Sample-efficient multi-agent reinforcement learning using learned world models

Daniël Willemsen

Technische Universiteit Delft

TU Delft

Sample-efficient multi-agent reinforcement learning using learned world models

by

Daniël Willemsen

to obtain the degree of Master of Science at the Delft University of Technology, to be defended publicly on Tuesday January 28, 2021 at 13:00.

Project duration:	Februay 3, 2020 – January 28, 2021	
Thesis committee:	prof. dr. G.C.H.E. de Croon,	Control and Simulation, TU Delft, supervisor
	ir. M. Coppola,	Control and Simulation, TU Delft, supervisor
	dr. E. van Kampen,	Control and Simulation, TU Delft
	dr. O.A. Sharpanskykh,	Air Transport and Operations, TU Delft

TU Delft

Acknowledgements

I would like to thank Dr. Guido de Croon, for providing guidance and direction throughout the project, whilst also giving me the freedom to explore ideas I found particularly interesting. I would also like to thank Mario Coppola, for your valuable insights throughout the project. You were always available to provide me with your time, support and feedback, even when you were already finished with your PhD at Delft University of Technology. Finally, I would like to thank my family, friends and roommates for their unconditional support throughout the project, this thesis would not have been possible without you.

Abstract

Having a group of robots cooperate to achieve certain goals has important benefits compared to designing a single robot to achieve that same task. These benefits include robustness, scalability, and flexibility. Manually designing individual behaviours for each robot in such a group is a complex task and gets exceedingly more difficult when the desired system-level behaviour gets more complex. Multi-agent robotic systems could thus benefit from teaching themselves how to act. It might be possible to do this through learning in a simulator, but this results in a problem known as the reality gap: robots learn to exploit inaccuracies in a simulator that do not exist in the real world, causing diminished real-world performance. A solution to this would be to get multi-agent robotic systems to learn through interacting in the real world. Learning in the real world, however, can be expensive both in time as well as money. It is not uncommon for agents to require tens of thousands of trials to learn desirable behaviours through the use of reinforcement learning. This not only takes an enormous amount of time but also results in a high probability of robots damaging themselves through selection of poor actions. Multi-agent robotic systems could thus benefit from reinforcement learning algorithms that are able to learn behaviours in a small number of trials, a property known as sample efficiency.

This research investigates if learned world models can help to improve sample efficiency of multi-agent reinforcement learning. We present a novel multi-agent model-based reinforcement learning algorithm: *Multi-Agent Model-Based Policy Optimization (MAMBPO)*, utilizing the Centralized Learning for Decentralized Execution (CLDE) framework. The CLDE framework is a framework that allows a group of agents to act fully decentralized after training, a desirable property for many systems comprising of multiple robots. Learning happens centralized, which helps to stabilize training in multi-agent environments. Current CLDE algorithms such as Multi-Agent Soft Actor-Critic (MASAC) suffer from limited sample efficiency, often taking many thousands of trials before learning desirable behaviours. This makes these algorithms impractical for learning in real-world robotic tasks. MAMBPO utilizes a learned world model to improve sample efficiency compared to these model-free alternatives. Model-based reinforcement learning goes through two steps that form an iterative loop. First, a model of the world dynamics is learned using real-world experience. Then, this so called dynamics model is used to generate additional data on which the policies of the agents can be trained. This allows a larger number of gradient steps to be performed without overfitting. Model-free learning, in contrast, directly trains policies on the data sampled from the real-world experience. In single-agent domains, model-based learning has shown to result in improvements of up to an order of magnitude in terms of sample efficiency. In multi-agent learning, however, there is only very limited research on model-based learning. To the best of our knowledge, our MAMBPO algorithm is the first application of learning with a generative world model within the CLDE framework.

We empirically demonstrate on a variety of multi-agent domains that our algorithm provides an up to 3.7× improvement in terms of sample efficiency compared to state-of-the-art model-free approaches. With this, we demonstrate that model-based learning deserves more attention in multi-agent reinforcement learning and could prove to be an important technique in making real-life learning for multi-agent robotic systems possible.

Contents

Acknowledgements iii

Abstract v

I Scientific Paper 1

II Preliminary Report 17

1 Introduction 21

2 An Overview of Swarm Robotics 23
2.1 Motivation and Inspiration . . . 23
2.2 Swarm Robotics within Multi-agent systems . . . 24
2.2.1 Taxonomies of multi-agent robotics . . . 24
2.3 Swarm Design. . . . 25
2.3.1 Behavior-based design methods . . . 25
2.3.2 Automatic design methods. . . . 26
2.4 Experimental Swarm Robotics . . . 27

3 Reinforcement Learning for Markov Decision Processes 29
3.1 Markov Decision Processes . . . 29
3.1.1 Policies. . . . 30
3.1.2 Optimality and Value. . . . 30
3.2 Basic learning algorithms for MDPs . . . 32
3.2.1 Model-Based Dynamic Programming . . . 32
3.2.2 Model-Free Reinforcement Learning. . . . 34
3.3 Taxonomies for Reinforcement Learning . . . 34
3.3.1 On-Policy and Off-Policy . . . 36
3.3.2 Bootstrapping . . . 36
3.3.3 Sample Backups and Full Backups . . . 36
3.3.4 Model-Based Planning and Model-Free Learning . . . 36
3.3.5 Function Approximation . . . 37
3.3.6 Discrete and Continuous. . . . 37
3.3.7 Actors and Critics . . . 37

4 Deep Reinforcement Learning 39
4.1 Deep Neural Networks . . . 39
4.2 Value Function approaches: Deep-Q Networks and Improvements . . . 39
4.3 Policy Gradient Approaches . . . 41
4.3.1 On-Policy Actor-Critics: A2C, A3C, TRPO and PPO . . . 43
4.3.2 Policy gradients for continuous action spaces . . . 44
4.3.3 DPG, DDPG and D4PG: Deterministic Policy Gradients . . . 44
4.3.4 Soft Actor-Critic: Off-Policy Policy Gradient with Entropy Regularization . . . 45
4.4 Policy Optimization without Gradients . . . 45
4.5 Benchmarking Deep Reinforcement Learning . . . 46
4.6 What algorithm is the best? . . . 47

5 Reinforcement Learning with Partial Observability 49
5.1 Partially Observable Markov Decision Processes . . . 49
5.2 Consequences of partial observability . . . 50
5.3 Reinforcement Learning with Partial Observability . . . 50

6 Reinforcement learning with a learned world model 53
6.1 Dyna: planning and learning with real and simulated experience . . . 54
6.2 Planning with continuous action spaces . . . 55
6.3 Planning with discrete action spaces . . . 56

7 Multi-Agent Reinforcement Learning 57
7.1 Decentralized Partially Observable Markov Decision Processes . . . 57
7.1.1 Consequences of decentralization . . . 58
7.2 Reinforcement Learning Algorithms for Multi-Agent Learning . . . 58
7.2.1 Decentralized Learning for Decentralized Execution . . . 58
7.2.2 Centralized learning for decentralized execution . . . 59
7.3 Benchmarking Multi-Agent Reinforcement Learning . . . 60

8 Literature Synthesis 63
8.1 Swarm Robotics . . . 63
8.2 Reinforcement Learning . . . 63
8.3 Model-Based Reinforcement Learning . . . 64
8.4 Multi-Agent Reinforcement Learning . . . 64
8.5 Model-Based Learning for Decentralized Agents . . . 64

9 Preliminary Analysis of PageRank for Online Learning in Swarms 67
9.1 An Analysis of PageRank and its relationship to MDPs . . . 67
9.2 Experiments & Results . . . 70
9.3 PageRank Discussion & Conclusion . . . 71

10 Conclusion, Approach, Planning 73
10.1 Conclusion . . . 73
10.2 General Approach . . . 74
10.3 Tasks & Nominal Planning . . . 74
10.3.1 Content Tasks: . . . 75
10.3.2 Auxiliary Tasks:. . . . 75
10.4 Expected Difficulties and Risks . . . 76

Bibliography 77

I

Scientific Paper

Sample-efficient multi-agent reinforcement learning using learned world models

Daniël Willemsen*, Mario Coppola†, Guido de Croon†

*MSc student, Faculty of Aerospace Engineering, Delft University of Technology
†Supervisor, Faculty of Aerospace Engineering, Delft University of Technology

***Abstract*—Multi-agent robotic systems could benefit from reinforcement learning algorithms that are able to learn behaviours in a small number trials, a property known as sample efficiency. This research investigates the use of learned world models to create more sample-efficient algorithms. We present a novel multi-agent model-based reinforcement learning algorithm: *Multi-Agent Model-Based Policy Optimization (MAMBPO)*, utilizing the Centralized Learning for Decentralized Execution (CLDE) framework, and demonstrate state-of-the-art performance in terms of sample efficiency on a number of benchmark domains. CLDE algorithms allow a group of agents to act in a fully decentralized manner after training. This is a desirable property for many systems comprising of multiple robots. Current CLDE algorithms such as Multi-Agent Soft Actor-Critic (MASAC) suffer from limited sample efficiency, often taking many thousands of trials before learning desirable behaviours. This makes these algorithms impractical for learning in real-world robotic tasks. MAMBPO utilizes a learned world model to improve sample efficiency compared to its model-free counterparts. We demonstrate on two simulated multi-agent robotics tasks that MAMBPO is able to reach similar performance to MASAC with up to 3.7 times fewer samples required for learning. Doing this, we take an important step towards making real-life learning for multi-agent robotic systems possible.**

***Keywords*—Model-Based, Reinforcement Learning, Multi-Agent, Decentralized Control, Deep Learning.**

I. Introduction

Reinforcement learning has become a popular tool for many artificial intelligence tasks and has been applied in a large variety of domains, including board games [1, 2, 3], video games [4, 5] and (virtual) robotics [6, 7]. In most situations, reinforcement learning considers a single agent acting and learning in a non-changing environment. However, there are several important potential applications of reinforcement learning that involve multiple agents acting and cooperating in a single environment. For example smart electricity markets [8], drone swarming [9], or general multi-robot control [10]. Standard reinforcement learning algorithms are often incapable of handling learning in multi-agent environments. One cause of this is the non-stationarity of

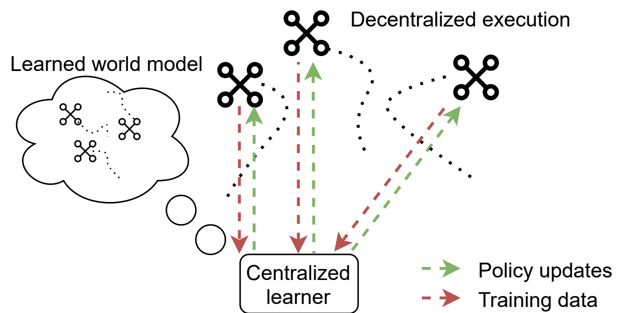


Fig. 1: Visual interpretation of how the proposed sample-efficient MAMBPO algorithm could be applied in a real-life scenario. Drones are able to act fully decentralized, sharing their experiences with a centralized learner when possible. The centralized learner learns a world model and uses this world model to imagine additional training data. This data is then used to train policies for the robots, that periodically receive updated policies.

the environment from the perspective of each agent [11]. This non-stationarity is especially problematic in deep reinforcement learning algorithms where replay buffers are used to learn from older data that is no longer representative for the current environment dynamics [12].

When designing deep reinforcement learning algorithms specifically for multi-agent decentralized control, two main approaches exist: that of *Independent Learners (IL)* and that of *Centralized Learning for Decentralized Execution (CLDE)*. In IL algorithms, each agent learns independently from the other agents and specialized techniques are used to reduce the instability caused by the non-stationarity of the environment. These techniques often reduce the size of the replay buffer or disable the buffer altogether [13], resulting in poor sample efficiency and stability of neural networks. Recent work does explicitly address the non-stationarity concerns, without disabling the replay buffer, but this approach has not yet been applied in domains with continuous action spaces, which are common for robotics tasks

[12]. The CLDE framework is more mature and has already resulted in a number of algorithms that are tested on domains with continuous action spaces [14, 15]. The CLDE framework allows centralized information to be used during training to improve the stability and performance of the algorithms. Then, during execution, no centralized information is required and all agents are able to act fully decentralized, a desirable characteristic in many multi-agent systems, such as robotic swarms [16]. Although these algorithms have been successful at a variety of tasks, such as cooperative navigation and predator-prey [17, 18, 19], their sample efficiency is still limited: they often require over 10 000 trials to solve relatively simple tasks.

Sample efficiency is an important performance measure for tasks where agents need to act in a real environment, such as robotics. Training agents through acting in the real world can be prohibitively expensive due to the large amount of time required, as well as the possibility of breaking hardware from trial and error of the algorithm [20]. Training in a simulator might be possible, but introduces the so-called reality-gap: a phenomenon where performance in a simulator does not fully translate to performance in the real world, due to simulator imperfections. Therefore, there is a need for algorithms that are able to train agents in a sample-efficient manner.

In single-agent reinforcement learning, model-based learning has recently shown to yield large improvements in sample efficiency, for example in virtual robotics tasks [21, 22]. Rather than directly optimizing the policy of agents, these algorithms first learn a model of the world dynamics. This is often easier to do as the world model is not dependent on an ever-changing policy, in contrast to, for example, a critic network whose target value depends on the current policy being followed. This world model is then used to generate additional training data for the policy, speeding up the learning process.

This research investigates the possibility of using model-based learning to improve sample efficiency of multi-agent reinforcement learning through the CLDE framework. To do this, we introduce a novel model-based multi-agent reinforcement learning algorithm: *Multi-Agent Model-Based Policy Optimization (MAMBPO)*, which is a multi-agent adaptation of the Model-Based Policy Optimization (MBPO) algorithm [22]. We demonstrate on two multi-agent domains that our algorithm provides an up to 3.7 times improvement in terms of sample efficiency compared to state-of-the-art model-free approaches. To the best of our knowledge, this algorithm is the first application of learning with a generative world model within the CLDE framework. A high-level visual interpretation of our algorithm can be found in Figure 1.

The remainder of this work is structured as follows. In Section II we discuss related literature and establish the novelty of our approach. Then, in Section III, we introduce our proposed MAMBPO algorithm, and show how it combines the strengths of CLDE learning and model-based reinforcement learning. In Section IV, we empirically demonstrate the performance of our algorithm in two benchmark domains: cooperative navigation and cooperative predator-prey. In Section V, we discuss our results, and reflect upon the advantages and limitations of the approach. Finally, in Section VI, we conclude the article with an emphasis on future work to be done to reach real-life learning for multi-agent robotic systems.

II. Related Work

Literature closely related to our research can be broadly divided into three categories: research into CLDE algorithms for continuous action spaces, research into single-agent model-based reinforcement learning and research into multi-agent model-based learning. We briefly discuss recent work in these three categories in this section.

In the past few years, a significant number of CLDE algorithms for continuous action spaces have been developed. Multi-Agent Deep Deterministic Policy Gradient (MADDPG) is a CLDE adaptation of the single-agent DDPG algorithm [17]. This adaptation, suitable for cooperative, competitive, and mixed domains, utilizes centralized critics that take the concatenated joint observations and actions of all agents as an input and outputs the estimated value for a single agent. The actors are decentralized, taking only the agents' own observations as an input and outputting a deterministic action. In another work, researchers propose two alternatives to MADDPG, one of which is called Multi-Actor-Attention-Critic (MAAC) [18]. They use an attention mechanism to improve the scalability of CLDE algorithms towards greater numbers of agents. The other alternative to MADDPG is a maximum entropy variant called Multi-Agent Soft Actor-Critic (MASAC). In maximum entropy reinforcement learning, agents have an incentive to maximize exploration within the policy. This can increase both stability as well as sample efficiency [18, 19]. MASAC serves as our model-free baseline and is used for policy optimization in our model-based algorithm.

Model-based learning has recently shown great sample-efficiency improvements in single-agent domains. Early model-based reinforcement learning approaches suffer from poor asymptotic performance. This is caused by a phenomenon known as model bias, where agents exploit inaccuracies in a model, resulting in suboptimal policies in the real environment. Recent work utilizes models that explicitly and simultaneously

take into account aleatoric and epistemic uncertainty, for example through the use of ensembles of stochastic neural network models [21]. This reduces the impact of model bias. These types of models have resulted in algorithms that are comparable to state-of-the-art model-free algorithms in terms of asymptotic performance, yet are able to learn with up to an order of magnitude greater sample efficiency [22].

Even though CLDE learning as well as single-agent model-based learning have recently been active fields of study, there is only limited work performed on combining the two. One work that utilizes model-based learning within the context of multi-agent learning is that of Krupnik et al.[23]. Their algorithm, however, performs Model-Predictive Control based on the learned world model. This requires the usage of the model during execution, which in turn requires centralized information. Their approach is therefore not suitable for decentralized execution. Other research investigates the possibility of utilizing a model to not only predict the environment but also predict the policies of opposing agents in 2-agent competitive domains [24]. This approach, however, is only evaluated on a task where the opponent agent has a fixed policy, rather than being an opponent that is learning concurrent to the agent itself. This removes the problems of non-stationarity from the domain. Hence, to the best of our knowledge, this work is the first to investigate model-based CLDE with all agents' policies being dynamic and decentrally executable.

III. Methodology

In this section, we introduce the Multi-Agent Model-Based Policy Optimization algorithm. First, in Section III-A, we introduce the Decentralized Partially Observable Markov Decision Process (Dec-POMDP) as a framework that describes the problem of cooperative multi-agent reinforcement learning. Then, in Section III-B, we introduce our novel MAMBPO algorithm and discuss its two main components, policy optimization through Multi-Agent Soft Actor-Critic (MASAC) and model-based learning through Model-Based Policy Optimization (MBPO) in more details.

A. Problem Description & Notation

In this research, we consider the problem of a group of agents interacting cooperatively within an environment, formalized in the Decentralized Partially Observable Markov Decision Process (Dec-POMDP) framework [25]. This framework is illustrated in Figure 2. At every timestep t, every agent $i \in \{1, ..., n\}$ simultaneously receives an individual observation o_t^i from a joint observation $\boldsymbol{o}_t := \left\{o_t^i\right\}_{i=1}^n$. An action a_t^i is then sampled from a (stochastic) policy (π^i), such that $a_t^i \sim \pi^i(\cdot|o_t^i)$. We use the $\pi^i(\cdot|o_t^i)$ notation to indicate the full probability distribution of π^i and we use the notation $\pi^i(a_t^i|o_t^i)$ to indicate the probability density function at a particular value a_t^i. The concatenation of the actions from all agents is the joint action, denoted $\boldsymbol{a}_t := \left\{a_t^i\right\}_{i=1}^n$. Note that we use superscripts ^{i} to indicate individual observations (o_t^i) and actions (a_t^i) rather than the joint observation ($\boldsymbol{o}_t$) and action ($\boldsymbol{a}_t$), which are indicated in bold and without a superscript. Similarly, we use $\boldsymbol{\pi}$ to indicate the joint policy of all agents. The environment takes the joint action and a hidden environment state s_t to (stochastically) produce a new hidden state through sampling from the state-transition function P, such that $s_{t+1} \sim P(\cdot|s_t, \boldsymbol{a}_t)$. The environment then produces a joint observation and reward through the observation function O and reward function R respectively[1]: $\boldsymbol{o}_{t+1} \sim O(s_{t+1})$, $r_{t+1} \sim R(s_{t+1})$. The goal for each agent is to find policy π^i that maximizes the expected return, which is the expected value of the cumulative reward, discounted with some factor γ over time until some terminal condition is reached: $\mathbb{E}[G_k] = \mathbb{E}[\sum_{t=k}^{t=t_{terminal}} \gamma^{k-t} r_{t+1}]$. The state-action value Q is a measure for the expected return of an agent given a certain starting state s, joint policy π and initial joint action $\boldsymbol{a}$: $Q^{\pi}(s, \boldsymbol{a}) = \mathbb{E}_{\pi}\left[\sum_{t=k}^{t=t_{terminal}} \gamma^{k-t} r_{t+1} \middle| s_k = s, \boldsymbol{a}_k = \boldsymbol{a}\right]$.

A Dec-POMDP can either have finite sets of states, actions, and observations, or it can have continuous state-, action-, and observation-spaces. In this article, we consider Dec-POMDPs of the continuous type.

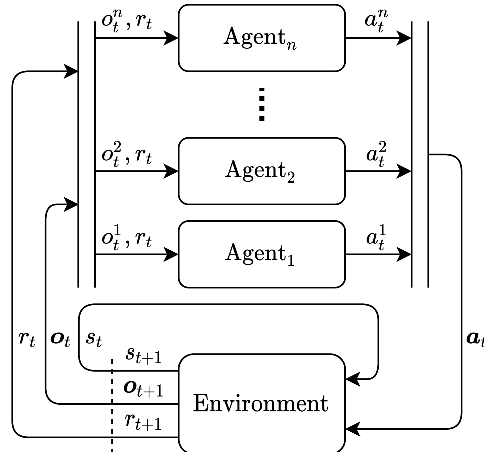


Fig. 2: Description of the interaction between the agents and the environment in a Dec-POMDP.

[1]We define the reward r_{t+1} as the reward received after taking action $\boldsymbol{a}_t$ and is received by the agents simultaneously with $\boldsymbol{o}_{t+1}$. This notation differs from some literature that would denote this reward with r_t as it is the reward that is a direct consequence of the action taken on time step t.

B. Multi-Agent Model-Based Policy Optimization

Our novel MAMBPO algorithm is a combination of model-based reinforcement learning and Centralized Learning for Decentralized Execution (CLDE). For model-based reinforcement learning, we use MBPO. The core idea is that it is easier for an agent to learn a model of the world than it is to optimize a policy. Therefore, MBPO learns a world model and then uses this world model to generate additional experience on which the agent's policy can be trained. In our case, we train this policy through MASAC, a Centralized Learning for Decentralized Execution variant of Soft Actor-Critic. This two-step process of model learning and policy optimization allows for improved sample efficiency compared to directly optimizing a policy. In the following three paragraphs, we explain the components in more detail. Section III-Ba introduces MASAC and Section III-Bb introduces MBPO. Finally, MAMBPO is then introduced in Section III-Bc.

a) Policy Optimization through MASAC: MASAC is a CLDE algorithm that uses an Actor-Critic architecture that includes an entropy maximization term to ensure sufficient exploration within a policy.

CLDE is an approach to circumvent problems associated with learning in multi-agent environments, such as an Dec-POMDP. In these environments, agents typically suffer from an apparent non-stationarity of the environment: from the perspective of a single agent, the other agents are part of the environment. However, when multiple agents are updating their policies simultaneously, it appears to each agent individually that the environment they are acting in is changing. The resulting difficulty can be illustrated intuitively. Reinforcement learning algorithms often use some estimation of state-values to improve their policies. When the environment changes, so should the values/performance measure for the given policy. As a result, the optimal policy changes. When the agent adjusts its policy based on this change in the environment, the environment has changed again, from the perspective of another agent, resulting in that other agent having to adapt its policy, resulting in a loop of changing environments and changing policies. The way CLDE avoids these problems is through the use of a centralized learning scheme. The idea behind centralized learning is to take into account global information from all agents (and all agents' policies) during learning, removing the non-stationarity of the environment, and thus stabilizing learning. When learning has finished, the agents themselves are able to act in a decentralized fashion, removing the need for centralized information during execution.

One way to achieve CLDE is through the use of an Actor-Critic architecture. Such an architecture splits up the agent into two separate components: an actor that performs action selection given an observation and a critic that serves as an estimator for the return given a selected joint observation and action. The critic for an agent i is thus a function of the joint observation and joint action of all agents and serves as an estimator for the expected return: $\hat{Q}_{w^i}(\boldsymbol{o}_t, \boldsymbol{a}_t) \approx Q^\pi(s_t, \boldsymbol{a}_t)$. This critic, through taking the joint action of all agents as an input, removes the non-stationarity of the environment. For this approximation to be a good estimator, the joint observation $\boldsymbol{o}_t$ must contain similar information to the full system state s_t. The actor of agent i stochastically selects an action given its individual observation: $a_t^i = \pi_{\phi^i}(\cdot|o_t^i)$. In this article, we consider both the actor and the critic to be neural networks, with weights described by the parameter vectors ϕ^i and w^i respectively.

Algorithm 1: Multi-Agent Soft Actor-Critic

```
1  Initialize actors π_{φ^i}, critics Q_{w^i}, for each agent i
2  Initialize environment replay buffer B_env
3  Initialize environment
4  for every timestep t do
5      Select joint action using π_{φ^i} for each agent i
6      Apply joint action in environment
7      Add transition to B_env
8      for G gradient updates do
9          for every agent i do
10             Update actor parameters φ^i using B_env
11             Update critic parameters w^i using B_env
12         end
13     end
14 end
15 return
```

A high level overview of a complete Multi-Agent Actor-Critic algorithm can be found in Algorithm 1. When training, the agents repeatedly perform training episodes in the multi-agent environment using their policies. During the training episodes, each transition tuple $(\boldsymbol{o}_t, \boldsymbol{a}_t, r_{t+1}, \boldsymbol{o}_{t+1})$ is stored in a replay buffer $\mathcal{B}_{env}$. After every step in the environment, the actors and critics are trained through taking one step of gradient descent using a batch of sampled transitions from the replay buffer. We use a Multi-Agent Actor-Critic version called MASAC that maximizes the entropy of the policy simultaneously to maximizing the expected return. This encourages exploration, stabilizing training and improving sample efficiency [19, 18]. The critic is trained through temporal difference learning, where the

target for the critic is set to the following:

$$y = r_{t+1} + \gamma \left(\hat{Q}_{w^i} \left(\boldsymbol{o}_{t+1}, \boldsymbol{a}_{t+1} \right) - \alpha \log \pi_{\phi^i}(a^i_{t+1} | o^i_{t+1}) \right), \quad \boldsymbol{a}_{t+1} \sim \boldsymbol{\pi}(\cdot | \boldsymbol{o}_{t+1}) \tag{1}$$

Without the log term, this function is equivalent to standard temporal difference learning used in reinforcement learning. The log term is an additional term used in (MA) Soft Actor-Critic and can be seen as a term that rewards policies that have a high entropy. Here, α is a weight that determines the level of entropy regularization. In our implementation, we automatically tune this weight during training, through tracking a reference entropy of the policy. The loss function for the critic is set to be the mean squared error of this target with the current Q-value estimator:

$$\mathcal{L}_{Q^i} \left(w^i, \mathcal{B} \right) = \left(\hat{Q}_{w^i} \left(\boldsymbol{o}_t, \boldsymbol{a}_t \right) - y \right)^2, \quad (\boldsymbol{o}_t, \boldsymbol{a}_t, r_{t+1}, \boldsymbol{o}_{t+1}) \sim \mathcal{B}, \quad \boldsymbol{a}_{t+1} \sim \boldsymbol{\pi}(\cdot | \boldsymbol{o}_t) \tag{2}$$

For the actor, the goal is to maximize the expected return and entropy of the agent's policy. The policy is thus updated through the policy gradient with the following loss function:

$$\mathcal{L}_{\pi^i} \left(\phi^i, \mathcal{B} \right) = - \left(\hat{Q}_{w^i}(\boldsymbol{o}_t, \boldsymbol{a}_t) - \alpha \log \pi_{\phi^i}(a^i_t | o^i_t) \right), \quad (\boldsymbol{o}_t) \sim \mathcal{B}, \quad \boldsymbol{a}_t \sim \boldsymbol{\pi}(\cdot | \boldsymbol{o}_t) \tag{3}$$

Within CLDE learning, an important distinction that needs to be made is that between *centralization of networks* and *parameter sharing of networks*. Whereas we define centralization as the use of global information when calling the network, we define parameter sharing of networks as the sharing of network parameters amongst different agents. In other words, when a network is shared, each agent has a network that outputs the same value given the same input. Although CLDE in MASAC requires the use of a centralized critic and a decentralized actor, CLDE does not require to have only one, shared, critic for the whole system and one, individual, actor for each agent. Rather, the designer is free to choose whether or not these network parameters are shared amongst agents. This does, however, have some important consequences in the types of behaviours that can be learned. When sharing the critic parameters between all agents, each agent will have an identical value estimate given a certain environment state. This can be a problem when the objectives of every agent are not fully aligned, i.e. a task that is not fully co-operative. The benefit of having only a single, shared, critic is the reduced amount of computation associated with training the critics. When utilizing an actor whose parameters are shared amongst all agents, every agent will act the same given a similar observation. In other words, all agents will be homogeneous. This will inhibit agents from learning specializations, which might be a disadvantage on certain tasks. In most commonly used implementations of CLDE actor-critics, neither the actor nor the critic is shared [18, 19]. This results in algorithms that are suitable for heterogeneous agents and objectives. To keep as close to the original implementations of MASAC as possible, we also utilize individual actors and critics for our implementations.

Algorithm 2: Model-Based Policy Optimization, adapted from [22].

```
1  Initialize actor π_φ, critic Q_w, world model p̂_θ,
     environment replay buffer B_env, model replay
     buffer B_model
2  Initialize environment
3  for every timestep t do
4      Take action in environment using π_φ
5      Add transition to B_env
6      if t % model train frequency = 0 then
7          Train model p̂_θ on B_env
8      end
9      for M model rollouts do
10         Sample s from B_env
11         Perform k-step rollout starting from s
             using π_φ and p̂_θ
12         Add transitions to B_model
13     end
14     for G gradient updates do
15         Update actor parameters φ using B_model
16         Update critic parameters w using B_model
17     end
18 end
19 return
```

MASAC can be used as a CLDE algorithm on its own, and this serves as our model-free baseline in our experiments. Our model-based MAMBPO algorithm uses the training targets and exploration of MASAC. The training data, however, is sampled from a replay buffer with more diverse data generated through our generative world model. This world model is trained using MBPO, as explained in the next paragraph.

b) Model-Based Learning through MBPO: MBPO is a single-agent model-based reinforcement learning algorithm for continuous action spaces that utilizes an ensemble of neural networks that serves as a simulator to generate additional experience for the actor and critic [22]. An overview of the algorithm can be found in Algorithm 2. In our MAMBPO algorithm, we use the model-learning architecture from MBPO, modified to suit multi-agent cooperative environments. The modi-

Algorithm 3: Multi-Agent Model-Based Policy Optimization

```
1  Initialize actors π_{φ^i}, critics Q_{w^i}, for each agent i
2  Initialize world model p̂_θ
3  Initialize environment replay buffer B_env, model
     replay buffer B_model
4  Initialize environment
5  for every timestep t do
6  |  Select joint action using π_{φ^i} for each agent i
7  |  Apply joint action in environment using
8  |  Add transition to B_env
9  |  if t % model train frequency = 0 then
10 |  |  Train model p̂_θ on B_env
11 |  end
12 |  for M model rollouts do
13 |  |  Sample s from B_env
14 |  |  Perform k-step rollout starting from s
   |  |    using π_{φ^i} for each agent i and p̂_θ
15 |  |  Add transitions to B_model
16 |  end
17 |  for G gradient updates do
18 |  |  for every agent i do
19 |  |  |  Update actor parameters φ^i using
   |  |  |    B_model
20 |  |  |  Update critic parameters w^i using
   |  |  |    B_model
21 |  |  end
22 |  end
23 end
24 return
```

fication that we do is that that we use a centralized model ($\hat{p}_\theta$) that predicts the reward and observations for the next time step given the joint observation of our current time step and the joint action of all agents: $\boldsymbol{o}_{t+1}, r_{t+1} \sim \hat{p}_\theta(\boldsymbol{o}_t, \boldsymbol{a}_t)$. The original MBPO implementation, on the other hand, only uses a single-agent observation and action as inputs. This is because it is a single-agent algorithm. The remainder of the model architecture, learning, and usage is performed similar to the original implementation, but briefly summarized here for comprehensiveness. Model bias is the phenomenon where inaccuracies in the model result in policies that exploit these inaccuracies, resulting in reduced performance of the agent in the real environment. To prevent model bias, MBPO explicitly incorporates the two types of uncertainty in the model: *aleatoric* and *epistemic* uncertainty. The aleatoric uncertainty is captured through the use of networks that predict not only the next observations and reward but predicts a distribution of these that can be sampled from. The network has means and standard deviations for each observation and reward as an output, together parameterizing a normal distribution that can be sampled from. The epistemic uncertainty, or the uncertainty in the parameters of the model, is captured through the use of an ensemble of models, from which is uniformly sampled when performing rollouts. The network is trained on the replay buffer as a maximum likelihood estimator for the next observations and reward. To ensure well-behaved neural networks, we bound the variance outputs to a fixed range, similar to how this is done in the original MBPO implementation [22] and its precursor [21].

c) The full algorithm: Combining MASAC and MBPO results in the MAMBPO algorithm shown in Algorithm 3. Agents interact with the environment and a centralized replay buffer ($\mathcal{B}_{env}$) is used to store the transitions. Periodically, a centralized world model ($\hat{p}_\theta$) is trained on this replay buffer. This model is then used to generate additional samples, stored in a separate replay buffer ($\mathcal{B}_{model}$). We supplement this second replay buffer with a small amount (in our case 10%) of real environment data, similar to the original MBPO implementation [22]. The model training and usage through MBPO is visible in line 10 through line 15 of Algorithm 3. The second replay buffer, $\mathcal{B}_{model}$, is used to train the actor and critic networks through MASAC, as shown in line 19 through line 20. The key insight that allows this algorithm to perform sample-efficient learning is that the larger diversity in the replay buffer from generating rollouts through the model makes it possible to increase the number of gradient updates G, without overfitting the actors or critics.

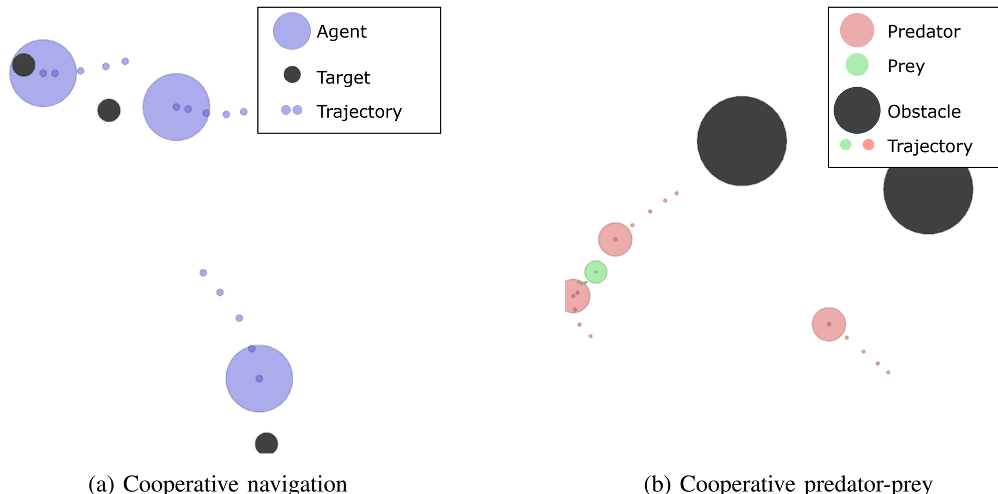
IV. Experiments & Results

In this section, we empirically demonstrate the efficiency of our algorithm on two benchmark domains: cooperative navigation and cooperative predator-prey. We discuss the details of these environments in Section IV-A. Then, we discuss the hyperparameter settings of our algorithms in Section IV-B. Finally, we show the results on both domains in Section IV-C.

A. Environment Descriptions

We test our algorithms on two benchmark domains from the multi-agent particle environments benchmark suite [26, 17]. This suite considers agents that can move around in a 2-dimensional space. We utilize a modified version of the suite that allows the use of continuous actions for each agent: agents output a desired acceleration in the x- and y-direction. This version is created by de Witt et al [15] [2]. Each episode consists of 25 time steps.

[2]The specific implementations of the environments can be found at https://github.com/schroederdewitt/multiagent-particle-envs. Here, cooperative navigation can be found as simple_spread and predator-prey can be found as simple_tag_coop



(a) Cooperative navigation

(b) Cooperative predator-prey

Fig. 3: Illustrations of the two benchmark domains.

a) Cooperative navigation: In the cooperative navigation domain, agents need to simultaneously cover three target locations whilst avoiding collisions with each other. Both the agents and targets are spawned at random locations at the start of each episode. The agents receive a cooperative reward based on the distance between each target and the closest agent to that target. The agents perceive the relative locations of all other agents as well as those of all targets. An illustration of the domain is shown in Figure 3a. This task is interesting from a multi-agent perspective as it requires coordination without communication to determine which agent should cover which target.

b) Cooperative predator-prey: In the cooperative predator-prey domain, three collaborating agents need to catch a single prey. This domain is a modification of the simple-tag domain of the multi-agent particle environments benchmark environment suite. The prey policy is replaced with a fixed heuristic to move away from predators when predators are close. The predators receive a reward for every time step at least one predator is on top of the prey. An illustration of the domain is shown in Figure 3b. Catching the prey requires less coordination amongst agents compared to the cooperative navigation domain. This domain, however, is of interest because it contains sparse rewards, which can complicate learning.

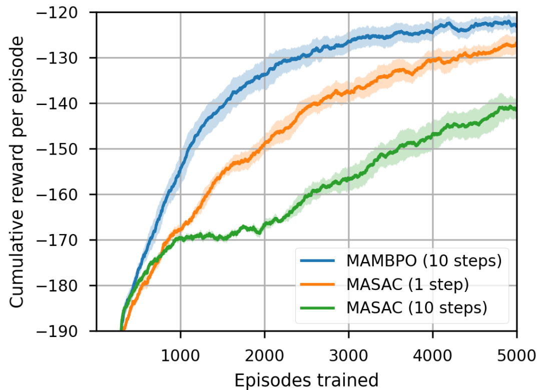
B. Hyperparameters and Experiment setup

In each of our experiments, we compare our model-based learner with its model-free counterpart. Since our interest lies in determining the benefits of model-based learning compared to model-free learning, we keep the hyperparameters constant between both environments and both algorithms to make the comparison between both algorithms as fair as possible. In our implementation, both algorithms are equivalent and utilize the same code with the exception of a switch that determines whether or not to train the model $\hat{p}_\theta$ and whether or not to use $\mathcal{B}_{model}$ for training the actors and critics[3]. The only hyperparameter that is varied between the two algorithms is the number of gradient steps performed per environment step, as we hypothesize that we can set this to a higher value to benefit from the larger diversity in the replay buffer in the case of model-based learning. For a fair comparison, we perform the experiments of the model-free learning algorithm using both the higher number of gradient steps (10) as well as the smaller number of gradient steps (1). Our low value for gradient steps is the standard value used in single-agent SAC algorithms, but it is still significantly higher than the values used in previous research on these multi-agent domains, as we found this to improve performance for both algorithms. In MAMBPO, we set the model rollout length to 1 time step, as this was found to work reasonably well in the original MBPO implementation and does not require any additional tuning of the parameter. The full hyperparameter settings can be found, for reference, in Appendix A.

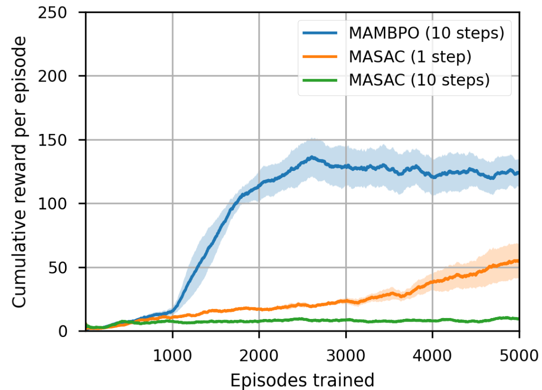
C. Results

In this subsection we show the performance of our full algorithm compared to the baseline model-free MASAC algorithm on the two benchmark domains. We do this through analysing both the cumulative rewards as well as a measure of successful task completion. In addition, we analyse the quality of the learned world model.

[3]The code used for all experiments is available at https://github.com/danielwillemsen/mambpo.



(a) Cooperative navigation. The average cumulative reward per episode of MASAC (1 step) after 5000 training episodes is reached by MAMBPO after only 2980 episodes. This corresponds to about a 1.7 times improvement in sample efficiency. An upper limit for the maximum reachable reward is -75, which is reached when all targets are covered for all 25 time steps without any collisions. This is not practically reachable as the agents need to travel to their targets whilst avoiding each other.



(b) Cooperative predator-prey. The average cumulative reward per episode of MASAC (1 step) after 5000 training episodes is reached by MAMBPO after only 1340 episodes. This corresponds to about a 3.7 times improvement in sample efficiency. An upper limit for the maximum reachable reward is 750, which is reached when the prey is covered by all three agents for all 25 time steps. This is practically not reachable as the predators need to travel to a prey that is actively avoiding them.

Fig. 4: Comparison between the performance of MAMBPO and MASAC on the two benchmark tasks. The number of steps indicates the number of gradient steps performed per real environment step. The results are smoothed using a moving average filter with a filter size of 200 episodes. The bold lines and shaded areas indicate the mean and standard error of the mean over 5 independent runs respectively.

Episodes trained for	Chance of success	
	MASAC	MAMBPO
1250	0.3%	3.2%
2500	3.8%	20.1%
5000	25.9%	37.1%

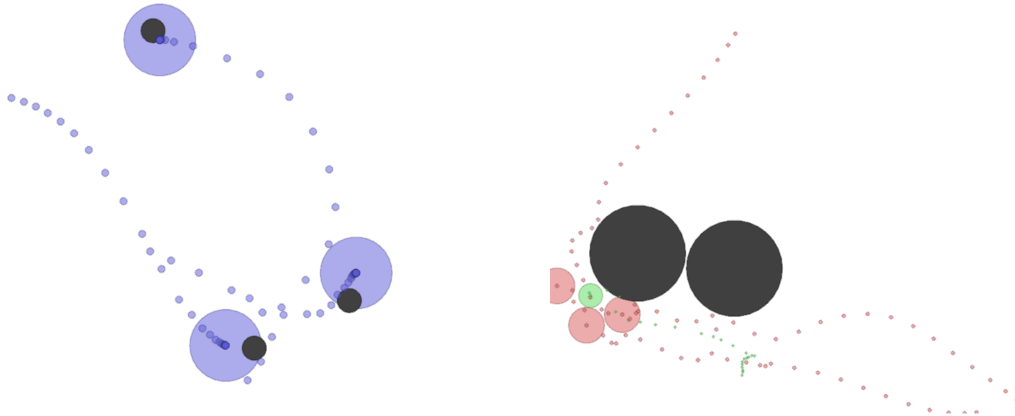
TABLE I: Comparison of the chance of covering all three targets for at least 1 timestep, without any collisions occurring during the episode. Evaluated over 5 independent runs and 250 evaluation episodes per run.

Episodes trained for	Chance of catching prey	
	MASAC	MAMBPO
1250	43.5%	72.1%
2500	55.0%	97.8%
5000	76.3%	98.8%

TABLE II: Comparison of the chance of covering the prey by at least one predator for at least 1 timestep. Evaluated over 5 independent runs and 250 evaluation episodes per run.

a) Rewards on benchmark domains: The results of testing our algorithm on the two test domains and comparing it against its model-free counterpart can be seen in Figure 4. We find that the MAMBPO algorithm results in higher rewards for a given number of episodes, and requires between $1.7\times$ (cooperative navigation) and $3.7\times$ (predator-prey) fewer episodes to reach a desired benchmark score compared to the standard (1 step) MASAC algorithm. Increasing the number of gradient steps performed in MASAC does not improve performance or, in the case of predator-prey, prohibits learning altogether.

b) Success rates on benchmark domains: For the cooperative navigation task, we define an episode to be successful if all three targets are simultaneously covered for at least 1 time step without any collisions occurring during the episode. We show the success rates at different stages of learning in Table I. Although the success rates for both algorithms remain relatively modest (at most 37.1%), we do see improvements of MAMBPO compared to MASAC of 16.3 percentage points halfway during training and 11.2 percentage points after finishing the training. Of the unsuccessful episodes still occurring after 5000 episodes of training, 39% is caused by collisions, 36% are caused by not covering all three targets and 25% fail due to both colliding as well as failing to cover all targets. When agents fail to cover all three targets, the agents often do come close to all three targets, but not close enough to fully cover them. The large amount of collisions might be caused by the relatively low negative reward (-2) associated with them, which might result an incentive for agents to go through



(a) Cooperative navigation. In this episode, the task performance is regarded as a failure as there occurs a collision in the 4th time step between the agent that ends up at the top and the one that ends up at the right.

(b) Cooperative predator-prey. In this episode, the agents manage to succeed in their task as there is at least one timestep where one predator is in contact with the prey.

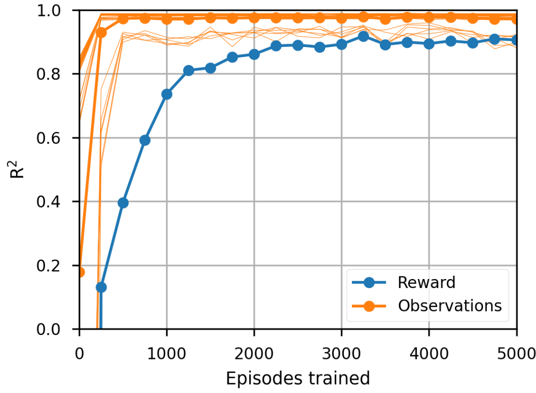
Fig. 5: Sample trajectories of learned behaviours of the MAMBPO agents after 5000 episodes of training in the two benchmark domains.

each other to more quickly reach their targets. For the predator-prey task, we look at the chance of covering the prey by at least one predator for at least one time step during an episode, counting this as "catching the prey". These results can be found in Table II. In this domain, the success rates are higher. The performance of MAMBPO agents trained for only 1250 episodes nearly equals the performance of MASAC agents trained for 5000 episodes, and reach a success rate of 98.8% when fully trained (compared to 76.3% for the MASAC agents). Sample trajectories of the learned behaviours can be found in Figure 5.

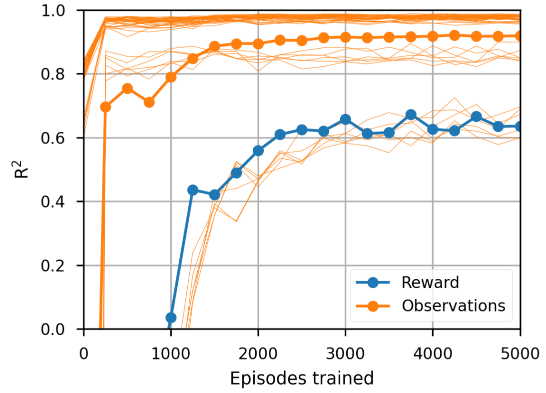
c) Model quality: The world models are periodically evaluated on their reward- and observation predictive capabilities. We do this by generating a number of trajectories in the environment using the current actor and measure the coefficient of determination (R^2) between the single-step predictions of the model and the true single-step rewards and observations. These results are shown in Figure 6. We find that, in both domains, after approximately 2500 episodes of training the model does not appear to improve any further. Also, it is visible that there are still large inaccuracies in the model, even after training for 5000 episodes. The model performs worse in predicting observations of velocities compared to positions. The velocity of the prey in the predator-prey domain appears to be the most difficult to predict. This is likely because the actions that the prey takes are not directly visible to the model. Rather, the prey uses a fixed heuristic to move around which is a part of the environment and only the relative velocity of the prey can be observed by the agents themselves. We also find that the model has more difficulty in predicting the reward of the predator-prey domain, compared to the navigation domain. This might be caused by the sparseness and highly non-smooth nature of the rewards in the predator-prey domain.

To see whether the inaccuracies in the models are hampering the performance of MAMBPO agents, we overlay the performance of agents with the quality of the model during training in Figure 7. In the cooperative navigation domain, agents are able to continue improving until the end of training. Indicating that the quality of the model is not bottlenecking the final performance of the agents. In the predator-prey domain, however, it appears that agents stop improving at a similar point in training as when the model quality stops improving. This might indicate that model-quality is limiting the final performance of agents. To check what the asymptotic performance of agents can be without limitations in model quality, we perform a number of longer training runs using the model-free MASAC algorithm on the predator-prey task. In these training runs, we find that agents are still only able to reach an average cumulative reward of 115 after 10000 episodes of training. This might indicate that it is not the model quality that is limiting performance, but rather the policy optimization algorithm itself or the difficulty of the task.

Another important aspect to look at with regards to the model quality is model bias. As mentioned in the methodology, ensembles of networks are used to reduce the amount of model bias in our algorithm. When there

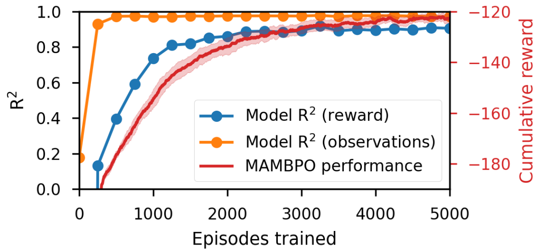


(a) Cooperative navigation. The model reaches a $R^2 > 0.95$ for all observations with the exception of the 6 observations that correspond to velocities of agents.

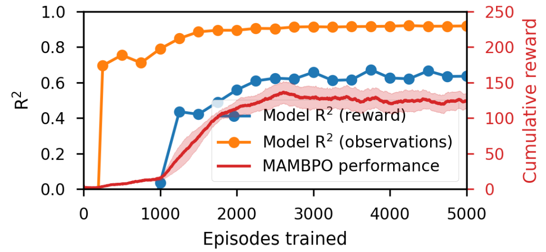


(b) Cooperative predator-prey. The model reaches a $R^2 > 0.95$ for all observations with the exception of the 6 observations that correspond to velocities of predators ($R^2 \approx 0.85$) and the 6 observations corresponding to velocity of the prey ($R^2 \approx 0.60$).

Fig. 6: Analysis of the model quality during training. We compute the R^2 value of one-step model predicted observations and rewards compared to the true observations and rewards received. The model predicts a joint observation and reward consisting of 54 (navigation) or 48 (predator-prey) observation variables and 1 reward variable. Each thin line corresponds to a single observation variable. The thick line indicates the mean R^2 of all observations. The results are obtained over 5 independent training runs and 250 evaluation episodes per run.



(a) Cooperative navigation. Even after around 2500 episodes, when the model appears to not improve significantly further, the agents still continue to learn. This might indicate the inaccuracies in the model are not bottlenecking the training.



(b) Cooperative predator-prey. The agent performance gains stops at a similar point as when the model quality stops improving. This might indicate that inaccuracies of the model are limiting the final performance of agents.

Fig. 7: Overlay of model quality and agent performance. We compare the R^2 value of the predicted rewards and observations, as displayed already in Figure 6, with the performance of the MAMBPO agents, as already seen in Figure 4.

is a model bias, agents learn to exploit inaccuracies in the model which results in the agents underperforming in the real task compared to their performance in the model. This bias can be made visible through comparison of the expected rewards of model predictions with real environment rewards. We show these results in Table III. We see that the differences in rewards are limited and we observe no substantial evidence of model bias using 1250 samples.

	Average Reward ($\pm 2\times$ standard error of the mean)		
Environment	Real	Predicted	Bias
Navigation	-4.95 ($\pm$ 0.11)	-4.94 ($\pm$ 0.11)	- 0.13%
Predator-prey	5.43 ($\pm$ 0.34)	5.51 ($\pm$ 0.29)	+ 1.39%

TABLE III: Bias of single-step model reward predictions for models trained for 5000 episodes. Each value was calculated based on 5 independent training runs and transitions were sampled from 250 evaluation episodes per training run.

V. Discussion

The primary goal of this research is to contribute towards making online learning for real-life robotic swarms possible. We do this through the proposal of a novel multi-agent reinforcement learning algorithm that utilizes model-based learning to improve sample efficiency within the centralized learning for decentralized execution framework. In this section, we reflect upon the contribution we make towards this primary goal.

Our experiments compared model-based multi-agent reinforcement learning with its model-free counterpart. The results indicated that, similar to single-agent reinforcement learning, model-based learning is able to improve the sample efficiency. The amount of improvement in sample efficiency varies per domain. This variance in performance gain has also been observed in the single-agent MBPO algorithm and is therefore not a surprising finding [22]. The improved efficiency was achieved through benefiting more from an increased numbers of gradient steps. This benefit is most likely resulting from the larger diversity in the replay buffer available to the actor and critic, as performing a similar number of gradient steps without a model does not improve performance. It appears that after a limited amount of training, the world models are unable to improve in performance further. Although this does not prevent us from beating model-free algorithms in sample efficiency, this does leave us with the question of whether improvements in model learning techniques or model architecture could result in even greater improvements in sample efficiency. When training robots in real-life scenarios, sample efficiency is of utmost importance due to the costs associated with robots breaking, wear and tear, and the time of running the robots. Although our algorithm is able to improve sample efficiency, it only does so by a factor of between 1.7 and 3.7 in both our benchmark tasks, still requiring up to 5000 trials to reach desirable performance. This number of trials can still be prohibitively expensive, especially when there is a risk of robots breaking during each trial. Thus, more improvements in sample efficiency are needed to reach our goal of real-life learning for robotic swarms.

Although the increased number of gradient steps improves performance, it also significantly reduces the computational efficiency of the algorithm, especially in the scenarios when the costs of running the actor- and critic- networks are high compared to the cost of the simulator. The simple 2-D domains that we tested our algorithm on are prime examples of this. This results in a near linearly proportional relationship between the number of gradient steps performed and the computing time required. Whether this is a problem fully depends on the application. In many robotic scenarios, the centralized learning could be offloaded towards an external system that has plenty of computing power available.

At first glance, the chosen framework of centralized learning for decentralized execution imposes significant limitations on the applicability towards online learning in real life, as some centralized infrastructure appears to be needed to perform the learning. Such infrastructure might not be available in real-life scenarios where robotic swarms can be used. One possible way to circumvent this problem is to only perform learning when the robotic swarm has returned to some form of base station where communication is available. The robots could collect data simultaneously whilst acting in the real world and only use and share their collected data to learn when there is a communication link. This could be done, for example, when the robots are charging. Such a method would naturally fit within our approach as the only information that is used during centralized learning is a concatenation of all observations and actions of each agent, thus requiring no external observer of the system.

VI. Conclusion

The goal of this study is to contribute towards making real-life online learning for robotic swarms possible, which requires the availability of sample-efficient multi-agent reinforcement learning algorithms. To study the potential benefits of model-based learning to improve sample efficiency in a multi-agent context, we design and evaluate a new centralized learning for decentralized execution algorithm. This algorithm is, according to our best knowledge, the first in its class to utilize learned world models to improve sample efficiency. Using this algorithm we are able to improve the sample efficiency of a model-free baseline by a factor of 1.7 to 3.7, as demonstrated on two benchmark domains, highlighting the potential benefits of model-based learning.

Even though our approach demonstrates a significant improvement in sample efficiency compared to model-free approaches, more improvements are needed to achieve sample efficiencies that make online-learning in multi-agent real-life scenarios feasible. One way to do this would be to improve the model learning capabilities of our algorithm. One could utilize recent advancements in supervised learning techniques and neural network architectures to improve model learning speed and quality (for example through regularization [27]) or scalability (for example through attention [18]). Another way would be to improve the policy optimization part of our algorithm. Our research has specifically focused on a model-based version of Multi-Agent Soft-Actor-Critic; however, model-based approaches might also be applicable to other model-free algorithms such as Attention-Actor-Critic [18] or Q-MIX [14].

A second direction for future research could be to perform fully decentralized model-based learning. As explained in the discussion, the CLDE framework can be practical in many real-life applications. However, in situations where robots cannot communicate with each other or with a base station at all, fully decentralized algorithms are still needed. Model-based learning could be investigated as a potential improvement for sample efficiency in these situations as well.

Finally, further research could investigate how these algorithms can be scaled up to perform in more complex, real-world tasks. A solution could be to combine sim-to-real learning[28] with model-based reinforcement learning, such that the agents do not need to start from scratch when acting in the real world.

As highlighted in the previous paragraphs, this article provides a first baseline and starting point for further research towards model-based learning in multi-agent systems, which might eventually result in realizing the end goal of making online real-life learning for multi-agent robotic systems possible.

REFERENCES

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

[2] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, and Adrian Bolton. Mastering the game of go without human knowledge. *Nature*, 550(7676): 354–359, 2017.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, and Thore Graepel. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

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

[6] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning*, pages 387–395, 2014.

[7] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[8] Markus Peters, Wolfgang Ketter, Maytal Saar-Tsechansky, and John Collins. A reinforcement learning approach to autonomous decision-making in smart electricity markets. *Machine learning*, 92 (1):5–39, 2013.

[9] Mario Coppola, Kimberly Mcguire, Christophe De Wagter, and Guido Croon. A survey on swarming with micro air vehicles: fundamental challenges and constraints. *Frontiers in Robotics and AI*, 7:18, 2020.

[10] Levent Bayindir. A review of swarm robotics tasks. *Neurocomputing*, 172, 2015.

[11] Pablo Hernandez-Leal, Michael Kaisers, Tim Baarslag, and Enrique Munoz de Cote. A survey of learning in multiagent environments: Dealing with non-stationarity. *arXiv preprint arXiv:1707.09183*, 2017.

[12] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1146–1155, 2017.

[13] Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in neural information processing systems*, pages 2137–2145, 2016.

[14] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485*, 2018.

[15] Christian Schroeder de Witt, Bei Peng, Pierre-Alexandre Kamienny, Philip Torr, Wendelin Böhmer, and Shimon Whiteson. Deep multi-agent reinforcement learning for decentralized continuous cooperative control. *arXiv preprint arXiv:2003.06709*, 2020.

[16] Erol Sahin, Sertan Girgin, Levent Bayindir, and Ali Turgut. Swarm Robotics. *Swarm Intelligence*, pages 87–100, 2008.

[17] Ryan Lowe, Yi I. 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*, pages 6379–6390, 2017.

[18] Shariq Iqbal and Fei Sha. Actor-attention-critic for multi-agent reinforcement learning. In *International Conference on Machine Learning*, pages 2961–2970. PMLR, 2019.

[19] Shubham Gupta and Ambedkar Dukkipati. Probabilistic View of Multi-agent Reinforcement Learning: A Unified Approach. 2019.

[20] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *International Journal of Robotics Research*, 32(11): 1238–1274, 2013.

[21] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models. *arXiv preprint arXiv:1805.12114*, 2018.

[22] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems*, pages 12498–12509, 2019.

[23] Orr Krupnik, Igor Mordatch, and Aviv Tamar. Multi-Agent Reinforcement Learning with Multi-Step Generative Models. *arXiv preprint arXiv:1901.10251*, 2019.

[24] Pararawendy Indarjo, Michael Kaisers, and Peter D. Grünwald. Deep State-Space Models in Multi-Agent Systems. Master's thesis, Leiden University.

[25] Frans A. Oliehoek. Decentralized POMDPs. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State-of-the-Art*, Adaptation, Learning, and Optimization, pages 471–503. Springer, Berlin, Heidelberg, 2012.

[26] Igor Mordatch and Pieter Abbeel. Emergence of grounded compositional language in multi-agent populations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[27] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

[28] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30, 2017.

APPENDIX A
FULL HYPERPARAMETER SETTINGS

The full set of hyperparameters used in the experiments described in Section IV is shown in Table IV.

Hyperparameter	Notation	Value
Learning rate of model	lr_{model}	0.01
Learning rate of actor & critic	lr_{ac}	0.003
L2 normalization of model	-	0.001
Hidden layer sizes model	-	4×200
Hidden layer sizes actor	-	2×128
Hidden layer sizes critic	-	2×256
Model training batch size	-	512
Model training gradient steps	-	500
Model training interval	-	250
Number of models in ensemble	-	10
Actor-Critic batch size	-	256
Actor-Critic gradient steps	-	5
Actor-Critic training interval	-	1
Actor-Critic target network update rate	τ	0.01
Initial exploration constant	α	0.1
Discount factor	γ	0.95
Target entropy	-	-2
Min / max log variance of model output	-	$-5/-2$

TABLE IV: Full set of hyperparameters used in the multi-agent tasks.

APPENDIX B
VERIFICATION ON SINGLE-AGENT CHEETAH TASK

When there exists only one agent in an environment, our MAMBPO algorithm is equivalent to the MBPO algorithm described by Janner et al. [22]. Thus, to verify the correctness of our implementation, we compare the performance of our MAMBPO algorithm on the MuJoCo HalfCheetah-v2 task with the performance reached by Janner et al. In this task, the agent must learn to control a virtual cheetah to run forward. Each episode consists of 1000 time steps. We use similar hyperparameters to the ones described in the original MBPO paper. The results can be seen in Figure 8. The increased level of noise in our results can be explained by the limited number of runs we do (3) and the fact that we evaluate our agent through its training episodes, whereas Janner et al. do not show the number of runs and or the number of greedy evaluation episodes.

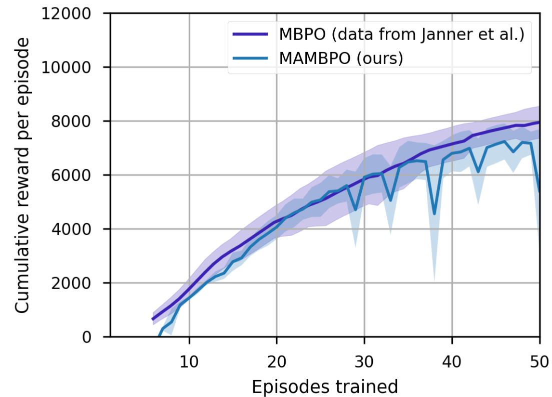


Fig. 8: Verification of our MAMBPO algorithm on the MuJoCo HalfCheetah task.

II

Preliminary Report

Authors remark: this part is a direct copy of the preliminary thesis report as submitted on 17-7-2020. The contents and direction of the thesis project have significantly changed after that date. The preliminary report should thus only be viewed as a source of additional information, rather than a direct preliminary study of the research as presented in Part I.

1
Introduction

Humans can achieve great things through cooperation. Robots can benefit from cooperation as well, but first they need to learn how to do so. From an early age, humans are social beings and learn to cooperate, achieving goals that they would not be able to reach on their own. Cooperative behaviour however, is far from unique to humans. Many animals, big and small, have their own forms of cooperation and social behaviours. Think of a pack of wolves that work together to hunt down prey much larger than they would be able to on their own. Or small creatures like ants working together in large colonies to forage for food. It would seem natural that robots, able to learn all kinds of complex behaviours, should also be able to cooperate with each other. One especially promising direction would be to have a large group of small and relatively simple robots cooperate to achieve complex tasks. Doing so can offer great benefits compared to creating a single robot able to do the task on its own. One example of this being that a single robot would be susceptible to failure. If one components breaks, the robot could fail in its task entirely. A large group of robots on the other hand, could be robust to individual failures.

The study of large groups of robots, known as *swarm robotics*, is motivated by the benefits that these large groups of robots could offer, and is inspired by swarms of insects [1–3]. To fully achieve the benefits of large groups, and be able to apply swarms of robots on complex tasks, some key challenges still remain to be solved. One of which is that of *swarm design*, which deals with designing the behaviours of the robots in a swarm. Although manually creating these behaviours is possible for simple tasks, this gets exceedingly more complex when the tasks get more difficult. To make this process less complicated for the human designer, *automatic swarm design* methods can be considered; methods that design or learn behaviours for robot swarms automatically. Being able to create behaviours without any human intervention would relieve the human designer of its exceedingly difficult task, and allow more complex behaviours to occur. However, the human is still required to create a swarm simulator which can be used for the automatic swarm design method to optimize behaviours on. Since reality is always slightly different from simulation, these optimized behaviours might turn out to perform worse in reality than they did in simulation, a problem known as the *reality gap*. Ideally, robots would learn their behaviours whilst acting in the real world, also known as *online learning*, and quickly learn desirable behaviours and how to solve tasks without any human intervention.

One promising area of research to look into to solve this challenge is that of *reinforcement learning*. This subfield of artifical intelligence studies how an agent can learn desirable behaviours automatically purely from interacting with an environment, receiving feedback in the form of rewards [4]. In recent years, the combination of reinforcement learning with *deep neural networks*, a class of function approximator loosely inspired by the human brain, has been able to achieve some remarkable feats. Being especially powerful in games, it has been able to outperform humans in Go, Chess [5–9], Atari games [9–12] and Starcraft-II [13]. Also on virtual robotics tasks such as learning to stand, walk or run as a virtual humanoid [14], reinforcment learning has made great progress. Learning these complex tasks, however, often takes thousands if not millions of trials, which is not viable on real robots: not only could failed trials damage the robot or equipment, it also takes an enormous amount of time. The amount of experience required by a reinforcement learning algorithm known as *sample efficiency*, a key aspect to improve to make learning on real robots possible [15]. One approach to do so is *model-based* reinforcement learning, where robots not only learn their behaviour,

but also learn a model of the world which can then be used to improve their behaviours faster. Techniques like these have allowed algorithms to solve simple robotic tasks in less than 10 trials [16].

In situations where multiple agents are simultaneously learning to cooperate in a single environment, reinforcement learning is more complex. The agents now not only have to learn about themselves and the environment, but also need to learn about the other agents. To avoid this increased complexity, one could design a central all-knowing entity that coordinates the behaviours of all agents. This effectively turns the problem into one, larger, single-agent problem where the central entity is considered the learning agent. For swarms of robots though, this would not be a good option. For one, the benefit robustness to individual failure would no longer exist. Every robot is relying on a single central entity. A more desirable approach would be for every agent to learn fully *decentralized*, without any central entity. For reinforcement learning, this is a complex task, as every agent only has limited information supply, and the environment, from the perspective of an agent, is not stationary due to changes in behaviour of other agents. Some standard reinforcement learning algorithms have been adapted to improve their performance in these situations, however; these approaches suffer from poor sample efficiency, and are thus unsuitable for learning on real robots.

Since model-based reinforcement learning has had such a significant effect on improving the sample-efficiency of single-agent reinforcement learning algorithms, we investigate if similar techniques can also be applied to improve sample-efficiency of decentralized multi-agent reinforcement learning. The combination of these two techniques has remained as a gap in ongoing Multi-Agent Reinforcement Learning Research [17]. Exploiting this combination can bring us one step closer to achieving online learning for robot swarms. Our main research question therefore is the following:

Can sample efficiency of decentralized, multi-agent reinforcement learning be improved through the use of a learned world model?

The main aim of this preliminary report is to provide the reader with the motivation, necessary background information and a clear approach to start answering this question. The core of the motivation part is Chapter 2, where we give a brief overview of the field of swarm robotics, with special attention to swarm design. Chapter 3 through Chapter 5 provide a comprehensive overview of a number of topics that can be considered prerequisite knowledge for the remainder of the report. Readers that are already familiar with these topics could skim through them or keep them as reference materials. These topics are as follows. In Chapter 3, we introduce the field of reinforcement learning, some basic reinforcement learning algorithms, key characteristics of these algorithms and the formal problem reinforcement learning strives to solve. Next, in Chapter 4, we introduce the combination of deep neural networks with reinforcement learning, also known as Deep Reinforcement Learning, introducing the reader to modern algorithms that serve as the basis for later multi-agent reinforcement learning algorithms and model-based reinforcement learning algorithms. One complicating aspect often occurring in complex and multi-agent tasks is that of partial observability, this topic is introduced in Chapter 5. The two primary components which the thesis will build upon are introduced in Chapter 6 and Chapter 7. In Chapter 6, we introduce model-based reinforcement learning algorithms. In Chapter 7, multi-agent reinforcement learning algorithms are introduced. The primary approach for this thesis can be seen as combining these two building blocks into a single algorithm that performs *decentralized, multi-agent, model-based reinforcement learning*. All the aforementioned topics are combined and summarized in the literature synthesis that is Chapter 8. This concludes the background part of this report. Then, in Chapter 9 we do a preliminary analysis on one potential approach towards online, model-based, learning for swarming that we call the PageRank approach, based on research by Coppola et al. [18] What remains is Chapter 10, the conclusion. Although this chapter is the conclusion, it primarily serves as a preview of the work to be expected in the final version of this thesis and the description of a practical approach on how to get there.

2

An Overview of Swarm Robotics

The field of swarm robotics studies the coordination of large groups of relatively simple robots through the use of local behavioural rules [1]. A key idea in swarming is the contrast between complexity at the individual level compared to the complexity at the system level (or swarm level). Simple individual robots are designed, often with simplistic behavioural rules. The system-level behaviour that emerges from these rules, however, can be complex. Simple agents can work together to achieve difficult and large tasks that are much larger than any one individual could perform on its own. This chapter gives a broad overview of the field. We start the chapter with a discussion on the inspiration and motivation of swarming. see that certain properties of social insects that form the source of inspiration can be highly desirable properties in robotic systems, leading to a number of advantages compared to standard single- or multi-agent systems. After this, in Section 2.2, we show how swarm robotics is embedded within the field of multi-agent robotics. Then, in Section 2.3, we discuss how the behaviour of the robots can be designed. This is known as swarm design. Finally, in Section 2.4 we discuss some issues that arise when applying swarm robotics to real robots.

2.1. Motivation and Inspiration

In this section, we detail the inspirational source of swarm robotics. In addition, we highlight some key advantages that a swarming approach to robotics can have compared to conventional (multi-agent) robotics in certain situations. Swarm robotics can be described as the application of swarm intelligence concepts into multi-robot systems. Swarm intelligence is the collective intelligence that arises from interactions within a large group of autonomous individuals [2]. Despite the individuals being relatively simple and lack of centralized coordination, the emergent behaviour at the system level can be impressive. Social insects, such as ants, wasps and termites, also show this type of system level behaviour, as these animals are able to coordinate their individual behaviours to achieve goals that greater than that which a individual can achieve [19]. The inspiration for swarm robotics stems from these kinds of insects [3, 20]. Although theses insects are a source of inspiration for many swarming algorithms and methods, imitation of nature does not have to be the main motivation for doing swarm robotics. Instead, research can be motivated by the advantages that swarming offers compared to conventional (multi-agent) robotics. Three properties that social insect swarms have that are also desirable for certain multi-robot systems are robustness, flexibility and scalability [3]:

- **Robustness** implies that swarm systems should be robust to failure or loss of any one individual or outside disturbances. If a single robot fails, the swarm should still be able to perform its task, albeit at a possibly lower level of performance. Four key factors can be identified that result in this robustness [3]. First, redundancy in the system: the large number of individuals results in the possibility of compensating the loss of a single individual by the collective group. Second, decentralized coordination; as the coordination is a shared task between the individuals, there is no central individual or infrastructure that could be a single failure point. Third, simplicity of the individuals; in general, complex robots or individuals have a greater chance of failure since their are more points within the robot that could cause such a failure. Fourth, multiplicity of sensing; distributed sensing by a large number of individuals results in a better signal to noise ratio.

- **Flexibility** implies that swarm systems should be able to perform different tasks, and be able to adapt

to changes in the environment. Individuals should also be able to take part in performing different tasks in contrast to being specialized on a single task only.

- **Scalability** implies that swarm systems should be able to achieve coordination and be able to perform tasks across a wide range of group sizes. This requires the coordination system that is in place to be able to work across different group sizes.

These three properties of social insects could be used as requirements for designing swarm robotics systems. Barca and Sekercioglu mention five other benefits of swarm robotics, that are highly related to and build upon the previously mentioned properties of social insects [2]:

- Utilization of sensing of many robots simultaneously allows for quick exploration of large areas. This is related to the previously described flexibility and scalability properties.
- Utilization of sensing from many robots simultaneously allows for a greater situational awareness of the swarm as whole. This advantage is also closely related to the previously described flexibility property.
- Robot swarms are more robust to failure of individuals as other individuals can take over unfinished work from the lost or failed individual. This advantage is very similar to the robustness property of insects.
- Workload can be distributed amongst individuals, resulting in greater results. This is especially beneficial when tasks need to be performed over a large spatial area. This relates to the scalability property of insects.
- Tasks can be carried out in parallel, this relates to the scalability property of insects.

In situations where these properties are desired, it might be useful to take a swarm robotics approach.

2.2. Swarm Robotics within Multi-agent systems

In this section, we place swarm robotics within the larger group of multi-agent robotics. To do this, we elaborate on two taxonomies of multi-agent systems and explain how swarm robotics fits into these taxonomies.

2.2.1. Taxonomies of multi-agent robotics

A taxonomy that describes multi-agent robotics along seven axes has been described by Dudek [21]. This taxonomy is also summarized in Table 2.1.

1. **Size of the collective.** This simply describes the number of robots within the system. Dudek identifies four categories within this axis. First, SIZE-ALONE implies a single-agent system. Second, SIZE PAIR, denotes a two-agent system. Third, SIZE-LIM means a system with multiple robots, where the number of robots is small compared to the size of the task. Fourth, SIZE-INF, where the number of robots is effectively infinite. The distinction between SIZE-LIM and SIZE-INF is dependent on the size of the environment or task. SIZE-INF provides a simplification by making the assumption that there is always a large supply of robots available to perform (part of) a task. SIZE-INF is typically assumed for swarm robotics [1].
2. **Communication range.** Dudek splits the communication range in three categories: COM-NONE, COM-NEAR and COM-INF, where there is no direct communication between robots at all, where robots can only communicate with other robots that are sufficiently nearby and where robots can communicate with any other robot respectively. Note that in the COM-NONE case, robots can still communicate indirectly through observing each other behaviour. COM-NEAR is typical for swarm robotics.
3. **Communication topology.** The communication topology axis categorizes multi-agent systems by structure in the communication network between the robots. Four categories are discerned here. First, TOP-BOARD, where agents broadcast their messages to all other agents. Second, TOP-ADD, where agents can address their messages to specific other agents. Third, TOP-TREE, where a tree structure determines the possible communication links. An agent can only directly communicate with agents they share an edge in the tree with. Fourth, TOP-GRAPH, where a graph determines robots available communication lines. This is a generalization of the TOP-TREE structure. Due to its decentralized nature, TOP-GRAPH is used in swarm robotics [1].

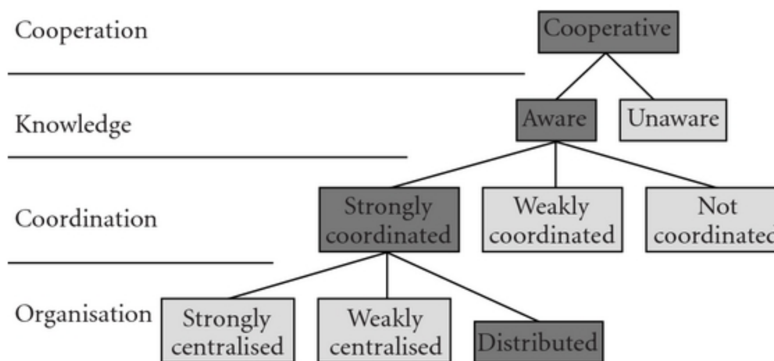


Figure 2.1: Classification of swarm robotics within multi-agent robotics using the taxonomy from Iocchi et al. [22] Figure taken from Navarro et al. [1] who in turn adapted it from Iocchi et al. [22]

4. **Communication bandwidth.** This axis categorizes systems by the cost of communication. BAND-INF implies that communication is effectively free. BAND-MOTION implies that costs are around the same order of magnitude as the costs of moving. BAND-LOW implies very high communication costs. Finally, BAND-ZERO implies no communication at all. Robots are in this case also unable to sense each other. Navarro states that BAND-MOTION is typical for swarm robots [1].

5. **Collective reconfigurability.** This axis deals with how quickly robots can reconfigure the group spatially. ARR-STATIC means a fixed topology. ARR-COM implies that rearrangement is only possible within communicating members. Finally, ARR-DYN implies that the formation can change arbitrarily. The dynamic nature of ARR-DYN is an integral part of swarm robotics [1].

6. **Individual processing ability.** This axis distinguishes systems between processing capabilities of the individual agents. Four categories are established. From low to high processing power these are: PROC-SUM, a non-linear summation unit, PROC-FSA, a finite state machine, PROC-PDA, push down automation and PROC-TME, Turing machine equivalent. Although processing power of swarm-robots typically is limited, they still use PROC-TME type processors [1].

7. **Collective composition.** This axis indicates the similarity between agents. CMP-IDENT implies that all agents are identical both in hardware and software. CMP-HOM implies that all agents are similar in physical capabilities. CMP-HET implies that agents can differ physically. Robots in a swarm are typically similar, they might be both CMP-HOM or even CMP-IDENT [1].

Another way of classifying multi-agent robotics has been introduced by Iocchi et al.[22] A taxonomy is proposed based on four properties, as is shown in Figure 2.1. A cooperative system is one in which multiple robots work together to achieve one system level goal. Awareness distinguishes whether or not robots are aware of the existence of the other robots. The level of coordination determines how strongly individuals react based on behaviour of other individuals. Finally, organization determines if the the coordination is done from a central level or in a distributed fashion.

2.3. Swarm Design

Swarm design deals with creating the algorithms that determine the behaviour for the agents to achieve a certain system-level goal. As explained in the previous section, coordination of swarm robotics is usually done in a distributed fashion. This implies that each agent has a set of rules that it follows and that these individual sets of rules determine the system-level behaviour. Designing these rules can either be done manually (also known as behaviour-based methods) or in an automatic fashion.

2.3.1. Behavior-based design methods

According to Brambilla et al., the most commonly used design methods are of the manual category [23]. Designing individual behaviours manually to result in the desired system-level behaviour is usually a trial and error process of evaluating and tuning the behavioural rules until the performance is satisfactory, this is also

Characteristic	Options	Notes
Size of the collective	ALONE, PAIR, LIM, **INF**	Swarms are typically described by a large number of agents.
Communication range	NONE, **NEAR**, INF	Communication in swarms typically only happens between agents that are close to each other.
Communication topology	BOARD, ADD, TREE, **GRAPH**	Swarms are highly decentralized and therefore typically have a graph structure.
Communication bandwidth	LOW, **MOTION**, INF	Generally, swarm robots are able to communicate, but there do exist limitations to the amount of communication possible.
Collective reconfigurability	STATIC, COM, **DYN**	Reconfigurability is a key benefit of swarms.
Individual processing ability	SUM, FSA, PDA, **TME**	This category might be slightly outdated with the recent increase in computing capabilities. Although many robots have Turing Machine Equivalent processors, their capabilities can still be limited by the amount of processing power.
Collective composition	**IDENT**, **HOM**, HET	Swarms generally are quite homogeneous, but behaviours don't necessarily need to be identical.

Table 2.1: The position of swarms within the taxonomy introduced by Dudek [21]. Typical characteristics of swarm-robotics are set in bold font.

known as a bottom-up or a code-and-fix approach [24]. These behaviour-based design methods can be categorized by the way in which they represent the behavioural rules. The two most popular representations will be discussed here in further detail. First, there are probabilistic finite state machines (PFSMs). Probabilistic finite state machines keep an internal state stored in the robot and updates these based on observations. Then, based on this internal state and these observations, actions are selected probabilistically. Second, there are virtual physics-based representations. These rules mimic physical laws, usually working with potential fields, resulting in behaviour of repulsion and attraction. These approaches are especially useful for formation flight[25] or obstacle avoidance [26] and can result in flocking-like behaviour [27]. In addition to these bottom-up approaches, some research has also been done on top-down approaches, such as property-driven design. In property-driven design, the starting point is a set of requirements for the system-level behaviour and the individual behavioural rules are then developed in a similar fashion to test-driven development [24].

2.3.2. Automatic design methods

There exist several approaches for automatic swarm design, the two main categories are both machine learning approaches: reinforcement learning and evolutionary robotics [23, 27]. The former of which will be discussed in more detail in the remainder of this literature study. In this section, we discuss some general characteristics, advantages and downsides to the general strategy of automatic swarm design. In general, automatic swarm design approaches can be categorized as a form of top-down design. The task of the designer here is to formulate the desired system-level characteristics in such a precise way that the automatic algorithm can optimize the individual behaviours to achieve these system-level characteristics. An additional task that needs to be done however; is creating the optimization algorithms needed to find the behaviour. It is desirable that these algorithms work across a wide variety of tasks and systems, such that the designer does not need to adjust the algorithm for every swarm system. Many of the machine learning approaches rely on evaluation of behavioural rules within simulations. These simulations should be sufficiently accurate that the behaviours can translate into the real world. A disadvantage of many of these methods is that the learned behaviour is difficult to interpret manually.

Describing automatic swarm design methods To describe automatic swarm design methods, four key characteristics of these methods are identified.

- **Online or offline learning.** Automatic swarm design methods usually perform some form of optimiza-

tion based on observed local or system behaviours. The difference between online and offline learning is whether this optimization is done based on simulation of the swarm, called offline, or if the method is designed to optimize over behaviours observed in the swarm acting in the real world, online. As simulators usually have a limited level of fidelity, there could occur a gap in performance of the swarm in the simulator and the performance when it is deployed in the real world. This problem is known as the reality gap. Online learning could be a solution to this. However, since using real robots to learn can be expensive to do for prolonged amounts of time[15], sample efficiency (a measure of the amount of real world experience the swarm needs before it reaches the desired behaviours) becomes an important characteristic of the algorithm.

- **Optimization algorithm.** The two primary categories of algorithms that perform behavioural optimization are that of Reinforcement Learning (RL) and Evolutionary Algorithms (EA). They are inspired by different principles yet tackle the same problem [28]. In recent years, the distinction between these two fields has become less pronounced as RL algorithms now sometimes incorporate optimization steps with close similarities to evolution as a non-gradient based optimization step within their policy optimization [29]. Although this study primarily focuses on standard (gradient-based) RL approaches, we will briefly discuss the possibility of using non-gradient optimizers such as EA in Chapter 4.
- **Level of centralization.** Even though swarms are supposed to have decentralized coordination, it is still possible for swarm systems to perform learning through a centralized agent. When the agent learns offline, this usually is not a problem. However, when online learning is done, this could require additional learning infrastructure, taking away from the decentralized nature of swarms.
- **Level of homogeneity.** In general, swarms are large groups of similar robots. However; it might still be possible for the robots to have different policies resulting in some amount of heterogeneity.
- **Policy representation.** Many choices are available for representing a policy (a function that determines behaviour of a robot). Some common methods include neural networks and behavioural trees. An important factor to consider is whether the representation is a direct mapping from an observation of the robot to a an action, or if the robot takes into account some form of history or memory.

2.4. Experimental Swarm Robotics

When dealing with real robots instead of simulated ones, there is a significant increase in complexity of the system to be designed. Design of the swarming behaviour is only possible once the robots are individually functioning correctly and autonomously, which can be a great challenge for many applications such as swarming of Micro Air Vehicles (MAVs). Coppola describes how low level characteristics of MAVs can influence the swarm design process [27]. Coppola depicts that the design of swarming MAVs can be split up into four levels of which only the highest level directly deals with the swarming behaviour. These four levels are the following:

1. MAV design
2. Local ego-state estimation and control
3. Intra-swarm sensing and avoidance
4. Swarm behaviour

The MAV characteristics at the lower levels set constraints on the possible behaviours at higher levels and vice versa: higher level behaviours place requirements on lower level characteristics. These relationships need to be taken into account when designing for swarm behaviour.

3

Reinforcement Learning for Markov Decision Processes

Reinforcement Learning (RL) is a subfield of artificial intelligence that studies agents acting in an environment. The agent receives observations and a feedback signal from the environment and needs to learn how to act based on these signals. This provides a promising research area for automatic swarm design. In this chapter, we strive to give a brief introduction to the topic and its main concepts. In later chapters, we use this knowledge to discuss more advanced concepts and techniques in reinforcement learning and show how RL could be applied to swarm robotics. In Section 3.1, we introduce the framework of Markov Decision Processes (MDP) which can be seen as a mathematical formulation of the problem studied by reinforcement learning. We roughly follow the notation used in the textbook by Sutton and Barto [4]. After this, in Section 3.2, we introduce some basic reinforcement learning algorithms used for learning within the MDP framework. Finally, in Section 3.3, we introduce taxonomies for reinforcement learning algorithms and define some broader concepts that explain design decisions when creating reinforcement learning algorithms.

3.1. Markov Decision Processes

Markov Decision Processes form a framework that is able to describe sequential decision making problems. The problems that we consider here are that of an agent that resides within a certain environment and needs to make decisions in order to reach some desired rewards. In this section, we discuss all the components that make up the Markov Decision Process in detail.

- **Agent & Environment** The two primary components of a MDP are the agent and the environment. The agent is the learner and decision maker. This can be a robot, a computer algorithm that plays board games and many other things. Everything outside the agent is called the environment. The agent and environment interact with each other at every time-step: the agent selects an action A_t, after which the environment provides the agent with a new state S_{t+1} and a reward R_{t+1}[4]. This interaction is illustrated in Figure 3.1. The main goal for the agent is to find a policy that maximizes its expected return (the sum of rewards over all time steps).

- **State** A state S_t is a characterization of the environment that fully describes all that is relevant to predict future rewards and states. In other words, knowledge of state is equally useful for predicting the future as knowledge of the full state history [4]. This is also known as the *Markov Property.* In finite MDPs, there is a finite set of all possible states: $\mathcal{S} = \{s^1, ..., s^M\}$. Here, M is the size of the state space.

- **Action** At every time step, the agent can select an action A_t from a finite set of possible actions $\mathcal{A} = \{a^1, ..., a^K\}$. Here, K is the size of the action space. These actions are the only way in which an agent can influence the environment. The set of actions that an agent can take can be dependent on the state that the agent currently is in. We name the set of actions that the agent can take in state $s : \mathcal{A}_s \subseteq \mathcal{A}$.

- **State transition function** When an action is selected in state s, the environment transitions to a new state s'. This transition can be described by the state transition function $P : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ that returns

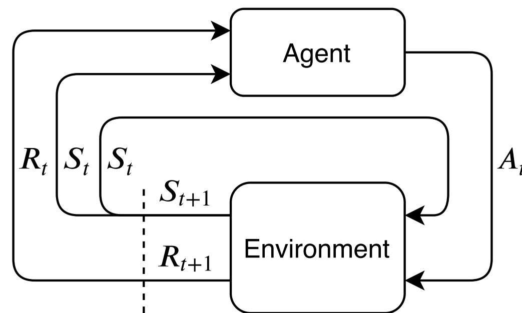


Figure 3.1: Description of the interaction between agent and environment. Figure adapted from Sutton and Barto [4].

the probability of transitioning to state s' given the previous state s and action a:

$$P\left(s', s, a\right) = \Pr\left\{S_t = s' | S_{t-1} = s, A_{t-1} = a\right\}$$

This function can also be represented as an array $\mathbb{R}^{M \times M \times K}$, where each element in the array contains the output of function P for a given s', s and a. We also call this array P, with elements $P^a_{ss'}$, but it should be clear from the context whether we are referring to the function or the array.

- **Reward function** The reward function $R : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ describes the reward R_t associated with transitioning to state s' from state s taking action a:

$$R_t = R\left(S_t = s', A_{t-1} = a, S_{t-1} = s\right) \tag{3.1}$$

In many problems, this reward function is not conditioned on all three of these variables, instead, a reward can be conditioned only on the resulting state s', or on a combination of two of these three variables.

Given these four elements, we can describe the complete Markov Decision Process as the following tuple[1][2]:

Definition 3.1.1. A **Markov Decision Process** is a tuple $\langle \mathcal{S}, \mathcal{A}, P, R \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ is a state transition function and $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a reward function.

Now that Markov Decision Processes are formally described, we go into more depth about the agent: how it selects an action through a policy and what the goal of the agent is.

3.1.1. Policies

The agents chooses actions to perform in the environment. It does this using a (stochastic) policy $\pi : \mathcal{S} \times \mathcal{A} \to [0,1]$ that maps a state action combination to a probability of choosing that action. As explained by [31], this policy is applied to the MDP in the following manner. First, an initial state s_0 is generated from the initial state distribution I. Then, an initial action a_0 is sampled from policy $\pi(s_0)$. This action is performed, and the environments generates a new state $s_1 \sim P(\cdot, s_0, a_0)$ and calculates the reward r_1 from $R(s_1, s_0, a_0)$. This process continues, resulting in the sequence $s_0, a_0, s_1, r_1, a_1, s_2, r_2, a_2, ...$ If the process is episodic, this process continues until a terminal state $s \in \mathcal{S}_{terminal}$ is reached. If the process is not episodic, this sequence continues on forever. This process is also illustrated in Algorithm 1. The application of a fixed policy to an MDP results in a Markov Reward Process $\langle \mathcal{S}, P^\pi, R^\pi \rangle$ where $P^\pi(s', s) = \sum_a P(s', a, s)\pi(s, a)$ and $R^\pi(s', s) = \sum_a R(s', a, s)\pi(s, a)$

3.1.2. Optimality and Value

An agent behaving in an environment seeks to gather rewards. However; the agent does not only care about the immediate reward, but also strives to maximize the rewards gathered in the future. This is represented

[1] Some descriptions of MDP's, such as the one by Silver [30], also include the discount factor in the MDP description. We refrain from doing this, and view the discount factor as part of algorithm design.

[2] The MDP is actually not fully described by this tuple as the reward array only contains the expected values of the rewards, and does not contain the full reward distribution. However; any MDP can be modelled in a deterministic reward form by splitting up the state with a separate state for every reward that can possibly be received.

```
Data: S, A, P, R, I, π
begin
    // Sample initial state
    s ← I
    repeat
        // Agent samples action
        a ← π(s)
        // Environment samples new state, reward
        s' ← P(s, a)
        r ← R(s', s, a)
        // Step forward in time
        s ← s'
    until forever
end
```

Algorithm 1: Application of a policy in an MDP.

in what is called the return, G_t. The objective of the agent thus is to maximize this return G_t. We describe three different kinds of returns (or objective criteria) that all describe the relationship of importance between future rewards and immediate rewards: *Finite horizon*, *Discounted infinite horizon* and *Average-reward*.

- The finite horizon objective strives to maximize the reward over some finite horizon h:

$$G_t = \sum_{k=0}^{h} R_{k+t+1}$$

- The discounted infinite horizon objective strives to maximize a discounted reward over an infinite time:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{k+t+1}$$

  Where $\gamma \in [0, 1)$ is a discount factor that determines the relative importance of rewards close in the future compared to rewards in the distant future. This discount factor also serves the mathematical purpose of bounding the infinite sum. This objective is perhaps the most used criterion in reinforcement learning research.

- The average-reward criterion strives to maximize the average reward over an infinite duration:

$$G_t = \lim_{h \to \infty} \frac{1}{h} \sum_{k=0}^{h} R_{k+t+1}$$

From here on, we by default assume usage of the discounted infinite horizon objective, as this is the most commonly used type of reward criterion.

In the next paragraphs, we will elaborate on some formulations of optimality. When considering the policy of the agent we denote the optimal policy, that is the policy that maximizes the expected return, with π^*:

$$\pi^*(s) = \arg\max_{\pi} (\mathbb{E}_{\pi} [G_t | S_t = s]) \tag{3.2}$$

It can be shown for all three criteria that if there exists an optimal policy, there exists an optimal policy that is deterministic. In the case of the two infinite-horizon criteria, it has also been shown that this optimal policy is stationary [32].

We now introduce two additional functions related to optimality that are used extensively in algorithms for solving MDPs: the *state value function* and the *state-action value function.* Starting with the state-value function, v:

$$v_{\pi}(s) = \mathbb{E}_{\pi} [G_t | S_t = s]$$

This value of state s can be related to the value of successor states in the following manner:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[G_t | S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \middle| S_t = s\right] \\ &= \sum_a \pi(s,a) \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma \mathbb{E}_\pi\left[G_{t+1} \middle| S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(s,a) \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma v_\pi(s')\right] \end{aligned} \tag{3.3}$$

The final line of this equation is what we call the *Bellman equation for* v_π and expresses the relationship between the value of the state and the value of its successor states. The *state-action value* function, also commonly known as the Q-value, is a similar concept to the state value, except that represents the value of starting in state s, taking action a and afterwards following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right]$$

This representation of value is commonly used in many RL algorithms and allows for selecting an action that maximizes the return, without requiring explicit knowledge of the state transition function.

3.2. Basic learning algorithms for MDPs

In this section we take a closer look at some standard approaches for solving MDPs. The techniques explained in this section are applicable to problems where the number of states and actions are limited, and therefore no function approximation is needed. These methods are also known as tabular solution methods, as the policies and other functions are usually represented in a tabular form. An important distinction in these approaches is whether the approach is *model-based* or *model-free*. In model-based approaches, full knowledge of the MDP is needed. In other words the state transition probability function P is directly used within the algorithms. In model-free approaches, this is not the case. Instead, the agent learns only through interaction with the environment.

3.2.1. Model-Based Dynamic Programming

In this section, we introduce two basic algorithms of the category dynamic programming that can be used to find an optimal policy in an MDP: *value iteration* and *policy iteration*. We explain the algorithms for the discounted infinite horizon objective. However, with some adjustments, they are also usable for other objective functions. Dynamic programming provides a structured way to organize the search for the an optimal policy π^* through the use of value functions [4]. Using a perfect MDP model of the environment directly within an algorithm to calculate an optimal policy is often referred to as dynamic programming (DP). Usually, finding such an optimal policy is done iteratively in a loop of *policy evaluation* and *policy improvement*.

Policy Evaluation The goal of policy evaluation is to find the value of a given policy for any given state. The Bellman equation (see Equation 3.3, repeated here for clarity) gives the relationship between the value of a state and its successor states:

$$v_\pi(s) = \sum_a \pi(s,a) \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma v_\pi(s')\right]$$

In the policy evaluation step, we solve this equation iteratively to find the value for any state as shown in Algorithm 2.

Policy Improvement Any policy can be improved by selecting greedily actions that result in the highest value. This step is known as the policy improvement step as shown in Algorithm 3.

Policy Iteration The complete dynamic programming algorithm that performs the policy evaluation and policy improvement steps iteratively until a converged optimal policy is found is known as the policy iteration algorithm and shown in Algorithm 4. It can be shown that this algorithm will always converge to an optimal policy [4].

Value Iteration The Value Iteration (VI) algorithm can be seen as a simpler alternative to Policy Iteration. Whereas PI consists of two separate iterative steps: Policy Evaluation and Policy Improvement, Value Iteration combines these two steps into a single iterative step. The value iteration algorithm is shown in Algorithm 5. In this algorithm, θ is a threshold for convergence.

Data: $\mathcal{S}, \mathcal{A}, P, R, \pi$
begin
  `// Initialize value array;`
  Initialize array $V(s) = 0$, for all $s \in \mathcal{S}$;
  repeat
    $\Delta \leftarrow 0$;
    for *every* $s \in \mathcal{S}$ **do**
      $v \leftarrow V(s)$;
      $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma V(s')\right]$;
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
    end
  until $\Delta < \epsilon$;
end
return $V \approx v^{\pi}$

Algorithm 2: Iterative Policy Evaluation algorithm, adapted from [4].

Data: $\mathcal{S}, \mathcal{A}, P, R, V$
begin
  for *every* $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_a \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma V(s')\right]$;
  end
end
return π

Algorithm 3: Policy Improvement algorithm, adapted from [4].

Data: $\mathcal{S}, \mathcal{A}, P, R$
begin
  `// Initialize policy and value array;`
  $V(s) \in \mathbb{R}, \pi_0(s) \in \mathcal{A}()$ arbitrarily for all $s \in \mathcal{S}$;
  $i \leftarrow 0$;
  repeat
    $i \leftarrow i + 1$;
    $V \leftarrow PolicyEvaluation(\pi)$;
    $\pi_i \leftarrow PolicyImprovement(V)$;
    if $\pi_i == \pi_{i-1}$ **then**
      return π_i, V
    end
  until;
end

Algorithm 4: Policy Iteration algorithm, adapted from [4].

Data: $\mathcal{S}, \mathcal{A}, P, R, \theta$
begin
  `// Initialize value array;`
  Initialize array $V(s) = 0$, for all $s \in \mathcal{S}$;
  repeat
    $\Delta \leftarrow 0$;
    for *every* $s \in \mathcal{S}$ **do**
      $v \leftarrow V(s)$;
      $V(s) \leftarrow \max_a \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma V(s')\right]$;
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
    end
  until $\Delta < \theta$;
end
return $\pi(s) = \arg\max_a \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma V(s')\right]$, *for all* $s \in \mathcal{S}$

Algorithm 5: Value Iteration algorithm, adapted from [4].

3.2.2. Model-Free Reinforcement Learning

The methods discussed in the previous subsection all directly use a model of the environment in their learning updates. That is, the backups utilize $P(s', a, s)$. In many cases, these transition dynamics of the environment are not known to the agent. Instead, the agent can only learn from experiences it gains trough exploring the environment. When this is the case, we can no longer directly utilize model-based reinforcement learning. Instead, model-free reinforcement learning can be used, where samples of the transitions are used for learning. Note that it might be still possible to use model-based reinforcement learning techniques if we first learn a model of the environment. These approaches, also known as indirect reinforcement learning will be postponed until Chapter 6. In this section, we will instead focus on three of the simplest model-free reinforcement learning algorithms: Monte-Carlo learning, Q-learning and SARSA. In Section 3.3 we will use these three algorithms as examples to illustrate some important characteristics of reinforcement learning algorithms that can help get more insight in the vast amount of reinforcement learning algorithms available. In later chapters, these more complex and generalized algorithms will be introduced that can improve performance.

A common theme between these three algorithms and a difference with the previously discussed DP approaches is that these approaches need to actively balance exploration and exploitation. Only exploiting what currently is thought of as the optimal policy (also known as acting greedily) can result in convergence to globally suboptimal policies. One aspect to guarantee convergence to a globally optimal policy, every state in the problem domain needs to be visited in the limit infinitely many times. This is often achieved through using an ϵ-greedy exploration strategy. That is: with a chance of ϵ, the agent selects a random action, and with a chance of $1-\epsilon$, the agent selects an action greedily, thus selecting the action with the highest estimated action-value. ϵ is often decreased over time to get a gradually more greedy policy.

Monte-Carlo Learning The simplest way of doing model-free reinforcement learning is that of Monte-Carlo learning. The idea behind Monte-Carlo learning is to sample episodes using some policy, and update a Q-Value table based on those samples. This has many similarities to the previously discussed Dynamic Programming methods with the key difference being that Monte-Carlo Learning uses samples of the environment to estimate a value instead of backing up the values through the state-space using the environment dynamics model. An implementation of a Monte-Carlo learning algorithm can be seen in Algorithm 6.

On-Policy Temporal Difference Learning: SARSA Instead of waiting until an episode has finished to update the value estimates, temporal difference (TD) methods update their value estimates based on the value estimates of the next occuring state(s). Therefore, TD learning bases estimated values on other estimated values. This is also known as *bootstrapping* and will be explained in more detail in the next section. A simple algorithm that does this is shown in Algorithm 7. Based on a learning rate, α, the the state-action value of a state action combination is updated towards the discounted state-action value of the state action combination encountered next, with the addition of the instantaneous reward observed. The policy that is used during the episodes is usually a ϵ-greedy policy. Therefore, the value that is being estimated is also influenced by the exploratory moves, resulting in unbiased value estimations for the ϵ-greedy policy, a key characteristic of on-policy algorithms that we will discuss in more detail later in this chapter.

Off-Policy Temporal Difference Learning: Q-Learning An off-policy alternative to SARSA known as Q-learning is shown in Algorithm 8. Instead of updating the action-value of a state based on the action-value of the next state-action combination, Q-learning updates the value based on the maximum over all policy action values for the next state. This implies that the values Q-learning learns are unbiased for a greedy policy, even though the agent is following a non-greedy policy. This is a key characteristic of off-policy algorithms, the policy that is being learned about is a different policy than the policy that is used to traverse the world during training.

3.3. Taxonomies for Reinforcement Learning

In this section, we expand on some important properties that make reinforcement learning algorithms differ from each other. In later chapters, where we discuss more complex RL algorithms, we will refer back to these terms to understand their behaviour and suitability for certain problems. These distictions are based on the dimensions higlighted by Sutton and Barto in their textook [4].

Data: $\mathcal{S}, \mathcal{A}, \theta$
begin
    `// Initialize state-action value array;`
    Initialize array $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$;
    Initialize list $Returns(s,a)$ empty;
    Initialize policy $\pi(a|s)$ arbitrarily as an ϵ-greedy policy.;
    repeat
        Generate an episode using π;
        for *each pair s, a in episode* **do**
            $G \leftarrow$ the return that follows the first occurrence of s, a;
            Append G to $Returns(s,a)$;
            $Q(s,a) \leftarrow$ average $(Returns(s,a))$;
        end
        for *each s in episode* **do**
            $a^* \leftarrow \arg\max_a Q(s,a)$;
            for *all* $a \in \mathcal{A}(s)$ **do**
                if $a = a^*$ **then**
                    $\pi(a|s) \leftarrow 1 - \epsilon + \epsilon/|\mathcal{A}(s)|$;
                else
                    $\pi(a|s) \leftarrow \epsilon/|\mathcal{A}(s)|$;
                end
            end
        end
    until *forever*;
end
return $\pi(s) = \arg\max_a \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma V(s')\right]$, *for all* $s \in \mathcal{S}$

Algorithm 6: Monte-Carlo Learning, adapted from [4].

Data: $\mathcal{S}, \mathcal{A}, \theta$
begin
    `// Initialize state-action value array;`
    Initialize array $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}$;
    for *each episode* **do**
        Initialize s;
        Choose a from s using policy derived from Q (such as ϵ-greedy) ;
        repeat
            Take action a, observe r, s';
            Choose a' from s' using policy derived from Q (such as ϵ-greedy) ;
            $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$;
            $s \leftarrow s'$;
            $a \leftarrow a'$
        until *s is terminal*;
    end
end
return $\pi(s) = \arg\max_a \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma V(s')\right]$, *for all* $s \in \mathcal{S}$

Algorithm 7: SARSA Algorithm, adapted from [4].

Data: $\mathcal{S}, \mathcal{A}, \theta$
begin
```
// Initialize state-action value array;
```
Initialize array $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}$;
for *each episode* **do**
Initialize s;
repeat
Choose a from s using policy derived from Q (such as ϵ-greedy) ;
Take action a, observe r, s';
$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \arg\max_{a'} \left[Q(s',a')\right] - Q(s,a)\right]$;
$s \leftarrow s'$;
until *s is terminal*;
end
end
return $\pi(s) = \arg\max_a \sum_{s'} P(s',a,s)\left[R(s',a,s) + \gamma V(s')\right]$, *for all* $s \in \mathcal{S}$

Algorithm 8: Q-Learning algorithm, adapted from [4].

3.3.1. On-Policy and Off-Policy

The first distinction we investigate is that of on-policy learning and off-policy learning. When an agent is acting in an environment it is following a certain policy, which we call the *behavioural policy*. When acting, the agent is learning, often it is learning values associated with a state. These values are estimations of the true value associated with following a certain policy from that state. In other words, we are learning values for a certain policy, which we call the *target policy*. In some cases, the target policy and the behavioural policy are the same, for example in SARSA. We are learning the value for the policy we are currently following. This type of learning is called *on-policy* learning. When the target policy is not the same as the behavioural policy, we denote this as *off-policy* learning. This is the case in for example Q-learning where the agent learns about a greedy policy.

3.3.2. Bootstrapping

In all discussed reinforcement learning algorithms, the agent updates its policy or value table based on some possible future states and rewards. This update could be done based on the final outcome of an episode (Monte-Carlo methods) or could be done based on some estimate of the values that will be encountered. Basing the estimate on another estimate is known as *Bootstrapping*. An obvious question that arises is: what is better and converges faster? There is no full answer to this question yet, although empirically, it has been found that bootstrapping often converge faster compared to Monte-Carlo methods in stochastic domains [4]. Therefore, most modern Deep Reinforcement Learning algorithms do perform bootstrapping as will be shown in Chapter 4.

3.3.3. Sample Backups and Full Backups

A second distinction related to how the values are updated is the differentiation between sample-backups and full-backups. Full-backups use the distribution of possible next states, actions and rewards to update a value target. This is the case in dynamic programming. However, when this distribution is not available, the algorithm has to instead rely on samples.

3.3.4. Model-Based Planning and Model-Free Learning

Learning based on a model of the environment is also known as planning. In dynamic programming, we assume availability of a full and perfect model of the environment which we use to update our values and policies. When this is not available, we rely on model-free learning. In some cases it might be valuable, instead of doing completely model-free updates, to build an imperfect model of the environment and use as a model for planning. This can improve the efficiency in some reinforcement learning algorithms and will be discussed in more detail in Chapter 6.

3.3.5. Function Approximation

In this chapter, we only discussed RL methods where the policy and/or values are stored in a tabular fashion. When the state or action space of a problem becomes large however; this is no longer possible, the algorithms become too slow to converge to their correct values or there is not enough memory available to store the tables. One approach to deal with this is through the use of function approximation, where instead of storing every value in a table, we store the values implicitly through the parameters of a model. Many model structures exist to do this with, two popular ones are a linear model based on hand-crafted features, or a neural network model which automatically learns the features. Especially the neural network model has become extremely popular over the last years, which is why we dedicate a chapter to this, Chapter 4.

3.3.6. Discrete and Continuous

The standard MDP formulation is only able to deal with discrete actions, where the actions available to take are all in a finite set $\mathcal{A}$. This, however, is very limiting, as in many tasks the actions that can be taken are continuous and therefore an infinite number of actions is possible. When this is the case, two general approaches are used: either the action space is discretized into bins, reducing the problem to a standard MDP, or function approximation is used to deal with continuous action spaces. This second approach is discussed in more detail in Chapter 4.

3.3.7. Actors and Critics

The final differentiation that we discuss here is that of actors, critics and actor-critics. An agent can keep a representation of a policy, state(-action) values or both. If an agent only contains a representation of a value function, we call this method a value function method, or a critic-only method. All methods discussed in this chapter are usually referred to as critic-only methods. These algorithms do not directly optimize for the actual objective of reinforcement learning (find a policy with the highest expected return), but instead use the knowledge that a policy that is greedy with respect to the values will be an optimal policy. Another approach is to only keep track of a policy function, this approach is called a actor-only method and are often known as direct policy search. These algorithms have not been discussed in this chapter, but some approaches that could fit in this category are evolutionary methods, genetic algorithms and some gradient based techniques. These algorithms perform a direct search on the actual objective of reinforcement learning: find a policy that has the highest return. These algorithms generally have good convergence properties, are easily extendible to continuous and large action spaces. However, they might be susceptible to local optima. The final possibility is to use a combination of an actor and critic, where both a value function as well as a policy are being learned. Such an approach might provide for faster and more stable training compared to actor-only approaches. This type of learner will also be discussed in more detail in Chapter 4.

4
Deep Reinforcement Learning

In this chapter, we discuss the use of Deep Neural Networks (DNN) as a function approximator in Reinforcement Learning. This is also known as Deep Reinforcement Learning (DRL). In Section 4.1, we give a brief introduction to Deep Neural Networks, the function approximator used in this type of reinforcement learning. Please note that this introduction does not strive to give a comprehensive introduction to DNN, but rather serves a refresher for someone already familiar with the topic. For a more comprehensive introduction to DNNs the reader is referred to the textbook on deep learning by Goodfellow [33]. After this introduction, critic-only DRL approaches are introduced in Section 4.2. Then, in Section 4.3, we introduce policy gradient methods for DRL. These methods, which are actor-only or actor-critic approaches, use a gradient based approach for policy optimization. We also show how these approaches can be used in domains that have continuous action spaces. In Section 4.4, we introduce some methods that perform policy optimization without direct use of the policy gradient. These algorithms include evolutionary approaches. Then, in Section 4.5 describe how DRL algorithms can be benchmarked and compared against each other. Finally, in Section 4.6 a number of insights are introduced that can help a DRL practitioner select a suitable algorithm for its task.

4.1. Deep Neural Networks

Deep Neural Networks (DNN) are a class of nonlinear function approximators loosely inspired by the human brain [33]. A visualization of DNN can be found in Figure 4.1. Typically, DNN are made up of an input layer, one or more hidden layers and an output layer. Every hidden layer consists of a number of neurons in which the following three steps are performed :

1. The output vector of the previous layer is multiplied with a weight matrix.
2. A bias vector is added to the output of this multiplication.
3. A non-linear activation function is applied to the vector.

When learning with DNN, the learning algorithm learns the parameters of the network (the weights and biases) to minimize some loss function. This minimization is usually done through stochastic gradient descent (SGD). In SGD, a mini-batch of training data is sampled and passed forward through the DNN. The outcomes of the DNN are compared to the target outputs for this mini-batch and this serves as an estimator for the loss function. The gradients of this loss estimation with respect to the parameters is then used to update the parameters by multiplying the gradient with some learning rate.

4.2. Value Function approaches: Deep-Q Networks and Improvements

In this section, we discuss the main value function (or critic-only) approaches used in Deep Reinforcement Learning. In general, these approaches often form the Neural Network based equivalent of Q-Learning. In the simplest case, one could replace the Q-value lookup table in Q-learning with a neural network. For example, you would have a neural network that maps a state to a Q-value for every action in that state. You could then, after every step in the environment, update the neural network in a similar fashion as is done in Q-learning. Stochastic Gradient Descent (SGD) for Neural Networks however, performs best if the samples

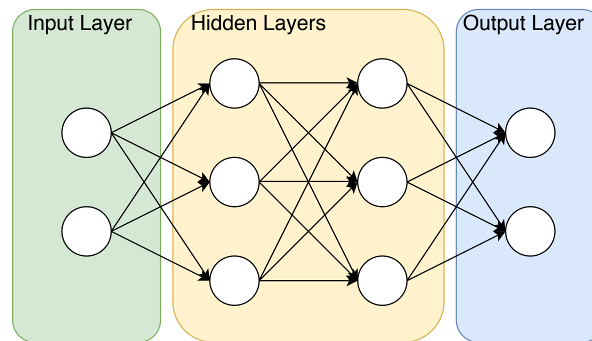


Figure 4.1: Visualization of a neural network. Data starts in the input layer, after which it is passed through the hidden layers. This is done after multiplication with a weight matrix (the arrows). In all nodes in the hidden layers, a bias is added, the inputs are summed and a nonlinear activation function is applied.

it receives are independent and identically distributed, which is not the case in samples that are received in order from single trajectories.

Deep Q-Networks (DQN) is a modification of the standard Q-learning with neural networks that solves this problem of receiving more independent samples. It has shown remarkable performance on the set of Atari games [12, 34]. DQN modifies the standard Q-learning with neural networks in two key ways. First, DQN uses a *replay buffer*. Once a step is taken in the environment, the transition and associated reward is stored in a replay buffer instead of directly performing SGD on this sample. Then, random sampling from the replay buffer is done, and stochastic gradient descent is performed on these samples instead. The replay buffer allows a single sample to potentially be used for multiple gradient descent steps, reducing the correlation between samples. The update is always based on the most recent Q-value estimate, as the target Q-value is calculated on the spot through the Neural Network. This last benefit could also be a disadvantage however, as unwanted feedback loops can start to occur between the value targets and the network parameters. The second modification strives to solve this problem of unwanted feedback loops through use of a *target network*. Instead of basing the target Q-value on the current neural network evaluation, the target values are based on target network evaluations, which is a network that is cloned from the Q-network and is then kept constant for a number of steps. These modifications have allowed DQN to achieve human-level performance on many Atari games.

In the next paragraphs, we will show how the DQN algorithm has been improved further through a series of modifications. An overview of how these modifications relate to and historically build upon each other is shown in Figure 4.2.

Double Deep Q-Networks In DQN, the action that is used as a training target is selected through a max operator over the Neural Network outputs, and the same Neural Network is used to evaluate the value. This can result in overestimation of the value. Double DQN reduces this through using the target network for evauation, and the current network for action selection [35]. This resulted in significantly improved performance in Atari.

Prioritized Experience Replay DQN samples uniformly from the replay buffer. Prioritized Experience Replay [10] changes this and instead samples frequently from samples where the last seen error was highest. In other words: sample experiences where there still is much to learn [11].

Dueling Deep Q-Networks DQN estimates the Q-value for every state-action combination. The Dueling Deep Q-Networks [36] architecture splits the Q-network into two streams with a separate head for Advantage and Value, where the advantage is improvement in value by selecting an action compared to the value of the state itself.

C51: Distributional Reinforcement Learning Standard reinforcement learning algorithms learn to predict the expected value of a state-action combination. Distributional Reinforcement learning instead learns a

begin
  Initialize value parameter w;
  Initialize target network parameters $w' \leftarrow w$;
  Initialize replay buffer R;
  for *each episode* **do**
    initialize s;
    while *s not terminal* **do**
      With probability ϵ select random action a;
      Otherwise $a = \arg\max_a(\hat{Q}_w(s,a))$;
      Take action a, observe s', r;
      Store transition (s,a,r,s') in R;
      Sample minibatch of $|B|$ transitions (s_i, a_i, r_i, s_{i+1}) from R ;
      Set $y_i = r_i + \gamma \max_{a'} \left(\hat{Q}_{w'}(s_{i+1}, a')\right)$;
      Update critic by minimizing loss: $L = \frac{1}{|B|}\sum_i (y_i - \hat{Q}_w(s_i, a_i))^2$;
      `// Update target network;`
      Every C steps: $w' \leftarrow w$;
    end
  end
end
return π

Algorithm 9: Deep Q-Networks

distribution of values. An algorithm that does this, known as C51 [37], has improved state-of-the-art performance on Atari. C51 learns the distribution represented as a fixed set of bins. Later research has improved this technique roughly through allowing the bin locations to adapt during learning (Quantile Regression or QR-DQN [38]) or through approximating the inverse cumulative distribution function, known as the quantile function, directly (Implicit Quantile Networks or IQN [39]).

Noisy Nets Performance on Atari has shown to improve through replacing ϵ-greedy action selection, with exploration by adding noise to network weights in a linear layer [40].

Rainbow: Combining Improvements The techniques mentioned in the previous sections, all improve the training of DQN in a different way. These five modifications have been combined in a single algorithm called Rainbow [11].

Ape-X: Distributed Learning Ape-X is a DQN-based algorithm, using many of the same components that Rainbow does, that explicitly seperates training the neural network from acting in the environment [41]. This allows multiple agents interacting with separate instances of the environment at the same time, all storing their experiences in a shared replay buffer and using a shared neural network. This allows faster training in terms of time elapsed during training, however, not necessarily in terms of sample efficiency.

R2D2: Recurrent Networks Recurrent Replay Distributed DQN (R2D2 [42]) improves the state-of-the-art on Atari in terms of maximum performance through the use of recurrent neural networks (RNN).

4.3. Policy Gradient Approaches

In contrast to the previously mentioned approaches that only keep track of a value network, policy gradient approaches use an explicit policy, either without any value network at all (REINFORCE), or in addition to a value network (Actor-Critic approaches). In this section, we give a brief overview of some important approaches within this category and show how they relate to each other. In contrast to DQN-like approaches, these algorithms are often are on-policy. Similarly to value function approaches, policy gradient approaches strive to find the optimal policy $\pi(s)^*$ (Equation 3.2, repeated here for clarity):

$$\pi(s)^* = \arg\max_{\pi}(\mathbb{E}_\pi[G_t|S_t = s])$$

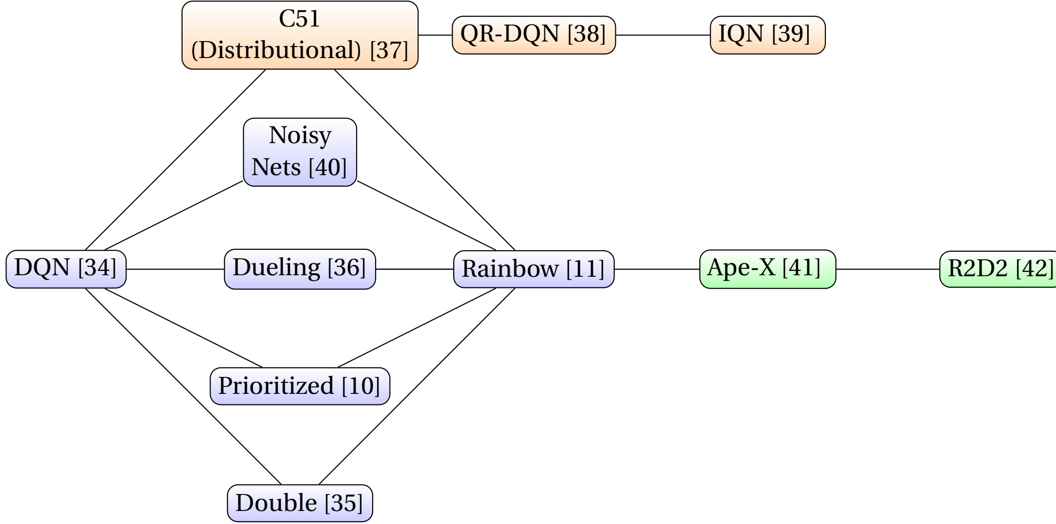


Figure 4.2: Historical relationship between different modifications of the DQN algorithm.

Policy gradient directly tries to find the optimal parameters (θ) of a policy (π_θ) for some representation through gradient ascent on a objective function based on the return:

$$J(\theta) = \sum_{s\in\mathcal{S}} \rho^\pi(s) V^\pi(s) = \sum_{s\in\mathcal{S}} \rho^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \tag{4.1}$$

Where V and Q are the true value for a state given policy π. ρ^π is the stationary distribution over the states, following policy π. In other words, how likely it is for our agent to be in a certain state. The goal for policy gradient algorithms is thus to find the values for parameters θ that maximize J_θ. Policy gradient algorithms do this through estimating the gradient of this objective function with respect to the parameters and performing gradient ascent on it:

$$\Delta\theta = \alpha \nabla_\theta J(\theta) \tag{4.2}$$

Calculating this gradient directly is difficult, however; thanks to the policy-gradient theorem, the gradient can be simplified to the following form [4]:

$$\begin{aligned} \nabla_\theta J(\theta) &\propto \sum_{s\in\mathcal{S}} \rho^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s) \\ &= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \mathbb{E}_\pi[Q^\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s)] \end{aligned} \tag{4.3}$$

Where the expectancy over π is the expectancy both over states and actions following policy π.

REINFORCE One of the most basic policy-gradient algorithm is that of REINFORCE. Using the knowledge that $\mathbb{E}_\pi[G_t|S_t, A_t] = Q^\pi(S_t, A_t)$, Equation 4.3 can be rewritten to:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(a|s)] \tag{4.4}$$

REINFORCE performs updates based on Monte-Carlo samples of this expectation:

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \ln \pi_\theta(a|s) \tag{4.5}$$

The full algorithm can be seen in Algorithm 10

REINFORCE with Baselines The Monte-Carlo REINFORCE algorithm suffers from high variance and thus slow learning. One method to reduce this variance is through the use of baselines. A baseline b is an action independent value that is subtracted from the Q-value estimate, modifying the policy gradient in the following way:

Data: $\mathcal{S}, \mathcal{A}, \theta$
begin
Initialize policy parameter θ;
for *each episode* **do**
Generate an episode $s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T$, following $\pi(\cdot|\cdot, \theta)$;
for $t = 0, 1, ..., T-1$ **do**
$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$;
$\theta \leftarrow \theta + \alpha + \alpha\gamma^t G \nabla_\theta \ln \pi_\theta(a_t|s_t)$;
end
end
end
return π

Algorithm 10: REINFORCE algorithm, adapted from [4]

$$\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_{s\in\mathcal{S}} \rho^\pi(s) \sum_{a\in\mathcal{A}} \left(Q^\pi(s,a) - b(s)\right) \nabla_\theta \pi_\theta(a|s) \\
&= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) \left(Q^\pi(s,a) - b(s)\right) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \mathbb{E}_\pi[\left(Q^\pi(s,a) - b(s)\right) \nabla_\theta \ln \pi_\theta(a|s)]
\end{aligned} \tag{4.6}$$

This does not modify the policy gradient itself, as shown below [4]:

$$\sum_a b(s) \nabla \pi_\theta(a|s) = b(s) \nabla \sum_a \pi_\theta(a|s) = b(s) \nabla 1 = 0$$

Adding such a baseline, however, can help reduce the variance of the Monte-Carlo updates, improving efficiency of REINFORCE. One suitable choice for the baseline would be an estimate of the state value $\hat{V}_w(s)$, where w are the parameters of this state value estimator.

Data: $\mathcal{S}, \mathcal{A}, \theta$
begin
Initialize policy parameter θ;
Initialize value parameter w;
for *each episode* **do**
Generate an episode $s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T$, following $\pi(\cdot|\cdot, \theta)$;
for $t = 0, 1, ..., T-1$ **do**
$\delta \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k - \hat{V}_w(s_t)$;
$\theta \leftarrow \theta + \alpha_\theta \gamma^t \delta \nabla_\theta \ln \pi_\theta(a_t|s_t)$;
$w \leftarrow w + \alpha_w \delta \nabla_w \hat{V}_w(s_t)$;
end
end
end
return π

Algorithm 11: REINFORCE algorithm with baselines, adapted from [4]

4.3.1. On-Policy Actor-Critics: A2C, A3C, TRPO and PPO

In this subsection, we introduce some standard on-policy actor-critic algorithms. Although REINFORCE with baselines does have an explicit value representation, it is generally not considered to be an actor-critic method, as it does not use the value representation to bootstrap from. If we instead do bootstrap from the value representation, we achieve the standard one-step actor-critic algorithm, shown in Algorithm 12. This algorithm can be seen as the baseline for most actor-critic algorithms. Two famous algorithms of this kind are Asynchronous Advantage Actor Critic (A3C) and Advantage Actor Critic (A2C). These algorithms are modifications to the standard one-step actor-critic algorithm to be more suitable for parrallelized learning using multiple CPU's, GPU's and instances of an environment [43]. Trust-Region Policy Optimization (TRPO) and

Proximal Policy Optimization (PPO) are improvements to the actor updates of actor-critics that both strive to answer the question: what is the greatest possible step that an actor can take as an update, without collapse of the policy performance? These techniques improve stability and efficiency of actor-critic approaches[44, 45].

Data: $\mathcal{S}, \mathcal{A}, \theta$
begin
  Initialize policy parameter θ;
  Initialize value parameter w;
  for *each episode* **do**
    initialize s;
    $I \leftarrow 1$;
    while *s not terminal* **do**
      $a \sim \pi_\theta(\cdot, s)$;
      Take action a, observe s', r;
      $\delta \leftarrow r + \gamma \hat{V}_w(s') - \hat{V}_w(s)$;
      $w \leftarrow w + \alpha_w \delta \nabla_w \hat{V}_w(s)$;
      $\theta \leftarrow \theta + \alpha_\theta I \delta \nabla_\theta \ln \pi_\theta(a|s)$;
      $I \leftarrow \gamma I$;
      $s \leftarrow s'$;
    end
  end
end
return π

Algorithm 12: One-step actor-critic, adapted from [4]

4.3.2. Policy gradients for continuous action spaces

In this chapter, we have not discussed adjustments to the policy to make these algorithms suitable for continuous action spaces. In the case of DQN-based algorithms, this is non-trivial to do, since DQN involves computing an $\arg\max$ of the state-action values over all actions. Policy gradient techniques however, do not contain such an argmax. Therefore, these algorithms can simply be extended to continuous action spaces. This is done by making the policy output a continuous distribution. One common way to do this is to have the policy output a mean vector μ_θ and a standard deviation vector σ_θ of a normal distribution, and sample from this normal distribution to select an action.

4.3.3. DPG, DDPG and D4PG: Deterministic Policy Gradients

In this section, we discuss Deterministic Policy Gradients. These algorithms use a deterministic policy representation, in contrast to the previously discussed algorithms which represent the policy as a probability distribution over all possible actions. To do this, we first rewrite the policy gradient in its continuous form [46, 47]:

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \mathrm{d}a \mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a) \right] \end{aligned} \tag{4.7}$$

Now consider a deterministic policy μ_θ that maps a state to an action deterministically, a similar policy gradient can be written down. Instead of integrating both over states as well as over actions however, this policy gradient only integrates over states since the actions are deterministic [47]:

$$\begin{aligned} \nabla_\theta J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a) \Big|_{a=\mu_\theta(s)} \mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\mu} \left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a) \big|_{a=\mu_\theta(s)} \right] \end{aligned} \tag{4.8}$$

In some cases, this approach can be directly applied in an on-policy learner. Often however, exploration is needed to discover the optimal policy. Therefore; an off-policy variant is needed, where a stochastic behaviour policy is followed whilst learning about the deterministic target policy. We name this behavioural policy β and the resulting policy gradient is the following [47]:

$$\begin{aligned}\nabla_\theta J_\beta(\mu_\theta) &\approx \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \mu_\theta(a|s) Q^\mu(s,a) \mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)\big|_{a=\mu_\theta(s)} \right]\end{aligned} \tag{4.9}$$

This can be incorporated into the one-step actor critic algorithm by adapting the updates to form the algorithm also known as DPG [47] and is shown in Algorithm 13.

Deep Deterministic Policy Gradient (DDPG) is an extension of DPG specifically tailored to the use of Neural Networks as function approximators [14]. This is shown in Algorithm 14. The main alterations are similar to the changes made to create DQN: it adds a replay buffer and target networks to aid in the stability of training. An extension to DDPG is Distributed Distributional DDPG (D4PG) [48] which adds multi-step returns, distributional RL and prioritized experience replay to the DDPG algorithm. In addition, multiple environment instances are used to act in simultaneously, reducing training time (but not necessarily improving computational efficiency).

begin
    Initialize policy parameter θ;
    Initialize value parameter w;
    for *each episode* **do**
        initialize s;
        $I \leftarrow 1$;
        while *s not terminal* **do**
            $a \sim \pi_\theta(\cdot, s)$;
            Take action a, observe s', r;
            $\delta \leftarrow r + \gamma \hat{Q}_w(s', \mu_\theta(s')) - \hat{Q}_w(s,a)$;
            $w \leftarrow w + \alpha_w \delta \nabla_w \hat{Q}_w(s,a)$;
            $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \mu_\theta(s) \nabla_a \hat{Q}_w(s,a)\big|_{a=\mu_\theta(s)}$;
            $s \leftarrow s'$;
        end
    end
end
return π

Algorithm 13: Off-policy Deterministic Policy Gradient

4.3.4. Soft Actor-Critic: Off-Policy Policy Gradient with Entropy Regularization

Soft Actor-Critic (SAC) is an algorithm that combines the off-policy learning of DDPG with the stochastic policy representation of on-policy actor critics [49]. Many on-policy actor-critic algorithms such as the previously discussed TRPO, PPO and A3C cannot utilize a replay buffer, requiring new samples from the environment in every time-step. This makes these algorithms perform poorly in terms of sample-efficiency. Off-policy actor critics such as DDPG are able to reuse samples through the use of a replay buffer, but can be brittle and sensitive to hyperparameter tuning [50]. The Soft-Actor Critic algorithm is in many regards similar to DDPG, with the main change being that it uses a stochastic policy of which the entropy is maximized through a regularization term. This removes the need for explicitly adding exploration noise to the policy, as the noise is already present in the policy itself and can be reduced or increased based in certain states by the learner itself. The authors of the article on SAC argue that this entropy regularization makes for a more sample-efficient algorithm that, in contrast to DDPG, is also very stable [49].

4.4. Policy Optimization without Gradients

The previously discussed policy gradient methods use gradient based optimization methods to find a (local) optimum for the policy, given a certain parameterization. It is also possible to do this optimization without direct use of this gradient. The core concept of these approaches is to perform some form of stochastic optimization directly on the policy parameters. These parameter settings are then evaluated on the task to assign a fitness score. In the simplest setting, random search can be used as an optimization algorithms [51]. Perhaps surprisingly, this approach has yielded competitive results on some benchmark reinforcement learning

begin
  Initialize policy parameter θ;
  Initialize value parameter w;
  Initialize target network parameters $\theta' \leftarrow \theta$, $w' \leftarrow w$;
  Initialize replay buffer R;
  for *each episode* **do**
    initialize s;
    while *s not terminal* **do**
      $a = \mu_\theta(s) + \mathcal{N}$;
      Take action a, observe s', r;
      Store transition (s, a, r, s') in R;
      Sample minibatch of $|B|$ transitions (s_i, a_i, r_i, s_{i+1}) from R ;
      Set $y_i = r_i + \gamma \hat{Q}_{w'}(s_{i+1}, \mu_{\theta'}(s_{i+1}))$;
      Update critic using: $\nabla_w \frac{1}{|B|} \sum_i (y_i - \hat{Q}_w(s_i, a_i))^2$;
      Update actor using policy gradient: $\nabla_\theta J \approx \frac{1}{|B|} \sum_i \nabla_a \hat{Q}_w(s, \mu_\theta(s_i))$;
      `// Update target networks;`
      $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$;
      $w' \leftarrow \tau w + (1-\tau) w'$;
    end
  end
end
return π

Algorithm 14: Deep Deterministic Policy Gradient

tasks. Many optimization algorithms commonly used in these approaches belong to the class of Evolutionary Algorithms (EA), algorithms inspired by the biological process of evolution [52, 53].

4.5. Benchmarking Deep Reinforcement Learning

Two popular benchmark suites for Deep Reinforcement Learning are a set of Atari games known as the Arcade Learning Environment (ALE) [54], and a set of continuous control tasks implemented in the MuJoCo physics simulator [55, 56]. ALE provides a set of 55 games, based on the Atari 2600 game computer system. Agents are usually trained based on the high-dimensional visual inputs. The games have a broad range in difficulty and since they were designed for human players, there exists a large variety in their goals and play-style. Continuous control algorithms are often benchmarked on virtual robotics tasks inmplemented in the MuJoCo physics simulator. Two popular implementations are those from the OpenAI Gym [55] and those from the Deepmind Control Suite [56]. An example environment can be seen in Figure 4.3. The goal in this environment is to move run forward by controlling the joints of the Cheetah. The input can be based on pixel observations or based on joint angle observations.

Figure 4.3: Cheetah environment from the Deepmind Control Suite [56].

Even though these benchmark suites exist, objectively comparing algorithms can still be difficult. Changes in hyperparameter tuning, stochasticity in the environments and performance evaluation can all have a significant effect on the apparent performance of an algorithm making reproducability a prevailing issue in deep

reinforcement learning research [57, 58].

4.6. What algorithm is the best?

In this chapter, a large number of Deep Reinforcement Learning algorithms have been introduced, in a varying degree of detail. The question that arises is then: what algorithm should one use when trying to solve a reinforcement learning problem? Sadly, it seems that this question is not straightforward to answer. In the previous section, we already discussed the difficulty of comparing reinforcement learning algorithms and their general sensitivity to hyperparameter tuning. In this section, we identify three insights that can help an user to select a suitable algorithm.

Critic-only approaches are most popular for discrete action domains, Actor-Critic approaches are most popular for continuous actions spaces. It seems that for most discrete action domains such as Atari, Critic-only approaches based on DQN are the most popular choices. The core principle that makes these approaches suitable for discrete action domains is the possibility to perform an $\arg\max$ operator over the Q-values of all actions possible from a certain state. In domains with continuous action spaces, this $\arg\max$ is in general impossible to perform[1]. For this reason, actors are introduced that form an estimator of this $\arg\max$ operator.

Off-policy learning is efficient, on-policy learning is robust. On-policy learning can only utilize samples from the environment that are found using the current behavioural policy. This makes use of a replay buffer impossible and significantly limits the amount of gradient updates one can do based on a single episode in the environment. Off-policy algorithms are able to use a replay buffer. When off-policy learning, function-approximation and bootstrapping are combined, as is done in these off-policy DRL algorithms, it is known that learning can diverge [60]. In practice however; these algorithms can still perform very well, but are known to be brittle and sensitive to hyperparameter tuning. Recent attempts at combining the strengths of on- and off-policy learning have resulted in algorithms that are arguably more stable and sample efficient [49].

Improved performance often comes at the cost of complexity. Both for critic-only methods, as well as for actor-critic methods, we see that many improvements to the baseline algorithms come from adding additional networks or hyperaparameters. For example: target networks, a second critic network, a delay hyperparameter for selecting how often to update the actor compared to the critic, etc. In some cases, such as a second critic network, this comes at the computational cost of having to do additional backpropagation. In most other cases, the improvements come at the cost of additional hyperparameters that need to be tuned. Even though the final performance of an "improved" algorithm can be an improvement compared to the original algorithm, the additional effort required to perform the hyperparameter tuning to extract this extra performance should not be neglected. Perhaps in some cases it might be better to select a "worse" algorithm that has less hyperparameters to tune or might be more robust to its hyperparameters. Apart from hyperparameters, the time to implement and debug algorithms also becomes greater as the complexity of an algorithm increases. Recently, some efforts are being made to make algorithms more robust to hyperparameters [49], or to create simpler algorithms altogether [61].

In the case where we want to apply model-based learning to improve sample efficiency of reinforcement learning for swarming, one might choose a sample-efficient algorithm as a baseline, and improve this through learning with a model. Since ideally, one would describe the swarming task as a continuous action task, sample-efficient off-policy algorithms such as DDPG or SAC might be suitable for their simplicity or robustness respectively. On the other hand, when a model is available to use for learning, it is possible that the sample-effiency of the RL algorithm itself is not an important characteristic anymore, due to the availability of a model that can provide a large amount of samples to the learner. From this perspective, a robust on-policy algorithm such as PPO might be the better choice.

[1] There are some exceptions to this, such as Normalized Advantage Functions that uses a special Q-network structure that makes the Q-value estimator quadratic in the selected action [59].

5
Reinforcement Learning with Partial Observability

In the previous chapters, we have considered reinforcement learning mostly for Markov Decision Processes, where the future states and rewards are only a function of the current state observed by the agent and the future actions taken by the agent. This is a property known as *Markovianness* of the environment. In many applications however, the agent is unable to observe the full state and only receives partial information about the state known as observations. This results in, from the perspective of the agent, the state transitions not only being dependent on the current observation and action, but also on earlier observations and actions. In this chapter, we first discuss the framework known as Partially Observable Markov Decision Processes (POMDPs) that describes these types of environments. We then discuss the consequences of this partial observability for reinforcement learning algorithms. Finally, in Section 5.3 some methodologies to deal with the consequences of partial observability are introduced.

5.1. Partially Observable Markov Decision Processes

Partially Observable Markov Decision Processes are an extension to the MDP framework. The framework is illustrated in Figure 5.1. In MDPs, the agent receives the full system state with all information that contributes to prediction of the future rewards and states. In selecting an optimal action, the state history can be ignored as the current state on its own already contains all relevant information. In other words, the Agent receives a *Markovian signal* for future rewards and states. In POMDPs this assumption is not valid any more. Instead of the full state, the agent now receives an observation. This observation contains some information about the state, but does no longer have to be a Markovian signal for future rewards and observations. Instead, the environment has a hidden Markov state. We denote the observation an agent receives at time t: $O_t \in \mathcal{O}$. Where $\mathcal{O}$ is the set of possible observations. This observation is generated from the observation function $O : \mathcal{O} \times \mathcal{S} \times \mathcal{A} \to [0,1]$:

$$O(o, s', a) = \Pr\{O_{t+1} = o | S_t = s, A_t = a\}$$

A full definition of a POMDP is given in Definition 5.1.1.

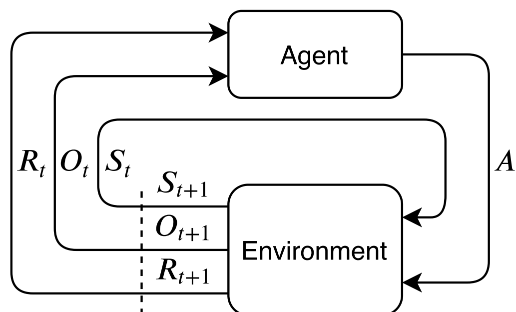


Figure 5.1: Description of the interaction between agent and environment in a POMDP.

Definition 5.1.1. A **Partially Observable Markov Decision Process** is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, P, R, O \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ is a state transition function, $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a reward function and $O : \mathcal{O} \times \mathcal{S} \times \mathcal{A} \to [0,1]$ is an observation function.

5.2. Consequences of partial observability

Partial observability usually originates from two sources: noisy sensors and multiple states resulting in the same sensor reading [31]. This second source can occur, for example, when a robot can only observe a small area of a larger environment. Partial observability can lead to *perceptual aliasing*: different parts of the environment look similar to the agent, but require different actions.

The fact that the environment no longer gives a Markovian signal to the agent has significant consequences on the algorithms available to solve the problems. Many standard reinforcement learning algorithm use a value function, which is a function of the current state, to which the agent no longer has access. Directly applying these algorithms to a partially observable problem is therefore usually not possible. Of course, there is a scale in how much observability the agent has, some problems are technically partially observable, but close to all state information can still be reconstructed from a single observation. In these cases, it might be possible to apply a standard RL algorithm, even though its performance might be slightly hampered. On the other end of the spectrum, observations can be extremely noise and contain only a very limited amount of information. This obviously determines what approaches are suitable for any given problem.

5.3. Reinforcement Learning with Partial Observability

A number of approaches have been developed of dealing with partial observability. In this section, we will elaborate upon some of the most important ones.

First, it is possible to simply ignore the partial observability and treat the observations as states. In this case, the agents policy will be a mapping $\pi : \mathcal{O} \times \mathcal{A} \to [0,1]$. Such a policy is also known as a *memoryless* policy. In general, these policies are suboptimal. The amount of performance loss, however, depends on the amount of observability. Also, the specific algorithm used for optimization of this memoryless policy can influence the converged performance of these policies. There exists some evidence that directly searching for the policy using a metaheuristic such as Evolutionary Algorithms can be better than value function based reinforcement learning algorithms [62]. An interesting question that remains is how gradient based policy search methods such as REINFORCE would perform in comparison. An additional aspect to take into account is that finding the optimal memoryless policy in a POMDP is no longer a convex linear optimization problem as it is for regular MDPs, instead generally being non-convex [63].

A second approach is to utilize memory, in the simplest case, the agent might not take only the current observation (o_t) into account, but also the previous observation (o_{t-1}) and the previously selected action (a_{t-1}) to select the current action a_t. One could increase the amount of previous observations taken into account further, creating a *finite-memory* policy, taking into account the last l observations and actions. One consequence of doing this would be that the dimensionality of the policy mapping increases rapidly: $\pi : \mathcal{O}^l \times \mathcal{A}^{l-1} \times \mathcal{A} \to [0,1]$. In addition, this combination of multiple observations could provide a signal that is closer to Markovian, but there are no guarantees about it. The hidden system state could be dependent on actions performed much earlier. To always be able to select the optimal action, the agent must take into account the full history: $h_t = \langle o_0, a_0, o_1, a_1, ...o_t \rangle$. For an infinite-horizon MDP this implies that the agent must have an infinite memory and the policy will be a mapping of: $\pi : \mathcal{O}^t \times \mathcal{A}^{t-1} \times \mathcal{A} \to [0,1]$. Directly using the full history can have significant practical drawbacks due to the scaling of the problem and the possible infinite growth of the policy.

The final approach is to compress the entire history into one fixed size variable, either perfectly or approximately. One approach to perform this compression is through the transforming the POMDP into a *belief MDP*. Such a belief MDP stores the history into what is known as a belief: $b(s)$ [31]. This belief represents the probability distribution over the states $\mathcal{S}$. This belief then serves as a Markovian state signal. Although this restores the Markov property of the system, the size of the belief space is infinite, as the belief of a state can take any value between 0 and 1. For this reason, standard dynamic programming approaches as described in Subsection 3.2.1 cannot be applied directly. The value function now needs to be computed over an continuous set of beliefs in contrast to the finite sets of states in a standard MDP. Fortunately, the value function can be parameterized by a finite number of vectors and has a convex shape [31]. This results in modified versions of dynamic programming algorithms being able to find the optimal policies for POMDP's. However, these

techniques are only computationally feasible for small domains. In addition, maintaining a belief requires a full model of the underlying POMDP, which in many applications is not available. If such a model is not available, and usage of memory is desirable, the agent can use learned hidden states as an alternative to beliefs. These hidden states can either be used to reconstruct a model of the underlying POMDP, on which a policy is learned, or these hidden states can directly be used in a policy representation. The indirect method of first reconstructing the POMDP has been used less extensively due to the complexity of both reconstructing such a model as well as the complexity of solving the resulting POMDP [31]. Of course, it is also possible to learn a value function in contrast to using only a policy or a full model. Recent work has investigated the use of recurrent neural networks (RNN) as a method for learning hidden states [64, 65]. RNNs are a type of NN that are able to handle an input of variable size, usually a time sequence [66]. At any point in the time sequence, these networks have a set of hidden nodes that contain some representation of all previous inputs. This naturally translates to the use of RNN in POMDPs, where a sequence of observations and actions could be compressed into a hidden state. This is done in a framework called recurrent policy gradient (RPG) [64]. Another, similar, approach is that of Deep Recurrent Q-Networks (DRQN) which is an adaption of DQN using RNN [67].

6

Reinforcement learning with a learned world model

This chapter focusses on how a world model can be learned to combine the benefits of model-based learning and model-free learning, increasing the sample-efficiency of learning algorithms. In Chapter 3, we saw that dynamic programming approaches can be used for training when a full model of the environment is available. For learning in an environment where this knowledge is not available, we showed in Chapter 3 and Chapter 4 that model-free reinforcement learning can be used. One issue that remains with model-free reinforcement learning is that, in general, a large number of trajectories of the environment need to be sampled during the learning process. This leads to poor sample efficiency. One promising direction for improving sample efficiency of model-free reinforcement learning is through the use of a learned world model [68]. We still denote this approach as model-based reinforcement learning. This strategy of learning a model and then using this model to plan actions has been shown to be more sample efficient than direct model-free learning [69]. In this work, we use the term *planning* to denote the use of a model to improve or generate a policy, similar to the usage of the term in Sutton and Barto's textbook [4]. Frameworks that use this approach often follow a structure similar to Figure 6.1. Any experience generated from the envionment can be used to learn a model and can be used to learn the policy or value estimations directly as well. The model-based part of learning is also known as indirect-RL whereas the model-free part is known as direct-RL. Two key aspects of any indirect-RL method are as follows [70]:

1. **Model learning & representation** When a model is learned, one of the most important questions to answer is how this model can be learned and how this model is represented. Of course, many different representations are possible, from simple tabular representations that directly try to estimate the state-transition array to neural network representations of all different forms. The word model can be ambiguous, one could argue that utilizing a replay buffer and re-sampling from this replay buffer as done by for example DQN is a form of model-based reinforcement learning, where the model is the replay buffer. In this chapter though, we will usually consider parametrized models that estimate the dynamics and rewards of the system. That is, we have a model $\mathcal{M}_\eta$ with parameters η that is an estimation of the forward dynamics of the systems, $\hat{p}_\eta \approx p$:

$$R_{t+1}, S_{t+1} \sim \hat{p}_\eta(r, s'|S_t, A_t) \tag{6.1}$$

   Within this category of parametrized models, three main types of model can be identified: expectation models, stochastic models (also known as generative models) and full models. Expectation models model the expected state outcome. When the state transitions are nondeterministic, this might lead to problems. For example, take an agent in a grid world that can select an action "up". When taking this action, the agent has a 50% chance of moving diagonally up to the left and a 50% chance of moving diagonally up to the right. Depending on the features used, or the parametrization of the model, the model might learn that the expected next location would be one grid straight up, a state that would be impossible to achieve in the real environment. Or the model might learn that the agent exists in two locations at once, another state that would not be achievable in the real environment, and that the value function could have difficulty estimating a value for. Therefore, in stochastic environments, a stochastic

model can be used. These models take some form of random noise as an additional input and generate samples of the state transitions from that. In the aforementioned example, it should learn to generate a left-up sample 50% of the time, and a right-up sample 50% of the time. The final type of model, full models, directly try to learn the full distribution function of state transitions. However; when the state space is large, this might be difficult to do. When dealing with any kind of model representation, it is unlikely that our model is a perfect representation of the environment. It is important to distinguish uncertainty in the model (epistemic uncertainty) from uncertainty associated with the stochasticity of the environment (aleatoric uncertainty). Using knowledge of the epistemic uncertainty can help to know how much trust can put into the model in certain parts of the state-space.

2. **Planning & Learning Integration** The second key aspect is the way in which learning and planning is integrated. One distinction that can be made here is the distinction between planning methods that plan only for the current state or trajectory (also known as planning over a learned model [70]), or planning methods that do planning to improve the policy over the entire state-space irrespective of the current state (also known as Model-Based RL with a learned model [70]). Both approaches have their own advantages and disadvantages. Planning only for the current state requires real-time computing power, whereas planning for all states allows the agent to learn at any moment it has excess computational power available. On the other hand, planning for all states might waste computing power on states that are not directly useful for the agent. In addition, it is not yet evident which of the two results in the greatest improvements in performance. Another distinction that can be made lies in the fact that algorithms for planning in continuous action spaces compared to planning in discrete action spaces often work differently.

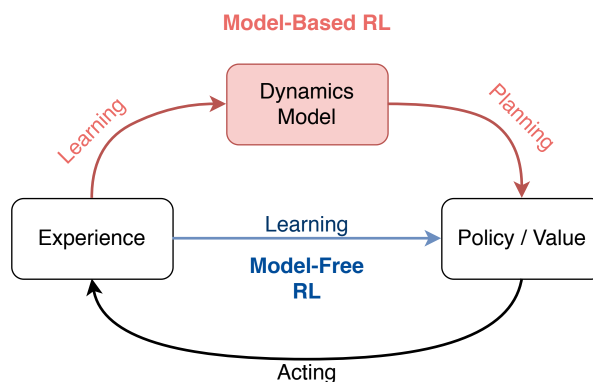


Figure 6.1: Comparison between model-free and model-based learning when there is no full system model available. Adapted from [71]

In the following sections, we will highlight some key approaches in indirect-RL and elaborate how they relate to the previous two questions. The first approach that will be discussed is that of Dyna. This is an early framework for the combination of model-based and model-free learning that serves as a basis for many contemporary model-based reinforcement learning algorithms. In the sections following this, modern approaches for model-based RL with continuous and discrete action spaces will be discussed.

6.1. Dyna: planning and learning with real and simulated experience

An early framework that utilizes the combination of model-free and model-based reinforcement learning is Dyna [72]. The core idea behind Dyna is to use a model as a simulator to generate additional experience for the policy/value learner. One implementation of this is the Dyna-Q algorithm [4] which is shown in Algorithm 15. This algorithm is an adaptation of the Q-learning algorithm that adds additional experience generated through the model as additional Q-learning updates. The model is represented as the P array for stochastic transitions or directly as state-action-state transition pairs for deterministic environments

The Dyna-framework itself, of which Dyna-Q is a specific implementation, does not specify what kind of learning algorithm or what kind of model representation has to be used. The key idea behind the framework is to both use simulated as well as real experience in a similar fashion to learn/plan a value function or a policy. It thus performs planning over the entire state-space. Other approaches close in nature to Dyna have been developed later, using localized features [73], linear function approximation [74, 75], deep neural networks

Data: $\mathcal{S}, \mathcal{A}, \theta$
begin
  `// Initialize state-action value array;`
  Initialize array $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}$;
  Initialize $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$;
  for *each episode* **do**
    Initialize s;
    repeat
      Choose a from s using policy derived from Q (such as ϵ-greedy) ;
      `// Direct-RL;`
      Take action a, observe r, s';
      $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \arg\max_{a'} \left[Q(s', a') \right] - Q(s, a) \right]$;
      $Model(s, a) \leftarrow r, s'$;
      $s \leftarrow s'$;
      `// Indirect-RL / Planning;`
      for *n times* **do**
        $s_{sim} \leftarrow$ random previously observed state;
        $a_{sim} \leftarrow$ random action previously taken in s_{sim};
        $r_{sim}, s'_{sim} \leftarrow Model(s_{sim}, a_{sim})$;
        $Q(s_{sim}, a_{sim}) \leftarrow\leftarrow Q(s_{sim}, a_{sim}) + \alpha \left[r_{sim} + \gamma \arg\max_{a'} \left[Q(s'_{sim}, a') \right] - Q(s_{sim}, a_{sim}) \right]$;
      end
    until *s is terminal*;
  end
end
return $\pi(s) = \arg\max_a \sum_{s'} P(s', a, s) \left[R(s', a, s) + \gamma V(s') \right]$, *for all* $s \in \mathcal{S}$

Algorithm 15: Dyna-Q algorithm, adapted from [4]

[76] and deep-belief networks [77]. When a neural network is used, it is possible not to model the full state or observation, but instead predict some encoded version of the state, known as a hidden or latent state, which has shown to be beneficial in stochastic environments [78].

6.2. Planning with continuous action spaces

Planning with a learned model in continuous action spaces has had significant attention over the years. A key motivation for this is that robotics, a possible application of reinforcement learning, could significantly benefit from sample efficient learning as doing trials on real robots can be expensive [15]. One approach for model-based reinforcement learning is probabilistic inference for learning control (PILCO) [16]. In this algorithm, Gaussian processes are used to model the dynamics, which allow for quantification of the epistemic uncertainties in the model. These Gaussian processes can be described as and are used as a full model. The controller is represented as a radial basis function network, and is optimized purely based on the model. This approach allowed to learn a cart-pole swing-up in less than 10 trials, or approximately 20 seconds of real world interaction. In later work, a similar general approach has been used. However, instead of using Gaussian processes as the dynamics model, this approach used Bayesian Neural Networks [79]. This model was used as a stochastic model, and used for sampling trajectories. Another approach is to use locally fitted linear models, as is done in Guided Policy Seach (GPS) [59, 80].

Recently, neural networks have become a popular method of representing a world model. One approach that does this uses a NN to predict a change in state based on an action [81]. This approach does not take into account uncertainties and only uses model-based learning as pre-training before model-free finetuning is performed to improve performance. Model Assisted DDPG (MA-DDPG) uses the neural model during the entirity of training, but only in situations when there is a large amount of uncertainty in the value estimates, limiting the usage of the model [82]. The model itself is a deterministic model. Chua et al. recognized the importance of modelling uncertainties in an approach called Probablistic Ensembles with Trajectory Sampling (PETS) [83]. Their neural networks output a Gaussian distribution of states which can be used to sample trajectories from. This deals with aleatoric uncertainty. Their model therefore is a full model of the environment.

To incorporate epistemic uncertainty into the model, they train an ensemble of models, sampling from each one when planning. In contrast to Dyna, planning in PETS is performed only for the current state and uses a Cross Entropy Method (CEM) planner as a form of Model Predictive Control (MPC). A later approach called Model Based Policy Optimization (MBPO) has a similar approach to PETS but instead performs planning for all states and combining it with a SAC learner, resulting in improved performance [84].

Another line of work has focussed on model-based learning for high dimensional observations. Two of these approaches are known as Embed to Control (E2C) [85] and Robust Controllable Embeddings (RCE) [86], where a low-dimensional latent dynamics representation is learned from high-dimensional image data. This dynamics model is then linearized locally to perform locally optimal control. An encoder-decoder network is used to learn this low-dimensional latent space. To perform better in more complex domains, Deep Planning Network (PlaNet) similarly uses an encoder to compress the high dimensional observation into a latent space [87]. Planning is then performed in this latent space, without linearization. This approach was suitable for POMDPs through the use of RNNs. In their dynamics model, both probabilistic and deterministic paths are incorporated, which was found to be beneficial for planning over longer horizons. Similar to PETS, planning is performed only for the current state. Another approach, called Dreamer, also performs planning in a latent space [88]. However; instead of using the learned model to plan using CEM for a single state, this approach performs simulated roll-outs to generate updates for a policy gradient actor, bearing similarities to the Dyna architecture. This resulted in improved performance compared to PlaNet, for similar tasks.

6.3. Planning with discrete action spaces

Model-based learning in complex discrete domains such as Atari has has limited success for a long time. Only recently, model-based approaches have been able to compete with their model-free counterparts. Value Prediction Network (VPN) learns a deterministic neural network model to predict latent states, values and rewards, conditioned on observations and actions [78]. A tree-like search is performed using the network to select actions during learning. In two algorithms called TreeQN and ATreeC, a tree-search planning method with a learned model is incorporated into DQN and an actor-critic approach respectively, outperforming these baselines [89]. These approaches are close in methodology to VPN. Muzero[9], another algorithm that performs tree search based on a learned, deterministic latent-space NN model, uses the search algorithm called Monte-Carlo Tree Search . In addition, this algorithm also uses a network that approximates both policy as well as value. The policy head of the neural network is then used to bias the tree search. This algorithm has been able to achieve state-of-the-art performance in Chess, Go, Shogi and Atari.

7

Multi-Agent Reinforcement Learning

In this chapter, we take a closer look into reinforcement learning with multiple agents. More specifically, we investigate cooperative Multi-Agent Reinforcement Learning (MARL). These cooperative reinforcement learning problems are often formulated in the form of a Decentralized (Partially Observable) Markov Decision Process. This type of process will be introduced in Section 7.1. In this section, we also briefly mention some complicating effects that make reinforcement learning in a decentralized setting more complex compared to single-agent reinforcement learning. After this, in Section 7.2, we elaborate upon a number of approaches for performing multi-agent reinforcement learning.

7.1. Decentralized Partially Observable Markov Decision Processes

Decentralized POMDPs (Dec-POMDPs) are an extension of POMDPs where there exist multiple agents that need to take actions, each having their individual observation. This is illustrated in Figure 7.1. Combining all individual observations together results in what is called the joint observation. Since every agent takes an action, the environment transition is no longer a function of the state and the action of one agent, instead it is a function of the state and the joint action. Dec-POMDPs are usually defined to be cooperative. That is, every agent receives the same, global, reward.

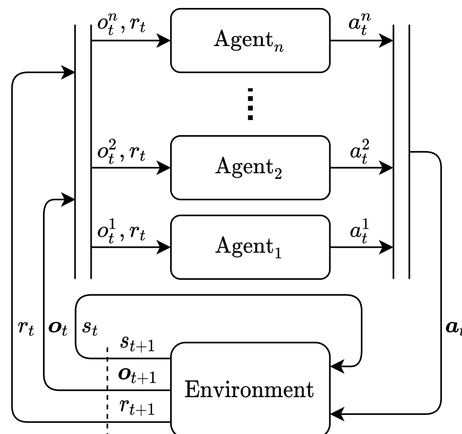


Figure 7.1: Description of the interaction between agent and environment in a Dec-POMDP.

Definition 7.1.1. A **Decentralized Partially Observable Markov Decision Process** is a tuple $\langle \mathcal{D}, \mathcal{S}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{O}}, P, R, O \rangle$ where $\mathcal{D} = \{1, \ldots, N\}$ is a finite set of N agents, $\mathcal{S}$ is a finite set of system states, $\boldsymbol{\mathcal{A}} = \times_{i \in \mathcal{D}} \mathcal{A}_i$ is a finite set of joint actions, where $\mathcal{A}_i$ is the set of actions available to agent i. $\boldsymbol{\mathcal{O}} = \times_{i \in \mathcal{D}} \mathcal{O}_i$ is a finite set of joint observations, where $\mathcal{O}_i$ is the set of possible observations available to agent i. $P : \mathcal{S} \times \boldsymbol{\mathcal{A}} \times \mathcal{S} \to [0,1]$ is a state transition function, $R : \mathcal{S} \times \boldsymbol{\mathcal{A}} \times \mathcal{S} \to \mathbb{R}$ is a reward function and $O : \boldsymbol{\mathcal{O}} \times \mathcal{S} \times \mathcal{A} \to [0,1]$ is an observation function.

A variation on Dec-POMDPs are Swarm Markov Decision Processes (Swarm-MDP), where every agent has the same characteristics and policy [90].

7.1.1. Consequences of decentralization

The assumption that an agent receives or is able to reconstruct a Markovian state signal does not hold for Dec-POMDPs. Since many RL algorithms are built upon this assumption, convergence problems can arise when applying standard RL to decentralized domains. In MDPs, the agent directly observes a Markovian state. In POMDPs, the agent is able to reconstruct a Markovian state through the use of beliefs or through taking into account the entire observation and action history. An agent acting in a Dec-POMDP is no longer able to reconstruct a Markovian state signal. This happens due to the fact that the state transitions depend not only on your own action, but also on (unobservable) actions of other agents. These actions are dependent on their policies which might change over time. This makes it impossible for the agent to construct a Markovian signal, unless the policies of all other agents are stationary, at which point the problem reduces to a POMDP. This problem is also known as non-stationarity of the environment [91]. The resulting difficulty can be illustrated intuitively. RL algorithms use some estimation of state-values or some other performance measure of a policy to improve their policies. When the environment changes, so should the values / performance measure for the given policy. As a result, the optimal policy changes. When the agent adjusts its policy based on this change in the environment, the environment has changed again, from the perspective of another agent, resulting in that other agent having to adapt its policy, resulting in a loop of changing environments and changing policies.

Two specific problems that arise with non-stationarity and a decentralized nature of an environment are *action shadowing* and the *equilibrium selection problem*[92]. Action shadowing is the phenomenon that occurs when one action appears better from the agents perspective, whereas another action could potentially be stronger if all agents select that action cooperatively. This issue is caused by the reward of an agent being dependent on the joint action selection, whereas the Q-value update is performed based on individual action selection. The equilibrium selection problem describes situations when coordination is required to select one of multiple global optimal solutions. This creates potential for agents to mis-coordinate resulting in poor performance.

7.2. Reinforcement Learning Algorithms for Multi-Agent Learning

Reinforcement learning for Dec-POMDPs is more complex than it is for POMDPs or MDPs, due to the aforementioned problems. Algorithms have been designed to specifically be able to deal with these problems. Often, these algorithms are adaptations from standard (deep) reinforcement learning algorithms. Two key directions can be identified in which reinforcement learning has been applied to Dec-POMDPs. The first approach is *centralized learning for decentralized execution (CLDE)* approach. In this approach, the assumption is made that some form of global information is available during learning, which would often be the case when learning is performed in simulations. CLDE approaches exploit this information to speed up and improve training. The final policies that are embedded into the agents, however, will only be conditioned on local observations, which allow them to perform decentralized execution. The second approach is *decentralized learning for decentralized execution (DLDE)*, also known as *independent learners*. Following this approach, no additional information is exploited during training that would not be present during execution. This complicates learning. For example, how would an agent know if the reward he receives is due to another agent taking a good action or due to the agent itself taking a good action? Although learning is more difficult, this approach is the only approach completely suitable for online learning on real robots in a real environment. In the following subsections, we explain how both of these approaches work, and review modern algorithms for each of those approaches.

7.2.1. Decentralized Learning for Decentralized Execution

We now discuss algorithms that belong to the DLDE approach. The simplest, but also naive, technique is to a standard reinforcement learning algorithm to the decentralized problem, where every agent learns on its own and the other agents are simply treated as part of the environment. As previously discussed, from the agent's perspective, the ennvironment is no longer stationary, as the behaviour of the other agents can change over time. This could result in convergence issues [93]. However, this does not necessarily results in worse practical performance of Q-learning-like algorithms in multi-agent settings when compared to algorithms that utilize deeper insights about multi-agent systems [94]. One example of this is the use of DQN to learn in two-player, cooperative Atari games [95]. This approach is also known as Indepenent DQN (IDQN). For problems with partial observability, recurrent nerworks have been used, resulting in an approach that has been called Reinforced Inter-Agent Learning (RIAL) [93], also known as Decentralized Recurrent Deep

Q-Networks (Dec-RDQN)[96]. In the same work, Differentiable Inter-Agent Learning (DIAL) is proposed, instead of agents purely sending discrete messages, these messages are also used as a way to propagate gradient from one agent to another, providing richer feedback for the agents. All these techniques, are based on DQN, which in the single-agent setting utilized replay buffers to stabilize training. In multi-agent learning however, replay buffers have been found to be detrimental to training due to the non-stationarity of the problem from the agents perspective [97]. Disabling replay buffers altogether can be done successfully as is shown by Foerster et al.[93].

A second way to deal with non-stationarity is through the insight that a sequence of actions resulting in a bad return could, in the multi-agent case, also be caused by exploratory moves of other agents, rather than by bad actions performed by the individual agent itself. This has been used to improve the convergence qualities of Q-learning in the multi-agent case. These approaches use optimism to remove (or reduce) the impact of these exploratory moves on estimated Q-values, whilst still being fully decentralized, such techniques are called optimistic approaches, as they assume decreases in value estimates are caused by exploratory moves of other agents. One approach that does this is known as distributed Q-learning and is (provable) able to find optimal policies in deterministic environments [98]. This (tabular) approach only updates the Q-values in the Q-table if the update results in an increase in value. In stochastic environments however, these algorithms are not able to differentiate between poor rewards from environment stochasticity and poor rewards from bad actions by cooperating agents, resulting in only the most rewarding state transition to be taken into account, instead of the expected value over all possible state transitions. Hysteretic Q-learning attempts to improve on this this through use of two separate learning rates, where one smaller learning rate is used for updating Q-values in the negative direction and a larger Q-value is used for positive updates [99]. Although this approach no longer guarantees optimal policies in deterministic domains, it can result in improved practical performance, especially in stochastic domains. The power of hysteric Q-learning and Dec-RDQN are combined in an algorithm known as Decentralized Hysteretic Recurrent Q-Networks (Dec-HDRQNs) [96]. Leniency is a similar technique to hysteretic learning. In this technique, the amount of "mistakes" accepted by the learner is reduced over time. This has demonstrated even better performance in some stochastic domains [100, 101]. This has also been used in a DQN adaptation known as Lenient Deep Q-Networkss (LDQN) [102]. Still however, these techniques are not able to explicitly differentiate between environment stochasticity and multi-agent exploration.

Distributional Reinforcement Learning, discussed in Chapter 4, has recently been investigated as an alternative to Hysteresis and Leniency and has shown improved performance in stochastic domains [91]. This approach, called Likelihood Hysteretic IQN (LH-IQN) uses two core techniques: Time Difference Likelihood (TDL) and Dynamic Risk Distortion (DRD). TDL measures the likelihood of a return distribution produced by the target Q-network given a distribution of the main Q-network. In other words, it is a measure of similarity between these two distributions. If the TDL is small, it is likely that this is caused by non-stationarity and teammate exploration. In this case, the learning rate is reduced. DRD is a separate technique to control risk sensitivity in distributional reinforcement learning. LH-IQN uses this to increase risk aversion over time.

The previously mentioned algorithms all build upon DQN and are therefore suitable for discrete action spaces, only limited research has been performed on decentralized learning in continuous action spaces. One work compared the performance of DDPG, TRPO and REINFORCE in a DLDE fashion with an adaptation of DDPG for CLDE [103]. This adaptation will be discussed in more depth in the next section. The CLDE algorithm unsurprisingly outperformed all DLDE algorithms as it was able to exploit more information and the DLDE algorithms were not adjusted to deal with non-stationarity.

7.2.2. Centralized learning for decentralized execution

The CLDE paradigm is a popular approach for automatic swarm design through reinforcement learning. In this approach, a policy is learned centrally, either using the global state, joint observations or through experience sharing. The policy that is learned, however, can be executed decentralized as it is only dependent on the localized observations.

One simple way to use global information is to share parameters of a neural network. This is done in a modification of the previously discussed RIAL algorithm, where the parameters of a Q-network are shared, and is applied in a communication task[93]. Value Decomposition Networks (VDN) [104] and QMIX [105] are two DQN based approaches that factorize the centralized Q-Network into Q-networks conditioned on individual observations. In VDN, this is done by composing the global Q-value to be the sum of individual Q-values. In QMIX, this is done through a mixing network that guarantees monotonically increasing global Q-values with respect to the individual Q-values.

A popular domain for continuous action CLDE algorithms is that of multi-agent collision avoidance and has spawned some algorithms designed specifically for this task. In this setting, two or more robots encounter each other whilst trying to reach their individual destinations. In general, the goal is to prevent collisions whilst minimizing the time to reach the destinations. Often, this path finding is performed through real-time planning, but reinforcement learning has been investigated as a method to reduce the online computational costs of collision avoidance associated with this real-time planning. The approaches differ in the exact type of RL algorithm used, the input of the centralized learner (global state, joint observations or shared experiences), and the type of observation data. Usually, these approaches learn a homogeneous policy over all agents. One work investigates the problem for a fixed number of agents in the observation [106]. The full system state is described by the combined individual states of a number of robots. The individual states of each robot is split up into two parts: an observable part that can be observed by every robot and a hidden part that can only be observed by the robot itself. The observable part contains information about the robots location, speed and size. The hidden part contains information about the robots intended goal position, preferred speed and heading angle. The RL method used is a model-free critic only method, sharing similarities with DQN. To make the method usable for continuous action spaces, the Q-network is replaced with a V-network. The learned value network is based on shared experiences of the agents and only has local observations as an input. Therefore, the value network can directly be used for action selection in a decentralized fashion. To do this, the other agent's state is propagated through using its filtered velocity. The reward signal consists of a local part based on collision avoidance and time to reach the target, and a global part based on the total time to reach the target for both agents. A core limitation of this work is that the state representation used in the value network requires a fixed number of agents. Later work has removed this constraint through the use of Long Short-Term Memory Networks (LSTMs), which are a type of NN able to handle inputs of variable size [107]. Another approach used raw (simulated) lidar data instead of the higher level features such as agent speed and location. This inherently made the observation space flexible to the amount of agents in the environment [108]. In this work, PPO is used and trained with shared experience.

Outside of the specific multi-agent collission avoidance approaches, the CLDE paradigm has also been used to find policies in more general (continuous action) swarming tasks. Hüttenrauch has investigated the use of reinforcement learning in a swarm-MDP. An actor-critic approach is used, where the full system state is available to the critic [109]. The agents have an actor that selects actions based on local history. However, this actor is updated centralized and therefore all agents will have the same policy. Huttenrauch has also investigated usage of TRPO for learning swarm behaviours. In this research, emphasis was placed on methods for representations of local observations, either through the use of histograms [110] or through the use of mean feature embeddings [111].

A CLDE version of DDPG called Multi-Agent DDPG (MADDPG) also uses a centralized critic and decentralized actor. This has been compared against standard decentralized DDPG in a suite of mixed competitive and cooperative games, showing improved performance compared to standard DDPG. COMIX is a DQN-based approach for continuous action CLDE. A centralized Q-Network is factorized into separate Q-networks for each agent that are only conditioned on their local observations [112]. DQN is made suitable for continuous action spaces through use of a Monte-Carlo sampling technique known as the Cross-Entropy Method (CEM).

7.3. Benchmarking Multi-Agent Reinforcement Learning

Whereas for single agent RL a number of domains exist that are extensively used for benchmarking RL algorithms, this is not the case in multi-agent reinforcement learning. This makes comparison of MARL algorithms even more difficult than is the case for single agent RL. Many researchers or research groups use their own benchmarks and domains. One reason for this might be that multi-agent algorithms are more commonly specialist algorithms designed to solve a specific set problems such as multi-agent navigation. It would be unfair to judge the performance of such an algorithm on its performance on some generic benchmark that has nothing to do with multi-agent navigation. In addition, there exist many different types and formulations of Multi-Agent problems (cooperative, competitive, etc.) and each of those has had significantly less attention than the standard single-agent MDP formulation, reducing the amount of comparison that can be done between algorithms. Even though it is therefore understandable that creating and conforming to generic benchmarks is difficult in multi-agent reinforcement learning, it is still good aspiration to utilize common benchmarks as much as possible. It is for this purpose, that we still show some domains that have been used in multiple articles. Similar to the rest of the chapter, we solely focus on cooperative domains, or domains

that can be made cooperative trivially.

One recently proposed benchmark domain is that of the Starcraft Multi-Agent Challenge (SMAC) [113] which provides scenarios of various difficulties for control of multiple agents in the strategy game Starcraft. These scenarios all feature discrete action space. Another recently proposed benchmark domain are robotics domains in MuJoCo, similar to the Deepmind Control Suite, modified for decentralized control, where separate agents control parts of a single robot [112]. Another continuous control set of environments is that of the Multi-Agent Particle Environments [103, 114]. In this set of environments, a number of tasks involving communication, cooperation and competition are implemented where the agents control a particle in a 2D continuous environment.

Specifically for swarming, few, if any, open sourced benchmark domains exists. Many works perform tasks such as formation forming [18], aggregation and pursuit [111], but implementations of these environments are often made for a single article and algorithm, providing limited, if any, comparison with other algorithms.

8

Literature Synthesis

This report has given a broad overview of topics, all related to online learning for swarm robotics. In this chapter, we discuss in more detail how each of those topics will contribute to the remainder of this thesis.

8.1. Swarm Robotics

Swarm robotics is a subfield of robotics characterized by the use of a large number of relatively simple robots, cooperating to achieve more complex goals than any individual robot could achieve. Usage of a larger number of robots in contrast to one or a few can improve robustness, flexibility and scalability of the system [1–3]. Typically, individual robots in a swarm are only able to observe the environment locally, only have local communication capabilities and are controlled in a decentralized fashion, that is: there is no single entity controlling the swarm. Instead, every robot makes decisions purely based on its own observations. Thus to achieve a global goal for the swarm, local behavioural rules need to be designed that contribute to this goal. Designing individual behaviours for these swarm-robots is a complex task. To achieve more capable swarms, automatic swarm design methods must be considered. Automatic swarm design strives to create algorithms that automatically optimize local behavioural rules. The two primary techniques that are used for this are Reinforcement Learning and Evolutionary Algorithms. Due to the existence of the so called reality gap, a decrease in performance noticed when deploying a swarm that was optimized in a simulator but deployed in the real world, it might be desirable to optimize the behaviours through real-world experience. This is known as online learning. However, online learning can be expensive due to the need to use real robots for prolonged amounts of time and the risk of damaging the robots [15]. To mitigate this as much as possible, it is important to use algorithms that can learn desirable behaviours with a limited amount of real-world experience, a property that is known as sample-efficiency.

8.2. Reinforcement Learning

Reinforcement Learning studies an agent that needs to make sequential decisions in an environment to maximize some expected average reward, also known as the return [4]. Markov Decision Processes (MDP) formalize the interface between the agent and its environment. At every time step, the agent observes a state and receives a reward from the environment, upon which the agent selects an action. Based on this action, the environment provides the agent with a new observation and reward, in a continuing cycle until some terminal state is reached. Usually, the agent internally keeps track of a value function (an estimate for the expected return) or a policy (a function that determines what action to select given a certain state). Based on experience, the value function and/or policy are improved over time, resulting in ever improving performance.

How to represent the policy or value function and how to adapt them based on experience is a primary factor discerning different reinforcement learning algorithms. In small domains where only a limited number of distinct states can be visited and a limited number of actions can be selected in any state, these functions might be stored in tabular form. In larger domains, some form of function approximation is needed. Recently, Deep Neural Networks have become a popular tool for this function approximation in a field that is known as Deep Reinforcement Learning. Function approximation also makes it possible to perform learning in continuous environments, where the actions or states are continuous instead of discrete. This is more realistic for many applications including swarm robotics.

The MDP framework assumes an environment that is fully observable: the agent receives an observation that is Markovian. Future states and rewards are only dependent on the current and future actions and the current state, not on past states. If this is not the case, the problem is a Partially Observable MDP (POMDP). This means the agent needs to take into account the full observation history in order to select the optimal action, which is non-trivial to do. In some cases, it is possible to ignore the partial-observability at minimal expense of some performance. Otherwise, one needs to explicitly deal with the partial observability. This could be done optimally through the use of belief states for small domains, or through the use of function approximators such as Recurrent Neural Networks (RNN) that are able to take sequential data as an input.

Although (Deep) Reinforcement Learning has already proven to be a powerful approach for single agent-tasks [11, 12, 34, 43], multi-agent tasks [93] and swarming tasks[109–111], it is usually reliant on the availability of a large amount of training data to achieve strong performance, especially in large and complex domains that deal with continuous action spaces, partial observability or multiple agents, all typical properties of swarm systems. Therefore, significant improvements in sample efficiency are necessary to make online learning for swarm robotics viable in anything but the simplest domains.

8.3. Model-Based Reinforcement Learning

One way to improve the sample efficiency of reinforcement learning is through the use of a learned world model. It might be easier to learn a model of the world than it is to learn a full policy or a value function. If this is the case, it can be beneficial to first learn a model of the world and then learn a policy and/or value function based on this world model. Although these methods do improve sample efficiency, this often comes at a cost of additional computation. Robotics, including swarm robotics, with it's high cost of real-world samples, could be a promising application for these techniques. Two key design choices are particularly important in this class of algorithms. The first is model-representation and architecture. Most recently, ensembles of DNN that incorporate both epistemic and aleatoric uncertainty have shown to be a powerful representation [83, 84]. The second is planning-learning integration: how can the model be used to learn to select better actions. The two primary methods for doing this are planning only for the current state (such as PETS [83]), or planning for the entire state-space (such as MBPO [84]). Both approaches have their own advantages and disadvantages. Planning only for the current state requires real-time computing power, whereas planning for all states allows the agent to learn at any moment it has has computational power available. On the other hand, planning for all states might waste computing power on states that are not directly useful for the agent.

8.4. Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning (MARL) studies Reinforcement Learning when multiple agents are acting and interacting within a single environment. One specific instance of this is when agents need to cooperate to achieve certain goals. Two primary ways exists in which reinforcement learning is performed in these situations. The first, centralized learning for decentralized execution, uses a central entity to guide the learning process. During deployment, the agents then act in a decentralized fashion. This approach is generally not suitable for online learning for swarming, as it is undesirable to have a central entity whilst learning in the real world for a swarm of robots. The second approach, decentralized learning for decentralized execution, is a fully decentralized appraoch. Here, no central entity is used during learning. Instead, each robot learns on its own through local interactions. Such an approach, however, can create problems for standard reinforcement learning algorithms. Most standard reinforcement learning algorithms are based on the assumption that the environment is stationary. In decentralized multi-agent reinforcement learning, this assumption is no longer directly satisfied: from the perspective of a single agent, all other agents are part of the environment. As the learning progresses, the policies of all agents change and thus also the environment changes from the perspective of a single agents. This can result in convergency problems for reinforcement learning [93]. Even so, in some practical settings, direct application of standard RL algorithms in decentralized environments have had decent results [94, 95]. If this does not provide satisfactory results, specialized techniques such as hysteresis and leniency can be used to make algorithms more robust to the non-stationarity of the environment [98, 99, 102].

8.5. Model-Based Learning for Decentralized Agents

Although both Multi-Agent Reinforcement Learning as well as Model-Based Reinforcement Learning are popular topics of research in their own regards, the combination of these two techniques for decentralized agents

has largely remained a gap in the current literature [17]. Given this gap in recent literature and the fact that improving the sample efficiency is a crucial ingredient towards making online learning for swarm robotics possible, this thesis strives to answer the following research question:

Can sample efficiency of decentralized, multi-agent reinforcement learning be improved through the use of a learned world model?

Direct application of a model-based reinforcement technique such as MBPO [84] or PETS [83] in a decentralized fashion is one way to approach this. These approaches could theoretically suffer from the same convergency problems as when applying standard model-free algorithms to decentralized environments, yet standard model-free algorithms do not always worse practical performance compared to more complex approaches of dealing with decentrality [94]. Similarly, direct application of model-based techniques on swarming problems might be a succesfull approach. Possibly, these algorithms could be adapted to be made more suitable for decentralized domains similar to how techniques such as hysteresis have made Q-learning more suitable for decentralized environments [99, 102].

A more specialized method for decentralized reinforcement learning would be to explicitly embed information about the decentralized nature of the environment within the model architecture. Such an approach could be desirable as opens up the possibility for an agent to reason differently about the non-stationary collaborating agents compared to the stationary environment. On the other hand, such an approach does embed more specific prior information on the model structure, possibly making the approach less generalizable.

9

Preliminary Analysis of PageRank for Online Learning in Swarms

PageRank, an algorithm originally used to rank websites based on the hyperlinks referring to those websites has recently been used in a novel approach to perform automatic swarm design[18]. In this chapter, we perform a preliminary analysis of this technique and its application to online learning for swarm-robotics. In Section 9.1 we perform a thorough analysis of the PageRank approach for swarming. We give an analysis on how the PageRank terminology relates to standard MDP terminology, we describe how the problem can be reformulated as a Linear Programming problem and give an example how the approach could be modified to make online learning possible. In Section 9.2, we experimentally demonstrate that the performance of the linear programming formulation is similar to the original PageRank method, with much less computational demand. Also, we show the practical implications of a discrepancy between the objective function of the PageRank optimization and the performance of the swarm. Finally, we demonstrate the effects of online learning with the Linear Programming formulation. In Section 9.3, we conclude this chapter from the perspective of usability of the PageRank method for the remainder of this thesis.

9.1. An Analysis of PageRank and its relationship to MDPs

A persisting problem that exists in (automatic) swarm design is that of the micro-macro link. The goal for a swarm system is usually defined on the swarm-level, comprising of a large number of robots, whereas the behavioural design is performed on the individual swarm-robot level. Investigating the system level performance is usually done through a global simulation of the system, which can be expensive. When doing automatic optimization of the individual behaviours, a large number of these global simulations are needed. Research by Coppola et al. [18] has used the PageRank as a method to link system-level behaviour with local behavioural rules, removing the need for any global simulation to find satisfactory behaviours.

The PageRank method for automatic swarm design implicitly models the swarming problem for any individual in the swarm as a fully observable, stochastic MDP. In the next four paragraphs, we show how the terminology introduced in the paper translates to standard MDP terminology. This translation makes it clear that the problem can be transformed into a linear programming problem that is efficiently solvable without the need for an Evolutionary Algorithm, which is done in the original research.

PageRank model for swarming The PageRank model to evaluate a given policy for swarming behaviour considers a graph $G_{\mathcal{S}} = (V, E)$ [18]. This graph models the local behaviour of a robot in a swarm. Every node in the graph is a state the robot can be in: $V = \mathcal{S}$. The edges contain all transitions that the robot can make in the local state-space. If the edges of this graph contain all transition probabilities, it describes a Markov Chain. Coppola splits these transitions up into two parts: *active* and *passive*.

- $G^a_{\mathcal{S}}$, the active part of the graph, contains the transitions that happen due to actions by the robot itself. This is the only part of the graph that is influenced by the stochastic policy π.

- $G^p_{\mathcal{S}}$, the passive part contains the transitions that happen due to changes in the environment [18]. That

is, changes that occur due to other agents in the swarm. In the method, it is assumed that this is independent from π.

Formally, the full graph can be described as the union of these two sub-graphs: $G_{\mathcal{S}} = G_{\mathcal{S}}^{a} \bigcup G_{\mathcal{S}}^{p}$. Effectively, these definitions transform the problem that the policy is being optimized for into a single-agent fully-observable stochastic MDP, as all transition probabilities are only dependent on the local state and the current action only.

The paper then introduces matrices $\mathbf{H}_{\pi}$, $\mathbf{E}$ and $\mathbf{D}$, which form matrix representations of the graph $G_{\mathcal{S}}$. $\mathbf{H}_{\pi} = \text{adj}(G_{\mathcal{S}}^{a})$ models the local state transitions of the robot through its own actions and has therefore the policy as a parameter. $\text{adj}(G)$ denotes the weighted adjacency matrix of graph G. $\mathbf{E} = \text{adj}(G_{\mathcal{S}}^{p})$ models the environment: the state transitions that can happen due to other robots taking actions. Finally, $\mathbf{D}$ models the environment when the robot itself cannot or will not take any actions. States in which this happens are named static sates: $\mathcal{S}_{static}$. Therefore: $\mathbf{D} = \text{adj}(G_{\mathcal{S}}^{p}(\mathcal{S}_{static}))$. To complete the PageRank model, an expansion factor α is introduced, which represents the relative chance of the robot taking an action compared to the environment causing a state transition. The so-called "Google matrix" is then constructed:

$$\mathbf{G}_{\pi} = \alpha\left(\mathbf{H}_{\pi} + \mathbf{D}\right) + (1-\alpha)\,\mathbf{E} \tag{9.1}$$

To calculated the PageRank centrality of all states, vector $\mathbf{R}$, the following procedures is performed iteratively until some convergence criterion is met:

$$\mathbf{R}_{k+1}^{T} = \mathbf{R}_{k}^{T}\mathbf{G} \tag{9.2}$$

A set of desired local states is defined, $\mathcal{S}_{des}$, which are problem specific. Finally, the fitness F of a policy is then calculated based on the average PageRank centrality of these desired states[1]. Note that in this chapter, we use R to denote the PageRank centrality and r to denote rewards in an MDP.

$$F = \sum_{s \in \mathcal{S}_{des}} R(s) \tag{9.3}$$

The Google Matrix is the transition matrix of the original graph As explained by Coppola [18], the Google Matrix is a stochastic matrix, which holds the probabilities of transitioning between nodes (or states) as it would be described by Markov chains. We argue that the original graph $G_{\mathcal{S}}$ is in fact the Markov Chain that is described by this Google Matrix. Therefore, Equation 9.2 describes the distribution propagation of state $\mathbf{R}_k$ to $\mathbf{R}_{k+1}$, where k indicates the timestep. Thus, given an initial state probability distribution $\mathbf{R}_0$, $R_k(s)$ describes the probability of being at state s at time k, and the final PageRank centrality simply approximates the probability of being in that state in the limit as time goes to infinity.

Fitness function as an average-reward Markov Chain At every time step in a Markov chain, the current state s transitions to a new state s'. It is possible that $s' = s$. Therefore, in the limit as time goes to infinity, probability of being in a state is equal to the probability of transitioning to that state: every time step that is spent in a state, has been preceded by transitioning to that state. The fitness function can therefore be transformed into a reward signal following the notation:

$$r(s') = r(s', s, a) = \begin{cases} 1, & \text{if } s' \in \mathcal{S}_{des} \\ 0, & \text{otherwise} \end{cases} \tag{9.4}$$

We can then define a return for this infinite horizon MDP:

$$G_t = \lim_{T\to\infty} \frac{\sum_{k=0}^{T} r(s_{k+t+1}, s_{k+t}, a_{k+t})}{T} \tag{9.5}$$

Intuitively, $G_t = F$. Therefore; finding a policy that maximizes the return G_t will also maximize the fitness of this policy.

[1] In the original paper, this contained some more terms, however; we argue that this can be simply reduced to multiplication with a constant since the sum of all PageRank centralities should be equal to 1.

The effect of the environment In the original problem description by Coppola et al. [18], the problem is formulated in such a way that there is an chance of α that the robot itself takes an action and there is a $1-\alpha$ chance that the environment takes, causes a state change instead. At first sight, this is not suitable for usage in an MDP, as it makes the definition of the state transition array P ambiguous: there are transitions that can happen without any action taking place. This, however; does not have to be a problem if we say that the agent always selects an action, but there is a chance of $1-\alpha$ that the state transition associated with this action does not get executed, but instead a state transition from the environment is executed. This is something that can modeled directly in the state transition array of a standard MDP.

Formulating PageRank as a linear programming problem As previously explained the PageRank centrality of state s, $R(s)$ is the steady state probability of being in this state. The fitness function can therefore be described as:

$$F = \sum_s R(s) r(s) \tag{9.6}$$

Which is equal to the total PageRank centrality of all states. We now call the steady state probability of being in a state s and taking action a to be $R(s,a)$. Then our policy becomes:

$$\pi(s,a) = \frac{R(s,a)}{R(s)} = \frac{R(s,a)}{\sum_{a'} R(s,a')} \tag{9.7}$$

As explained in the previous paragraph, we should be able to construct a state-transition array P with the following elements:

$$P^a_{ss'} = p\left(s', s, a\right) = \Pr\left\{S_t = s' | S_{t-1} = s, A_{t-1} = a\right\}$$

Finally, we can formulate this entire problem as a linear programming problem, removing the need of an evolutionary algorithm:

$$\begin{aligned} F = \quad \max \quad & \textstyle\sum_s \sum_a r(s) R(s,a) \\ \text{s.t.} \quad & \textstyle\sum_s \sum_a R(s,a) = 1 \\ & \textstyle\sum_a R(s',a) = \sum_s \sum_a R(s,a) p(s',s,a) \quad \forall s' \\ & R(s,a) \geq 0 \quad \forall s, \forall a \end{aligned} \tag{9.8}$$

Once $R(s,a)$ is found, the optimal policy can easily be obtained through Equation 9.7. Note that finding this optimal policy is exactly the same as globally maximizing the fitness function. The benefit of this notation as a Linear Programming problem is that an exact optimal policy can be found quickly compared to finding a near optimal policy through an evolutionary algorithm.

Effect of determinism There exist optimal solutions to the previously described linear programming problem that are deterministic. One of the assumptions that is made by the MDP model described by the PageRank approach is that the passive part of the system graph is independent of the used policy. This is not true when all agents utilize the same policy. Especially when a deterministic optimal policy is used, there can be a significant discrepancy between the expected performance based on the fitness function in the MDP and the true performance in a real swarming task. One workaround that can be used to reduce this effect is to add a source of stochasticity to policies, for example through using a policy that uses a weighted average between a uniform policy and the found optimal policy. This is the approach we follow in our experiments. Although this does reduce the fitness value of the policy, this can improve the performance of the real swarm. This indicates a discrepancy between the fitness function and the real swarm performance and will be demonstrated in the experiments in the next section.

Online Learning using Linear Programming The state transition array P used in the linear programming approach described in the previous section can be derived based on the assumptions on passive and active parts of the system graph. This is done in the offline version of PageRank as well as the linear programming approach. It is also possible to adjust this array based on experience gained whilst performing the swarming task. When this adapted array is then used to determine a new optimal policy, either through an EA or through LP, this creates an online learning version of the algorithm.

9.2. Experiments & Results

We evaluate the PageRank approach and linear programming approaches in the consensus task described by [18]. A number of agents are placed on a grid. Each agent starts with a randomly selected opinion from a set of possible opinions. At every time step, a random agent is selected that can decide to change its opinion to one of the other possible opinions or keep its opinion as it was. The goal is for all agents on the grid to get to the same opinion, a consensus, in the least amount of time steps. The difficulty lies in the fact that an agent can only observe the opinions of itself and its direct neighbours, not the opinions of all agents on the grid. Once full consensus is reached, an episode is considered finished. The environment is reset and the number of time steps it took to reach consensus is stored.

Experimental Comparison between PageRank and Linear Programming The first experiment is a comparison between the PageRank approach as described in [18] and a modified approach in which the Evolutionary Algorithm of the PageRank approach is replaced with a linear programming solver, as described in the previous subsection. As the optimal solution determined by the linear programming solver will largely be deterministic, an additional uniform policy is added to the optimal policy. The effect of this uniform policy and a comparison to the solution found through the standard PageRank approach are shown in Figure 9.1. The computing time needed to find the policy was 0.067 seconds for the LP approach and 83.8 seconds for the PageRank with EA approach. It is found that the Linear Programming approach reaches similar performance to the PageRank with EA approach. In addition, it is found that increasing the amount of stochasticity of the policy up to a certain degree improves the performance of the swarm. From this, it can be noticed that the local fitness in terms of PageRank centrality of desired states is not necessarily a good measure for real swarming performance. This indicates that there exists a discrepancy between the real swarming problem and the problem that is solved through the local MDP optimization.

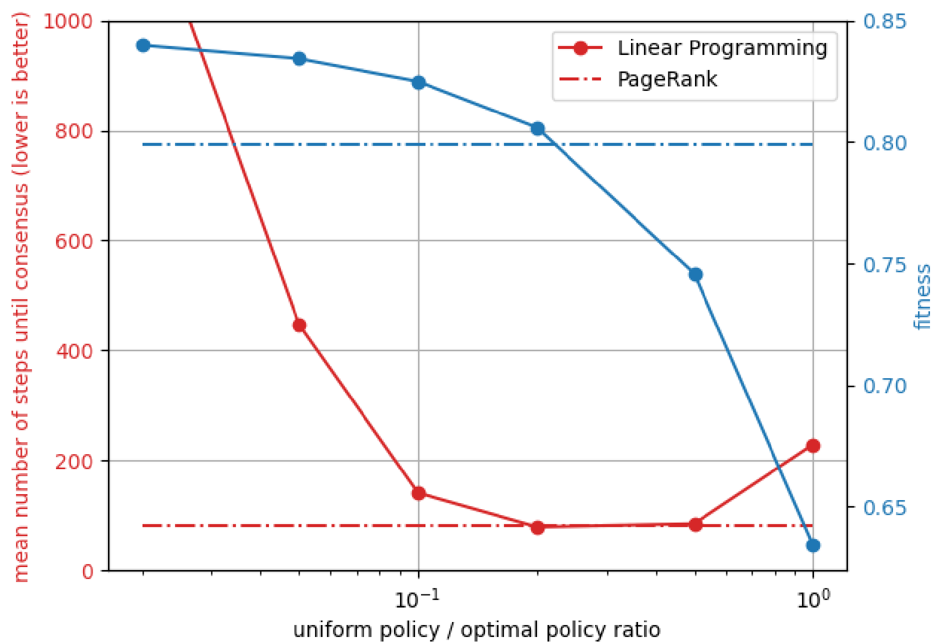


Figure 9.1: Comparison between the optimal solution found through Linear Programming and the solution found through PageRank with an Evolutionary Algorithm. For the Linear Programming solution, the used policy was a weighted average between the optimal policy and a uniform policy. The task is a consensus task of 10 robots with 2 possible opinions, more details on the task can be found in [18]. The performance is averaged over 250 random environment initializations.

Experimental Investigation of Online Learning with Linear Programming The second experiment serves as an investigation for the potential of online learning using a MDP representation as a model for a swarming task. We perform the previously described online LP approach where the P array is adjusted based on expe-

rience. Adaptation of the policy is performed once after every 100 episodes. This adaptation is done through solving the new LP problem described by the new P array. The results of this can be seen in Figure 9.2. The first few hundreds episodes experience a decrease in performance compared to the initial offline-learned policy. After this however, the number of steps needed to reach consensus decreased considerably.

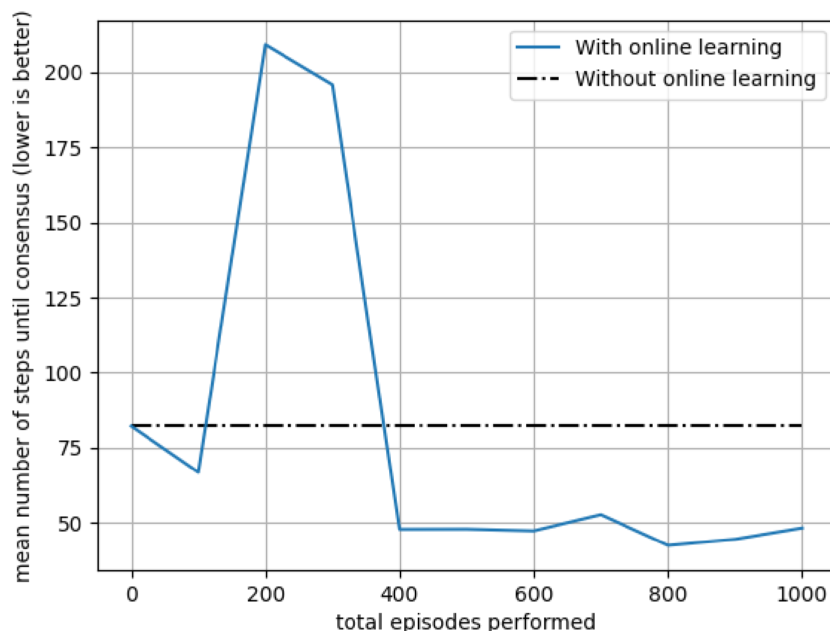


Figure 9.2: Comparison between the optimal solution found through Linear Programming and an approach that performs online learning in combination with Linear Programming. The task is a consensus task of 10 robots with 2 possible opinions, more details on the task can be found in [18].

9.3. PageRank Discussion & Conclusion

The PageRank approach described by Coppola seems to combine two key contributions:

1. Learning swarm behaviours purely based on a local MDP model of the environment. This requires a transformation of global rewards to local rewards, which is done through use of desired states.
2. Using an Evolutionary Algorithm to optimize the behaviour for the swarm.

Through analysis and experiments, we have shown that the Evolutionary Algorithm in the PageRank approach can be replaced with a Linear Programming solver for the MDP. In addition, we have demonstrated that online learning can improve the performance of the resulting policy compared to a fully offline-learned policy. The usage of local rewards in contrast to global rewards is a powerful idea that could aid in improving decentralization of RL. Through these experiments however; some limitations were also identified with the PageRank approach. The most prominent of which is the discrepancy between the local fitness and the real swarm performance. Even though online learning did improve the real swarm performance, perhaps decreasing this discrepancy, there still exist inherent limitations due to the usage of a tabular MDP model. The problem size that is solvable by such a model is limited. In addition, there is not a direct way to deal with partial observability or the effects of decentralization. Therefore; the remainder of this thesis will not directly pursue this model structure. Instead, we will investigate in more depth what model representations could be used in decentralized reinforcement learning and how these models can be used to learn policies in a more sample efficient manner. More specifically, we will focus on other contemporary model-based reinforcement learning algorithms and investigate how to modify them to make them applicable in decentralized domains.

10
Conclusion, Approach, Planning

This chapter serves two main purposes: summarizing what has been done and introducing what will be done. First, in Section 10.1 we take a look back at the work done in this preliminary study and briefly summarize the conclusions found both in the literature review as well as in the preliminary experiments. Second, in Section 10.2, Section 10.3 and Section 10.4, we translate these conclusions into an approach and planning that will be followed for the remainder of this thesis.

10.1. Conclusion

Swarm robotics is an important field of study, since swarms of robots offer inherent advantages compared to their single-robot counterparts: robustness, flexibility and scalability [1–3]. Designing individual behaviours for these swarm-robots is a complex task. To achieve more capable swarms, automatic swarm design methods must be considered. Reinforcement learning [4] is a promising technique to use and is already sucessfull for single agent-tasks [11, 12, 34, 43], multi-agent tasks [93] and swarming tasks[109–111]. To fully exploit the benefits of swarming, robots should be able to learn online, whilst acting in a real environment. Most reinforcement learning techniques however, take thousands of trials before the agent learns desiraopvragenble behaviours, rendering them unfeasible for learning on real robots [15]. Partial Observability and nonstationarity of the environment, typical phenomena encountered in multi-agent swarming tasks, complicate learning even further. One way to improve this so called sample-efficiency is through the use of learned world models, and has become increasingly popular in recent years [9, 59, 77, 79–83, 85–89]. Especially for tasks with continuous action spaces, typical for robots, these techniques have resulted in dramatic increases in sample efficiency. Although these model-based techniques have been applied in settings with partial observability, their applicability to decentralized multi-agent reinforcement learning has, according to the authors' best knowledge, not yet been investigated. As improving the sample efficiency is a crucial ingredient towards making online learning for swarm robotics possible, this thesis strives to answer the following research question:

Can sample efficiency of decentralized, multi-agent reinforcement learning be improved through the use of a learned world model?

One model-based swarming technique was investigated in more detail. This algorithm used PageRank and relied on formulation of the swarming problem from the perspective of a single agent as a tabular Markov Decision Process and optimized a policy within this MDP. It was found that there exists a discrepancy between the optimal solutions to the MDP and good solutions to the actual swarming problem. This raises the question whether a Tabular MDP is actually the best representation for a world model in a swarming problem. Not only due to the limitations on size of the problem when using a tabular representation, but also due to the inherent limitations of using a fully-observable MDP to represent a Decentralized, Partially Observable MDP.

The question what a suitable model structure and model usage to tackle decentralized model-based reinforcement learning with is a more fundamental question than how to adapt PageRank towards online learning and approaches could be found that do not suffer from the same limitations as the PageRank approach does. One key building block for this would be contemporary single agent deep model-based reinforcement

learning techniques. In these techniques, usage of an ensemble of neural networks to represent a world model has recently become more popular. These algorithms, such as MBPO [84] or PETS [83], allow for solving larger model-based reinforcement learning tasks. Yet, in a decentralized setting they might still suffer from the same convergency problems as when applying standard model-free algorithms or a tabular MDP approach. Interestingly, standard model-free algorithms do not always have worse practical performance in large decentralized tasks compared to more complex approaches of dealing with decentrality [94]. Similarly, direct application of model-based techniques on swarming problems might be a succesfull approach. Possibly, these algorithms could be adapted to be made more suitable for decentralized domains similar to how techniques such as hysteresis have made Q-learning more suitable for decentralized environments [99, 102]. A more specialized method for decentralized reinforcement learning would be to explicitly embed information about the decentralized nature of the environment within the model architecture. Such an approach could be desirable as this opens up the possibility for an agent to reason differently about the non-stationary collaborating agents compared to the stationary environment. On the other hand, such an approach does embed more specific priors on the model structure, possibly making the approach less generalizable. Both these approaches are worthy of further investigation and how this will be done is elaborated upon in the next sections.

10.2. General Approach

To answer the core research question, this thesis will design and evaluate model-based reinforcement learning algorithms for decentralized reinforcement learning. These decentralized techniques will be based on the model-based algorithms discussed in Chapter 6. Two primary model architectures are identified to make these algorithms suitable for decentralized learning:

1. *World models that implicitly take into account the existence of other agents.* In these world models, other agents are purely seen as part of the environment. Although this approach could be as simple as directly applying single-agent model based RL to a decentralized domain, it makes the environment that the world model needs to learn non-stationary. This might have negative consequences on the performance of these types of world models, similar to the issues discussed in Chapter 7.

2. *World models that explicitly take into account the existence of other agents.* Instead of modelling the other agents implicitly as part of the environment, it might also be possible to model them as an explicit part of the environment. One way to do this through splitting up the world model into two parts: an individual agents model that models the action selection of agents, and a model that represents the consequences of these actions on the environment. The action selection part will be the only non-stationary aspect of the model, and the effect of these actions on the environment will remain stationary. Such a world model places some additional constraints on the problem; the action selection of other agents should somehow be able to be derived from the observations. In addition, this model structure is more complex and deviates more from the methods describe in previous works.

Both approaches rely on modification of existing model-based reinforcement learning algorithms. Therefore; the first identified step is to implement these algorithms and evaluate the benefits of model-based reinforcement learning in single-agent domains. After this, the focus will shift towards adapting the algorithms for decentralized domains following the approaches explained above. A third topic of investigation can also be identified that focuses more on the learning process rather than the model structure:

3. *Hybrid models that utilize both model-based and model-free reinforcement learning.*

Although the first two topics could already provide opportunity for an interesting and novel investigation, this third topic might be able to improve performance. For this reason it will, for now, remain within the scope of this thesis.

10.3. Tasks & Nominal Planning

To be able to reach the desired goals of the thesis, designing and evaluating model-based decentralized RL algorithms, concrete tasks have been specified. These tasks are split up into two major categories. First, content tasks, which are the tasks that contribute to new insights and will form the majority of the actual contribution of this thesis. Second, auxiliary tasks, these tasks deal with the communication, organizatorial and writing tasks of the thesis. In addition, every subtask has been labelled according to the type of work that

it is: I(mplementation and design), E(xperimental) and W(riting). All these tasks have been scheduled over the remaining thesis period as shown in Figure 10.1. In the subsections below, the tasks corresponding to the numbering in the planning diagram are elaborated upon.

10.3.1. Content Tasks:

C1. **Create Necessary Infrastructure to perform analysis and evaluation of RL Algorithms**

- C1.1. I: Create single interface for single-agent and multi-agent algorithms.
- C1.2. I: Create infrastructure to quickly run experiments with different configurations.
- C1.3. I: Create suitable decentralized environments.

C2. **Analysis of Model-Based RL in single agent domains.** Desired outcome: a comparison between model-based and model-free RL that shows the benefit of model-based RL for sample efficiency in single agent reinforcement learning.

- C2.1. **I: Implement single agent RL algorithm (DDPG or SAC).**
  Potential difficulty: debugging can take a significant amount of time due to computing time required for running the algorithms and their tendency to fail silently.
- C2.2. **I: Implement model-based version of single-agent RL algorithm.** Potential difficulty: debugging can take a significant amount of time due to computing time required for running the algorithms and their tendency to fail silently.
- C2.3. **E: Create comparison between the model-based and model-free algorithms.** Potential difficulty: hyperparameter tuning can influence performance greatly. Tuning them can take significant time due to resource intensity of running the algorithms.

C3. **Design and Evaluation of Decentralized Model-Based RL using world models that implicitly take into account the existence of other agents.**

- C3.1. I: Design detailed algorithm
- C3.2. E: Evaluate performance on decentralized environments.

C4. **Design and Evaluation of Decentralized Model-Based RL using world models that explicitly take into account the existence of other agents.**

- C4.1. I: Design detailed algorithm
- C4.2. E: Evaluate performance on decentralized environments.

C5. **If no major delays in other tasks: Design and Evaluation of Hybrid Model-Based RL - Model-Free Algorithm that utilizes both simulated experience as well as real experience to learn.**

- C5.1. D: Design detailed algorithm
- C5.2. E: Evaluate performance on decentralized environments.

10.3.2. Auxiliary Tasks:

A1. **Prepare Preliminary**

- A1.1. W: Revise Planning & Approach
- A1.2. W: Finalize results section.
- A1.3. W: Proofreading & Formatting.
- A1.4. W: Prepare Preliminary Presentation.

A2. **Prepare Midterm**

A3. **Prepare Draft Thesis**

A4. **Prepare Final Thesis**

A5. **Prepare Defence**

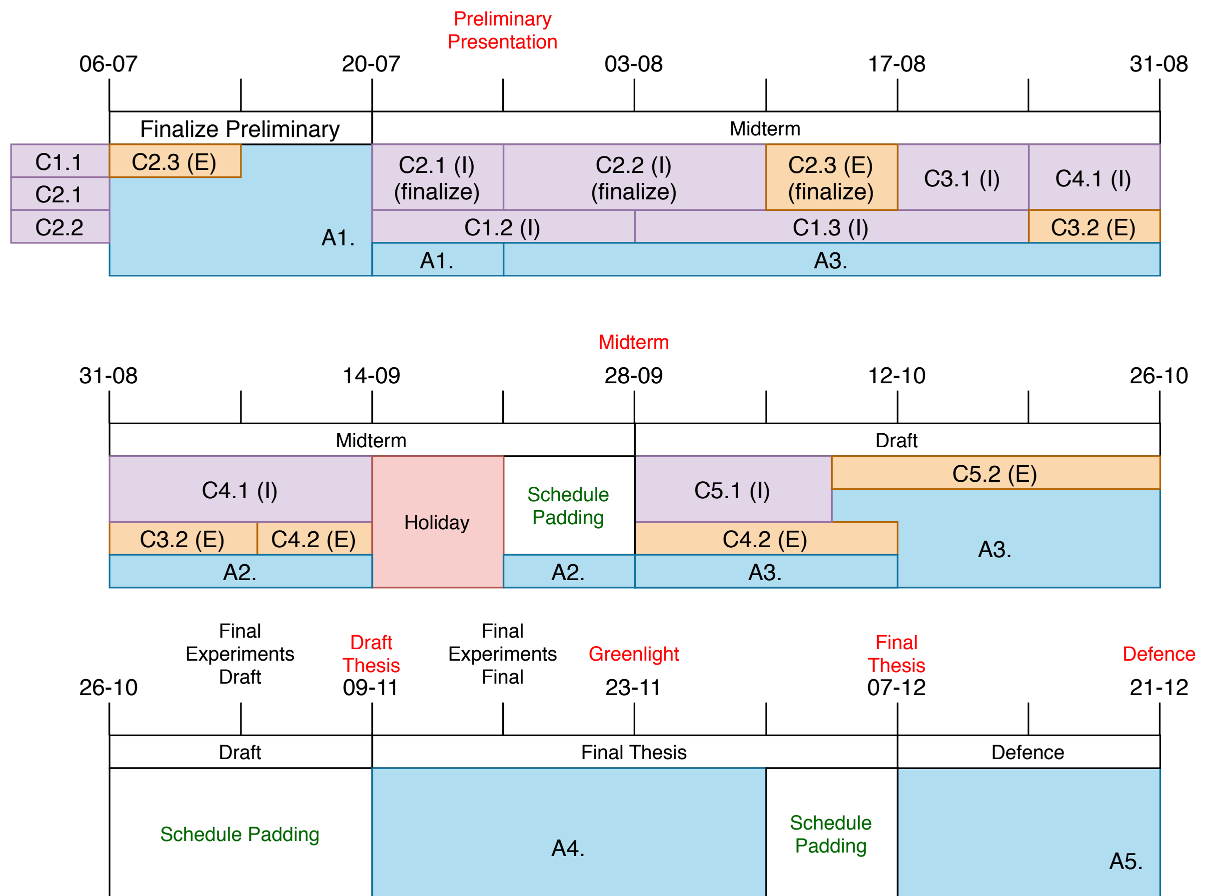


Figure 10.1: Planning for the remainder of this thesis

10.4. Expected Difficulties and Risks

A number of potential difficulties and risks have been identified with the previously described approach:

1. **Some task takes much longer than expected.**
   Risk mitigation: 1. Schedule some additional time in each phase of the thesis.

2. **Benefit of model-based RL is smaller than expected.**

3. **Computation time.** Doing experimentation can take a significant amount of time due to the running time of code. This is exacerbated by the potential need for extensive hyperparameter analysis.
   Risk mitigation: 1. Reduce computation time through finding small environments that are still an effective test-bed. 2. Schedule tasks that can be performed in parallel to computation of experiments. 3. Utilize machines with more computing power.

4. **Dependency on the right implementation details.** The performance of RL algorithms in general can be highly dependent on implementation details. Investigating these details and experimenting what works best can take a significant amount of time.
   Risk mitigation: 1. Create modular code in which implementation details can easily be swapped out.

5. **Debugging in combination with computation time.** Reinforcement learning is prone to failing silently. It could seem that the code works, and an agent learns very slowly, when in fact their are errors present in the code that reduce performance dramatically. These errors can be difficult to find due to their limited visibility.
   Risk mitigation: 1. Same risk mitigation as for the computation time risk. 2. Re-use code as much as possible. 3. Utilize git effectively.

Bibliography

[1] Iñaki Navarro and Fernando Matía. An Introduction to Swarm Robotics. *ISRN Robotics*, 2013.

[2] Jan Carlo Barca and Y. Ahmet Sekercioglu. Swarm robotics reviewed. *Robotica*, 31(3):345–359, 2013.

[3] Erol Şahin. Swarm Robotics: From Sources of Inspiration to Domains of Application. In Erol Şahin and William M. Spears, editors, *Swarm Robotics*, Lecture Notes in Computer Science, pages 10–20, Berlin, Heidelberg, 2005. Springer.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, and Adrian Bolton. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[6] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, and Thore Graepel. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

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

[8] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, and Thore Graepel. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[9] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, and Thore Graepel. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

[10] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv preprint arXiv:1511.05952*, 2016.

[11] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *arXiv preprint arXiv:1710.02298*, 2017.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

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

[14] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[15] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[16] Marc Deisenroth and Carl E. Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011.

[17] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep Reinforcement Learning for Multiagent Systems: A Review of Challenges, Solutions, and Applications. *IEEE Transactions on Cybernetics*, pages 1–14, 2020.

[18] M. Coppola, J. Guo, E. Gill, and G. C. H. E. de Croon. The PageRank algorithm as a method to optimize swarm behavior through local analysis. *Swarm Intelligence*, 13(3):277–319, 2019.

[19] Erol Sahin, Sertan Girgin, Levent Bayindir, and Ali Turgut. Swarm Robotics. *Swarm Intelligence*, pages 87–100, 2008.

[20] Vito Trianni, Stefano Nolfi, and Marco Dorigo. Evolution, Self-organization and Swarm Robotics. In Christian Blum and Daniel Merkle, editors, *Swarm Intelligence: Introduction and Applications*, Natural Computing Series, pages 163–191. Springer, Berlin, Heidelberg, 2008.

[21] Gregory Dudek, Michael R. M. Jenkin, Evangelos Milios, and David Wilkes. A taxonomy for multi-agent robotics. *Autonomous Robots*, 3(4):375–397, 1996.

[22] Luca Iocchi, Daniele Nardi, and Massimiliano Salerno. Reactivity and Deliberation: A Survey on Multi-Robot Systems. In Markus Hannebauer, Jan Wendler, and Enrico Pagello, editors, *Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, Lecture Notes in Computer Science, pages 9–32, Berlin, Heidelberg, 2001. Springer.

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

[24] Manuele Brambilla, Carlo Pinciroli, Mauro Birattari, and Marco Dorigo. Property-driven design for swarm robotics. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, pages 139–146. 2012.

[25] William M. Spears, Diana F. Spears, Jerry C. Hamann, and Rodney Heil. Distributed, Physics-Based Control of Swarms of Vehicles. *Autonomous Robots*, 17(2):137–162, 2004.

[26] Martin Saska, Jan Vakula, and Libor Preucil. Swarms of micro aerial vehicles stabilized under a visual relative localization. pages 3570–3575, 2014.

[27] Mario Coppola, Kimberly Mcguire, Christophe De Wagter, and Guido Croon. A survey on swarming with micro air vehicles: fundamental challenges and constraints. *Frontiers in Robotics and AI*, 7:18, 2020.

[28] Stephane Doncieux, Nicolas Bredeche, Jean-Baptiste Mouret, and Agoston E. (Gusz) Eiben. Evolutionary Robotics: What, Why, and Where to. *Frontiers in Robotics and AI*, 2, 2015.

[29] Freek Stulp and Olivier Sigaud. Robot skill learning: From reinforcement learning to evolution strategies. *Paladyn, Journal of Behavioral Robotics*, 4(1):49–61, 2013.

[30] David Silver. Lecture 2: Markov Decision Processes. *UCL. Retrieved from www0. cs. ucl. ac. uk/staff/d. silver/web/Teaching_files/MDP. pdf*, 2015.

[31] Marco Wiering and Martijn van Otterlo, editors. *Reinforcement Learning: State-of-the-Art*. Adaptation, Learning, and Optimization. Springer-Verlag, Berlin Heidelberg, 2012.

[32] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., USA, 1st edition, 1994.

[33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

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

[35] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[36] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv preprint arXiv:1511.06581*, 2016.

[37] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 449–458. JMLR. org, 2017.

[38] Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[39] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit Quantile Networks for Distributional Reinforcement Learning. *arXiv preprint arXiv:1806.06923*, 2018.

[40] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, and Olivier Pietquin. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[41] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[42] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. 2018.

[43] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[44] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust Region Policy Optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

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

[46] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[47] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning*, pages 387–395, 2014.

[48] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva Tb, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.

[49] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[50] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[51] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

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

[53] Verena Heidrich-Meisner and Christian Igel. Neuroevolution strategies for episodic reinforcement learning. *Journal of Algorithms*, 64(4):152–168, 2009.

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

[55] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[56] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, and Andrew Lefrancq. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[57] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[58] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[59] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

[60] Hado Van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648*, 2018.

[61] Rasool Fakoor, Pratik Chaudhari, and Alexander J. Smola. DDPG++: Striving for Simplicity in Continuous-control Off-Policy Reinforcement Learning. *arXiv preprint arXiv:2006.15199*, 2020.

[62] G de Croon, M F van Dartel, and E O Postma. Evolutionary Learning Outperforms Reinforcement Learning on Non-Markovian Tasks. page 12.

[63] Kamyar Azizzadenesheli, Alessandro Lazaric, and Animashree Anandkumar. Open Problem: Approximate Planning of POMDPs in the class of Memoryless Policies. *arXiv preprint arXiv:1608.04996*, 2016.

[64] D. Wierstra, A. Forster, J. Peters, and J. Schmidhuber. Recurrent policy gradients. *Logic Journal of IGPL*, 18(5):620–634, 2010.

[65] Xiujun Li, Lihong Li, Jianfeng Gao, Xiaodong He, Jianshu Chen, Li Deng, and Ji He. Recurrent Reinforcement Learning: A Hybrid Approach. *arXiv:1509.03044*, 2015.

[66] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[67] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

[68] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A Survey of Deep Reinforcement Learning in Video Games. *arXiv preprint arXiv:1912.10944*, 2019.

[69] C.G. Atkeson and J.C. Santamaria. A comparison of direct and model-based reinforcement learning. In *Proceedings of International Conference on Robotics and Automation*, volume 4, pages 3557–3564 vol.4, 1997.

[70] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Model-based Reinforcement Learning: A Survey. *arXiv preprint arXiv:2006.16712*, 2020.

[71] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Learning Multimodal Transition Dynamics for Model-Based Reinforcement Learning. *arXiv preprint arXiv:1705.00470*, 2017.

[72] Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. 1991.

[73] David Silver, Richard S. Sutton, and Martin Müller. Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th international conference on Machine learning - ICML '08*, pages 968–975, Helsinki, Finland, 2008. ACM Press.

[74] Richard S. Sutton, Csaba Szepesvári, Alborz Geramifard, and Michael P. Bowling. Dyna-style planning with linear function approximation and prioritized sweeping. *arXiv preprint arXiv:1206.3285*, 2012.

[75] Kavosh Asadi Atui. Strengths, Weaknesses, and Combinations of Model-based and Model-free Reinforcement Learning. page 59.

[76] Baolin Peng, Xiujun Li, Jianfeng Gao, Jingjing Liu, Kam-Fai Wong, and Shang-Yu Su. Deep Dyna-Q: Integrating Planning for Task-Completion Dialogue Policy Learning. *arXiv preprint arXiv:1801.06176*, 2018.

[77] Ryan Faulkner and Doina Precup. Dyna planning using a feature based generative model. *arXiv preprint arXiv:1805.10129*, 2018.

[78] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In *Advances in Neural Information Processing Systems*, pages 6118–6128, 2017.

[79] Yarin Gal, Rowan McAllister, and Carl Edward Rasmussen. Improving PILCO with Bayesian neural network dynamics models. In *Data-Efficient Machine Learning workshop, ICML*, volume 4, page 34, 2016.

[80] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014.

[81] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566, 2018.

[82] Gabriel Kalweit and Joschka Boedecker. Uncertainty-driven imagination for continuous deep reinforcement learning. In *Conference on Robot Learning*, pages 195–206, 2017.

[83] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models. *arXiv preprint arXiv:1805.12114*, 2018.

[84] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems*, pages 12498–12509, 2019.

[85] Manuel Watter, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. In *Advances in neural information processing systems*, pages 2746–2754, 2015.

[86] Ershad Banijamali, Rui Shu, Mohammad Ghavamzadeh, Hung Bui, and Ali Ghodsi. Robust locally-linear controllable embedding. *arXiv preprint arXiv:1710.05373*, 2017.

[87] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*, 2018.

[88] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[89] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. Treeqn and atreec: Differentiable tree-structured models for deep reinforcement learning. *arXiv preprint arXiv:1710.11417*, 2017.

[90] Adrian Šošić, Wasiur R. KhudaBukhsh, Abdelhak M. Zoubir, and Heinz Koeppl. Inverse Reinforcement Learning in Swarm Systems. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, pages 1413–1421, São Paulo, Brazil, 2017. International Foundation for Autonomous Agents and Multiagent Systems.

[91] Xueguang Lu and Christopher Amato. Decentralized Likelihood Quantile Networks for Improving Performance in Deep Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:1812.06319*, 2019.

[92] Nancy Fulda and Dan Ventura. Predicting and Preventing Coordination Problems in Cooperative Q-learning Systems. In *IJCAI*, volume 2007, pages 780–785, 2007.

[93] Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in neural information processing systems*, pages 2137–2145, 2016.

[94] Erik Zawadzki, Asher Lipson, and Kevin Leyton-Brown. Empirically evaluating multiagent learning algorithms. *arXiv preprint arXiv:1401.8074*, 2014.

[95] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4), 2017.

[96] Shayegan Omidshafiei, Jason Pazis, Christopher Amato, Jonathan P. How, and John Vian. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2681–2690, Sydney, NSW, Australia, 2017. JMLR.org.

[97] Afshin OroojlooyJadid and Davood Hajinezhad. A Review of Cooperative Multi-Agent Deep Reinforcement Learning. *arXiv preprint arXiv:1908.03963*, 2019.

[98] Martin Lauer and Martin Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer, 2000.

[99] Laetitia Matignon, Guillaume J. Laurent, and Nadine Le Fort-Piat. Hysteretic Q-learning : an algorithm for Decentralized Reinforcement Learning in Cooperative Multi-Agent Teams. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 64–69, 2007.

[100] Liviu Panait, Keith Sullivan, and Sean Luke. Lenient learners in cooperative multiagent systems. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems - AAMAS '06*, page 801, Hakodate, Japan, 2006. ACM Press.

[101] Ermo Wei and Sean Luke. Lenient Learning in Independent-Learner Stochastic Cooperative Games. *Journal of Machine Learning Research*, 17(84):1–42, 2016.

[102] Gregory Palmer, Karl Tuyls, Daan Bloembergen, and Rahul Savani. Lenient Multi-Agent Deep Reinforcement Learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, pages 443–451, Stockholm, Sweden, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

[103] Ryan Lowe, Yi I. 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*, pages 6379–6390, 2017.

[104] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, and Karl Tuyls. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *Proceedings of the 17th international conference on autonomous agents and multiagent systems*, pages 2085–2087. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[105] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485*, 2018.

[106] Yu Fan Chen, Miao Liu, Michael Everett, and Jonathan P. How. Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 285–292, 2017.

[107] Michael Everett, Yu Fan Chen, and Jonathan P. How. Motion Planning Among Dynamic, Decision-Making Agents with Deep Reinforcement Learning. *arXiv preprint arXiv:1805.01956*, 2018.

[108] Pinxin Long, Tingxiang Fan, Xinyi Liao, Wenxi Liu, Hao Zhang, and Jia Pan. Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6252–6259, 2018.

[109] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. Guided Deep Reinforcement Learning for Swarm Systems. *arXiv preprint arXiv:1709.06011*, 2017.

[110] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. Local Communication Protocols for Learning Complex Swarm Behaviors with Deep Reinforcement Learning. In Marco Dorigo, Mauro Birattari, Christian Blum, Anders L. Christensen, Andreagiovanni Reina, and Vito Trianni, editors, *Swarm Intelligence*, Lecture Notes in Computer Science, pages 71–83, Cham, 2018. Springer International Publishing.

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

[112] Christian Schroeder de Witt, Bei Peng, Pierre-Alexandre Kamienny, Philip Torr, Wendelin Böhmer, and Shimon Whiteson. Deep multi-agent reinforcement learning for decentralized continuous cooperative control. *arXiv preprint arXiv:2003.06709*, 2020.

[113] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The starcraft multi-agent challenge. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2186–2188. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[114] Igor Mordatch and Pieter Abbeel. Emergence of grounded compositional language in multi-agent populations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.