Linköping University | Department of Computer and Information Science
Master thesis, 30 ECTS | Datateknik
202017 | LIU-IDA/LITH-EX-A--2017/001--SE

Intelligent Formation Control using Deep Reinforcement Learning

Rasmus Johns

Supervisor : Olov Andersson
Examiner : Patrick Doherty

External supervisor : Fredrik Bissmarck

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

In this thesis, deep reinforcement learning is applied to the problem of formation control to enhance performance. The current state-of-the-art formation control algorithms are often not adaptive and require a high degree of expertise to tune. By introducing reinforcement learning in combination with a behavior-based formation control algorithm, simply tuning a reward function can change the entire dynamics of a group. In the experiments, a group of three agents moved to a goal which had its direct path blocked by obstacles. The degree of randomness in the environment varied: in some experiments, the obstacle positions and agent start positions were fixed between episodes, whereas in others they were completely random. The greatest improvements were seen in environments which did not change between episodes; in these experiments, agents could more than double their performance with regards to the reward. These results could be applicable to both simulated agents and physical agents operating in static areas, such as farms or warehouses. By adjusting the reward function, agents could improve the speed with which they approach a goal, obstacle avoidance, or a combination of the two. Two different and popular reinforcement algorithms were used in this work: Deep Double Q-Networks (DDQN) and Proximal Policy Optimization (PPO). Both algorithms showed similar success.

Acknowledgments

Thanks to all employees at FOI for always making me feel welcome and part of your organization. Special thanks to Fredrik Bissmarck, who was my supervisor at FOI; Fredrik provided many great ideas and entered into discussions with me on a daily basis. Also, thanks to Jonas Nordlöf for always being helpful and positive about my work. Finally, thanks to all other students at FOI for contributing to a fun environment which encouraged curiosity and pursuit of knowledge.

From Linköping University, I want to thank my examiner, Patrick Doherty, for believing in the project from the start and for his advice on how to do a master's thesis. Furthermore, I must acknowledge Olov Andersson, my university supervisor, for being an invaluable guide into the field of deep reinforcement learning – without him this project would not have been possible.

Contents

List of Figures

List of Tables

Abbreviations

RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
FC	Formation Control
MDP	Markov Decision Process
ANN	Artificial Neural Network
PPO	Proximal Policy Optimization
DDQN	Deep Double Q-Network
ARG	Agent Reached Goal
OC	Obstacle Collision
FD	Formation Displacement
PR	Path Ratio
OCF	Obstacle Collision Frequency
AFD	Average Formation Displacement

1 Introduction

With ever cheaper, smaller and more powerful processors, multi-agent systems are increasingly becoming a feasible solution to many tasks. These tasks include anything from using robot swarms to handle packages in warehouses, as done by Amazon [1], to coordinating drones into performing spectacular shows, as done by Intel at the Olympic Games 2018 [2].

Common to many of these applications is the importance of formation control; when multiple agents are involved in a task, they often are required to, or at least benefit from, traveling in formations. By using formations, a single agent can be tasked with navigating an entire group toward a destination. Furthermore, the risk of agents colliding decrease as they maintain a spatial relationship within the group. Formations can also enhance a group's capacity; for instance, formations can be used to increase signal power, push objects with greater force, or search areas faster.

Since the field of multi-agent systems is relatively young, many approaches remain untested. This thesis explores the possibility of improving formation control. By combining reinforcement learning and formation control in a multi-agent environment, existing formation control paradigms are challenged.

Reinforcement learning is method based on letting agents explore their environments by trial and error. Agents are not told which actions to take, but is instead encouraged or discouraged by a numerical reward signal when interacting with the environment.

1.1 Motivation

Today, many multi-agent systems rely on formation control algorithms. Although these algorithms allow agents to move as a group, they are not adaptive and often require an expert to configure the agents' control parameters when the agents face a new scenario – this is time consuming, difficult and inconvenient. By integrating formation control with reinforcement learning, it may be possible to add more intelligent behaviors while also eliminating the need of tuning these parameters.

[1]https://www.amazonrobotics.com

[2]https://www.intel.com/content/www/us/en/sports/olympic-games/drones.html

1.2 Aim

The goal of this thesis is to investigate and evaluate the possibility of combining formation control with deep reinforcement learning. Furthermore, the thesis aims to answer how deep reinforcement learning could assist current formation control paradigms; what are the benefits of combining the fields and how does the reward function determine which behaviors are taught? Finally, the thesis aims to study the impact of hidden layers in modern reinforcement learning using neural networks.

1.3 Research Questions

The purpose of this report is to answer the following questions:

1. How can formation control be integrated with deep reinforcement learning?
2. What are the benefits of using deep reinforcement learning in conjunction with formation control versus solely using formation control?
3. What reinforcements should be employed for such an algorithm?
4. How does the hidden layer of neural networks impact the training of agents learning formation control?

1.4 Scope

The thesis work was conducted in a simulated environment. Furthermore, only a line formation was used and the Assignment Problem was ignored; both these limitations are described in Section 3.4.

The reinforcement learning algorithms used were limited to one method based on value iteration, Double Deep Q-Networks, and one method based on policy iteration, Proximal Policy Optimization. Both these algorithms are described in Section 2.4.

2 Theory

This chapter presents the necessary background theory to understand the work of the thesis. This includes formation control, reinforcement learning, artificial neural networks, and deep learning.

2.1 Formation Control

The idea of formation control is to create a system capable of moving agents in a predefined spatial pattern.

In general, formation control is a system which allows agents to travel as a coherent group without collision between agents. In addition, given some formation control system, only a single agent in a group needs to navigate; other agents in the group can rely on keeping their position in the formation. Maintaining a formation can also be beneficial in groups of mobile agents trying to achieve maximum sensor coverage. Moreover, one can imagine a situation where groups of agents need to align themselves into the formation which maximizes their signal throughput.

Commonly, three different types of formation control systems dominate the literature: leader-following, behavioral, and virtual structures. Each one of these approaches come with a different set of strengths and weaknesses [2].

2.1.1 Leader-following

The leader-following solution is based on dividing the agents' roles into two: one leader and one following. Commonly, the leader has different movement programming than its followers; furthermore, the leader is the only one knowing where the group is heading. Unlike the leader, the following is a group of agents aiming to maintain a spatial relationship. Then, as the leader moves to the goal, the following naturally moves with the leader. [2]

The leader-following system has been proven to be able to ensure that a group of decentralized agents stay connected and do not lose track of one and another [12]. Ensuring that agents stay connected is known as *the agreement problem* or *the consensus problem* – a problem that is at the heart of other problems, such as swarming, flocking, or rendezvous.

A common decentralized control law to control a follower is

$$\hat{x}_i = -k_i \sum_{j \in nbhd_i} w(||x_i - x_j||)(x_i - x_j) \tag{2.1}$$

where x_i represents a movement vector for follower i, $nbhd_i$ is the neighboring set containing all agents near follower i, k_i is a constant and $w : \mathbb{R}^n \rightarrow \mathbb{R}_+$ is a symmetric weight function based on the displacement $||x_i - x_j||$. w is some times known as the tension function. [4, 12]

The agents relative formation positions y can then be incorporated into Equation 2.1, resulting in

$$\hat{x}_i = -k_i \sum_{j \in nbhd_i} w(||(x_i - y_i) - (x_j - y_j)||)((x_i - y_i) - (x_j - yj)) \tag{2.2}$$

The control laws in Equation 2.1 and 2.2 are reciprocal, meaning the attraction/repulsion between two connected neighbors is symmetric. [4]

The strength and weakness of leader-following is strongly connected: it is solely directed by one leader [2]. Relying one just the leader to navigate is powerful, seeing how the leader specifies the behavior of the entire group; yet, the group has no explicit feedback to the formation. For instance, if the leader moves too fast, the following agents may lose their connection and the system risks breaking.

2.1.2 Behavior-based

Behavior-based formation control assigns a set of desired behaviors to each agent. The behavior of an individual agent is be calculated as the weighted average of the control for each desired behavior. Typically, these behaviors include collision avoidance versus other agents in the group, obstacle avoidance, goal seeking and formation keeping.

Behavior-based formation control has successfully been implemented in many experiments. For instance, a behavior-based formation control has been used to steer DARPA's unmanned ground vehicles and maintain several different formations [1]. To steer the DARPA unmannered ground vehicles, the behaviors listed as examples above were used.

To implement a behavior-based formation control approach, a formation position technique must be used [1]. The formation position is used by each agent to determine where the formation demands them to be. One method of positioning the formation is to set the formation center to be the average x and y positions of all agents involved in the formation, known as *Unit-center-reference*. Another method is to promote one agent to be the group leader. This agent does not attempt to maintain the formation; instead, the other agents are responsible for keeping the formation. This approach is called *Leader-reference*. Finally, another method of centering the formation is to solely rely on data about the neighbors, meaning a *Neighbor-reference* [1].

The greatest advantage of the behavior-based approach is its ability to naturally merge multiple competing objectives. Furthermore, unlike leader-following, there is an explicit feedback as every agent reacts to the position of its neighbors. The behavior-based approach is also easily implemented in a decentralized system. Another feature of using a behavior-based approach is the fact that alien agents, such as a humans, easily can interact with the formation by using a leader-referenced formation position.

The primary weakness of behavior-based formation control is that the group dynamic cannot be explicitly defined; instead, the group's behavior emerges from each agent's rules. This also means that this approach is difficult to analyze mathematically, and the stability of a formation can seldom be guaranteed [2].

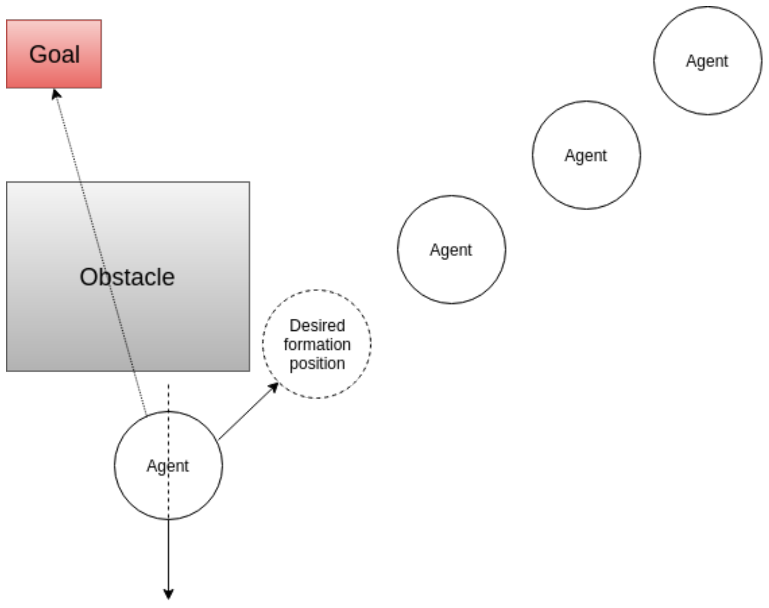


Figure 2.1: A visualization of an agent using a behavior-based approach. Vectors show different competing objectives.

2.1.3 Virtual Structures

The third approach is virtual structures. Using virtual structures, the entire formation is regarded as a single structure. The procedure for using a virtual structure approach, which is visualized in Figure 2.2, is as follows [27]:

1. Align the virtual structure with the current robot positions.
2. Move the virtual structure by distance Δx and orientation $\Delta\theta$.
3. Compute the individual robot trajectories to move each robot to its desired virtual structure point.
4. Configure the robots current velocity to follow the trajectory in Step 3.
5. Goto Step 1.

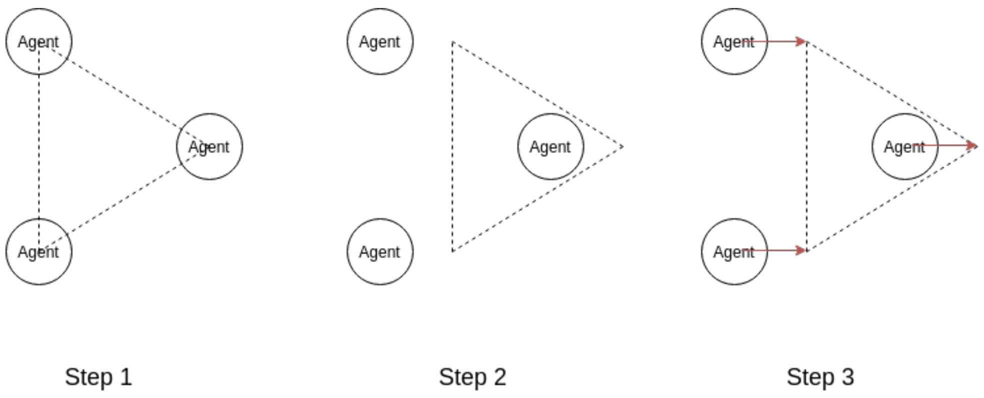


Figure 2.2: The first three steps of virtual structures.

The main benefit of using virtual structures is its simplicity. Moreover, feedback to the virtual structure is naturally defined. However, since the virtual formation requires the group to act as a single unit, the intelligence, flexibility, and therefore applications of the system are limited. [2]

2.1.4 Assignment Problem

When a group of multiple agents are assigned a formation, the question of which agent should fill which position in the formation needs to be asked. This problem is known as the *assignment problem*. Although there are decentralized solutions, such as the work by *Macdonald* [19], the assignment problem is often solved using centralized solutions. Commonly, the problem is solved using The Hungarian Algorithm [19].

2.2 Reinforcement Learning

Reinforcement learning is the process of finding a mapping between situations and actions using a numerical reward signal. Generally, an agent interacts with an environment without being told which actions to take or what their effect would be on the environment. Instead, the agent has to learn a behavior by trial and error. [26]

For the agent to gain a perception of what is to be considered good, a reward function r has to be implemented. This reward function is used to communicate the value of an agent's state transition, meaning the change in environment, to a numerical reward signal. The agent's reinforcement model should then learn to choose actions that tend to increase the long-term sum of rewards. As the agent acts to maximize the long-term sum of rewards, the agent has to plan a sequence of actions ahead of its current view of the environment. [14]

An agent's state s is the information given to the agent about the environment. The complexity of this state can range from using the arrays of a simple grid world, as done when designing the first Go program capable of beating the reigning human world champion [25], to using raw pixel data from images, as was done when a single model learned to play a wide variety of Atari 2600 games on a human level [20].

The process of reinforcement learning is depicted in Figure 2.3.

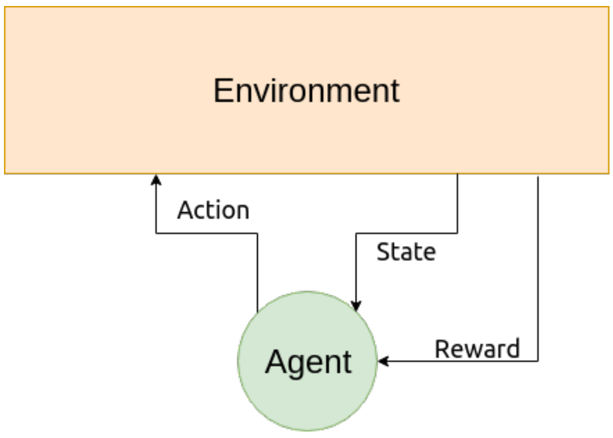


Figure 2.3: The basis of reinforcement learning.

2.2.1 Markov Decision Process

A Markov Decision Process (MDP) is a discrete time stochastic control process used to formalize decision problems such as reinforcement learning. At each discrete time step, the MDP is in a state s and able to choose any action a available in s. Upon selecting an action a, the MDP transitions by a probability $p^a_{s,s'}$ from state s to a new state s', yielding a numerical reward signal $r^a_{s,s'}$. An MDP has to satisfy the Markov Property, meaning it must be a stochastic process in which the conditional probability distribution of future states only depend upon the present state. Therefore, an MDP does not depend on the sequence of events that preceded it [26]. An example of a MDP can be seen in 2.4

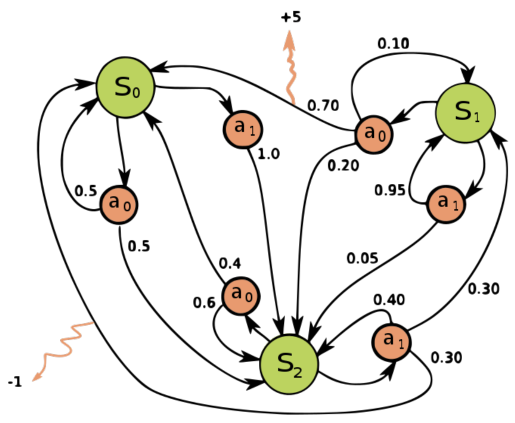


Figure 2.4: A simple MDP with three states (green circles), two actions (orange circles), and two positive rewards (orange arrows). [1]

2.2.2 Dynamic Programming

When faced with an MDP, the goal is to find the behavior yielding the highest future sum of reward. This behavior, which can be considered a mapping between which action to take in each state, is called a policy π.

Dynamic programming is a class of algorithms which, given a perfect model of the environment as an MDP, is able to compute the optimal policy $\pi*$. Therefore, dynamic programming provides an essential foundation to reinforcement learning.

Yet, although dynamic programming is able to compute $\pi*$, it does so at a high computational cost. In addition, unlike reinforcement learning, dynamic programming is only a solution to known MDPs. Reinforcement learning generalizes the methodology of dynamic programming to unkown MDPs, meaning the transition probabilities of an MDP $p^a_{s,s'}$ do not have to be known.

In many ways, reinforcement learning can be seen as a way to achieve results similar to dynamic programming, only with less computation and without the assumption of a known MDP. The essence of dynamic programming, and also of reinforcement learning, is to utilize a value function $V(s)$ to structure the search for a good policy. [26]

To find a good policy, either Policy Iteration or Value Iteration methods are typically used.

Policy iteration methods is a subclass of dynamic programming focusing on optimizing an agent's policy. Generally, this process is done in two steps.

First, the state-value function $V^\pi(s)$ for an arbitrary policy π is computed. $V^\pi(s)$ represents the future sum of rewards from the state s if the policy π is used. The process of computing $V^\pi(s)$ is called *policy evaluation*. The state-value function V^π can be computed as

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} p^a_{s,s'} \left(r^a_{s,s'} + \gamma V^\pi(s')\right) \tag{2.3}$$

[1]Image by Waldo Alvarez licensed under CC BY-SA 4.0

where $\pi(s,a)$ is the probability of taking the action a in a discrete state s, $p^a_{s,s'}$ is the probability of going from state s to state s' with action a, $r^a_{s,s'}$ is the reward, and γ is the discounting factor used to determine the importance of future rewards [26].

The second part of policy iteration is the *policy improvement* step. This step aims to find a new policy π' with a higher $V^{\pi'}(s)$ than the current state-value function $V^{\pi}(s)$. Improving the current policy can be done by selecting an action a in s and afterwards follow the existing policy π. Acting in such a way would give the value

$$Q^{\pi}(s,a) = \sum_{s'} p^a_{s,s'}\left(r^a_{s,s'} + \gamma V^{\pi}(s')\right) \tag{2.4}$$

which is the same as Equation 2.3, but without the first action selection.

By then comparing $Q^{\pi}(s,a)$ and $V^{\pi}(s)$, the new policy π' can be calculated. After all, if $Q^{\pi}(s,a) \geqslant V^{\pi}(s)$, the new behavior should be encouraged and vise versa.

Having improved policy π using V^{π} to find a superior policy π', it is then possible to repeat the same behavior with $V^{\pi'}$ to find an even better policy π'' and so on. This process is called *policy iteration*. [26]

Another fundamental method in dynamic programming is *value iteration*. This method operates similarly to policy evaluation, but replaces the action selection with a greedy selection of the best possible action, as seen in Equation 2.5.

$$V_{t+1}(s) = \max_a \sum_{s'} P^a_{s,s'}\left[r^a_{s,s'} + \gamma V_t(s')\right] \tag{2.5}$$

Like policy iteration, value iteration is able to converge the optimal value function given its existence [26].

2.2.3 Q-learning

Q-learning is an important algorithm in reinforcement learning. It is Temporal Difference (TD) algorithm, meaning values are updated throughout an epoch, instead of Monte Carlo learning, which updates values only at the end of an epoch. [26, 30]

Q-learning is a good example of how reinforcement learning is based on dynamic programming. At its core, Q-learning is just value iteration. The difference between value iteration and Q-learning is that Q-learning does value iteration on a Q-function, as seen in Equation 2.6, instead of the value function V.

Basically, the Q-learning operates by trial and error; it explores different actions in different states and record the success. The algorithm uses an exploration rate ϵ, which typically declines during training, to decide between doing the best possible action or a random action. Acting in this manner allows Q-learning to find better options.

Just like value iteration, the agent improves iteratively by using an update rule

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha\left[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right] \tag{2.6}$$

where s is a state, a is an action, α is the rate of learning, r is the received reward, and γ is the discount factor used to prioritize short-sightedness against long-sightedness [20, 26].

In other words, Q-learning acts to maximize

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a')|s,a\right] \tag{2.7}$$

Traditionally, Q-learning stores its mapping Q in a large lookup table, commonly known as the Q-table. While being a simple implementation, the Q-table has several disadvantages. First, the table must be initialized before the training commences to cover all possible state-action combinations. This causes the tables to scale poorly in continuous environments and often lead to memory issues. In addition, since an agent only looks at one specific cell in the

Q-table, using a lookup table can be detrimental to an agent if it enters a state which it never encountered during training. [29]

As Q-learning assumes to know the action in the next state is the greedy action a' (meaning it also assumes there is no exploration), Q-learning is said to be a off-policy algorithm. In spite of this assumption, Q-learning can still be proven to converge to the optimum action-values mapping for any finite MDP. [26, 31]

2.3 Artificial Neural Network

Finding the perfect feature representation is an impossible task. Yet, tools such as Artificial Neural Networks (ANNs) does a good job of utilizing a set of simple features and to find more complex features [16]. ANNs are feed-forward networks consisting of connected neurons, and are inspired by the structure of a biological brain. The networks have an input layer used to feed the network with data, a number of hidden layers used to find more complex patterns in the feature space, and a final layer which outputs the networks prediction. Using ANNs with several hidden layers is called *Deep Learning* [9]. An example of an ANN can be seen in Figure 2.5.

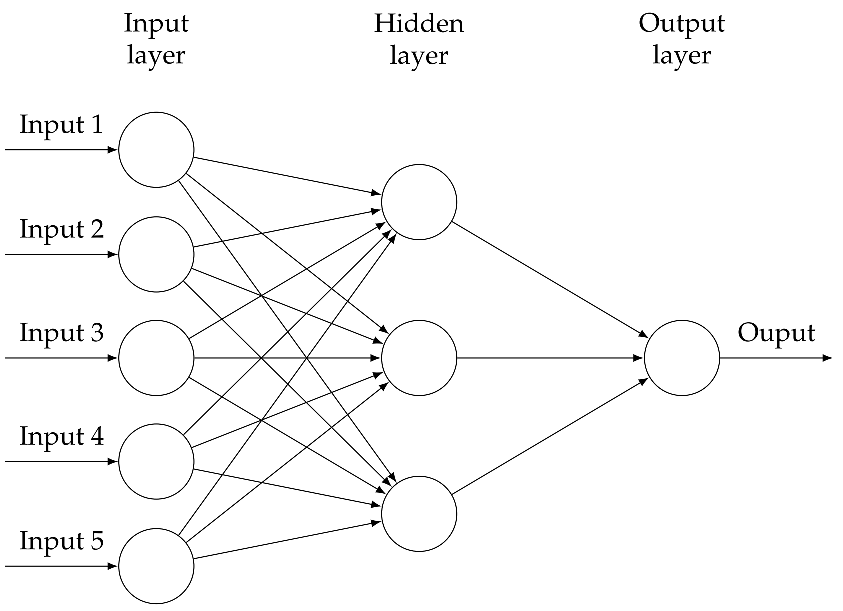


Figure 2.5: An artificial neural network with an input layer consisting of five nodes, one hidden layer consisting of three nodes and a final layer consisting of a single node.

The neurons in the hidden layers are connected to the previous layer by connections with weights. These weights w_j act as multipliers to the previous signal x, effectively allowing the network to reduce or increase the importance of different combination of neurons – thus yielding complex features. The output of a neuron y_j is calculated as

$$y_j = f(b + \sum_{i=1}^{n} w_{ij}x_i) \tag{2.8}$$

where f is an activation function, b is a potential bias which can be added and n is the number of connections to the previous layer. The purpose of the activation function f is

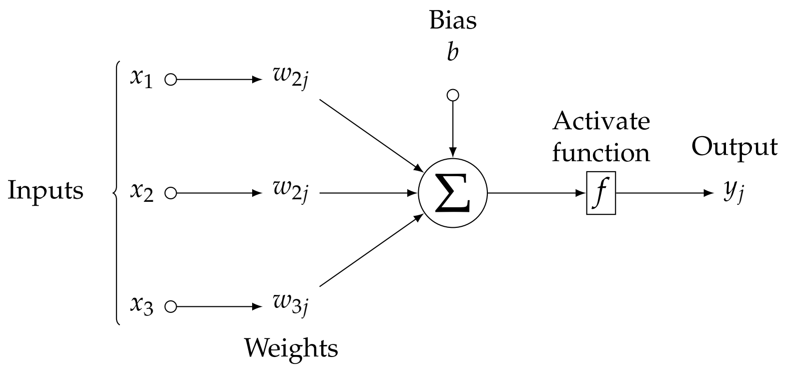


Figure 2.6: A neuron y_j which is connected to three other nodes in x.

primarily two-fold: they enable the ANN to do non-linear functional mappings and they can be used to limit the magnitude of y_j. A neuron's calculations is visualized in Figure 2.6. Passing input through the network to produce an output is called *forward-propagation*. [16]

When training an ANN, the set of weights between neurons is updated to reduce the network's performance. This process starts by letting a data sample forward propagate through the network. Once the ANN calculated its output for a data sample, it is able to calculate the difference between the network's output and the desired output. This difference is known as the *loss*. In order to minimize the loss, the network updates its weights by calculating the loss' gradient using the *back-propagation* algorithm – an computationally efficient algorithm which backpropagates errors through the network and calculates the weight gradient. [16]

As ANNs can learn to approximate an arbitrary function $y = f(x)$ given a large set of examples, ANNs can be used instead of tables in reinforcement learning. For instance, an ANN can replace the Q-table in Section 2.2.3.

2.4 Deep Reinforcement Learning

Deep reinforcement learning is the result of combining reinforcement learning paradigms with deep learning.

In this thesis, two deep reinforcement algorithms, both representing different ways of solving the problem, were used. These two algorithms were Deep Q-learning (based on value iteration) and Proximal Policy Optimization (based on policy iteration).

2.4.1 Deep Q-learning

Before Q-learning with function approximations was popular, it was clear that tabular Q-learning scaled poorly and function approximations were required to replace the Q-table in complex problems [29]. Yet, despite some progress using function approximations with Q-learning, Google DeepMind's Deep Q-learning algorithm was groundbreaking. The Deep Q-learning algorithm used a neural network as a function approximator to beat several Atari games in 2013 [20].

Deep Q-learning deviates from traditional Q-learning in two major ways.

First, the Q-table is replaced by a neural network acting as the Q-function approximator $\hat{Q}$. The neural network takes a state, such as four preprocessed images of an Atari game in DeepMind's case, and produces an output corresponding to the approximated $\hat{Q}$-values otherwise found in the Q-table. The neural network can be trained by minimizing a sequence of loss function $L_i(\theta_i)$ according to

$$L_i(\theta_i) = \mathbb{E}_{s,a}\left[(y_i - \hat{Q}(s,a;\theta_i))^2\right] \tag{2.9}$$

where, looking at Equation 2.7, the target for iteration i is similarly

$$y_i = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_{i-1})|s, a\right] \tag{2.10}$$

In Equation 2.10, we see that the weights from the previous iteration θ_{i-1} are frozen when optimizing $L_i(\theta_i)$. Unlike supervised methods, the target y_i in Deep Q-learning depend on the network weights $\hat{\theta}$ through the use of the dynamic programming update in Equation 2.10 [20].

Calculating a gradient for $L_i(\theta_i)$ yields

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a;s'}\left[\left(r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_{i-1}) - \hat{Q}(s, a; \theta_i)\right)\nabla_{\theta_i}\hat{Q}(s, a; \theta_i)\right] \tag{2.11}$$

The second improvement introduced into deep Q-learning is a replay buffer. The replay buffer saves each experience $e_t = (s_t, a_t, r_t, s_{t+1})$ which previously were used to perform TD-updates. The buffer is instantiated with a capacity N and overwrites old experience when filled. From the experience buffer, samples are drawn at random to update the neural network according to Equation 2.11. Using a replay buffer does not only allow the same experience to be used several times, increasing the data efficiency, but it also reduces the correlation between samples. Learning from consecutive samples, as done by traditional Q-learning, causes a strong correlation between data samples. As experiences e are drawn from the replay buffer at random, the variance of the updates are reduced. Finally, the replay buffer can be used to escape local minimums; for instance, if the best action is to move left, an agent will only move left, causing training samples to be dominated by samples from the left-hand side. By using a replay buffer in such a scenario, the training distribution can become larger and local minimums some times avoided. In short, the replay buffer is used to speed up the credit/blame propagation and allow the agent to refresh what it has learned before. [17, 20]

Finally, a popular adjustment to the deep Q-learning is to use *Double Q-learning* [10]. The purpose of Double Q-learning is to reduce the impact of selecting overestimated values, which can result in overoptimistic value estimates. This overestimation is the result of using one function approximator both to select and evaluate an action. Separating the selection and evaluation in Q-learning's target expression, meaning Equation 2.10, produces

$$y_i = \mathbb{E}_{s'}\left[r + \gamma \hat{Q}(s', \underset{a}{\operatorname{argmax}}\, \hat{Q}(s', a; \theta_{i-1}|s, a); \theta_{i-1}|s, a)\right] \tag{2.12}$$

Equation 2.12 highlights how the same weights θ_{i-1} are used to update both selection and evaluation. To correct this issue, it is possible to apply Double Q-learning. Double Q-learning introduces two value function, meaning two set of weights, θ and $\dot{\theta}$. One value function is used for evaluation and the other for selection. The new target then becomes

$$y_i = \mathbb{E}_{s'}\left[r + \gamma \hat{Q}(s', \underset{a}{\operatorname{argmax}}\, \hat{Q}(s', a; \theta_{i-1}|s, a); \dot{\theta}_{i-1}|s, a)\right] \tag{2.13}$$

When performing updates in Double Q-learning, either θ or $\dot{\theta}$ is chosen at random to be updated. Furthermore, each iteration θ and $\dot{\theta}$ are randomly assigned selection or evaluation. [10]

2.4.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a policy gradient method based on policy iteration. Policy gradient methods attempt to estimate a policy gradient onto which gradient ascent can then be applied. The policy improvement of PPO is done through Conservative Policy Iteration, expressed as

$$y = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)}\hat{A}_t\right] \tag{2.14}$$

where π_θ is a policy using the weights θ, π_old is the previous policy, and $\hat{A}_t$ is an advantage function at timestep t [24].

In PPO, the advantage function used is Generalized Estimated Advantage (GAE) [23]. GAE is computed as

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{t}^{\infty} (\gamma\lambda)^l \delta_{t+l}^{\hat{V}} \tag{2.15}$$

where l is a number of timesteps and $\delta_t^{\hat{V}}$ is calculated as

$$\delta_t^{\hat{V}} = -\hat{V}(s_t) + r_t + \gamma \hat{V}(s_{t+1}) \tag{2.16}$$

In Equation 2.15, λ can be set to any value in the interval $\lambda \in [0, 1]$. Yet, for the edge cases of this interval, GAE is defined as

$$GAE_t(\gamma, 0) = r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \tag{2.17}$$

$$GAE_t(\gamma, 1) = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - \hat{V}(s_t) \tag{2.18}$$

GAE(γ,1) is always γ-just, meaning it does not introduce bias, at the cost of having a high variance due the sum of terms. GAE(γ,0), on the other hand, introduces a bias in all cases but $\hat{V} = V^{\pi,\gamma}$. By varying the value of λ, it is then possible to compromise between bias and variance [23]. In the case of reinforcement learning and PPO, high variance refers to noisy, but on average accurate value estimates $\hat{V}$; in contrast, high bias refers to a stable, but inaccurate $\hat{V}$.

The hope of a high variance learning, such as Monte-Carlo learning or GAE(γ,1), is that a large number of action trajectories will average out the high variance from any one trajectory and provide an estimate of the "true" reward structure. Yet, as this is time consuming, it is in some cases desirable to lower the variance at the cost of a higher bias. In other words, one could say that the GAE allows for an interpolation between pure TD learning and pure Monte-Carlo sampling using a λ parameter [23].

To ensure that the policy updates of PPO are not too large, PPO optimizes the loss of a clipped surrogate objective. The optimized clipped objective function L^{CLIP} is

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t, \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)}, 1-\epsilon, 1+\epsilon \right) \hat{A}_t \right) \right] \tag{2.19}$$

where *min* is a function returning the smallest of two values and *clip* is a function clipping a value between two range values dictated by the hyperparameter ϵ [24].

The reasoning behind using a clipped surrogate objective is as follows. The first term inside the *min* function is the Conservative Policy Iteration from 2.14. Yet, if this was the sole objective, without any clipping or penalizing, maximization of the objective would lead to excessively large policy updates. Therefore, the *clip* function is introduced in conjunction with the *min* function. Together, they keep the policy updates small, which increases the update stability [24].

Yet, for PPO to function, the estimated value function – used by GAE – needs to be optimized as well. This objective can be expressed as

$$L_t^{VF}(\theta) = \left(\hat{V}_\theta(s_t) - V^{\text{target}_t} \right)^2 \tag{2.20}$$

As these two objectives are optimized, an entropy bonus S is added as well to introduce exploration. This yields the final objective function

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{2.21}$$

The PPO algorithm then uses Equation 2.21 with fixed-length trajectory segments. Before each update iteration, the agent collect T timesteps of data. Having collected the experience, the surrogate loss is calculated and optimized using stochastic gradient descent or another any other optimization method using K epochs, meaning PPO performs K updates of π_θ using the same $\hat{A}_t$ [24].

The final PPO algorithm can be read in Algorithm 1.

Algorithm 1 PPO

```
1: for iteration=1,2... do
2:     Run policy $\pi_{\theta_{old}}$ for $T$ timesteps
3:     Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$
4:     Optimize surrogate objective $L^{CLIP+VF+S}$ wrt $\theta$, with $K$ epochs and minibatch
       size $M$
5: end for
```

2.4.3 Multiagent Systems

A multi-agent system consists of multiple individual agents whose task is to interact to achieve a specific goal. The form of these agents could be anything from physical robots to virtual components in a software system.

Generally, applying machine learning to multi-agent systems presents a number of difficulties.

First, the state space and complexity has a tendency to grow larger as the number of agents increases. Unless a clever solution is implemented, which is able define a informative state space without including information about every other agent, the state space grows proportionally to the number of agents [32].

Second, the matter of the *credit assignment* problem, which refers to the problem of determining which specific action was responsible for the environments feedback, is very prominent in multi-agent systems. Traditionally, this problem refers to a single agent executing a long sequence of actions and then struggling to learn which action in the sequence was the important one. However, multi-agent systems also introduces the factor of other agents. For instance, an agent which just collided with another agent and received poor feedback might not be responsible for the collision at all [32].

Finally, there is a challenge of balancing the rewards of a multi-agent system. In order to construct a system that promotes cooperation, a full system reward structure is often used. A full system reward gives all agents in a system equal rewards when doing something desirable. Although effective, this reward structure often boost noise in large multi-agent systems, seeing how the credit assignment problem grows more difficult. Another approach is to use a local reward, meaning agents are rewarded individually. Typically, this reward is easier for multi-agent systems to learn; however, it is easier to learn at the expense of *factoredness* – referring to the problem of each agent optimizing their own behavior according to a local reward might not promote coordinated system behavior [32].

Despite these difficulties, several multi-agent systems have been trained successfully. Already in 1993, the potential, and problems, of applying reinforcement learning to a multi-agent system was studied [28]. In this experiment, two agents traversed a grid world looking for food. It was found that sharing sensations between agents can be beneficial at the cost of slower learning rates, seeing how the state space grew.

Since then, other researchers have continued exploring this area. Recently, an algorithm capable of micromanaging agents in the real-time strategy game Starcraft was developed [8]. This environment had a large state space and used a decentralized reinforcement learning model. Similarly, decentralized actors have been taught, using a centralized value iteration approach, to teach a group of agents to play something similar to tag [18]. Furthermore, the non-profit research company OpenAI are currently attempting to solve the complex game of DOTA 2, a game of two teams consisting of five players, using multi-agent reinforcement learning [22].

Some work have been done on investigating how to use reinforcement learning models to teach multiple agents solving tasks. One work compared several deep reinforcement learning paradigms from a multi-agent perspective [13]. Primarily, three approaches were compared: *centralized learning*, *concurrent learning* and *parameter-sharing*. The first approach, using one joint model for the action and observations of all agents, was not very successful. The second approach, using concurrent learning where each agent learns its own individual policy by training a personal model, did better. However, the most successful approach was to use parameter-sharing, meaning each agent trained a single model. [13]

Yet, the research of combining formation control with reinforcement learning has been scarce. One interesting work used reinforcement learning to teach agents how to get into formations using no control theory; instead, the agents taught themselves how to arrange into specified formations [6].

3 Method

To tackle the complexity of formation control, this thesis suggest the use of deep reinforcement learning. Since traditional formation control already provides a robust and effective system, the aim of this thesis was not be to replace it. Instead, deep reinforcement learning was to be used to enhance already functional systems. As a baseline, a simple traditional formation control system was implemented as well.

This chapter starts with an overview of used frameworks and hardware. Then, the setting of the experiments is detailed. The chapter then explains how the deep reinforcement learning was implemented and the reasoning behind important design decisions. Finally, the specific experiments and evaluation metrics used to test the algorithms are outlined.

3.1 Framework, Platform and Hardware

The deep reinforcement algorithm was implemented using OpenAI's Baselines package [7]. Baselines is a set of state-of-the-art Python implementations of reinforcement learning algorithms running on top of Tensorflow[1]. Baselines was chosen because of its wide variety of accessible reinforcement algorithms; furthermore, since Baselines builds on Tensorflow, the computations can run either CPU or GPU. For this thesis, the reinforcement learning algorithms were initially trained on the GPU using CUDA[2] – a parallel computing platform developed by NVIDIA to be used in conjunction with their GPUs. The used GPU utilized was an NVIDIA GeForce GTX 1080 Ti with 11 GB memory. However, for the problem structure in this thesis, the Baselines algorithm were executed faster using only the CPU, which was a Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz – a CPU with 6 cores.

In order to apply Baselines to a reinforcement learning problem, the problem must be formulated according to OpenAI's Gym-interface [3]. Therefore, the world used in this thesis was built on top of the work by Mordath et al [21], which implements the Gym-interface in a multi-agent environment.

[1]Tensorflow, `https://www.tensorflow.org/`

[2]CUDA, `https://developer.nvidia.com/cuda-zone`

3.2 Environment

The environment consisted of a group of agents existing in a world. Each agent in the world was a point mass able to apply a force in any direction in the 2D world. Next, there existed a goal which marked groups desired destination. Every agent was aware of the relative distance to the goal, but not the obstacles potentially blocking the direct path. A total of 11% of the environment's space was covered in circular obstacles.

While moving towards a goal, the agents were also instructed to hold specific formations. An image of the world can be seen in Figure 3.1.

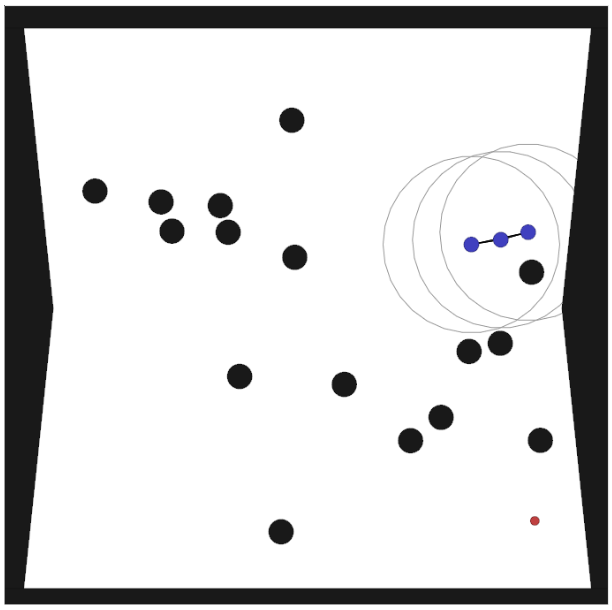


Figure 3.1: The thesis' simulation environment. The blue circles are agents with their respective sensor ranges visible in light gray. Between the agents, lines are drawn to illustrate agents' potential formation displacement. The red circle is the goal and black circles and boarders are obstacles.

An agent moved by applying a force $F_{i,control}$ to itself. The full equation to calculate an agent's velocity was

$$v_i(t+\delta) = \beta * v_i(t) + \frac{F_{i,control} + F_{i,collision}}{m_i} \tag{3.1}$$

where β was a dampening factor, $F_{i,control}$ was a custom force to be applied by the chosen control algorithm, $F_{i,collision}$ was a possible force generated by any elastic collision, and m_i was the agent's mass.

In Equation 3.1, $F_{i,control}$ was calculated as a combination of the sum of all behavior-based control vectors $F_{i,bb}$ (as listed in Section 3.3) and the reinforcement learning's action force $F_{i,rl}$. Section 3.5 describes how these forces were combined.

3.3 Behavior-based Formation Control

The behavior-based control algorithm in this thesis was based on previous work by Balch and Arkin [1]. All behaviors used in Balch and Arkin's work were implemented according to their formulas (although the hyperparameters had to be adjusted to suit the dimensions in this thesis' environment). The behaviors were: move-to-goal, maintain-formation-position, avoid-obstacle, and avoid-neighbor-collision. This algorithm served both as the baseline which the reinforcement learning algorithms was compared with, as well as the base from which the reinforcement learning algorithms started their training.

3.4 Formations

For this thesis, the assignment problem was ignored. Instead, agents had fixed positions in every given formation, meaning that a group in an inverted symmetric formation would have to flip its positions even if the group changed into a seemingly identical formation. Ignoring this problem simplified the algorithm at the cost of reducing the group's accumulated efficiency. Introducing the assignment problem conjointly with deep reinforcement learning could result in more interesting results; yet, for the scope of this thesis, the assignment problem was deemed to complex.

To test the algorithms, a line formation with fixed distances was used. A line formation formation, as seen in Figure 3.2, demands high intelligence. After all, a line formation maximizes the risk of at least one agent running into an obstacle. The line, constructed by the line formation, was always desired to be orthogonal to the angle between the formation center – which in this case was unit center referenced – and the goal.

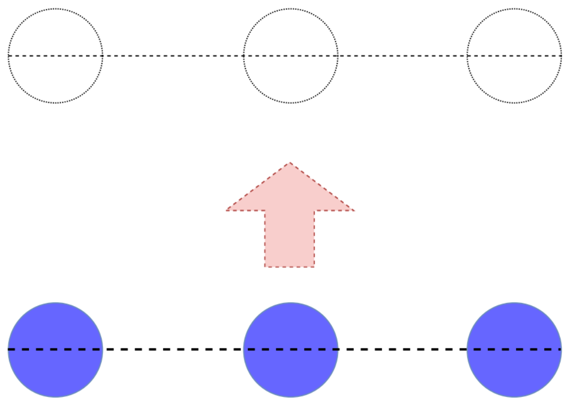


Figure 3.2: A line formation in which three agents (blue circles) are lined up and moving along the red arrow to their new positions.

3.5 Combining Formation Control with Reinforcement Learning

As previously stated, the goal of the thesis was to use a reinforcement learning algorithm to optimize a previously existing formation control approach. Of the three common approaches – leader-following, behavior-based, and virtual structures – behavior-based seemed to have the greatest potential to merge with reinforcement learning.

By using a behavior-based control algorithm with reinforcement learning added as behavior factor, meaning it outputted a $[x, y]$ vector $F_{i,rl}$, it was possible to train the reinforcement learning algorithm robustly. The vector of $F_{i,control}$ in Equation 3.1 was then calculated as combination of the behavior-based control algorithm's force $F_{i,bb}$ and the reinforcement learning model's action $F_{i,rl}$ according to Algorithm 2. In this algorithm, $clip$ represents a clipping function similar to the one used in PPO.

This approach allowed the agents to start with the behaviors of the traditional formation control algorithm, while still having the possibility to explore actions and improve the initial behavior with reinforcement learning.

Algorithm 2 Behavior-based reinforcement learning movement

```
1: procedure MOVE($F_{bb}, F_{rl}$)
2:     if $||F_{bb}|| > 1$ then
3:         $F_{bb} \leftarrow clip(F_{bb}, -1, 1)$
4:     end if
5:     if $||F_{rl}|| > 1$ then
6:         $F_{rl} \leftarrow clip(F_{rl}, -1, 1)$
7:     end if
8:     $F_{control} \leftarrow F_{bb} + F_{rl}$
9:     if $||F_{control}|| > 1$ then
10:        $F_{control} \leftarrow clip(F_{control}, -1, 1)$
11:    end if
12:    return $F_{control}$
13: end procedure
```

Another approach would have been to use reinforcement learning to optimize the weights of the existing behavior-based objectives; yet, such an approach would have limited the agents' potential to learn any new behaviors.

3.6 Deep Reinforcement Learning

Since managing decentralized agents through a map with obstacles in formations is a complex problem with known solid, yet limited, solutions, the ambition of DRL in this thesis was to optimize, rather than replace, these.

Two different models from OpenAI's Baseline were tested: the Deep Double Q-Network (DDQN) and the Proximal Policy Optimization (PPO2) algorithm [7].

3.6.1 Deep Double Q-Network

DDQN was applied to the environment using the default hyperparameters:

$$learning\_rate = 5 * 10^{-4}$$
$$\gamma = 0.99$$

The learning rate, meaning the size of updates of the Q-network, was set rather small. This prevented unstable updates which can otherwise be an issue, especially in multi-agent learning. γ was set high to promote long-sightedness.

The algorithm utilized a replay buffer of size $N = 50000$ and retrieved 32 experiences from the buffer each time the objective loss in Equation 2.11 was calculated.

The Q-network had one input unit for each state value. These were then fully connected to a hidden layer consisting of 64 units with an hyperbolic tangent function *tanh*. This layer was then fully connected to the final layer which had a size of 5, corresponding to the discrete actions: do-nothing, go-left, go-up, go-right, and go-down. These actions were then used as $F_{i,rl}$ in Algorithm 2.

3.6.2 Proximal Policy Optimization

PPO was applied to the environment using the following parameters:

As with the DDQN, the learning rate was kept small and γ was set high. Furthermore, c_1 and c_2 were set to values forcing $L_t^{\text{CLIP+VF+S}}$ to mostly depend on L_t^{CLIP}. Furthermore, as the environment was rather complex, meaning a large l and small K was needed in order not to

$$
\begin{aligned}
&l = 2048\\
&K = 2\\
&\lambda = 1\\
&\epsilon = 0.15\\
&c_1 = 0.5\\
&c_2 = 0.1\\
&learning\_rate = 5 * 10^{-4}\\
&\gamma = 0.99
\end{aligned}
$$

overfit the policy. Furthermore, as the cost of bias in a multi-agent environment was deemed too large, λ was set to 1 in order to reduce the bias at the cost of higher variance.

The policy network and value network of PPO both had an input layer with an input unit for each state value. These were then fully connected to a hidden layer of 64 units using *tanh* as its activation function. This hidden layer was connected to a similar layer. The policy and value networks only had different output layers; the output layer of the policy network consisted of two units yielding continuous values for $F_{i,rl}$, whereas the value network only had one output unit to predict $\hat{V}(s)$.

3.6.3 Parameter Sharing

Since maintaining a formation is a skill where each agents share the same kind of goal and has the same actions, one parameter-shared policy and value function was used and trained by all agents (see Figure 3.3).

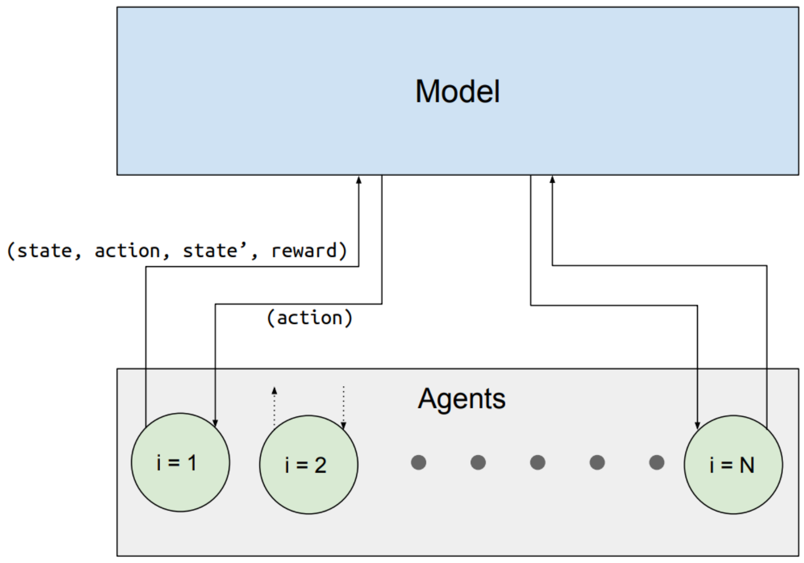


Figure 3.3: The parameter shared DRL model used by all agents. The model is updated by each individual agent and used when an agents needs to take an action.

3.6.4 State Representation

To honor the decentralized aspect of formation control, the state could not contain data about the surroundings other than what an individual agent could possibly perceive by itself.

Therefore, the agents' state was to be limited to data known to the specific agent, such as sensor data.

The state space S was shaped to capitalize on the objective vectors of behavior-based formation control, while also adding sensor data. Each behavior vector, of the predefined formation control behaviors, was added as a state feature. Additionally, each agent was equipped with 8 uniformly distributed sensors, amounting to a sense of 360 degree vision. Each degree block in the array contained two values: one was the sensor power to a potential neighbor, whereas the other was the sensor power to a potential obstacle. The sensor power was the power of a measured distance by a sensor. If nothing was found within a sensors range, the sensor power was set to 0. The relationship between measured sensor distance and sensor power can be seen in Figure 3.4. Furthermore, the part of the state space consisting of sensor data is visualized in Figure 3.5.

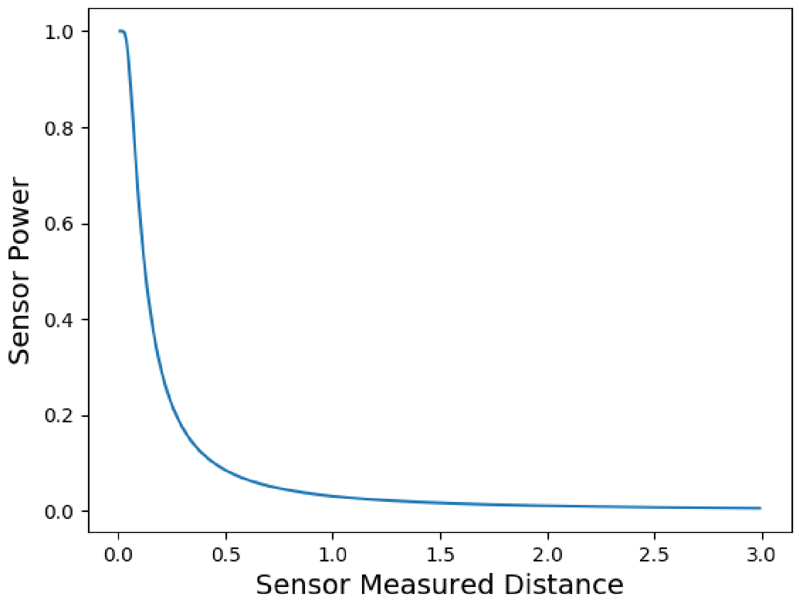
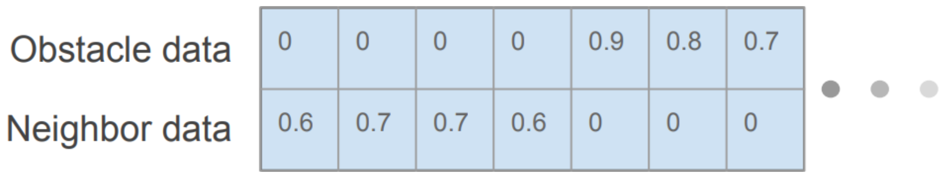


Figure 3.4: The relationship between the distance measured by a sensor and the sensor's power in an agents state space.

The measured distance of a sensor was converted to sensor power in order to simplify the state space representation. By using sensor power, any measured distance within the sensor range $d = 1$ could be recalculated into a sensor power using the relationship in Figure 3.4. In addition, when a sensor did not detect anything in range, the sensor power could be set to 0.



Obstacle data	0	0	0	0	0.9	0.8	0.7
Neighbor data	0.6	0.7	0.7	0.6	0	0	0

Figure 3.5: An example of how the agent's sensor state can look at a given timestep.

Finally, each agent included its unique id to its state. As the agents used parameter-sharing and trained the same model, adding agents' id as a feature allowed for personal behaviors.

3.6.5 Reinforcements

As the group of agents had to accomplish several objectives, the reward function was designed to enforce each objective. These objectives were: move to goal, avoid obstacle, and remain in formation. The reward function was split into one factor for each objective; these reward factors were then weighted by a coefficient k to increase or decrease the importance of different objectives.

To motivate the agents to reach the goal, two different rewards were used. First, a reward for Agent Reached Goal R_{ARG} was given to an agent reaching the goal.

$$R_{ARG} = \begin{cases} k_{ARG}, & \text{if agent reached goal} \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

In addition, a full reward, Group Reached Goal R_{GRG}, was awarded to the entire group when one agent from the group reached goal.

$$R_{GRG} = \begin{cases} k_{GRG}, & \text{if any group member reached the goal} \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

Next, in order to teach agents to avoid obstacles, a negative reward for Obstacle Collision R_{OC} was added.

$$R_{OC} = \begin{cases} k_{OC}, & \text{if agent colldided with an obstacle} \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

To encourage a group of agents with no pre-defined behavior to utilize formations, they either must be explicitly told to get a reward from keeping a formation, or they must benefit indirectly from maintaining formations. The latter could be the case if they for instance were rewarded for moving large objects, but indirectly had to charge the objects in formation to muster the force to move it. For this thesis, there was no clear way of introducing an incentive to maintain formation other than directly penalizing agents for deviating from their given formation. This resulted in a negative reward for Formation Displacement R_{FD}. This reward, which was the negative average sum of all agent's formation displacement over an episode, was given to the agents at the end of each episode according to Equation 3.5. Penalizing agents in this manner forced them to factor in their formation position while moving towards the goal.

$$R_{FD} = \frac{-k_{FD}}{N} \sum_{i=1}^{N} ||x_{i,t,desired} - x_{i,t}|| \tag{3.5}$$

Where N was the number of agents and x was their position relative to the formation center.

The reward function was then expressed as

$$R = R_{ARG} + R_{GRG} + R_{OC} + R_{FD} \tag{3.6}$$

3.7 Reward Function Experiments

The experiments were set up to challenge the learning algorithms and methodically test which characteristics different reward functions resulted in.

Id	Environment		Reward Function			
	Fixed obstacle pos.	Fixed start/end pos.	K_{ARG}	K_{GRG}	K_{OC}	K_{GFD}
1	Yes	Yes	100	0	0	0
2	Yes	Yes	0	100	0	0
3	Yes	Yes	0	100	0.5	0
4	Yes	Yes	0	100	1.5	0
5	Yes	Yes	0	100	0	0.25
6	Yes	Yes	0	100	0	1
7	Yes	Yes	0	100	0.5	0.5
8	Yes	No	0	100	0.5	0.5
9	No	No	0	100	0.5	0.5

Table 3.1: Reward Function Experiments.

As the agents' ability to generalize their knowledge to other maps were uncertain, obstacle positions was set either as fixed or changing between episodes. In addition, the agent and goal start/end position was set to either fixed or changing between episodes.

Next, the reward function was designed to encourage different behaviors. For this purpose, the reward function coefficients, seen in Equation 3.2, 3.3, 3.4, and 3.5, were altered.

The complete list of experiments mapped out in this report can be seen in Table 3.1.

3.8 Experiment about Use of Hidden Layers

To study the usage of hidden layers in the deep reinforcement models when trained in the described environment, DDQN was retrained once with no hidden layers. The experiments used to test the usage of the hidden layers can be seen in Table 3.2. Notice these correspond to Experiments 7 and 8 in Table 3.1.

Id	Environment		Reward Function			
	Fixed obstacle pos.	Fixed start/end pos.	K_{ARG}	K_{GRG}	K_{OC}	K_{GFD}
10	Yes	Yes	0	100	0.5	0.5
11	Yes	No	0	100	0.5	0.5

Table 3.2: The tested experiments trained by a DDQN model with no hidden layers.

3.9 Evaluation Metrics

In order to compare DDQN and PPO against regular formation control, evaluation metrics had to be established. These metrics aimed to answer how successful the agents were at reaching the goal, how fast they reached the goal, how many collisions they had along the way, and how well they remained in formation. To answer these questions, the evaluation metrics from Balch and Arkin [1] were employed.

3.9.1 Path Ratio

The Path Ratio (PR) was the average distance traveled by the agents divided by the Euclidean distance between the start and end position. This metric was used to evaluate how efficient the agents were at approaching the goal.

3.9.2 Obstacle Collision Frequency

The Obstacle Collision Frequency (OCF) explained the proportion of time the agents were in contact with an obstacle. A value of zero would imply no agent ever touched an obstacle during the episode, whereas a value of one would indicate that every agent was constantly touching an obstacle during the episode.

3.9.3 Average Formation displacement

By studying the agents' average Formation Displacement (FD) from the given formation, a sense of how well the algorithm were at keeping the formation could be established. This displacement was calculated as

$$E_{FD} = \frac{1}{TN} \sum_{t=0}^{T} \sum_{n=0}^{N} |x_{n,desired}(t) - x_n(t)| \tag{3.7}$$

where T was the number of timesteps and N was the number of agents.

3.9.4 Success

The most important measurement was the percentage of successful attempts to reach the goal. Even if the goal position was not reached in a perfect formation or in the fastest time, the fact that the goal position was reached was considered crucial.

3.9.5 Iterations to Goal

To study how well the agent made use of its ability to move fast, the number of iterations needed to successfully reach the goal was monitored.

4 Results

This section presents the results from the behavior-based formation control algorithm and the improved version using reinforcement learning.

The results are presented in two formats. First, the metrics described in Section 3.9 are used to evaluate the results. Second, a trajectory map of a sample episode is used to gain an understanding of how the algorithm behaved; for instance, the behavior-based formation control algorithm's trajectory map is seen in Figure 4.1. Each agent trajectory is a dotted line with a gradient; by studying these lines, the reader can get a sense of how the agents moved, their velocities, and how synchronized they were. Furthermore, the sensor range of each agent is visualized with a light circle. In addition, a line is drawn between agents to easier read their current formation displacement. Not all trajectory maps are included in this section; however, all trajectory maps can be studied in Appendix A.

4.1 Results Behavior-based Formation Control

With a behavior-based FC algorithm, good results were achieved. Without optimizing the algorithm, which is time-consuming, the use of decent hyperparameters were still very efficient. In Figure 4.1, the trajectories of agents using the control algorithm is illustrated.

4.2 Reward Function Experiment Results

This section show the results of the experiments using different reward functions, as seen in Table 3.1. All results in this section are results of using DDQN and PPO with greedy policies, meaning no exploration.

In this section, the mean reward is outlined for the different experiments' greedy policies. Furthermore the difference in reward between the learned behavior and the initial FC behavior is written in parenthesis next to the mean reward.

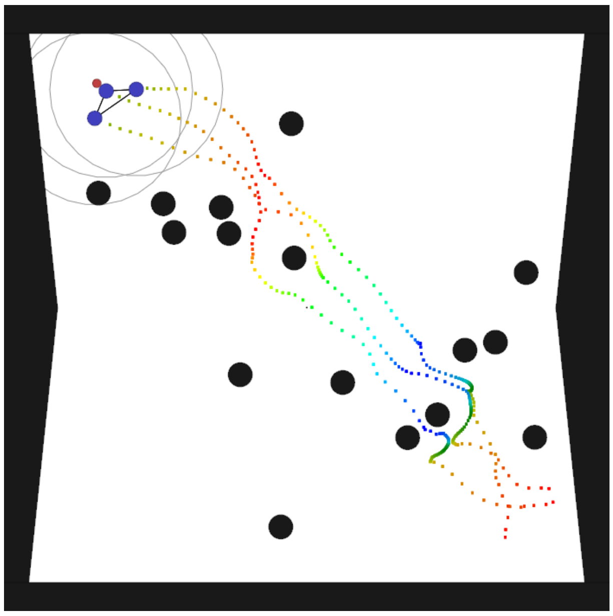


Figure 4.1: Trajectory map of agents in a line formation using a behavior-based formation control algorithm.

4.2.1 Fixed Obstacle Positions and Fixed Start and End Positions

As mentioned in Table 3.1, seven experiments was conducted to test different reward functionsin a world with fixed obstacle positions and fixed start and end positions. These seven experiments can be seen in Table 4.1.

Id	K_{ARG}	K_{GRG}	K_{OC}	K_{GFD}
1	100	0	0	0
2	0	100	0	0
3	0	100	0.5	0
4	0	100	1.5	0
5	0	100	0	0.25
6	0	100	0	1
7	0	100	0.5	0.5

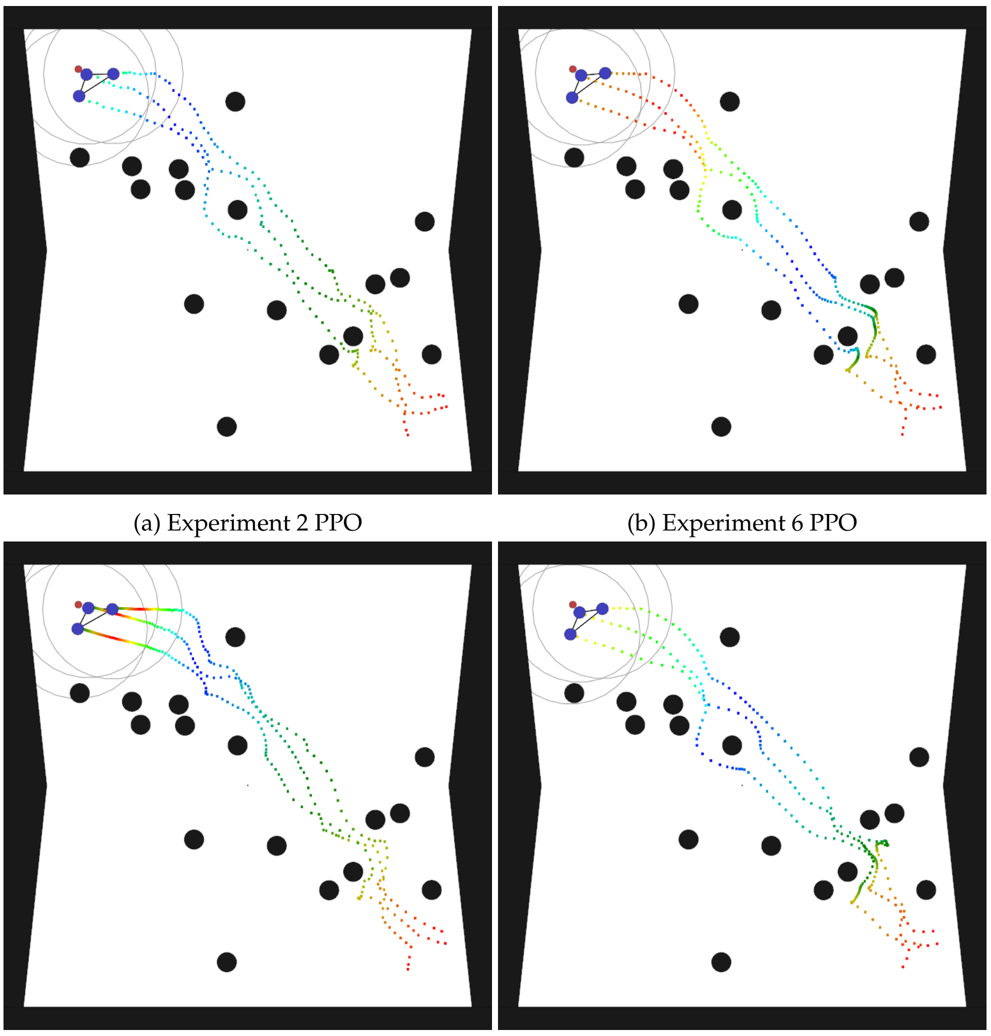
Table 4.1: Reward Function Experiments in a static environment.

The results of FC, DDQN, and PPO in a world with fixed obstacles and random start and end positions are seen in Table 4.2.

Id	Algorithm	PR	OCF	AFD	Success	Iterations to Goal	Mean Reward
–	FC	1.10	2.1%	0.15	100%	549	–
1	PPO	2.74	25.4%	0.24	0%	–	0.00 (-0.06)
1	DDQN	2.25	30.1%	0.28	0%	–	0.00 (-0.06)
2	PPO	1.06	2.0%	0.18	100%	277	0.36 (+0.17)
2	DDQN	1.11	3.0%	0.18	100%	418	0.24 (+0.5)
3	PPO	1.06	3.6%	0.19	100%	277	0.24 (+0.07)
3	DDQN	1.13	2.0%	0.18	100%	402	0.33 (+0.16)
4	PPO	0.32	0.0%	0.04	0%	–	0.00 (-0.15)
4	DDQN	1.20	0.0%	0.18	100%	574	0.17 (+0.02)
5	PPO	1.12	3.9%	0.15	100%	492	0.12 (+0.01)
5	DDQN	1.07	3.4%	0.18	100%	320	0.22 (+0.11)
6	PPO	1.09	3.1%	0.15	100%	507	-0.11 (+0.01)
6	DDQN	1.14	0.0%	0.14	100%	600	-0.12 (+0.00)
7	PPO	1.12	0.0%	0.14	100%	423	0.10 (+0.09)
7	DDQN	1.20	0.0%	0.16	100%	474	0.05 (+0.04)

Table 4.2: Experiment results in a world with fixed obstacle positions and fixed start and end position.

Arguably, the most successful experiment in Table 4.2 were Experiment 2 PPO, Experiment 6 PPO, Experiment 6 DDQN, and Experiment 7 PPO, as seen in Figure 4.2.


(a) Experiment 2 PPO (b) Experiment 6 PPO

(c) Experiment 6 DDQN (d) Experiment 7 PPO

Figure 4.2: The trajectories of the four most successful experiments in a world with fixed obstacles and fixed start and end position.

In Figure 4.3, the training progress of one conducted experiment, Experiment 2, is visualized. In the image, the progress of PPO is plotted against the number of passed episodes. Furthermore, the final result of PPO – using a greedy policy – is also shown.

4.2.2 Fixed Obstacle Positions with Random Start and End Positions

To test the performance in a world with fixed obstacle positions and random start and end positions, one reward function was tested: one which penalized obstacle collisions and formation displacement, while rewarding the group for reaching the goal. This is the same reward function seen in Experiment 7.

The results of FC, DDQN, and PPO in a world with fixed obstacles and random start and end positions.

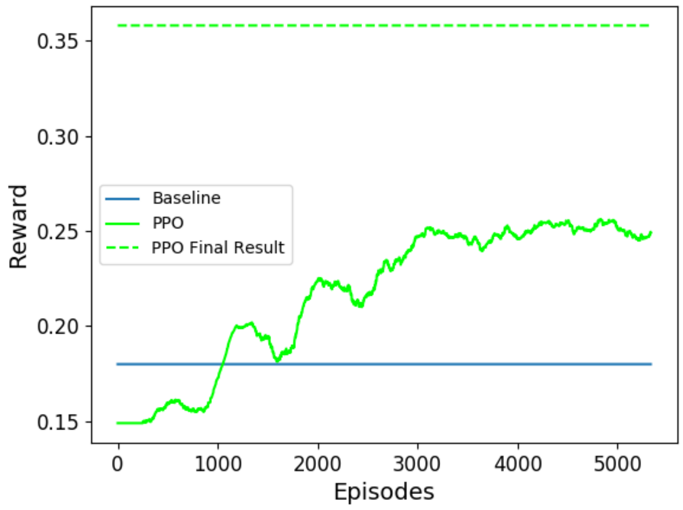


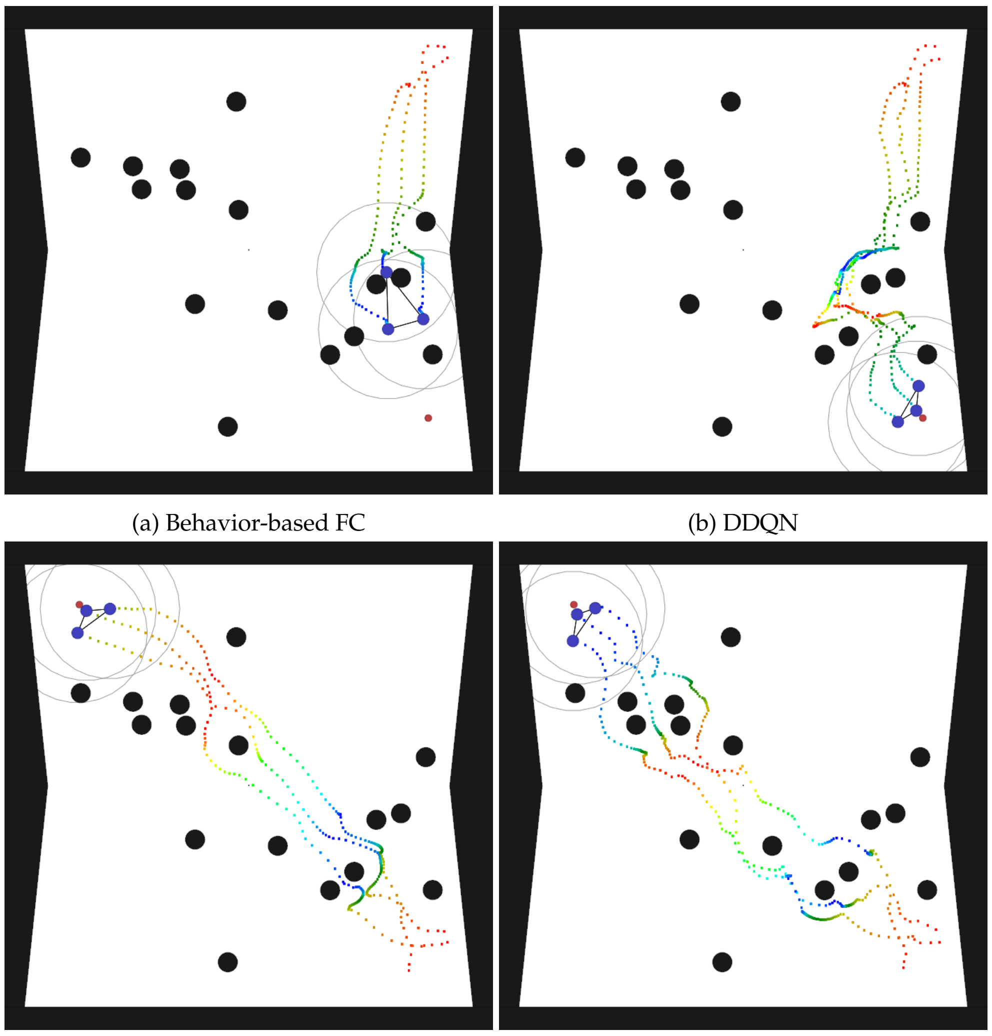
Figure 4.3: The training progress of PPO in Experiment 2.

Id	Algorithm	PR	OCF	AFD	Success	Iterations to Goal	Mean Reward
–	FC	1.12	3.5%	0.13	91%	349	–
8	PPO	1.20	5.7%	0.13	65%	329	-0.15 (-0.28)
8	DDQN	1.18	0.0%	0.15	94%	480	0.12 (-0.01)

Table 4.3: Experiment results in a static world with random start and end position.

As seen in Table 4.3, DDQN had the highest performance metrics. In Figure 4.4, it is clear that DDQN has managed to solve some of the problems the behavior-based formation control algorithm had. For instance, in Figure 4.4a it is clear that the untrained algorithm runs into an issue when an agent gets trapped between two obstacles and cannot move to its formation position or towards the goal. In the trained version, Figure 4.4b, the group takes a detour which does take a long time, but they do reach the goal.

Similarly, the difference between Figure 4.4c and 4.4d highlights how DDQN has learned to completely avoid obstacles at the cost of the time to reach the goal.


(a) Behavior-based FC (b) DDQN

(c) Behavior-based FC (d) DDQN

Figure 4.4: The difference between the default behavior-based formation control algorithm and the trained algorithm from Experiment 8 DDQN (as seen in Table 4.3).

4.2.3 Random Obstacle Positions and Random Start and End Positions

When the algorithms were instead trained in world with random obstacle positions, as well as random start and end positions, these were the results using the same reward function as in Experiment 7 and 8:

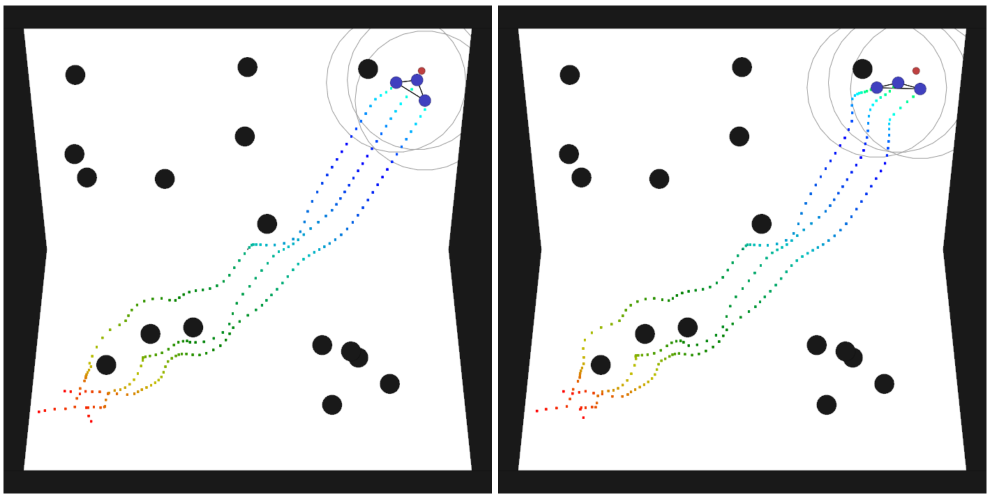
Id	Algorithm	PR	OCF	AFD	Success	Iterations to Goal	Mean Reward
–	FC	1.10	3.3%	0.13	91%	336	–
9	PPO	1.09	3.8%	0.14	89%	353	-0.01 (-0.16)
9	DDQN	1.10	1.3%	0.13	91%	391	0.11 (-0.4)

Table 4.4: Experiment results in a world with random obstacle position as well as random start and end position.

Table 4.4 shows that DDQN and PPO were quite similar in their result. Yet, PPO did not learn any new behaviors; instead, when the trained PPO algorithm ran with a greedy policy at the end of training, it chose only to utilize 10% of its action power. Thus, PPO chose to rely almost solely on the behavior-based control algorithm. Similarly, DDQN chose to perform action *do-nothing* 80% of the time.

Comparing the results between learning and no learning, it is clear DDQN challenges the behavior-based control model, whereas PPO's results are slightly worse than those of the behavior-based control algorithm. DDQN did learn to avoid obstacles at the cost of reaching the goal slightly slower. The displacement was identical.

As both DDQN and PPO relied heavily on the initial behavior-based control algorithm, it was difficult to find interesting differences in trajectories. In the end, their results were almost identical, as seen in Figure 4.5.



(a) Behavior-based FC (b) DDQN

Figure 4.5: The similarities between the default behavior-based formation control algorithm and the trained algorithm from Experiment 9 DDQN (as seen in Table 4.4).

4.3 Experiment Results of Hidden Layer Usage

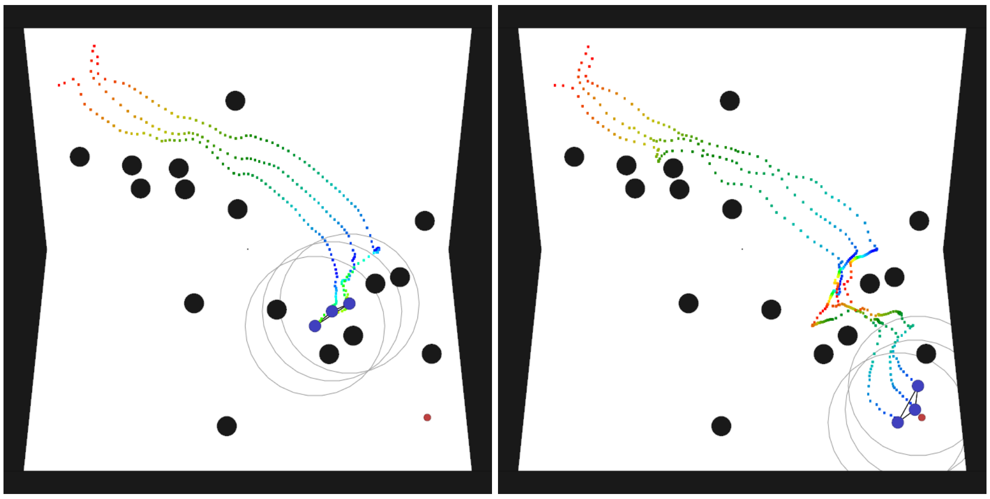
To study the importance of the hidden layer in DDQN, a model without any hidden layer was used in a static environment with random start and end positions, as described in Section 3.8. The results this Double Q-network are seen in Table 4.5. For these experiments, the same reward function used previous

Id	Algorithm	PR	OCF	AFD	Success %	Iterations to Goal	Mean Reward
10	Double QN	1.19	3.2%	0.14	100%	367	0.11 (+10)
11	Double QN	1.34	5.0%	0.17	50%	292	-0.03 (-0.18)

Table 4.5: Results of a Double Q-Network with no hidden layers trained on a fixed world with random start and end positions.

Looking at Table 4.5, it is clear that the model with no hidden layer only succeeds in Experiment 10: the basic case. In the more complex case, Experiment 11, the model only reaches the goal half the time. The only times the model manages to reach the goal, the goal is reachable in a short distance and thus more likely not blocked by obstacles.

Comparing these results to the DDQN trained in the same environment, it is clear where the model with no hidden layers fail. This is illustrated in Figure 4.6, where a comparison, in a static world with random start and end positions, is made.



(a) No Hidden Layer (b) With Hidden Layer

Figure 4.6: The difference between DDQN trained with zero and one hidden layer.

5 Discussion

This chapter contains discussions about the results, analyzing them in the context of the thesis aims, theory, and related work. Furthermore, an outline of potential future work is discussed.

5.1 Reward Function Experiments

The results to these experiments, seen in Section 4.2, shows that it is possible for a reinforcement learning algorithm to assist a behavior-based formation control algorithm. Of the conducted experiments, some results were more interesting than others.

For instance, using a reward function that only rewards the agent which reaches the goal, R_{ARG}, yielded disastrous results. As one agent always had a head start in reaching the goal – the agent which reached the goal first with only the default behavior-based control algorithm – the other two agents had to ruin the privileged agent's chances of reaching the goal. However, as the default behavior-based control algorithm assisting the learning algorithm always forced the agents to remain somewhat close, the group found itself staggered in the end, caught in a deadlock where no one reached the goal. One of the agents might, while looking for a way to reach the goal first, found a behavior which ruined the other two agents chances of being first.

If R_{GRG} was used in place of R_{ARG}, the result was completely different. The group instead converged to a behavior where they reached the goal quickly, although with a slightly worse average formation displacement. Yet, this is maybe the main contribution of this thesis. After all, the formation was maintained surprisingly well by solely the behavior-based control foundation; at the same time, the speed at which PPO reached the goal was more than the double initial speed and the collision rate was roughly the same. In short, Experiment 2, using only R_{ARG}, reached one of the best results while using the simplest kind of reward function.

The best result, depending on which characteristics were desirable, was most likely achieved by DDQN in Experiment 7. The behavior found by the algorithm – which used a reward function that rewarded reaching the goal and penalizing obstacle collisions and formation displacement – never collided with obstacles, remained in a very good formation and reached the goal faster than the behavior-based control algorithm. The only slight issue was a higher path ratio; yet, the path ratio is most likely an indicator of the algorithm adjusting the formations to not collide with any obstacles. This belief was supported by Experiment 4

DDQN, which was the only other experiment reaching such a collision frequency and which shared the same path ratio. The usability of DDQN with this reward function was further confirmed when it outperformed the behavior-based control algorithm in a scenario with random start and end positions, as seen in Figure 4.4.

The finding of Experiment 8, as seen in Figure 4.4, might present actual use. After all, agents could potentially be trained in any environment, and potentially released into the physical realm. For instance, agents could be trained to move in formations within a warehouse, where they together move heavy boxes, or a farm, where they either move objects, harvest fields or perform other duties.

Another interesting finding was Experiment 3, in which PPO failed to find the objective due to a very high penalty for colliding with obstacles. Instead PPO converged to a behavior in which it remained in the starting location in a perfect formation, thus never risking collision with an obstacle. DDQN did successfully find a way of reaching the goal while never colliding with the goal. Perhaps PPO could have found a similar solution given more time.

Finally, in the most complex environment tested (Experiment 9), the world with random obstacles, random start, and random end position, both PPO and DDQN managed to more or less maintain the performance of the behavior-based models they assisted by learning to do nothing at all. Although it would have been desirable to achieve better performance than the initial behavior-based control results, it was still comforting to know the algorithms were intelligent enough to step back when they were clueless. This suggests that an addition of the reinforcement learning might be a good thing in most cases; after all, performance improved in most experiments.

The difference in reward between the behavior-based control algorithm and reinforcement learning algorithm was definite in experiments featuring a world with fixed obstacle positions and fixed start and end positions. However, as soon as the world became more dynamic, both DDQN and PPO struggled to improve the reward. There are many reasons which may explain this behavior. The agents could have overfitted to individual episodes, or they could have required more training time. Furthermore, the complexity of the environment, although visually simple, is rather complex; agents have a continuous state and action space, there are multiple agents concurrently attempting to learn new behaviors, and the world is changing between episodes.

5.2 Rewarding social behavior

Defining a reward function which promotes a specific social behavior, such as moving in formations, is difficult. Since designing a shaped reward function require great knowledge of the task, the social behavior must be understood very well. For instance, rewarding an agent tasked with greeting people is not trivial; when is the agent rewarded for hugging a person, and when is it rewarded for shaking a hand? Using a sparse reward may be possible by checking if the encountered person smiles after performing a handshake or hug (which may have many flaws), but a shaped reward would require the programmer to define the task of greeting someone in numeric values.

Similarly, rewarding good formation control is difficult. When is it acceptable to deviate from the formation, and how should the reward signal promote robust formations? Moreover, most tasks require more than simply lining up in formation; most tasks require agents to maintain formations while pursuing additional objectives, such as moving towards a goal, obstacle avoidance or other tasks.

The simplest way of promoting agents to move in formations, shaping a reward signal that penalizes agents for their direct formation displacement, has several disadvantages. First, as the agents most likely are doing something else while maintaining formations, such a reward signal must be extremely balanced to encourage formations while not disturbing the other objectives. Second, a reward directly penalizing displacement is only as strong as its weakest link. For instance, suppose an agent has to make a large deviation from the formation to pass an obstacle. In such a scenario, the centroid of the group will be skewed and every agent will, unless they move to follow the deviating agent, be displaced from their formation position. One way of addressing this issue is to fix the formation centroid to a leader; however, the leader could be the agent who has to deviate from its position, resulting in a similar scenario where the entire group has to make huge adjustments to the leaders every move.

Another strategy for teaching multiple agents to move in formation, which was not possible to attempt within the timespan of this thesis, would have been to create an environment in which agents' success depend on their ability to hold a formation. For instance, in a scenario where a group of shield bearers are being bombarded with arrows from every direction, the reward could simply be a positive value given after a specific duration. Possibly, the agents would then learn how to survive by forming a Roman Testudo Formation (see Figure 5.1). Such a scenario could then be extended by rewarding agents for winning a battle, potentially resulting in the switching of formations. Yet, an issue with this kind of reward signals is that the best behavior for survival, or winning the battle, may not include any formations at all. In addition, in complex tasks such as formation control during movement and obstacle avoidance, there may not be any direct way to change the environment to promote formations without changing the entire scenario.

Another element to formations is that they may be a precaution for the worst scenario. For instance, moving in a specific formation might be beneficial when getting ambushed; but how often does a squad get ambushed? The vast majority of the time, the squad might be fine doing something optimal for non-combat. However, the rare occasion the squad gets attacked and demolished, the reward function should yield a terrible score. Training an algorithm to keep formations would then force the programmer to choose how often the catastrophic scenario occurs, thus introducing a bias.

Finally, in a military context, formations may simply be a social construct by man used to promote understandable order among units. These formations and the theory behind them may not be optimal, and agents receiving sparse rewards could learn a completely different system.

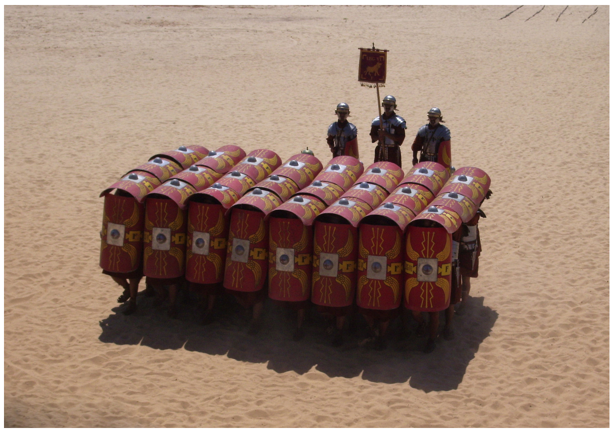


Figure 5.1: A Testudo formation in Roman reenactment. [1]

5.3 Experiments about Use of Hidden Layers

From the results of this thesis, the hidden layers of the modern reinforcement algorithms seem to matter. The model with no hidden layer failed as soon as the environment got slightly more complex (by introducing random start and end positions).

[1]Image by Neil Carey licensed under CC BY-SA 2.0

This finding strengthens the legibility of existing research into modern reinforcement learning, which to a large extent is based on models multiple with hidden layers.

5.4 Environment

The environment, which in many ways was very complex, was also basic. Complex attributes, such as continuous state space comprising of mostly local sensor data, the complexity of moving point masses, and the fact that it is a multi-agent scenario, were many and obvious. Yet, the environment and experiments conducted inside of it mostly featured tests in a static environment. Furthermore, the agents could apply forces in any direction with no delay, meaning there was less need to plan ahead; the agents could be successful while being somewhat reactionary.

5.5 Combining Formation Control with Reinforcement Learning

In this thesis work, a behavior-based formation control algorithm was combined with reinforcement learning. Although this approach showed some success, it would have been interesting to also examine the use of a leader-following approach. However, most obvious ways of combining leader-following formation control with reinforcement learning seem to suffer from flaws.

For instance, one approach would be to control the tension function w, as seen in Equation 2.1, with reinforcement learning. Allowing the reinforcement learning to control this function can be problematic for several reasons. First, the tension function has to be reciprocal, meaning every agent has to use the same tension function every timestep. With exploration and decentralization, meaning every agent uses a different tension function each timestep, the stability of leader-following is no longer guaranteed and agents could easily lose their connection to the group. Therefore, the reinforcement learning algorithm would have to pick one tension function for all agents at each timestep, meaning the reinforcement learning algorithm would have to use the states of all agents to make one decision. However, using each agent's state in a centralized manner violates the decentralized aspect of leader-following. Another reason as to why this approach is very instable is the tension function's sensitivity to noise. As the agents explore their action space, their behavior can become very erratic, making learning impossible, if the noise affects the tension function too much.

The second approach to integrate reinforcement learning with leader-following could be to use pre-determined tension function w. The reinforcement learning could then have the option to add a small force to the agent, on top of the force generated by the consensus equation. This solution would resemble how the behavior-based formation control algorithm was combined with reinforcement learning in this thesis' work. However, this approach is also problematic due to the exploration of the action space, which can cause agents to lose track of the leader. Furthermore, using such an approach would also lead to a final behavior which is not reciprocal.

5.6 Comparison with State-of-the-art

Because of primarily two reasons, it is difficult to compare deep reinforcement learning results between different labs.

First, a problem with reinforcement learning, and science in general, is the publication bias. Presenting a compelling story – a positive result – peaks interest. On the other hand, a negative result rarely gets notoriety. This phenomenon has resulted in a publication bias; scientists are encouraged to pursue positive results. Among psychology studies, one study even suggested as many as 50% of studies cannot be replicated [5]. Although the magnitude of publication bias in the field of reinforcement learning has not yet been measured, some results show that positive multi-agent reinforcement learning results may, even if they visually look stunning, use very overfitted models [15].

Disregarding from publication bias, deep reinforcement learning also suffers from sensitivity to noise. One paper suggests the random seed of state-of-the-art algorithm may cause a failure rate of up to 25%, which is seen in Figure (c) in [11]. This finding has been supported by other scientists such as Andrej K, Tesla's Director of AI [2].

This not only makes comparisons between related works more difficult, it also makes discoveries less reliable. Although the sources in this thesis consists of papers from the highest academic institutions in the field, it is in most cases not clear which values were used for different hyperparameters, how much computational power was required, and how overfitted the end result was.

One of the most similar multi-agent learning works was conducted by researchers at Stanford [13]. They showed that a policy gradient method closely resembling PPO, Trust Region Policy Optimization (TRPO) with parameter-sharing, proved most efficient for multi-agent learning. Yet, they still concluded: *"Despite the advances in decentralized control and reinforcement learning over recent years, learning cooperative policies in multi-agent systems remains a challenge"*. According to their findings, the difficulties stem from high-dimensional observation spaces, partial observability, the fact that it is multi-agent, and continuous action-spaces. Although some of these difficulties could be combated, by for instance giving agents full observability of the environment, such changes would defeat the purpose of many applications [13].

Their findings support much of the findings in this thesis. Although the results of some experiments were successful, the learning was far from trivial and the results were initially expected to be better.

5.7 The Work in a Wider Context

In essence, formations are something we take part of every day. Regardless if we are strolling next to a friend in a park, playing American Football, or moving through the woods in a line formation to search for a lost child, we engage in some kind of agreement of how to move in unison. Yet, for us to interact in such a way with silicon-based entities, we must first teach them how to move naturally in a multi-agent scenario – such as the world is. This work brings us a step closer to understanding how we could transfer our desirable formations to digital agents.

5.8 Future Work

It would have been interesting to investigate four other aspects of this work.

First, using a leader-referenced formation position technique would have been very interesting. While it is uncertain whether a group of agents could be trained using a leader-referenced formation position – due to a prominent weak link, which is the leaders exploration – there might have been other ways to test a leader-referenced technique. For instance, the group could possibly have been trained by a unit-center-reference, as done in this thesis, but then tested in a leader-referenced scenario. Preferably, this should be done with physical agents and a human leader; such a scenario could highlight how natural the groups dynamics actually are. It would also be a easy way to challenge the group, as the person embracing the leader role could forcefully test the group's strengths and weaknesses.

A second change which would make the environment a lot more complex, but also more interesting, would have been to use agents with more realistic movement abilities. Instead of

[2]https://news.ycombinator.com/item?id=13519044

being able to react with a force in any direction, perhaps the agents could have had a turn-rate limiting the speed which they could change direction with. At the same time, the point mass scenario used in this thesis did resemble some physical agents found in air and water. Furthermore, it could be argued that humans can apply a force in any direction.

Third, it would have been interesting to investigate other ways of merging formation control with reinforcement learning. Perhaps it would have been interesting to study how the agents behavior would have changed if the reinforcement learning could choose to ignore the other behaviors by adding a vector with high magnitude. Yet, such a solution would have to employ a reward function taking formations in consideration; otherwise, the algorithm would be likely to consider the formation a burden and always choose to overpower the formation behavior.

Finally, a very interesting change would be to investigate scenarios where the agents end success depend on their ability to actually learn a formation, such as described in Section 5.2. Only in such a scenario could we learn something from the agents, instead of the opposite.

6 Conclusion

In this thesis, a behavior-based formation control algorithm was merged with deep reinforcement learning to improve performance. By using deep reinforcement learning as an added behavior in the behavior-based control algorithm, the reinforcement learning algorithm naturally gained a foundation from which it could learn and improve.

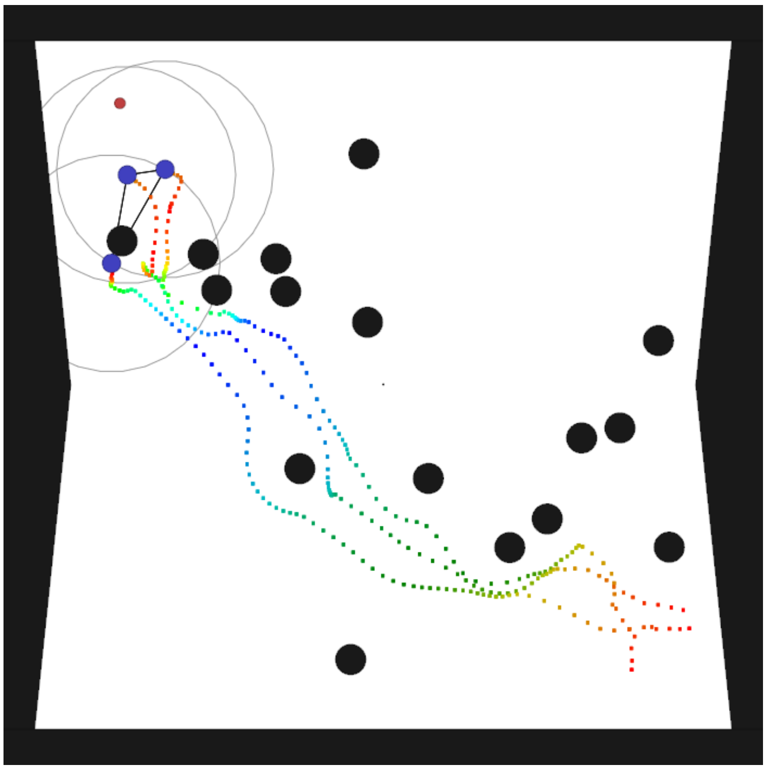
The benefits of integrating deep reinforcement learning with the behavior-based control algorithm varied. Depending on the scenario the agents faced and the reward function utilized, the agents were able to improve in different ways. For instance, they could be taught to move faster or avoid obstacles better. In some cases, they also learned to reach the goal more often and avoid deadlocks otherwise prominent in the behavior-based control algorithm. The only thing which was never drastically improved was the formation displacement; yet, the end result in some experiments was clearly superior as the formation displacement remained low as the agent more than doubled its speed or completely avoided obstacles.

Furthermore, the results of this thesis suggest the hidden layers of deep reinforcement learning models are helpful in complex experiments. However, in the most simple experiments, the addition of a hidden layer was not beneficial.

Shaping a reward signal for multi-agent cooperation, such as the environment of this thesis, is difficult. Unless the environment somehow allows for a sparse reward signal, a shaped one must be used. Balancing the shaped reward signal incorrectly can yield disastrous results, as shown in Experiment 4 PPO, where agents refused to leave their starting position out of fear of colliding with obstacles. Yet, one finding of this thesis suggested that the most simple reward signals, such as only using a full reward given to the entire group upon arriving to a goal destination, may be one of the best options.

A Trajectory Images

This appendix present the trajectory images corresponding to the results seen in Table **??**.


(a) PPO

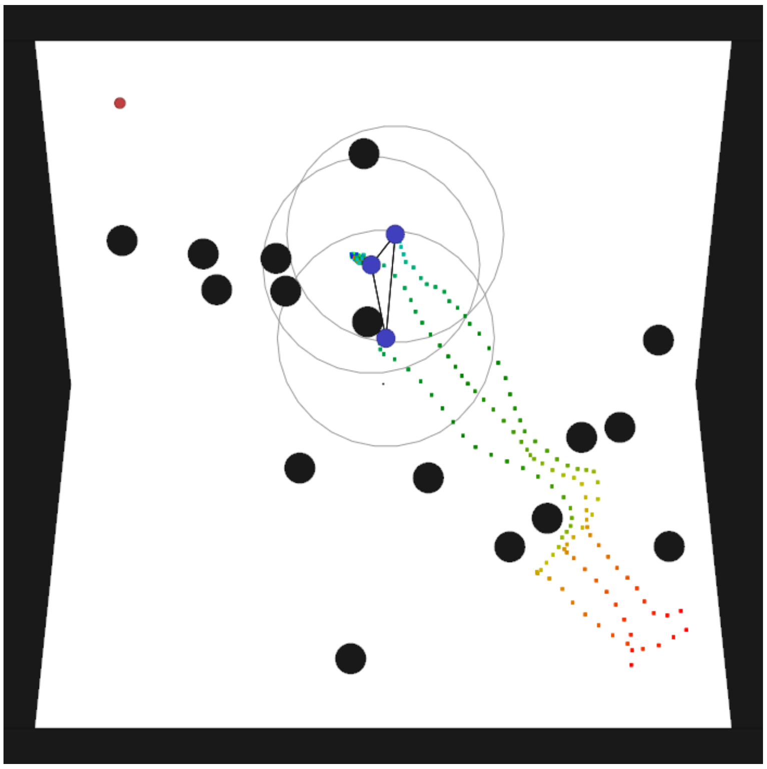
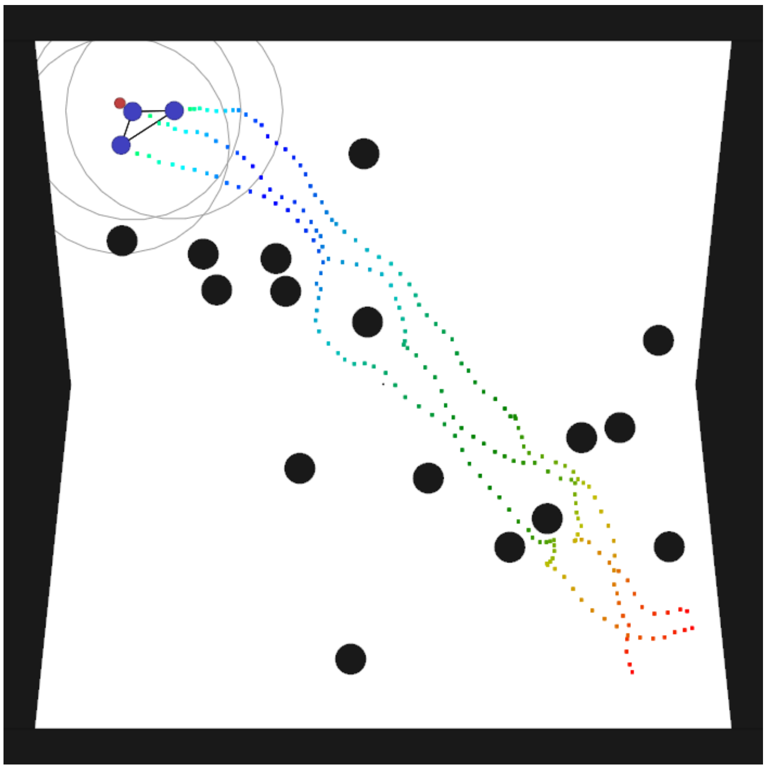

(b) DQN

Figure A.1: Experiment 1.


(a) PPO

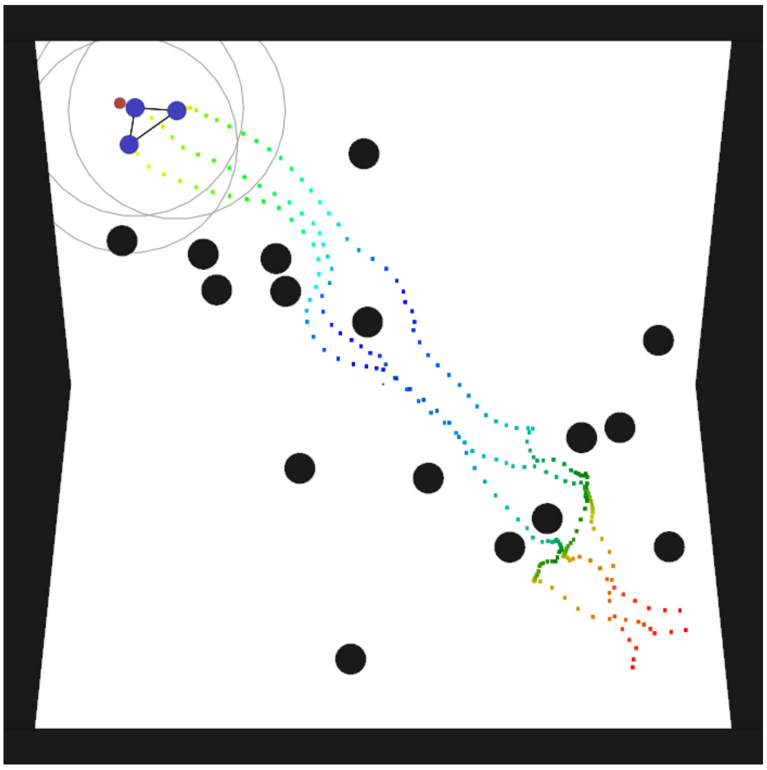
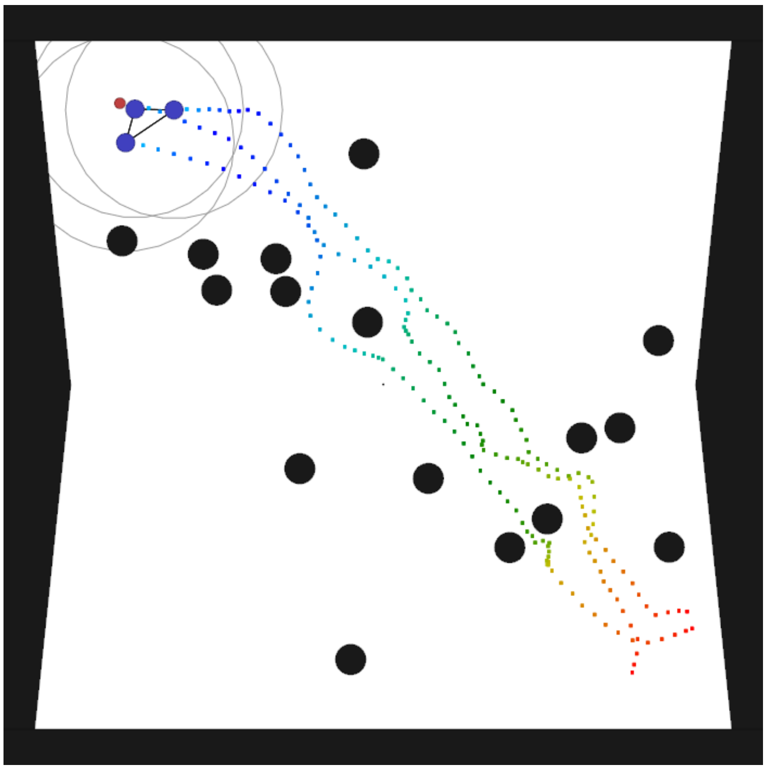

(b) DQN

Figure A.2: Experiment 2.


(a) PPO

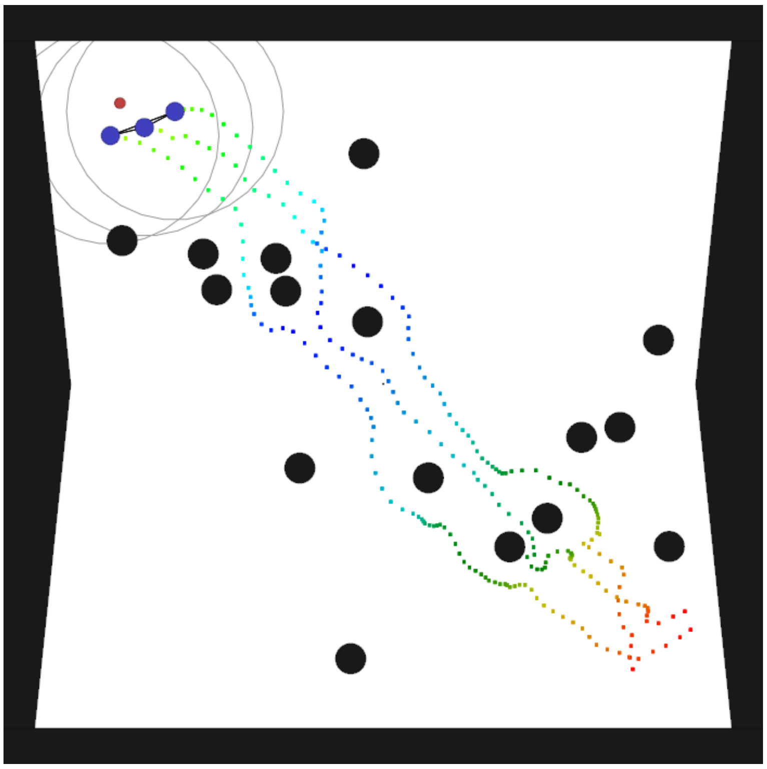
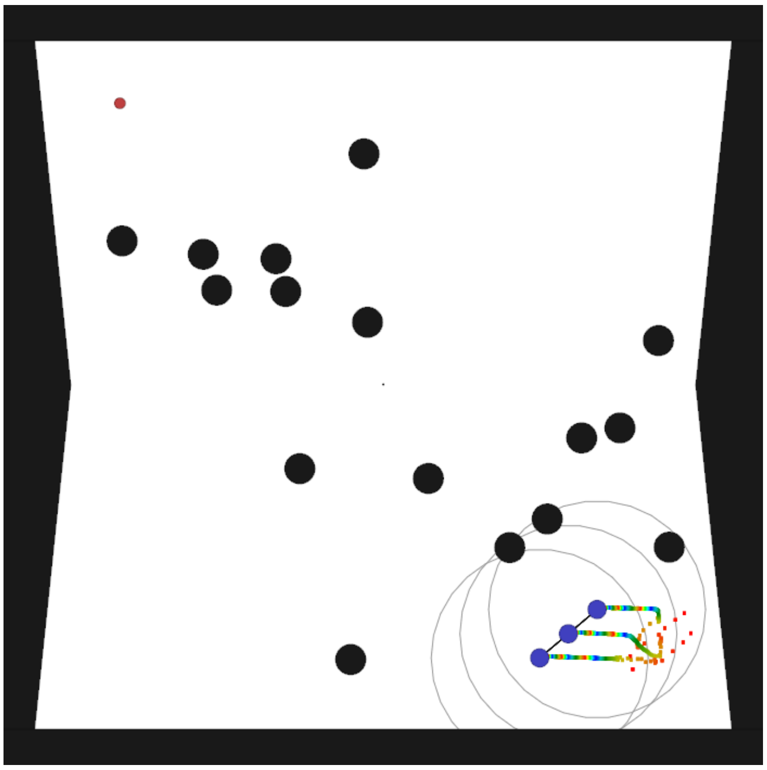

(b) DQN

Figure A.3: Experiment 3.


(a) PPO

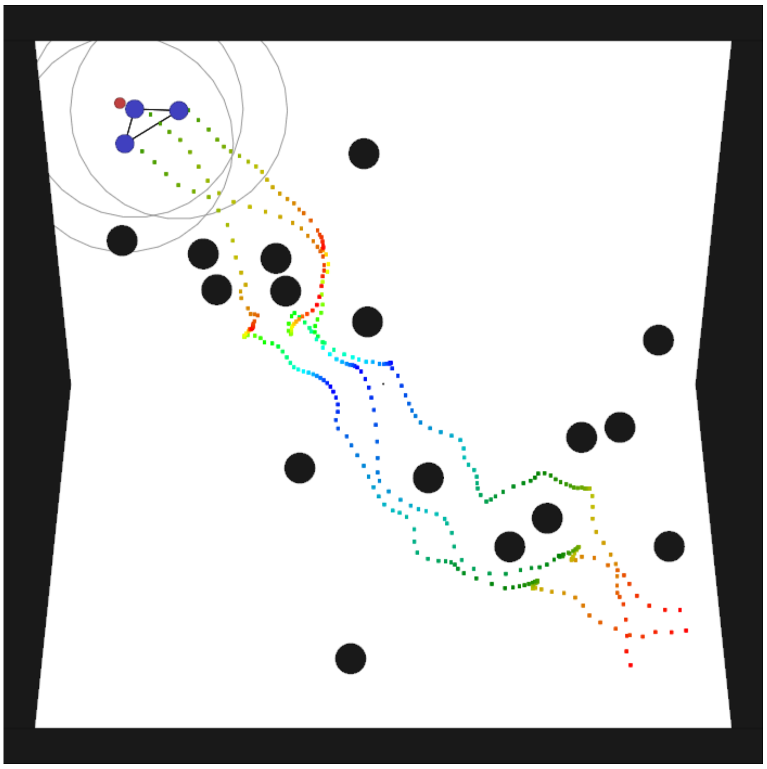
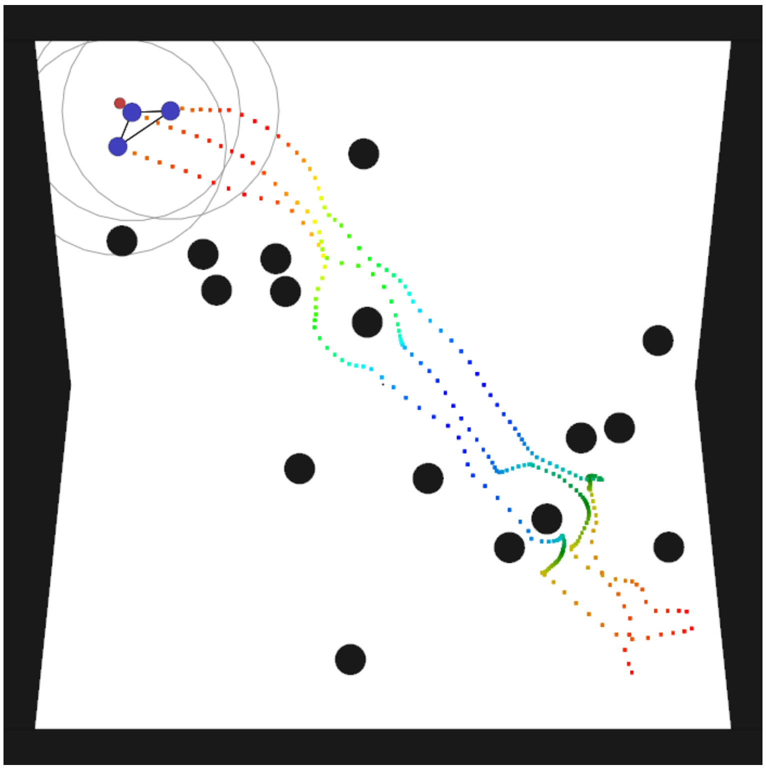

(b) DQN

Figure A.4: Experiment 4.


(a) PPO

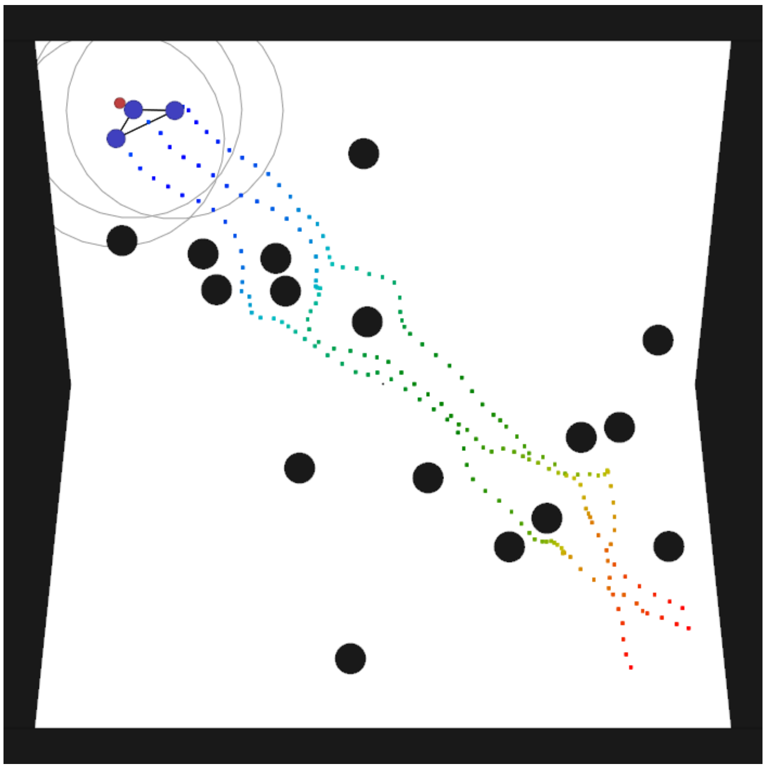
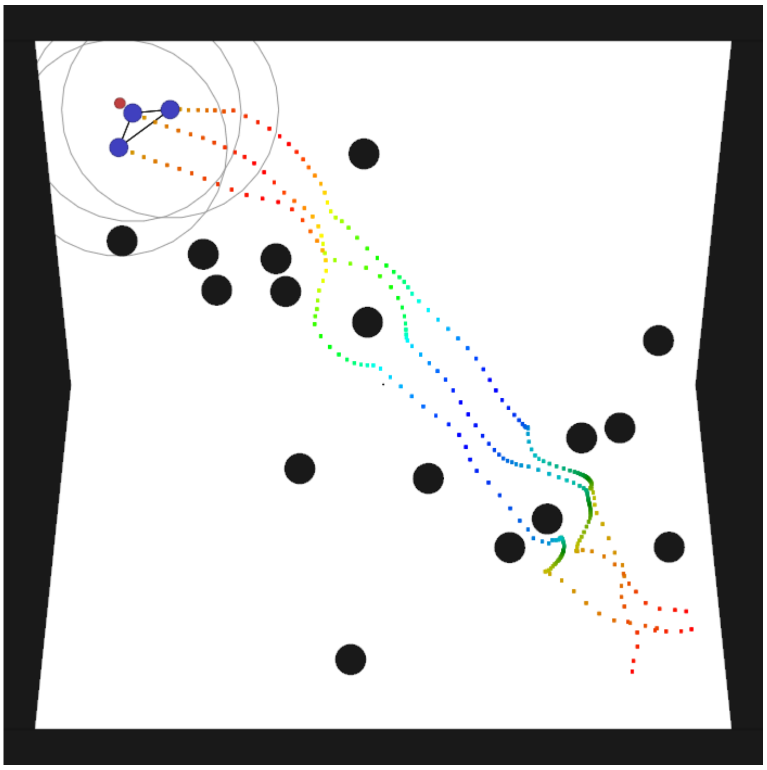

(b) DQN

Figure A.5: Experiment 5.


(a) PPO

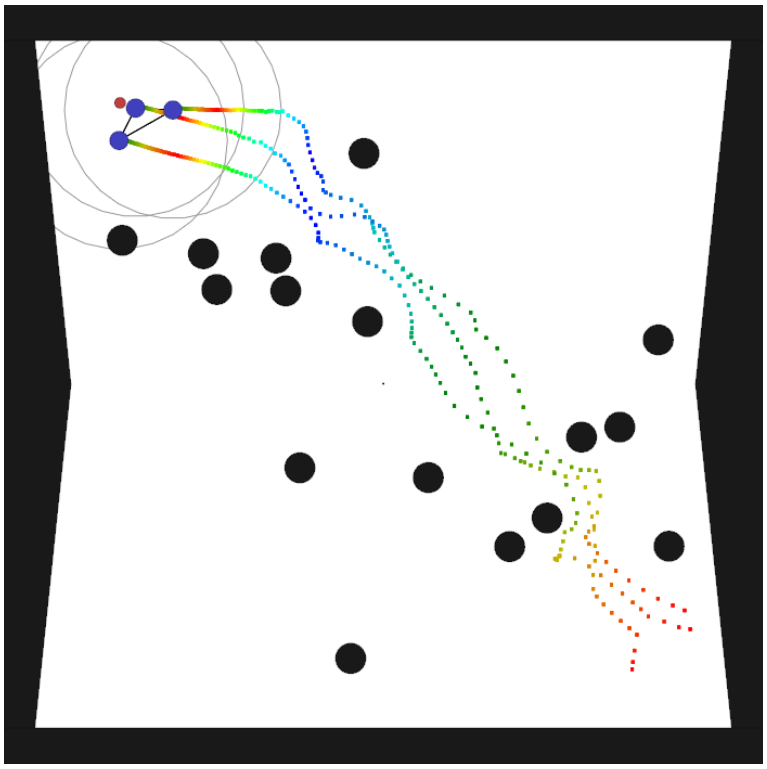
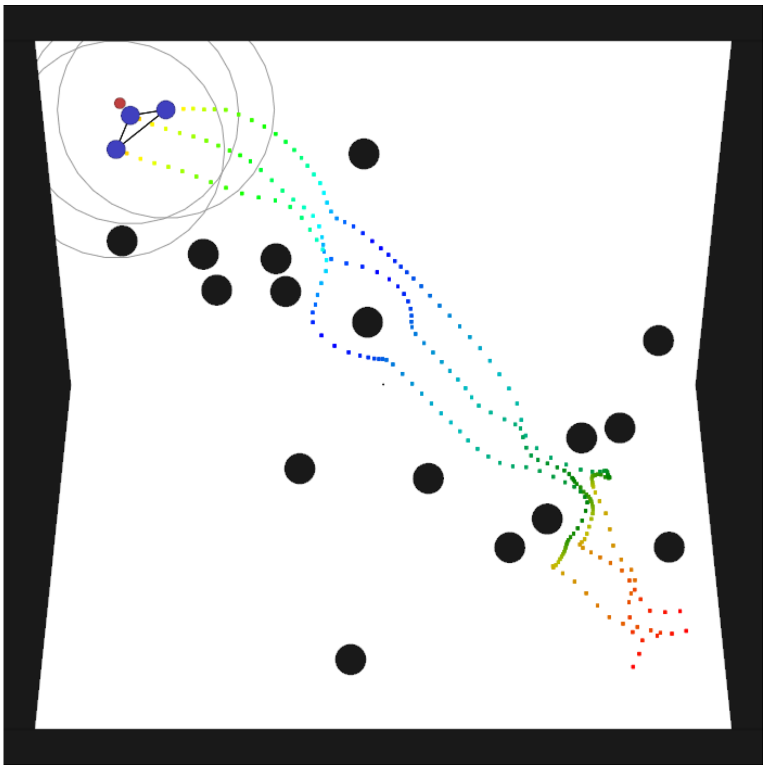

(b) DQN

Figure A.6: Experiment 6.


(a) PPO

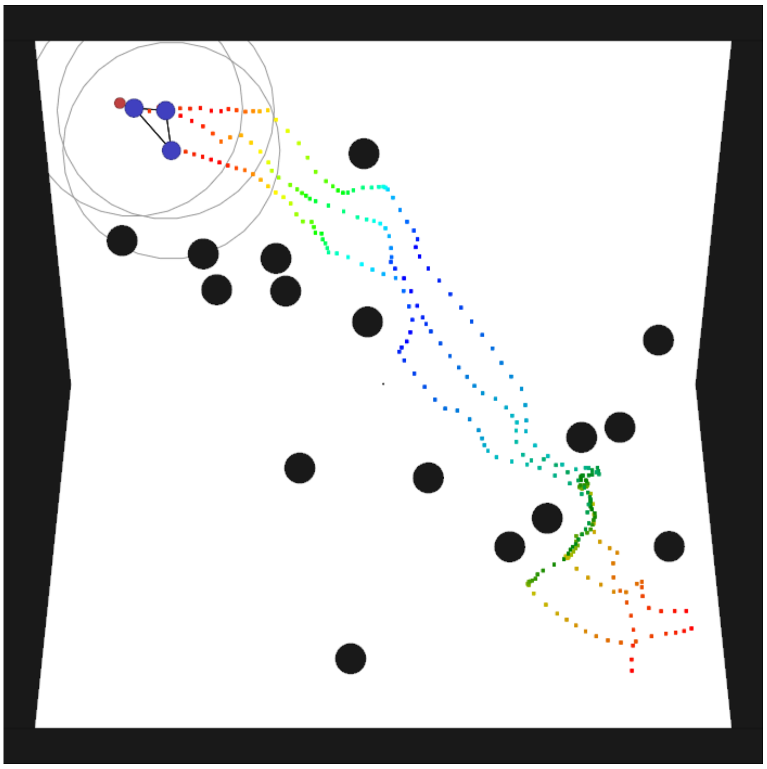
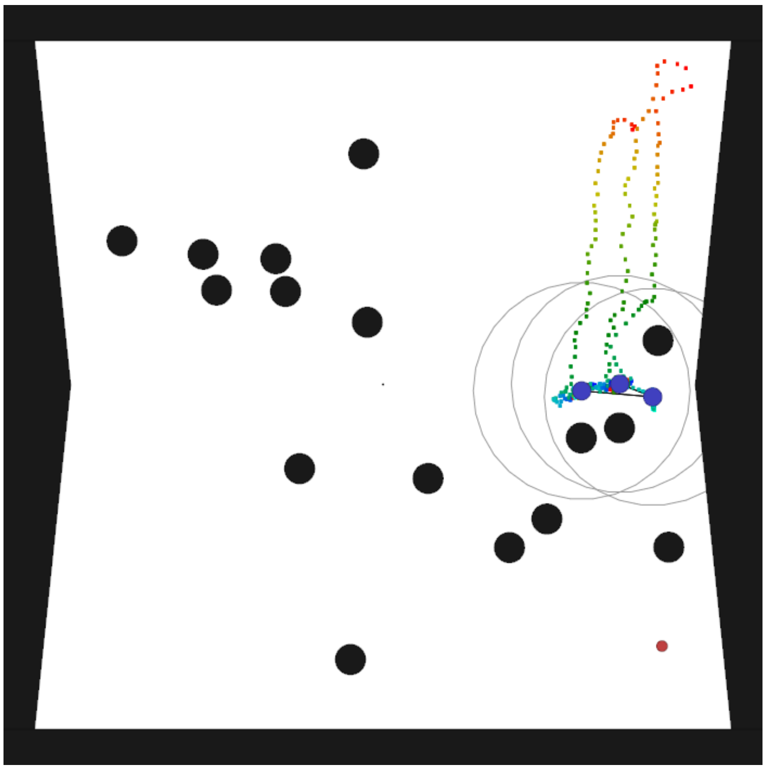

(b) DQN

Figure A.7: Experiment 7.


(a) PPO

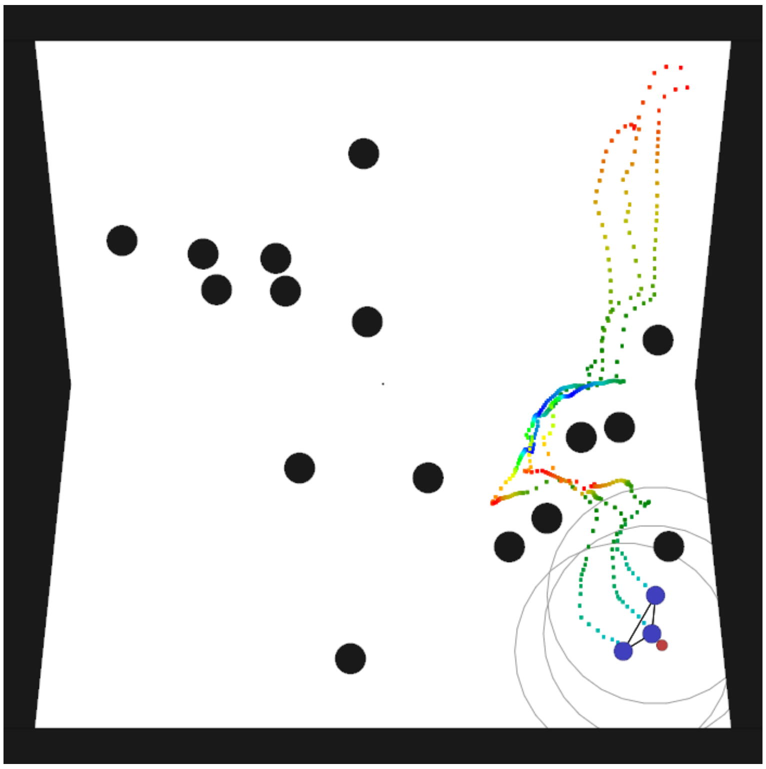
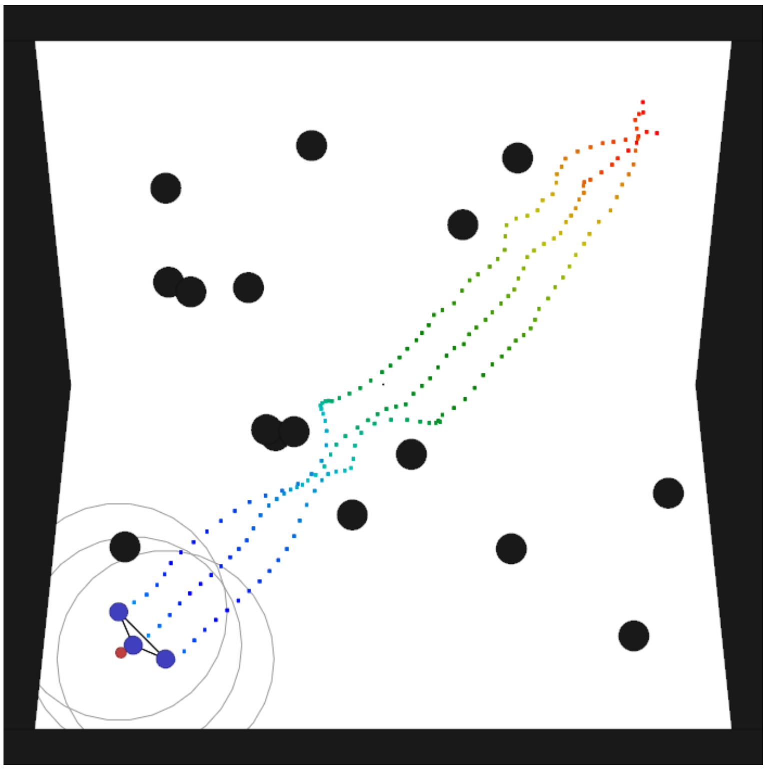

(b) DQN

Figure A.8: Experiment 8.


(a) PPO

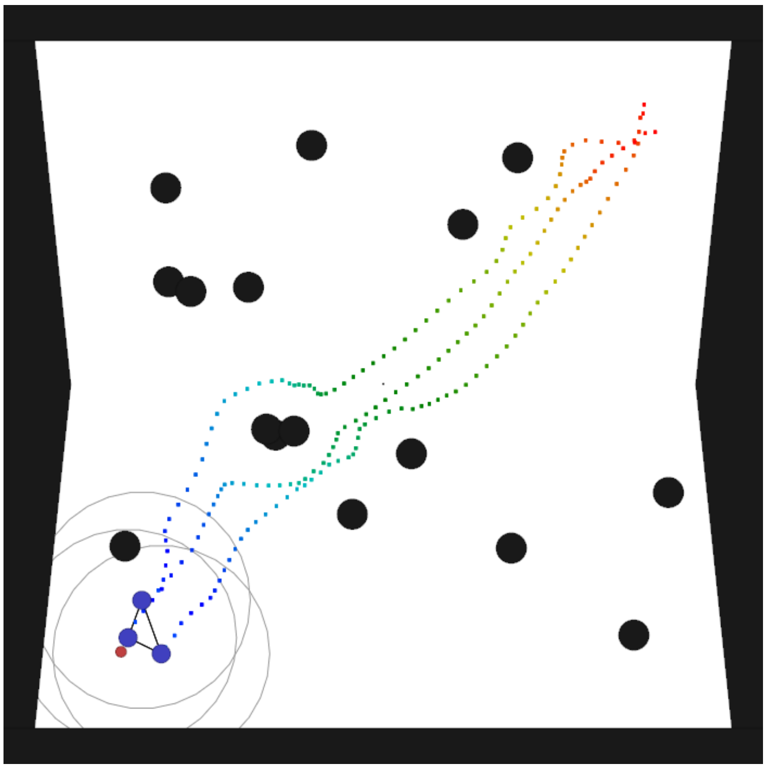

(b) DQN

Figure A.9: Experiment 9.

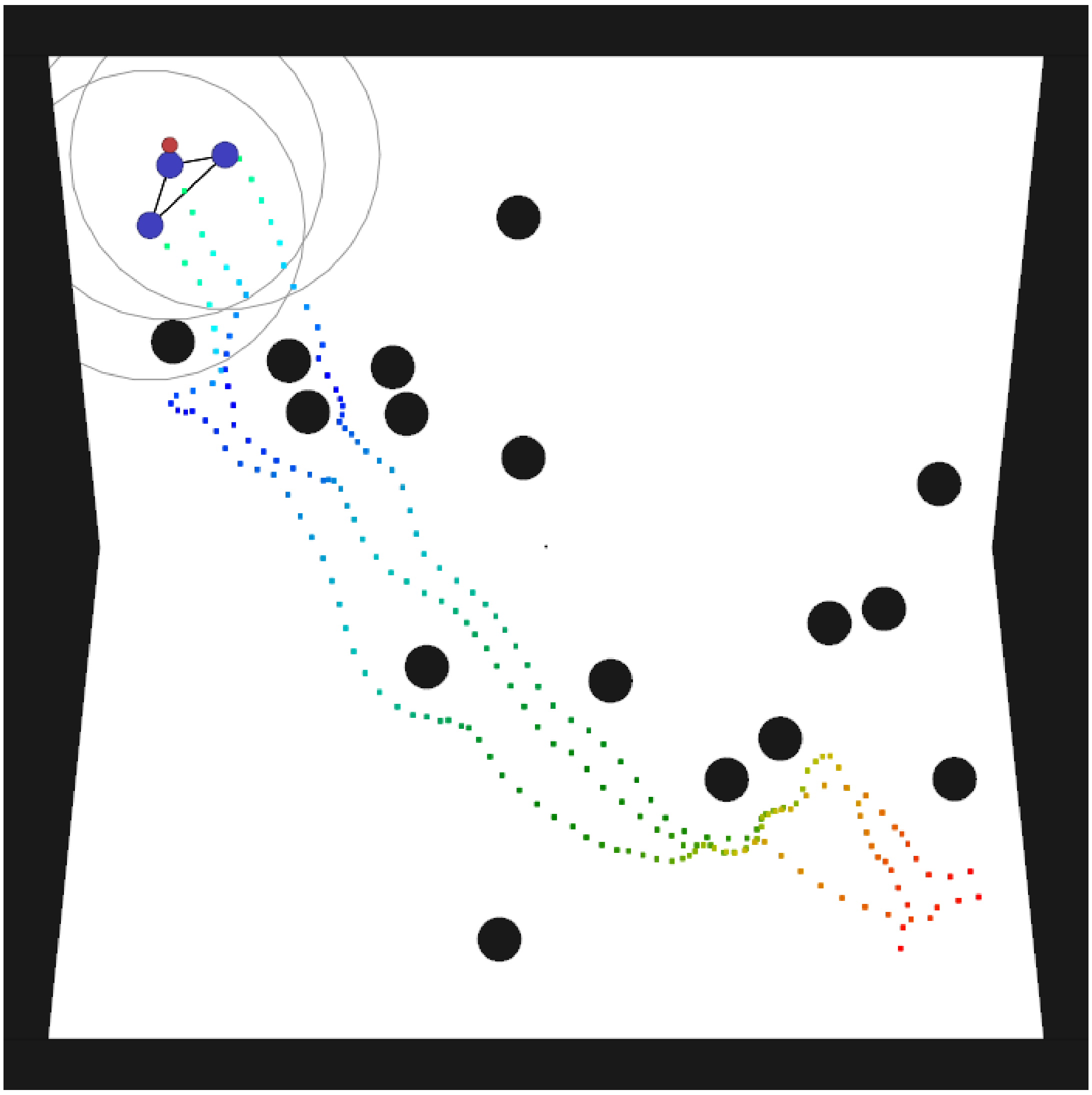


Figure A.10: Experiment 10.

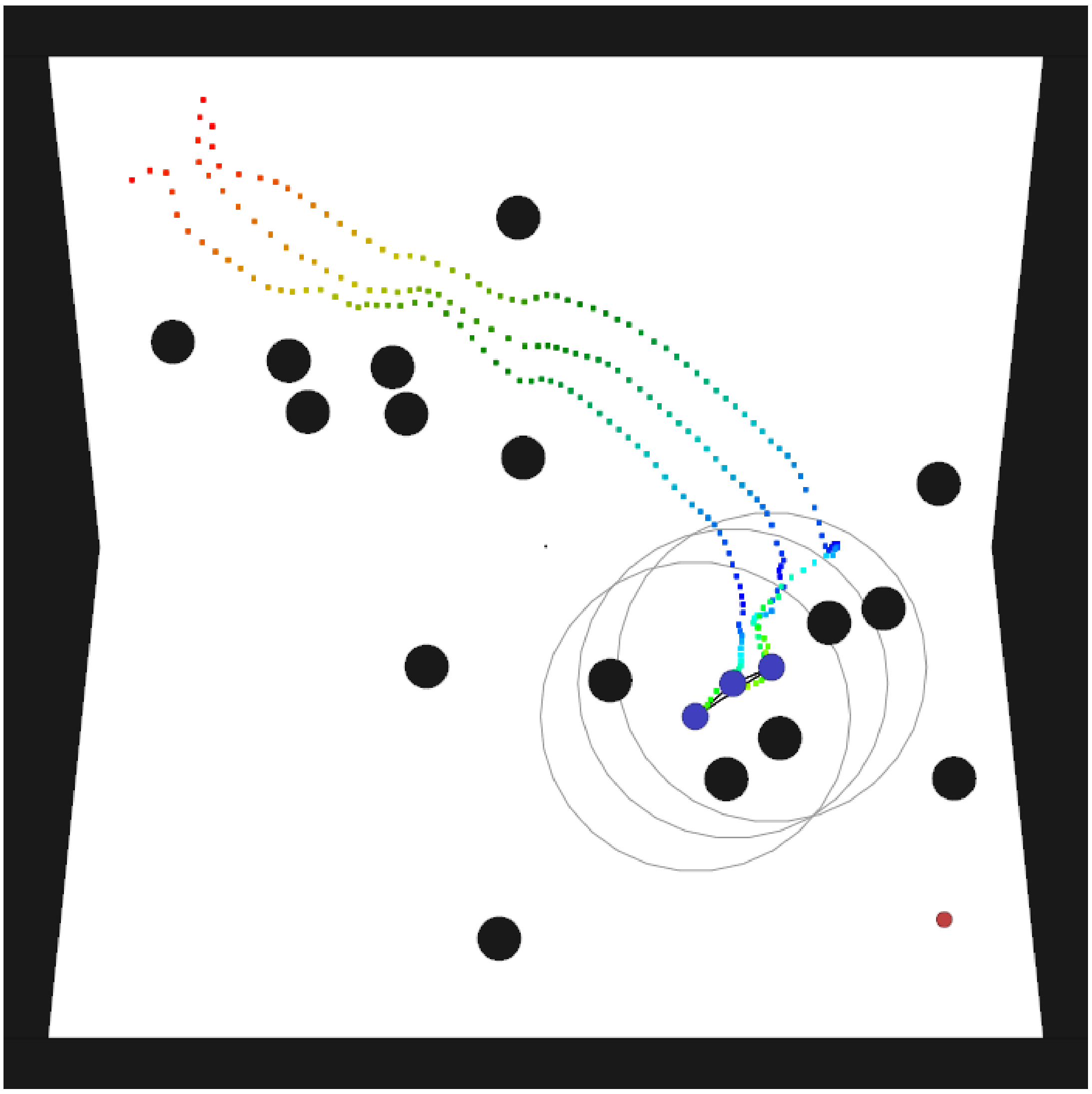


Figure A.11: Experiment 11.

Bibliography

[1] T. Balch and R. C. Arkin. “Behavior-based formation control for multirobot teams”. In: *IEEE Transactions on Robotics and Automation* 14.6 (Dec. 1998), pp. 926–939. ISSN: 1042-296X. DOI: `10.1109/70.736776`.

[2] R. W. Beard, J. Lawton, and F. Y. Hadaegh. “A coordination architecture for spacecraft formation control”. In: *IEEE Transactions on Control Systems Technology* 9.6 (Nov. 2001), pp. 777–790. ISSN: 1063-6536. DOI: `10.1109/87.960341`.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[4] X. Chen and R. W. Brockett. “Centralized and decentralized formation control with controllable interaction laws”. In: *53rd IEEE Conference on Decision and Control*. Dec. 2014, pp. 601–606.

[5] Open Science Collaboration et al. “Estimating the reproducibility of psychological science”. In: *Science* 349.6251 (2015).

[6] Vali Derhami and Yusef Momeni. “Applying Reinforcement Learning in Formation Control of Agents”. In: *Intelligent Distributed Computing IX. Studies in Computational Intelligence* (Oct. 2016). DOI: `10.1007/978-3-319-25017-5_28`.

[7] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. *OpenAI Baselines*. `https://github.com/openai/baselines`. 2017.

[8] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. “Counterfactual multi-agent policy gradients”. In: *arXiv preprint arXiv:1705.08926* (2017).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[10] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-Learning”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI’16. Phoenix, Arizona: AAAI Press, 2016, pp. 2094–2100.

[11] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. “Vime: Variational information maximizing exploration”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 1109–1117.

[12] M. Ji and M. Egerstedt. “Distributed Coordination Control of Multiagent Systems While Preserving Connectedness”. In: *IEEE Transactions on Robotics* 23.4 (Aug. 2007), pp. 693–703. ISSN: 1552-3098. DOI: `10.1109/TRO.2007.900638`.

[13] Jayesh K. Gupta, Maxim Egorov, and Mykel Kochenderfer. “Cooperative Multi-agent Control Using Deep Reinforcement Learning”. In: *International Conference on Autonomous Agents and Multiagent Systems* (Nov. 2017), pp. 66–83.

[14] Leslie P. Kaelbling, Michael L. Littman, and Andrew P. Moore. “Reinforcement Learning: A Survey”. In: *Journal of Artificial Intelligence Research* 4 (1996), pp. 237–285.

[15] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Julien Perolat, David Silver, Thore Graepel, et al. “A unified game-theoretic approach to multiagent reinforcement learning”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 4193–4206.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature* 521 (May 2015), 436 EP –.

[17] Long-Ji Lin. “Reinforcement Learning for Robots Using Neural Networks”. UMI Order No. GAX93-22750. PhD thesis. Pittsburgh, PA, USA, 1992.

[18] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. “Multi-agent actor-critic for mixed cooperative-competitive environments”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 6382–6393.

[19] Edward A. Macdonald. “Multi-Robot Assignment and Formation Control”. Masters of Science in the School of Electrical and Computer Engineering. Georgia Institute of Technology, Aug. 2011.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. “Human-level control through deep reinforcement learning”. In: *Nature* 518 (Feb. 2015), 529 EP –.

[21] Igor Mordatch and Pieter Abbeel. “Emergence of grounded compositional language in multi-agent populations”. In: *arXiv preprint arXiv:1703.04908* (2017).

[22] OpenAI. *More on Dota* 2. Aug. 2017. URL: `https://blog.openai.com/more-on-dota-2/`.

[23] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. “High-dimensional continuous control using generalized advantage estimation”. In: *arXiv preprint arXiv:1506.02438* (2015).

[24] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[25] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (Oct. 2017), pp. 354–.

[26] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.

[27] Kar han Tan. “Virtual structures for high-precision cooperative mobile robot control”. In: *Autonomous Robots* 4 (1997), pp. 387–403.

[28] Ming Tan. “Multiagent Reinforcement Learning: Independent vs. Cooperative Agents”. In: *: Proceedings of the Tenth International Conference on Machine Learning*. University of Massachusetts, 1993, pp. 330–337.

[29] Sebastian Thrun and A. Schwartz. “Issues in Using Function Approximation for Reinforcement Learning”. In: *Proceedings of the 1993 Connectionist Models Summer School*. Erlbaum Associates, 1993.

[30] Christopher Watkins. “Learning from Delayed Rewards”. PhD. King’s College, 1989.

[31] Christopher J. C. H. Watkins and Peter Dayan. “Q-learning”. In: *Machine Learning* 8.3 (1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698.

[32] Gerhard Weiss. *Multiagent Systems*. The MIT Press, 2013. ISBN: 0262018896, 9780262018890.