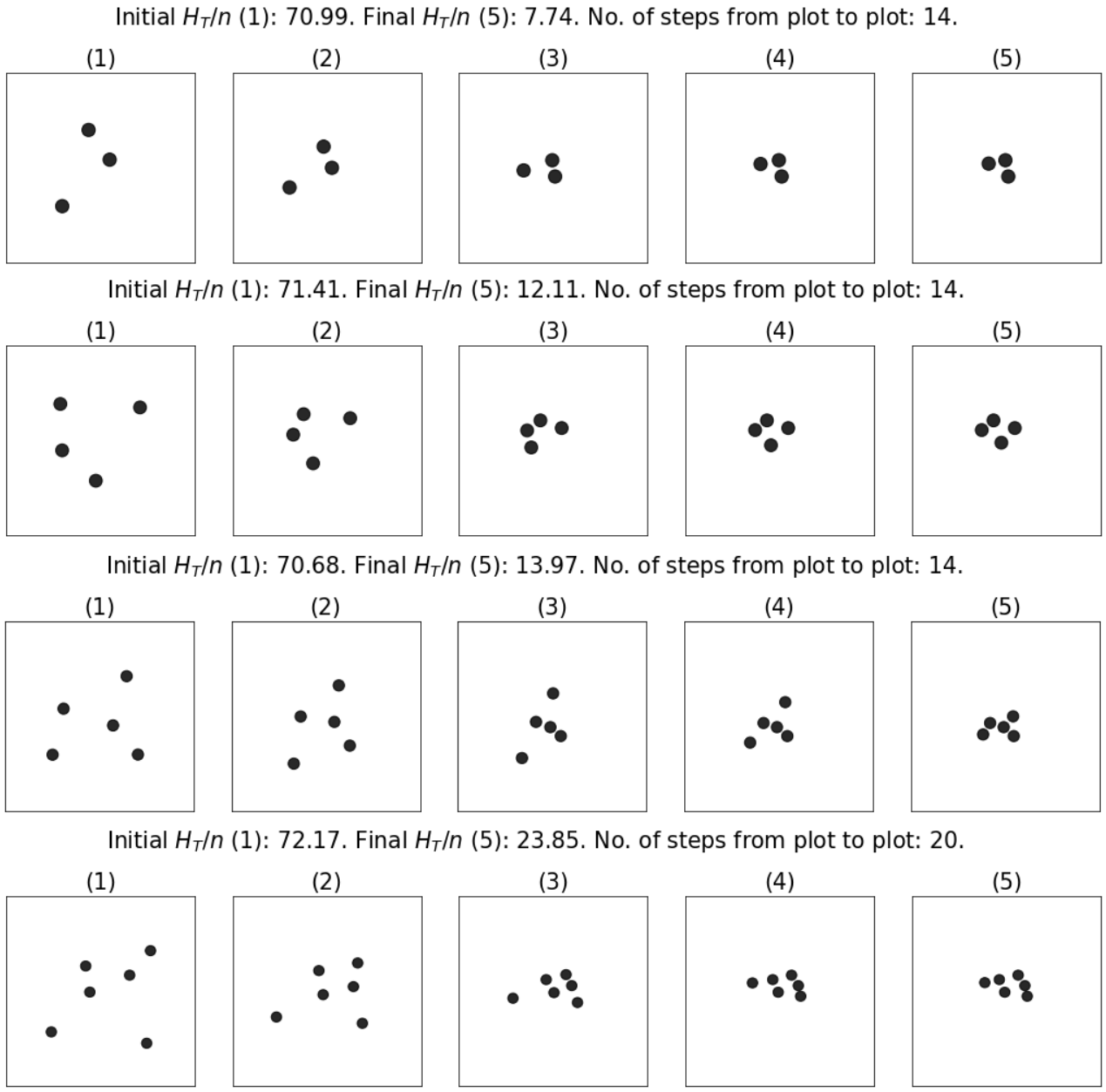


Figure 6.11: Examples of tests with agents performing the aggregation task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle. that indicates the area that the robot can watch. Author's concept.

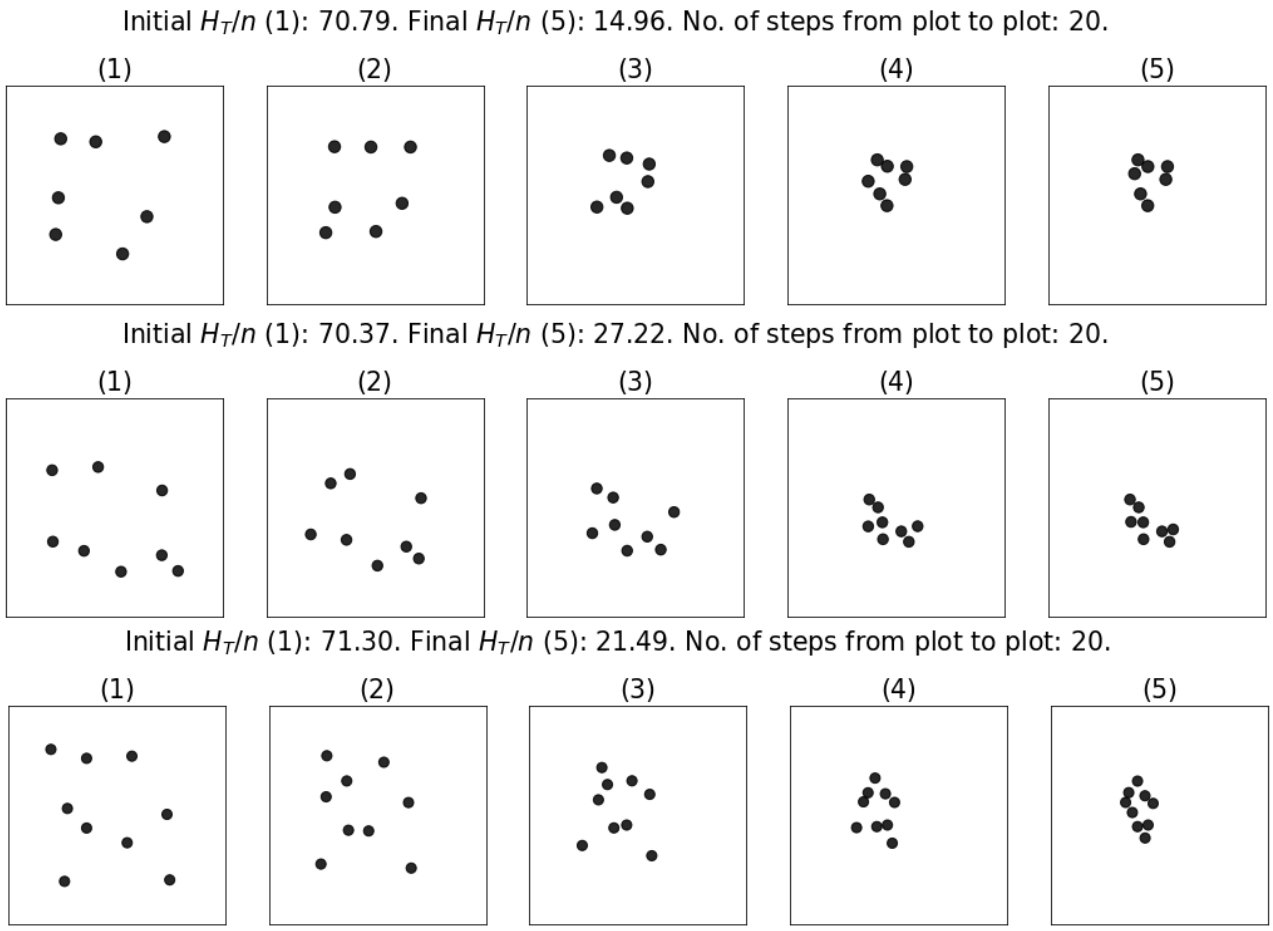


Figure 6.12: More examples of tests with agents performing the aggregation task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle. Author's concept.

6.3.4 Chain formation task

Again, we will perform 35 tests with different numbers of agents ranging from 3 to 9. Table 6.9 lists all the results obtained. Also in this task, the agents start at random positions as it was described at the beginning of this chapter.

No. of agents n	**No. of tests**	**Average initial** J_T/n	**Average final** J_T/n	**No. of collisions**	**Average no. of steps to end**
3	5	27.00	0.35	0	93
4	5	28.99	6.44	0	153
5	5	26.64	8.30	0	251
6	5	27.95	2.52	0	346
7	5	27.73	5.83	1	288
8	5	28.50	4.99	3	538
9	5	28.04	7.02	2	680

Table 6.9: Summary of tests performed for the chain formation task.

The results shown in the table can be better interpreted by Figures 6.13, 6.14 and 6.15. In the first two figures, regardless of the size of the swarm, the agents complete the task with very satisfactory results. Notice that we didn't require collinearity between all agents of the swarm, instead, we required piecewise collinearity, which is observed in the figures. Furthermore, it is very important to notice that agents formed the chains without any prior specification as to where each agent should be located at. This means that, even with minimal communication and absolutely no central coordination, the agents act in harmony with each other deciding about their locations in the chain at runtime based on their neighbors.

Out of all 35 tests, we noticed four cases - shown in Figure 6.15 - where the formed chains have more than three extremities. This shows that in rare cases (less than 12% of the cases in our tests) the swarm is trapped in sub-optimal solutions. As we have explained for the square lattice formation task, these sub-optimal solutions can be approached by increasing the training.

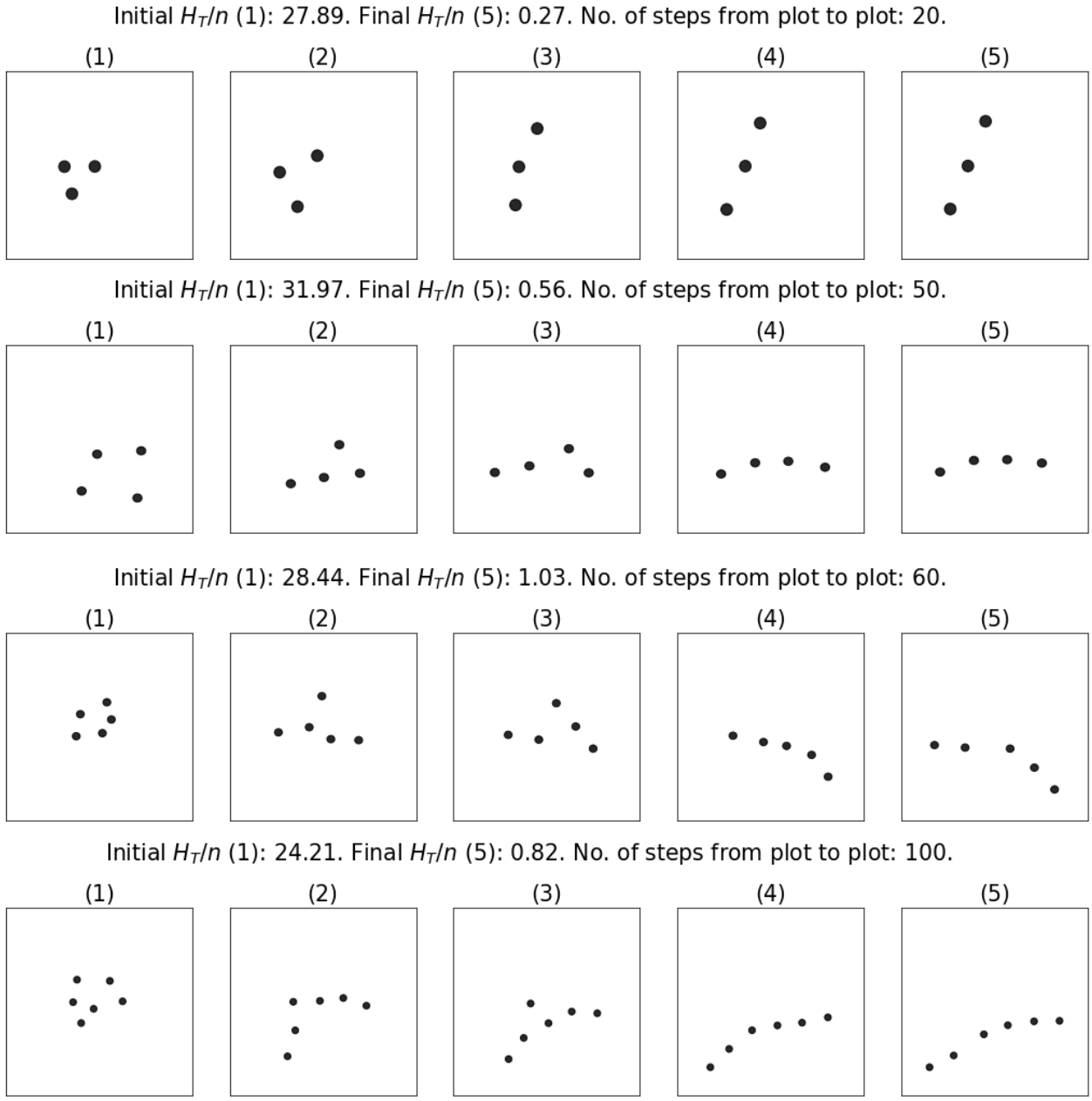


Figure 6.13: Examples of tests with agents performing the chain formation task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle. that indicates the area that the robot can watch. Author's concept.

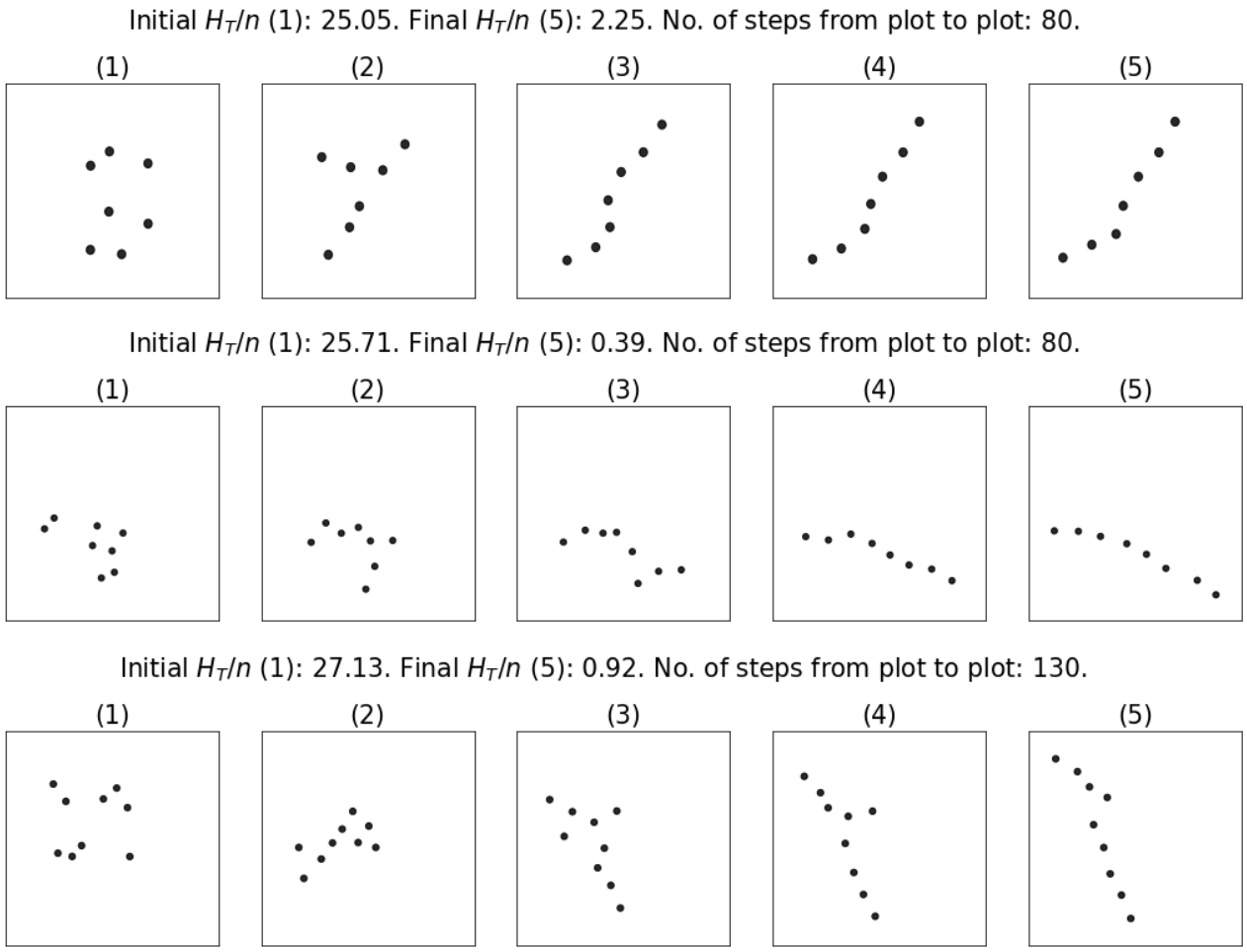


Figure 6.14: More examples of tests with agents performing the chain formation task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle. Author's concept.

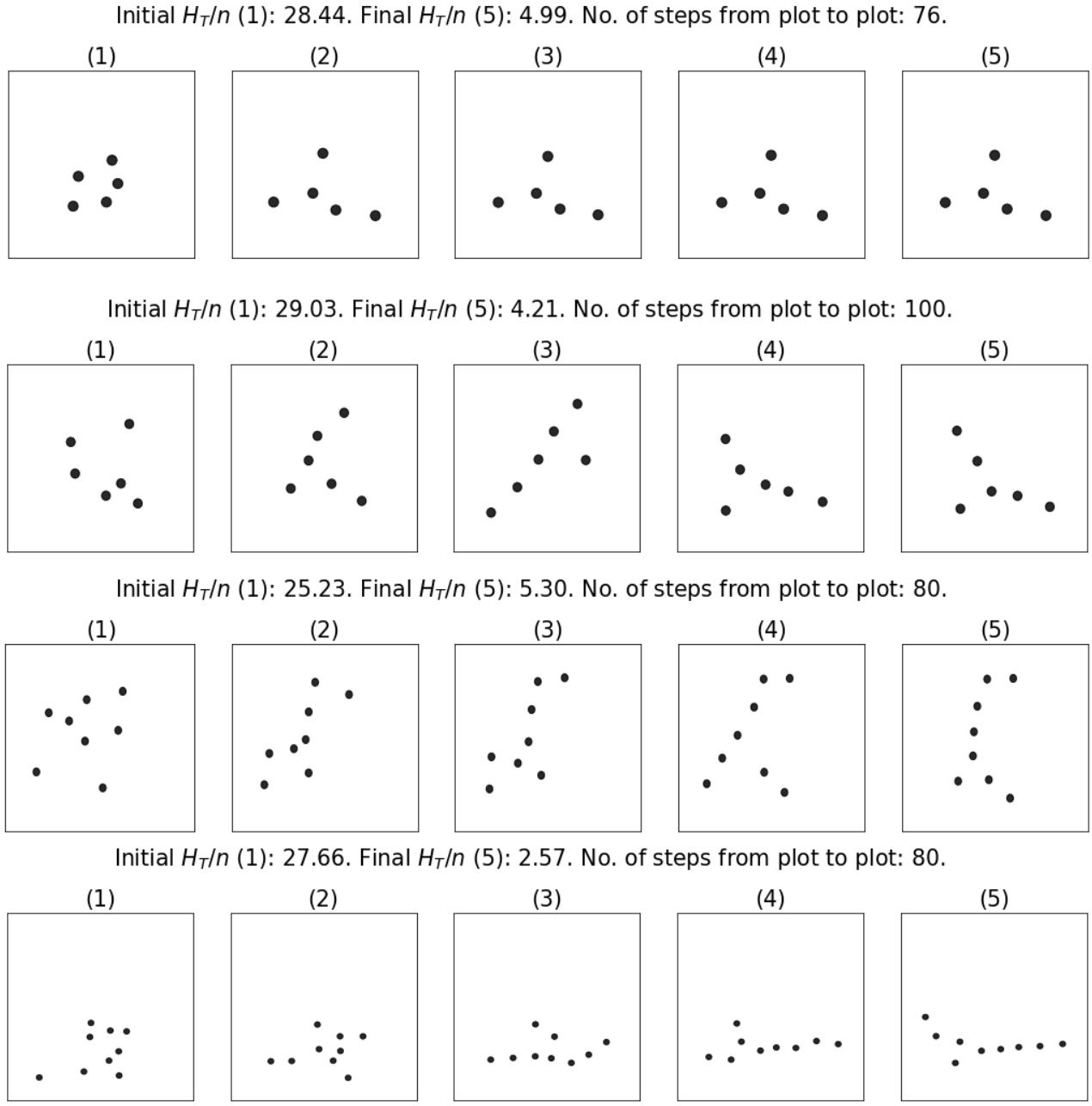


Figure 6.15: Few cases of undesirable (chains with more than two extremities) results in the chain formation task (top view). (1) represents the swarm at the beginning of the task, while (5) represents the swarm at the end of the task. (2), (3) and (4) are intermediate positions. Each robot is represented by a black circle. Author's concept.

6.4 Further comments

During the training phase, we have defined four quantities that indicate if the agents are learning the task. They provide means to decide whether or not to stop the training and look for possible errors and inconsistencies in the algorithm. All the results obtained during the training phase were on par with the expectations.

As for the evaluation phase, we have obtained very good results for all tasks, even though the training time ranged from roughly three to eleven hours[3]. In particular, we've seen that organization and order emerge seamlessly in the tasks of square lattice formation and chain formation, even though they seem fairly complex to be solved without a central coordination that observes the whole state of the swarm, plans the motion of every agent in it and dictates the actions of them. We've also witnessed that the collision avoidance method used is effective and results in the agents preserving a safe distance from their partners. Furthermore, out of the 140 tests performed, very few sub-optimal solutions (corresponding to less than 5% of the cases) appeared which we believe that could be extinguished with additional training. Based on these results, we define the benefits and limitations of our method as follows:

6.4.1 Benefits

- Instead of designing by hand a controller that takes into consideration the sensory readings, the task, the environment state, the robot's kinematics and the robot's state, the robots themselves learn this controller in an end to end fashion. In other words, the robots are never told how a certain task is to be achieved, instead, they learn how to make sense of their inputs and how to perform the task by a sequence of interactions with the environment via their outputs. [31] This can be particularly advantageous when the input space is formed by more complex sensor readings - such as from RGB or RGB-D cameras - or a fusion of different types of sensors, as well as when the task itself is complex.

- The algorithm is flexible as it can be applied to any task by solely designing a discrepancy function for the task.

[3]Impressive results in deep reinforcement learning are often achieved after several days or even weeks of training.

- The algorithm expands the boundaries of swarm robotics to deep learning and the most recent and impressive advances in deep learning such as transfer learning, imitation learning, continual learning and curriculum learning.

6.4.2 Limitations

- Training the agents to perform a task takes several hours and demands high computational costs. Also, sub-optimal solutions can appear if is not enough training is performed. During inference, depending on the hardware that the robots possess, the forward pass of the deep neural network might take long enough to hinder real-time operations.

- Selecting the best combination of hyper-parameters - such as those involving the architecture of neural network architecture or those related to the DDQN rule - is a burden that has to be tackled by trial and error.

- Since the agents are controlled by neural networks and these can be considered as black boxes, performing slight changes in the behavior of the swarm requires retraining or, in the best scenario, workarounds. For example, if we wish to modify the desired distance in our dispersion task, a workaround that would avoid the burden of retraining is preprocessing the sensory inputs by multiplying them by a constant. If our new desired distance is smaller than the previous one, the constant should be greater than one, which means that we enforce the robots to perceive the environment as being further away than it actually is. While this might work for specific tasks and for certain types of sensory inputs (as ultrasonic readings), in general, retraining is necessary.

7

Conclusions

In this work, we have proposed a general model-free deep reinforcement learning algorithm for swarm robotics. The algorithm is built upon the latest advances in deep reinforcement learning with modifications to fit within swarm robotics such as: shifting from a single-agent to a multi-agent application, multi-stage training phase with transfer learning to meet the scalability requirements in swarm robotics, non-resettable prioritized experience memory to avoid catastrophic forgetting at the transitioning between stages, and the use of LSTM cells in the neural networks as well as communication between the agents to reduce the partial observability.

The algorithm provides a flexible and distinct paradigm to the development of behaviors within swarm robotics, by allowing the agents to learn the task themselves in an end to end approach. We proposed the algorithm in such a way that it can be easily adapted to any task by solely designing a discrepancy function for the task, without worrying about aspects as the sensory inputs and the robots' kinematics. The validation of the algorithm was performed in computer simulation for four different tasks and produced very satisfactory results with a relatively short training time. As future work, we cite the following topics.

7.1 Future work

Comparison with other algorithms in swarm robotics

In order to provide better descriptions of benefits and limitations of the algorithm proposed in this work, it is necessary to perform a comparison with common paradigms within swarm robotics such as virtual physics and probabilistic finite state machines. In fact, the literature of swarm robotics also lacks a comparison between such common paradigms.

Noisy targets when training the policy network

The reward functions proposed in this thesis awards each agent solely based on its own actions. In more complex tasks, devising such type of reward function may not be always possible, leaving us with the only option of using reward functions that consider the joint actions of some or all agents when awarding each agent. With the latter type of reward function, an agent can receive a negative reward, even though it has done the best it could, in other words, the reward an agent receives can be incoherent. These incoherent rewards lead to noisy targets when training the policy neural network. This is undesirable, as it certainly harms the learning process. It is a topic for future work to provide ways to detect and discard noisy targets when training the policy network.

Continuous action space

Here we have reduced the complexity of the proposed algorithm by discretizing the action space into a few values. Future improvements could arrive by considering, instead, the latest researches in deep reinforcement learning for continuous action spaces.

Parallel computing

Recently, various authors [10, 28, 48, 51] have proposed deep reinforcement learning algorithms for parallel computing systems. These algorithms utilize several distributed CPUs and GPUs for greatly reducing the training time with respect to single-CPU and single-GPU baselines. Efficiently incorporating parallelization into the algorithm proposed here is not a

trivial task, but worth of study as it could provide great results in more complex situations as when using continuous action spaces, vast number (10+) of agents, and sparse reward functions[1].

Improved communication

We have proposed a simple communication idea in which each agent shares its sensory inputs with neighboring agents. Future work could implement more sophisticated communication methods as done by Foerster *et al.* [17], where agents use deep reinforcement learning itself to also learn how to communicate efficiently.

Imitation learning

Imitation learning in reinforcement learning [36, 77] consists of leveraging expert demonstration data to assist a reinforcement learning agent in the process of learning, i.e., instead of learning a task entirely by itself, the agent is presented with data that demonstrates how the task is to be achieved. The aim of imitation learning is to explore methods of using such demonstration data to accelerate the process of learning, thus reducing the training time. Therefore, allying imitation learning with deep reinforcement learning for swarm robotics can be an interesting and fruitful research topic.

Continual learning

Here we proposed the use of a prioritized replay memory that is never reset when transitioning from one stage to the other. The reason for this was to prevent the agents from unlearning to behave with sizes of the swarm from past stages (we call this continual learning without catastrophic forgetting). One clear downside of this approach is the increase of the size of the prioritized replay memory that leads to a growing computational expense of the training algorithm and longer training duration. More complex and effective methods for continual learning without catastrophic forgetting are reviewed by Parisi *et al.* [55], which could be applied to this work as an improvement.

[1] Sparse reward functions are reward functions that provide rewards sparsely, instead of at every time instant.

Real-life scenarios

Naturally, our ultimate goal is to apply the algorithm on real hardware. The first issue that can arise when transferring the algorithm to real cases is the processing power limitation imposed by the hardware of the robots. Depending on the complexity, deep neural networks can be impossible to be implemented on certain robots for real-time operations. Hence, the user needs to be aware of this before using the algorithm proposed here.

As we discussed before, our algorithm is comprised of two phases: the training phase and the inference phase. Based on [77], we argue here that is possible to carry out all the training phase in simulations and directly deploy the trained target neural network on real hardware, expecting a similar performance in real scenarios in comparison with the performance obtained in simulated environments. However, for this, it is necessary that the simulations reproduce real scenarios with very good fidelity, including factors as the varying battery levels of the robots, possible delays, packet loss and data corruption during communication and noisy sensor measurements. In case the simulations do not provide such high level of fidelity, a performance degradation is expected; and, in such case, we propose to fine tune the model performing an extra but shorter training phase on real cases.

Performing training on real scenarios requires the implementation of an observer that is aware of the global state of the environment, which requires the use of computer vision algorithms as we discussed in Chapter 5. Also, the robots are prone to collisions when training in real scenarios. To avoid damage, one should employ low-level algorithms that immediately stop the robot when a collision is imminent.

When training in real scenarios, the learner, and the prioritized replay memory have to be implemented on a computer that possesses enough computational power and storage. The prioritized replay memory occupies a significant amount of space that hinders it from being implemented onto the robots. Also, the learner is a very computational demanding part of the algorithm that samples batches of experiences from the prioritized replay memory, applies the Double DQN rule, calculates the gradients of the online neural network, performs the optimization step and updates the neural network weights. Therefore, it is best if the online and the target neural networks are implemented on the computer, instead of on the robots. If they are implemented on the robots, the algorithm can suffer a significant slow-downs due

to delays caused by the communication (wireless) between the learner and the prioritized replay memory. Not to mention the possibilities of packet loss during the communication.

If the neural networks are implemented on the computer, there isn't any reason to not do the same for the exploration and exploitation strategy. Hence, during the training phase, the robots do not perform any computation. They only communicate with the computer by periodically sending their sensory inputs and receiving as a response the joint velocities to use.

Depending on the available computational power, the training of the online neural network can be either performed during the execution of the episodes or at the end of them. If the training happens during the execution of the episode, it is necessary to avoid race conditions that appear when accessing the neural network by the training algorithm and by the inference algorithm[2].

The implementation of communication networks between the agents of the swarm for real-time operations has been well explored in the literature of swarm robotics. For this intent, the reader is advised to refer to works as [11, 14, 46].

Finally, during the inference phase, the robots need to have their neural networks implemented on their hardware, since we do not want to limit the operation of the swarm to a central device.

[2]Given some inputs, the inference algorithm performs a forward pass on the neural network and obtains its outputs. During training, inference happens in the decision process of determining joint velocities to the robots.

Acknowledgments

I hereby express my gratitude to my supervisor, Prof. D.Sc. Teresa Zielińska, who has always assisted in the quality of this work, and to Prof. Marco Baglietto who has provided me with additional advice before the final submission of this work.

In addition, I thank the EMARO+ program, all the personnel involved in it and the funding for making this work possible.

Bibliography

[1] Aleksandar Jevtic and Diego Andina. (2007) Swarm Intelligence and Its Applications in Swarm Robotics. In: 6th WSEAS Int. Conference on Computational Intelligence, Man-Machine Systems and Cybernetics, pages 41-46, Tenerife, Spain.

[2] Aleksis Liekna and Janis Grundspenkis. (2014). Towards practical application of swarm robotics: Overview of swarm tasks. Engineering for Rural Development. 13. pages 271-277.

[3] Arulkumaran K., Deisenroth M. P., Brundage M., Bharath A. A. (2017) Deep Reinforcement Learning: A Brief Survey. In: IEEE Signal Processing Magazine, vol. 34, no. 6, pages 26-38.

[4] Bennet D. J., McInnes C. R. (2010) Distributed control of multi-robot systems using bifurcating potential fields. In: Robotics and Autonomous Systems, Volume 58, Issue 3, pages 256-264.

[5] Blum C., Groß R. (2015) Swarm Intelligence in Optimization and Robotics. In: Springer Handbook of Computational Intelligence, pages 1291-1309. Springer, Berlin, Heidelberg.

[6] Blum C., Li X. (2008) Swarm Intelligence in Optimization. In: Swarm Intelligence - Introduction and Applications, Springer, pages 43-85.

[7] Brambilla M., Ferrante E., Birattari, Dorigo M. (2013) Swarm robotics: a review from the swarm engineering perspective. In: Swarm Intelligence, vol. 7, Issue 1, pages 1-41.

[8] Busoniu L., Babuska R., De Schutter B. (2010) Multi-agent reinforcement learning: An overview. In: Innovations in Multi-Agent Systems and Applications, chapter 7, vol. 310 of Studies in Computational Intelligence, pages 183-221, Springer, Germany.

[9] Carmel, D., Markovitch, S. (1999) Exploration Strategies for Model-Based Learning in Multi-agent Systems. Autonomous Agents and Multi-agent systems. 2(2), pages 141-172.

[10] Clemente A. V., Castejón H. N., Chandra, A. (2017) Efficient Parallel Methods for Deep Reinforcement Learning. Published in ArXiv: abs/1705.04862.

[11] Cianci C.M., Raemy X., Pugh J., Martinoli A. (2007) Communication in a Swarm of Miniature Robots: The e-Puck as an Educational Tool for Swarm Robotics.Swarm Robotics. SR 2006. Lecture Notes in Computer Science, vol 4433, pages 103-115. Springer, Berlin, Heidelberg

[12] Dada E.G., Ramlan E.I. (2015) A Hybrid Primal-Dual-PSO (pdipmPSO) Algorithm for Swarm Robotics Flocking Strategy. The Second International Conference on Computing Technology and Information Management (ICCTIM2015). Malaysia, IEEE, pages 93-98.

[13] 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. (2016) Mastering the Game of Go with Deep Neural Networks and Tree Search. Nature, 529(7587), pages 484-489.

[14] Di Caro G.A., Ducatelle F., Gambardella L.M. (2009) Wireless Communications for Distributed Navigation in Robot Swarms. In: Applications of Evolutionary Computing. EvoWorkshops 2009. Lecture Notes in Computer Science, vol 5484, pages 21-30, Springer, Berlin, Heidelberg.

[15] Di Mario E., Navarro I., Martinoli A. (2016) Distributed Learning of Cooperative Robotic Behaviors Using Particle Swarm Optimization. In: Experimental Robotics. Springer Tracts in Advanced Robotics, vol 109, pages 591-604. Springer, Cham.

[16] Don Miner. (2007) Swarm Robotics Algorithms: A Survey. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.4450. Access date: 11 November 2017

[17] Foerster J.N., Assael Y.M., de Freitas N., Whiteson S. (2016) Learning to Communicate to Solve Riddles with Deep Distributed Recurrent Q-Networks. Published in ArXiV: abs/1602.02672.

[18] Garnier S., Jost C., Gautrais J., Asadpour M., Caprari G., Jeanson R., Grimal A., Theraulaz G. (2008) The embodiment of cockroach aggregation behavior in a group of micro-robots, Artificial Life 14(4), pages 387-408.

[19] Glorot X., Bengio Y. (2010) Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics, pages 249-256.

[20] Guo H., Meng Y., Yaochu J. (2011) Swarm robot pattern formation using a morphogenetic multi-cellular based self-organizing algorithm. In: Proceedings - IEEE International Conference on Robotics and Automation 2011, pages 3205-3210.

[21] Ernest H. Williams. (2005) The Nature Handbook: A Guide to Observing the Great Outdoors. First edition. Oxford University Press.

[22] Fikret E.M., Li X., Liang X.M. (2010) A regular tetrahedron formation strategy for swarm robots in three-dimensional environment. In: Hybrid artificial intelligence systems, lecture notes in computer science, vol. 6076, pages 24-31, Springer.

[23] Hado van Hasselt, Arthur Guez, and David Silver. (2016) Deep reinforcement learning with double Q-Learning. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16). AAAI Press, pages 2094-2100.

[24] Hamann H., Wörn H. (2007) An Analytical and Spatial Model of Foraging in a Swarm of Robots. In: Swarm Robotics. SR 2006. Lecture Notes in Computer Science, vol. 4433, pages 43-55. Springer, Berlin, Heidelberg.

[25] Hettiarachchi S., Spears William M. (2009) Distributed adaptive swarm for obstacle avoidance. International Journal of Intelligent Computing and Cybernetics 2009, 2(4), pages 644-671.

[26] Hessel M., Modayil J., van Hasselt H., Schaul T., Ostrovski G., Dabney W., Horgan D., Piot B., Azar M., Silver D. (2017) Rainbow: Combining Improvements in Deep Reinforcement Learning. Published in ArXiv: abs/1710.02298.

[27] Hochreiter S. and Schmidhuber J. (1997) Long short-term memory. In: Neural Computation, vol. 9, no. 8, pages 1735-1780.

[28] Horgan D., Quan J., Budden D., Barth-Maron G., Hessel M., van Hasselt H., Silver D. (2018) Distributed Prioritized Experience Replay. Published in ArXiv: abs/1803.00933.

[29] Hüttenrauch M. (2016) Guided Deep Reinforcement Learning for Robot Swarms. Master Thesis. Published in ArXiv: abs/1709.06011.

[30] Hüttenrauch M., Šošić A., Neumann G. (2017) Guided Deep Reinforcement Learning for Swarm Systems. Published in ArXiv: abs/1709.06011.

[31] Jahanzaib Shabbir, and Tarique Anwer. (2018) A Survey of Deep Learning Techniques for Mobile Robot Applications . Published in arXiv: abs/1803.07608.

[32] Kaelbling L. P., Littman M. L., Moore A. W. (1996) Reinforcement Learning: A Survey. In: Journal of Artificial Intelligence Research, Vol 4, pages 237-285.

[33] Kanagaraj G., Ponnambalam S.G., Yogeswaran, M. (2013) Reinforcement learning in swarm-robotics for multi-agent foraging-task domain. 2013 IEEE Symposium on Swarm Intelligence (SIS), pages 15-21.

[34] Kemker R., Abitino A., McClure M., Kanan C. (2017) Measuring Catastrophic Forgetting in Neural Networks. Published in ArXiv: abs/1708.02072.

[35] Kingma D.P., Ba J. (2014) Adam: A Method for Stochastic Optimization. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. ArXiv: abs/1412.6980.

[36] Le H. M., Jiang N., Agarwal A., Dudík M., Yue Y., Daume H. (2018) Hierarchical Imitation and Reinforcement Learning. Published in ArXiv: abs/1803.00590.

[37] Levent Bayındır, A Review of Swarm Robotics Tasks. (2016) Neurocomputing, Volume 172, pages 292-321.

[38] Lopes, Y.K., Trenkwalder, S.M., Leal, A.B. et al. (2016) Supervisory control theory applied to swarm robotics. Swarm Intelligence Volume 10, Issue 1, pages 65-97.

[39] Ludwig L., Gini M. (2006) Robotic swarm dispersion using wireless intensity signals. In: Proceedings of the International Symposium on Distributed Autonomous Robotic Systems, pages 28-34.

[40] Marc J. Klowden. (2013) Chapter 12 - Communication Systems. In Physiological Systems in Insects (Third Edition), pages 603-647, Academic Press, San Diego.

[41] Martinson E., Payton D. (2005) Lattice Formation in Mobile Autonomous Sensor Arrays. In: Şahin E., Spears W.M. (eds) Swarm Robotics. SR 2004. Lecture Notes in Computer Science, vol 3342, pages 98-111, Springer, Berlin, Heidelberg.

[42] Mathews E. (2012) Self-organizing ad-hoc mobile robotic networks. Ph.D. thesis, Paderborn, University of Paderborn.

[43] Mayet R., Roberz J., Schmickl T., Crailsheim K. (2010) Antbots: A Feasible Visual Emulation of Pheromone Trails for Swarm Robots. In: Dorigo M. et al. (eds) Swarm Intelligence. ANTS 2010. Lecture Notes in Computer Science, vol 6234, pages 84-94. Springer, Berlin, Heidelberg.

[44] McLurkin J., Smith J. (2007) Distributed Algorithms for Dispersion in Indoor Environments Using a Swarm of Autonomous Mobile Robots. In: Distributed Autonomous Robotic Systems 6, pages 399-408. Springer, Tokyo.

[45] Michael Rubenstein, Alejandro Cornejo, Radhika Nagpal. (2014) Programmable self-assembly in a thousand-robot swarm. In Science, vol. 345, issue 6198, pages 795-799.

[46] Ming Li, Kejie Lu, Hua Zhu, Min Chen, Shiwen Mao and B. Prabhakaran. (2008) Robot swarm communication networks: Architectures, protocols, and applications. In: Third International Conference on Communications and Networking in China, pages 162-166.

[47] Mondada F., Bonani M., Raemy X., Pugh J., Cianci C., Klaptocz A., Magnenat S., Zufferey J.-C., Floreano D. and Martinoli A. (2009) The e-puck, a Robot Designed for Education in Engineering. In: Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions, 1(1), pages 59-65.

[48] Mnih V., Badia A. P., Mirza M., Graves A., Lillicrap T. P., Harley T., Silver D. and Kavukcuoglu K. (2016) Asynchronous Methods for Deep Reinforcement Learning. Published in ArXiv: abs/1602.01783.

[49] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D. (2015) Human-level control through deep reinforcement learning. Nature, 518, pages 529-533.

[50] Morihiro K., Isokawa T., Nishimura H., Matsui N. (2006) Emergence of Flocking Behavior Based on Reinforcement Learning. In: Knowledge-Based Intelligent Information and Engineering Systems. KES 2006. Lecture Notes in Computer Science, vol 4253, pages 699-706. Springer, Berlin, Heidelberg.

[51] Nair A., Srinivasan P., Blackwell S., Alcicek C., Fearon R., De Maria A., Panneershelvam V., Suleyman M., Beattie C., Petersen S., Legg S., Mnih V., Kavukcuoglu K., Silver D. (2015) Massively Parallel Methods for Deep Reinforcement Learning. Published in ArXiv: abs/1507.04296.

[52] Morlok R., Gini M. (2007) Dispersing robots in an unknown environment. In: Distributed Autonomous Robotic Systems 6, pages 253-262. Springer, Tokyo.

[53] Nasseri M.A., Asadpour M. (2011) Control of flocking behavior using informed agents: an experimental study. In: IEEE symposium on swarm intelligence, pages 1-6.

[54] Navarro I., Matía F. (2013) An Introduction to Swarm Robotics. ISRN Robotics, v. 2013, Article ID 608164, 10 pages.

[55] Parisi G. I., Kemker R., Part J. L., Kanan C., Wermter S. (2018) Continual Lifelong Learning with Neural Networks: A Review. Published in ArXiv: abs/1802.07569.

[56] Reynolds C.W. (1987) Flocks, herds and schools: a distributed behavioral model. ACM SIGGRAPH Computer Graphics, 21(4), pages 25-34.

[57] Richard S. Sutton. (1990) Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In Proceedings of the 7th international conference on Machine Learning, pages 216-224, Morgan Kaufman, San Mateo CA.

[58] Richard S. Sutton and Andrew G. Barto. (2017) Reinforcement Learning: An Introduction. Second edition. The MIT Press.

[59] Sadat S. A., Vaughan R. T. (2010) SO-LOST: An ant-trail algorithm for multirobot navigation with active interference reduction. In: Proceedings of the Alife XII Conference, pages 687-693.

[60] Şahin E., Girgin S., Bayindir L., Turgut A.E. (2008) Swarm Robotics. In: Blum C., Merkle D. (eds) Swarm Intelligence. Natural Computing Series, pages 87-100, Springer, Berlin, Heidelberg.

[61] Schaul T., Quan J., Antonoglou I, Silver D. (2016) Prioritized Experience Replay. In: International Conference on Learning Representations (ICLR). ArXiv: abs/1511.05952.

[62] Shucker B., Bennett J. K. (2007) Scalable control of distributed robotic macrosensors. In: Distributed Autonomous Robotic Systems 6, pages 379-388, Springer, Japan.

[63] Spears W.M., Spears D.F., Hamann J.C. et al. (2004) Distributed, Physics-Based Control of Swarms of Vehicles. Autonomous Robots (2004) 17 (2-3), pages 137-162.

[64] Soysal O., Bahçeci E., Şahin E. (2007). Aggregation in swarm robotic systems: evolution and probabilistic control. Turkish Journal of Electrical Engineering and Computer Sciences, 15(2), pages 199-225.

[65] Soysal O., Şahin E. (2005) Probabilistic aggregation strategies in swarm robotic systems. In Proceedings of the IEEE swarm intelligence symposium. Piscataway: IEEE Press, pages 325-332.

[66] Tarquino P. and Nickels K. (2014) Programming an E-Puck Robot to Create Maps of Virtual and Physical Environments. Advances in Intelligent Systems and Computing, 274, pages 13-28.

[67] Trianni V., Campo A. (2015) Fundamental Collective Behaviors in Swarm Robotics. In: Springer Handbook of Computational Intelligence, pages 1377-1394. Springer, Berlin, Heidelberg.

[68] Trianni V., Groß R., Labella T.H., Şahin E., Dorigo M. (2003) Evolving Aggregation Behaviors in a Swarm of Robots. In: Advances in Artificial Life. ECAL 2003. Lecture Notes in Computer Science, vol 2801, pages 865-874. Springer, Berlin, Heidelberg.

[69] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver and Daan Wierstra. (2016) Continuous Control with Deep Reinforcement Learning. Published in ArXiv: abs/1509.02971.

[70] Tuyls K. and Weiss G. (2012) Multiagent Learning: Basics, Challenges, and Prospects. AI Magazine, 33(3), pages 41-52.

[71] Ugur E., Turgut A. E., Şahin E. (2007) Dispersion of a swarm of robots based on realistic wireless intensity signals. In: Proceedings of the 22nd International Symposium on Computer and Information Sciences, pages 1-6.

[72] Vaughan R. T. , Støy K., Sukhatme G. S. , Mataric M. J. (2002). Lost: Localization-space trails for robot teams. IEEE Transactions on Robotics and Autonomous Systems, 18(5), pages 796-812.

[73] Vaughan R. T. , Støy K., Sukhatme G. S. , Mataric M. J. (2000) Whistling in the dark: cooperative trail following in uncertain localization space. In: Proceedings of the fourth international conference on Autonomous agents, ACM, pages 187-194.

[74] Wang Z., Schaul T., Hessel M., Van Hasselt H., Lanctot M., De Freitas N. (2016) Dueling network architectures for deep reinforcement learning. In: Proceedings of the 33rd International Conference on International Conference on Machine Learning, vol. 48, pages 1995-2003.

[75] Watkins C.J.C.H., Daya P. (1992) Q-learning. Machine Learning 8, pages 279-292.

[76] Ying Tan. (2015) Handbook of Research on Design, Control, and Modeling of Swarm Robotics (Advances in Computational Intelligence and Robotics). Information Science Publishing.

[77] Zhu Y., Wang Z., Merel J., Rusu A., Erez T., Cabi S., Tunyasuvunakool S., Kramár J., Hadsell R., de Freitas N., Heess N. (2018) Reinforcement and Imitation Learning for Diverse Visuomotor Skills. Published in ArXiv: abs/1802.09564.

[78] Tan Y., Zheng Z. (2013) Research Advance in Swarm Robotics. In Defence Technology, Volume 9, Issue 1, 2013, pages 18-39.

Appendices

A

Code

The code developed for this work is available at:

https://github.com/claytonfk/A_DRL_algorithm_for_SR

Requirements

To run the code, the following requirements should be met. The code has not been tested with earlier or later versions of software and libraries described below, thus forward and backward compatibilities are not guaranteed, however, they are expected.

- The simulation software V-REP educational version 3.5.
- A CUDA-enabled graphics processing unit.
- The TensorFlow machine learning library for Python, version 1.3.0 as well as Python programming language, version 3.6.
- Other necessary libraries are matplotlib (2.2.2), numpy (1.14.2) and scipy (1.1.0).
- Windows 64bit, Linux 64bit or Mac OSX operating system.

Usage

As the first step before running the code - either the training or the evaluation code - it is necessary to execute V-REP and set port 19999 for the communication between the Python code and the software. This can be done by inputting the following command into the console:

On Windows: start vrep.exe -gREMOTEAPISERVERSERVICE_19999_FALSE_TRUE

On Linux: ./vrep.sh -gREMOTEAPISERVERSERVICE_19999_FALSE_TRUE

On Mac: ./vrep.app/Contents/MacOS/vrep -gREMOTEAPISERVERSERVICE_19999_FALSE_TRUE

After properly executing V-REP, the user is required to open a scene file containing the number of robots that will be used. The scene files are located inside the folder named *scenes* on the Github repository.

Training

Before running the training file *train.py*, the user needs to define all options related to the training by editing the file and changing the values of its global variables. Such global variables are described in the following tables. The *train.py* file can be ran by executing the *python train.py* command in the console.

Restore options		
Name of global variable	**Description**	**Type**
restore_model	Initialize the weights of the network from a saved model	boolean
restore_em	Initialize the experience memory from a saved results file	boolean
path_to_model_to_restore	Path to the saved model from which the weights of the network will be restored	string
path_to_results_to_restore	Path to the results file from which the experience replay memory will be restored	string

Table A.1: Options related to restoring past models and results.

Save options		
Name of global variable	**Description**	**Type**
save_model_frequency	Frequency (in episodes) of saving models	integer
max_to_keep	Limit of models to keep saved in disk. Older models are replaced with new ones once this limit is exceeded	integer
path_to_model_to_save	Path to the model file which will be saved during training	string
path_to_results_to_save	Path to the results file which will be saved at the end of the training	string

Table A.2: Options related to saving models and results.

Stage options		
Name of global variable	**Description**	**Type**
num_agents	Number of agents used for training	integer
num_episodes	Number of episodes for the stage	integer
max_discrepancy	Discrepancy above which the episode is finished	integer
min_discrepancy	Discrepancy below which the episode is finished	integer
steps_limit	Number of steps in each episode	integer
desired_distance	Desired distance l in meters (not used in aggregation)	float
task	Task ID. 0: dispersion 1: square formation 2: aggregation 3: chain formation	integer

Table A.3: Options related to the multi-stage training.

Experience replay memory and Boltzmann exploration and exploitation options		
Name of global variable	**Description**	**Type**
em_capacity	Experience replay memory capacity	integer
alpha	Exponent alpha	float
beta	Initial value of the exponent beta	float
final_beta	Final value of the exponent beta	float
initial_b_temperature	Initial Boltzmann temperature	float
final_b_temperature	Final Boltzmann temperature	float

Table A.4: Options related to the experience replay memory and the Boltzmann exploration and exploitation.

Network options		
Name of global variable	**Description**	**Type**
training_frequency	Frequency (in time steps) at which the network is trained	integer
batch_size	Batch size used to train the network	integer
time_steps	Number of time steps used for the LSTM cell	integer
lstm_units	Number of units of which the LSTM cell is comprised	integer
num_neurons	Number of neurons of the multi-layer perceptron Example: [50, 50] means two layers containing 50 neurons each	list
copy_weights_frequency	Frequency (in time steps) at which the weights are copied from online network to target network	integer
discount_factor	Discount factor of the deep reinforcement learning algorithm	float
learning_rate	Learning rate of the optimization algorithm	float

Table A.5: Options related to the neural networks.

Evaluating

Evaluation options		
Name of global variable	**Description**	**Type**
path_to_model_to_restore	Path to the saved model from which the weights of the network will be restored	string
time_steps	Number of time steps used for the LSTM cell	integer
lstm_units	Number of units of which the LSTM cell is comprised	integer
num_neurons	Number of neurons of the multi-layer perceptron Example: [50, 50] means two layers containing 50 neurons each	list
num_agents	Number of agents used for evaluating	integer
num_episodes	Number of episodes for the evaluation	integer
max_discrepancy	Discrepancy above which the episode is finished	integer
min_discrepancy	Discrepancy below which the episode is finished	integer
steps_limit	Number of steps in each episode	integer
desired_distance	Desired distance l in meters (not used in aggregation)	float
task	Task ID. 0: dispersion 1: square formation 2: aggregation 3: chain formation	integer

Table A.6: Options related to the evaluation of the results.

Similarly, before running the *evaluation.py* file, it is necessary to edit the file by changing the global variables (Table A.6) according to how the user chooses to perform the evaluation. After this has been done, the evaluation file can be run by executing *python evaluation.py* in the console.