Guided Deep Reinforcement Learning for Robot Swarms

Guided Deep Reinforcement Learning für Roboter Schwärme

Master-Thesis von Maximilian Hüttenrauch aus Rüdesheim am Rhein

August 2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Guided Deep Reinforcement Learning for Robot Swarms
Guided Deep Reinforcement Learning für Roboter Schwärme

Vorgelegte Master-Thesis von Maximilian Hüttenrauch aus Rüdesheim am Rhein

1. Gutachten: Prof. Dr. Gerhard Neumann
2. Gutachten: Prof. Dr.-Ing. Abdelhak Zoubir
3. Gutachten: M.Sc. Adrian Šošić

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 7. August 2016

____

(Maximilian Hüttenrauch)

Thesis Statement

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, August 7, 2016

____

(Maximilian Hüttenrauch)

Abstract

In this thesis, we provide the framework of multi-agent guided deterministic policy gradient for reinforcement learning in cooperative multi-agent systems. The multi-agent scenario is formulated as a decentralized partially observable Markov decision process with continuous state and action spaces and homogeneous agents.

After deriving the theory of multi-agent deterministic policy gradient, we show two possible algorithmic implementations: the multi-agent guided deep deterministic policy gradient algorithm and the multi-agent recurrent deterministic policy gradient algorithm. For the implementation of the first algorithm we use feed-forward neural networks to approximate both the Q-function and the policy, in the second one we use recurrent neural networks to model the policy.

The algorithms are tested in a simulated robot swarm environment. The agents are modeled to move and communicate inspired by the abilities of Kilobots, a small-scale swarm robotics platform. The performance of the algorithms is evaluated on the task of cooperative localization. Experimental results indicate that system performance increases with the number of agents involved during learning.

Zusammenfassung

In der vorliegenden Masterarbeit stellen wir das Konzept des ‚multi-agent guided deterministic policy gradient‘ für Reinforcement Learning in kooperativen Multi-Agenten Systemen vor. Das Multi-Agenten Szenario wird als dezentralisiertes partiell beobachtbares Markov Entscheidungsproblem mit kontinuierlichen Zustands- und Aktionsräumen und homogenen Agenten formuliert.

Zunächst wird die Theorie für den ‚multi-agent deterministic policy gradient‘ hergeleitet. Anschließend zeigen wir zwei mögliche Implementierungen: den ‚multi-agent guided deep determinisitc policy gradient‘ Algorithmus und den ‚multi-agent guided recurrent policy gradient‘ Algorithmus. In der Implementierung des ersten Algorithmus werden sowohl die Q-Funktion als auch die Policy durch vorwärtsgerichtete neuronale Netze approximiert, im zweiten Algorihtmus wird die Policy mit rekurrenten neuronalen Netzen modelliert.

Die Algorithmen werden in einer simulierten Schwarmroboter Umgebung getestet. Die Fähigkeiten der Agenten sind an die Bewegungs- und Kommunikationsmöglichkeiten von Kilobots, einer kleinformatigen Schwarmroboter Plattform, angelehnt. Der Algorithmus wird an Hand der Aufgabe einen Ziel Agenten kooperativ zu finden analysiert. Die Versuchsergebnisse deuten darauf hin, dass die Aufgabe besser erfüllt wird, je mehr Agenten Teil des Lernprozesses sind.

Contents

1. Introduction 1
1.1. Related Work 2
1.2. Outline 3

2. Deep Reinforcement Learning 4
2.1. Reinforcement Learning 4
2.2. Deep Learning 7
2.3. Deep Reinforcement Learning 11

3. Guided Deep Reinforcement Learning for Homogeneous Multi-Agents Systems 18
3.1. The Decentralized POMDP Framework 18
3.2. Multi-Agent Guided Deterministic Policy Gradient 19
3.3. Multi-Agent Guided Deep Deterministic Policy Gradient 19
3.4. Multi-Agent Guided Recurrent Deterministic Policy Gradient 20

4. Application in Simulated Robot Swarms 24
4.1. Implementation 24
4.2. Task: Cooperative Localization 25
4.3. Experimental Setup 26

5. Results and Evaluation 29
5.1. Evaluation of the MAGDDPG Algorithm 29
5.2. Evaluation of the MAGRDPG Algorithm 41

6. Conclusion and Future Work 42

Bibliography 43

A. Appendix 47
A.1. Multi-Agent Guided Deterministic Policy Gradient 47

Figures and Tables

List of Figures

1.1. A Kilobot robot. 2

2.1. Simple model of an artificial neural network. 8
2.2. One neuron with three inputs and one output. 9
2.3. Examples of common activation functions. 9
2.4. Flow diagram of an LSTM cell. 10

4.1. 11 basis functions with standard deviation $\sigma = 0.25$ equally spread over the communication radius. 25
4.2. Network architectures for the MAGDDPG algorithm. 27
4.3. Actor using recurrent LSTM. 28

5.1. Mean learning curve for one-agent learning scenario, shown with two times standard deviation. 30
5.2. Mean learning curve for two-agent learning scenario, shown with two times standard deviation. 30
5.3. Mean learning curve for three-agent learning scenario, shown with two times standard deviation. 31
5.4. Mean learning curve for four-agent learning scenario, shown with two times standard deviation. 31
5.5. Mean learning curve for six-agent learning scenario, shown with two times standard deviation. 31
5.6. Mean learning curve for eight-agent learning scenario, shown with two times standard deviation. 32
5.7. Mean learning curve for six-agent learning scenario, shown with two times standard deviation (decreased base learning rate). 32
5.8. Mean learning curve for eight-agent learning scenario, shown with two times standard deviation (decreased base learning rate). 32
5.9. Trajectories of policies learned and executed by four agents after different numbers of update iterations. 33
5.10. Trajectories of policies learned and executed by four agents after different numbers of update iterations. 34
5.11. Mean reward over time in a scenario of 16 agents using policies learned by 1, 2, 3, 4, 6 and 8 agents. 36
5.12. Boxplots of returns in a scenario of 16 agents using policies learned by 1, 2, 3, 4, 6 and 8 agents. 36
5.13. Mean reward over time in a scenario of 32 agents using policies learned by 1, 2, 3, 4, 6 and 8 agents. 37
5.14. Boxplots of returns in a scenario of 32 agents using policies learned by 1, 2, 3, 4, 6 and 8 agents. 37
5.15. Mean learning curve for two-agent learning scenario with a horizon of 1, shown with two times standard deviation. 38

5.16. Mean learning curve for two-agent learning scenario with a horizon of 2, shown with two times standard deviation. 39
5.17. Mean learning curve for two-agent learning scenario with a horizon of 3, shown with two times standard deviation. 39
5.18. Mean learning curve for two-agent learning scenario with a horizon of 4, shown with two times standard deviation. 39
5.19. Mean learning curve for two-agent learning scenario with a horizon of 5, shown with two times standard deviation. 40
5.20. Mean learning curve for two-agent learning scenario with a horizon of 10, shown with two times standard deviation. 40

List of Tables

2.1. Comparison between deep reinforcement learning algorithms. 17

Abbreviations, Symbols and Operators

List of Abbreviations

Notation	Description
BPTT	backpropagation through time
DDPG	deep deterministic policy gradient
Dec-POMDP	decentralized partially observable Markov decision process
DPG	deterministic policy gradient
DQN	deep Q-network
LSTM	long short-term memory
MAGDDPG	multi-agent guided deep deterministic policy gradient
MAGRDPG	multi-agent guided recurrent deterministic policy gradient
MAS	multi-agent system
MDP	Markov decision process
POMDP	partially observable Markov decision process
RDPG	recurrent deterministic policy gradient
RNN	recurrent neural network
SGD	stochastic gradient descent

1 Introduction

Nature provides many examples where the resulting effort of a collective of limited beings exceeds the capabilities of one individual. Ants transport prey of the size no single ant could carry, termites build nests of up to nine meters in height and bees are able to regulate the temperature of a hive. Common to all these phenomena is the fact that each individual has only basic and local sensing of its environment and limited communication capabilities.

A field of research where cooperative behavior between multiple agents can be studied in a well-defined environment is swarm robotics. A common approach to decentralized problem solving is to extract rules from the observed behavior. Kube et al [1], for example, investigate cooperative prey retrieval of ants to infer rules on how a swarm of robots can fulfill the task of cooperative box-pushing. Further work may be found in [2, 3, 4].

In this thesis, we follow a different approach to solving cooperative multi-agent tasks. Instead of finding distinct rules we concentrate on a reinforcement learning solution based directly on the locally sensed information of an agent. This is a challenging task, since the dimensionality of the problem grows exponentially with increasing numbers of agents. Furthermore, the environment that an agent perceives is no longer static since observations change, depending not only on the current local state of the agent, but also depending on the state of all other agents. One novelty of our approach is to use a compact representation of the full system state to learn the Q-function while the actions of the agents are solely based on the local observation of each agent. In theory, this should greatly simplify the task of learning the Q-function since the full system state obeys the Markov property.

Our approach follows recent developments in deep reinforcement learning, which successfully combines techniques from deep learning with algorithms from reinforcement learning. Particularly appealing to this approach is the ability to learn policies in an end-to-end fashion, as mappings from high-dimensional sensory input to actions, without the need for complex feature engineering. The framework of policy gradient methods is especially useful since not only a Q-function (i.e. a measure of quality of actions taken in a certain state under a certain policy), but also a distinct parametrized policy is learned. Using deep neural networks as function approximators for the Q-function and the policy, diverse problems, some of which will be presented in Section 1.1, have already been solved.

The scope of this thesis is to develop a deep reinforcement learning algorithm for cooperative multi-agent systems which can handle continuous state and actions spaces. The performance of the algorithm will be evaluated on a simulated robot swarm environment. The agents have capabilities which are based on a real robot platform called Kilobot. A short introduction to multi-agent systems and Kilobot robots follows hereafter.

Multi-Agent Systems

Environments with more than one agent present are called *multi-agent systems (MAS)*. The agents' perception of the environment can be complete but also limited by their sensing abilities. Depending on the goals of the agents and how they interact, one can identify different types. Agents in a MAS can act *cooperatively* to fulfill a common task or *competitively* to satisfy their own interests. Cooperative systems can be further categorized on how to achieve the goal. The first one is *team learning* where a single learner discovers a joint solution for all agents to solve the task. Team learning can further be differentiated into three different approaches. *Homogeneous team learning*, where the learning mechanism tries to

find a solution for agents with identical behavior, *heterogeneous team learning*, for agents with different behavior, and *Hybrid team learning*, which separates the agents into a smaller number of groups such that each group can discover its own behavior. The second one is *concurrent learning* and is basically the opposite of team learning. Here, each agent is assigned its own behavior by a learning routine for each agent [5].

Kilobots

The *Kilobot* robot platform provides an environment where the mechanisms of such systems can be applied and evaluated. These robots are of small scale, comparatively low cost and are equipped with very basic capabilities like differential drive locomotion, ambient light sensing, and the ability to communicate with each other in a small radius. The Kilobot (Figure 1.1) is a three legged robot on a 33 mm diameter circular base. It moves with the help of two vibrating motors which can be controlled independently. These motors can be set to 255 different power levels to let the robot move forward with a maximum speed of 1 cm/s and turn with 45 deg/s. On-board computing is possible with an ATmega 328p 8bit processor at 8 MHz and 32 kB flash and 1 kB EEPROM memory. An ambient light sensor is placed on top of the robot and two infrared sensors, one for transmitting and one for receiving messages, beneath. Messages travel by reflection off the ground surface up to a distance of 7 cm, depending on the composition. It is also possible to determine the distance (but not the direction of arrival) to the transmitting robot based on the intensity of the received message's signal. Furthermore, a Kilobot can sense its own battery voltage level. Finally, each robot has an RGB LED with three levels of brightness pointed upward. An API makes it possible to control the basic functions of the robot such as motor speed, distance measurement and the communication module. Rule based behaviors can be stored as files and executed on the robot.

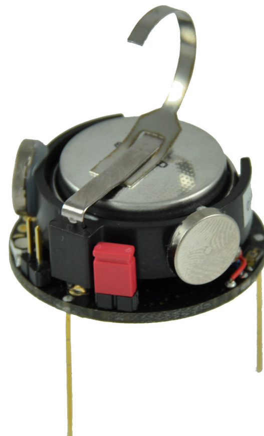


Figure 1.1.: A Kilobot robot.

1.1 Related Work

In this Section, we present some of the work whose core ideas are kept or extended in this thesis and point out, which problems the authors were able to solve.

The framework which is probably most famous among current deep reinforcement learning algorithms is *deep Q-networks (DQN)* [6, 7]. Mnih et al were able to learn policies based on a learned Q-function for Atari 2600 games solely on the information provided by the game screen and score using an algorithm called deep Q-learning. Policies for 49 different games were learned with a single set of hyper parameters. The algorithm takes downscaled screenshots of a game and uses deep convolutional neural networks to obtain a lower dimensional representation of current and past states of a game. Using this

representation they learned a Q-function for a set of discrete actions and were able to outperform previous approaches in 43 out of 49 games tested. Furthermore, the learned policies perform as well or even better than a skilled human player in 29 games.

A desirable property for algorithms in many problems, for example robotics, is the ability to cope with continuous actions spaces. Instead of discretizing the action space, which results in an exponential growth in the number of actions per dimension added, another approach taken by Lillicrap et al [8] is to combine policy gradient methods with the ideas behind DQN. They provide a framework called *deep deterministic policy gradient (DDPG)* based on the *deterministic policy gradient (DPG)* by Silver et al [9], where they use deep neural networks to learn a Q-function and a distinct policy at the same time. They evaluated their algorithm in a physics simulator called MuJoCo on classical reinforcement learning tasks, such as cartpole or locomotion tasks. Policies were learned on low dimensional state representations, such as joint positions and velocities, as well as on high dimensional visual input in the same fashion as DQN.

An extension of DDPG to partially observable problems are *recurrent deterministic policy gradient (RDPG)* by Heess et al [10]. This extension is particularly interesting since in the later introduced multi-agent problem the true state is not available to the agents to base their decisions on. In their paper they solve a simplified version of the Morris water maze [11], where a single agent is looking for a hidden platform in a closed two-dimensional state space. This problem is similar to the task treated in this thesis, which will be formulated for multiple agents.

Some work is already done combining deep reinforcement learning with multi-agent scenarios, for example Tampuu et al [12] use the DQN framework to independently train two agents to play the game of Pong by adjusting the reward scheme for both agents. In this fashion they were able to learn both competitive as well as cooperative behavior.

Finally, Foerster et al [13] propose *deep distributed recurrent Q-networks* in order to solve communication-based coordination tasks in partially observable multi-agent tasks.

1.2 Outline

This thesis is structured as follows. We start by introducing foundations of machine learning and important concepts of deep reinforcement learning in Chapter 2. The algorithms developed in the course of this thesis are presented in Chapter 3. Chapters 4 and 5 show the task, on which the algorithms were tested, and the results we were able to produce. Finally, Chapter 6 concludes this thesis and gives an overview of open questions and problems.

2 Deep Reinforcement Learning

Machine learning is a field of computer science whose goal it is to let computers learn to make decisions based on collected data without explicitly programming how these decisions are made. The decisions can be as simple as deciding between whether a sample of data belongs to one class or another, to recognizing speech or faces in images, or telling you what movie you probably would like to watch next on Netflix. A machine learning task can be roughly categorized by the way the computer learns patterns in data. If we have labeled training data the task category is called *supervised learning* and one can further differentiate between *classification* and *regression*, depending on if the computer will predict discrete or continuous values. Without labels it is called *unsupervised learning*. If we get feedback for the decisions a program makes and an algorithm can improve on these, the category is called *reinforcement learning*. Common to all these problems is the fact that almost always the program will not see every possible variation of data that exists and thus has to generalize beyond the samples that it is trained on.

Our contribution is in the field of *deep reinforcement learning*, a new approach to machine learning combining techniques from supervised learning and reinforcement learning, which has become popular over the last years. This chapter first introduces the main ideas of reinforcement learning and deep learning as well as basic algorithms from these fields. Subsequently, deep reinforcement algorithms on which the algorithm in this thesis is built on are shown.

2.1 Reinforcement Learning

Reinforcement learning is a general mathematical framework where an *agent* (the learner or decision maker) interacts with an *environment* (everything that is outside the control of the agent) whose goal is to learn how to maximize a performance measure, or to minimize a cost respectively, based on the interactions. In addition to the agent and the environment one can identify four main subelements of a reinforcement learning system [14]: a *policy*, which defines how the agent reacts (with its actions) to the current state of the environment; a *reward* signal, indicating how good or bad an action is in a certain state; *value functions*, an expectation of the long term reward under the current policy starting from a certain state; and, optionally, a *model* of the transition dynamics of the environment.

At each point in time the agent receives a representation of the environment E's current *state* s_t from a set of all possible states $\mathcal{S}$ and, based on this state, chooses an *action* a_t from the set of all possible actions $\mathcal{A}$. The agent's policy gives a mapping from states to actions. It can either be a stochastic policy $\pi(a|s)$, which is the probability of taking action $a_t = a$ when being in state $s_t = s$, or a deterministic one, where $a = \mu(s)$. In the next time step the agent receives the scalar reward $r_t = R(s_t, a_t)$ from the set of rewards $\mathcal{R} \subset \mathbb{R}$. The transition dynamics are in general expressed as a probability distribution

$$\Pr\left\{s_{t+1} = s' \mid s_0, a_0, s_{t-1}, \ldots, a_{t-1}, s_t, a_t\right\},$$

where the next state s_{t+1} is dependent on all previous sates and actions. If the next state can be determined only using the current state and the current action, the environment is said to be *Markovian* and the transition dynamics reduce to

$$p(s'|s,a) := \Pr\left\{s_{t+1} = s' \mid s_t = s, a_t = a\right\}.$$

The reinforcement learning task can be expressed as a *Markov decision process (MDP)* which is a tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$. If the set of states $\mathcal{S}$ and the set of actions $\mathcal{A}$ are finite the MDP is also said to be finite.

The core problem is to find a policy for the agent which maximizes the cumulative long term reward. In the case of *episodic* tasks of length T it is the *return* $G_t = \sum_{k=0}^{T} r_{t+k}$. In case of tasks with infinite length we consider the *discounted return* $G_t^\gamma = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, where $\gamma \in [0,1)$ is the so-called *discount factor* ensuring convergence of the sum.

2.1.1 Value Functions

The return is used to define a *state-value function* $v_\pi(s)$ indicating how good it is to be in state s under policy π, and an *action-value function* $q_\pi(s,a)$ indicating how good it is to take action a in state s under policy π in terms of the expected return. The action-value function is also called *Q-function*. For MDPs considering the discounted return these functions are expressed as

$$v_\pi(s) := \mathbb{E}_\pi[G_t^\gamma \mid s_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s\right]$$

and

$$q_\pi(s,a) := \mathbb{E}_\pi[G_t^\gamma \mid s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s, a_t = a\right],$$

where the expectation is taken with respect to the states the agents visit under policy π which can be dropped if the policy is deterministic. Both value functions can also be expressed recursively, i.e.

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[G_t^\gamma \,\middle|\, s_t = s\right] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_t + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_t + \gamma \mathbb{E}_\pi[G_{t+1}^\gamma \,\middle|\, s_{t+1} = s']\right] \\
&= \mathbb{E}_\pi\left[r_t + \gamma v_\pi(s')\right], \qquad (2.1)
\end{aligned}$$

and, using the same derivation,

$$\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t^\gamma \,\middle|\, s_t = s, a_t = a\right] \\
&= \mathbb{E}_\pi\left[r_t + \gamma \mathbb{E}_\pi\left[G_{t+1}^\gamma \,\middle|\, s_{t+1} = s', a_{t+1} = a'\right]\right] \\
&= \mathbb{E}_\pi\left[r_t + \gamma q_\pi(s', a')\right]. \qquad (2.2)
\end{aligned}$$

Equations (2.1) and (2.2) are referred to as *Bellman equations* and the corresponding *optimal* value functions are defined as

$$v_*(s) := \max_\pi v_\pi(s)$$

and

$$q_*(s,a) := \max_\pi q_\pi(s,a),$$

where q_* can also be expressed using v_* and vice versa by the expressions

$$q_*(s,a) = \mathbb{E}[r_t + \gamma v_*(s_{t+1}) \mid s_t = s, a_t = a]$$

and

$$v_*(s,a) = \mathbb{E}\left[q_*(s_{t+1}, a_) \,\middle|\, s_t = s\right].$$

An optimal policy is one that maximizes the action-value function and is thus given by the relation

$$\pi_*(s) := \arg\max_a q_*(s,a). \qquad (2.3)$$

Q-Learning

Calculating the optimal value function, and eventually the optimal policy from it, requires knowledge of the transition dynamics of the environment. If no model of the dynamics is known an optimal value function can be estimated using the Bellman equation. One famous algorithm solving this task for finite MDPs is *Q-learning* [15]. It learns a policy *off-policy*, which means that the samples for the algorithm are generated by a behavior policy $\beta(a|s)$ to guarantee *exploration* of the state space. Tuples $\langle s_t, a_t, r_t, s_{t+1} \rangle$ of states s_t, actions a_t taken in the state s_t, the reward r_t for taking the action a_t in state s_t and following state s_{t+1} are collected and, based on so-called *temporal difference errors*

$$\delta_t = r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t),$$

an estimate of a tabular action-value function $Q_t(s, a)$ can be iteratively updated. With an arbitrary initialization of $Q_0(s, a)$ the update rule with step size α is given by

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha \delta_t.$$

Given enough update iterations, a bounded reward signal and a learning rate $0 \leq \alpha < 1$, $Q_t(s, a)$ converges to the optimal action-value function with probability 1 [16].

In an environment with a discrete finite action set the ϵ-greedy strategy is often used for exploration. Here, with probability ϵ a random action is chosen and the rest of the time the *greedy* action $a = \arg\max_{a'} Q(s, a')$ action is taken:

$$\beta(a|s) = \begin{cases} 1 - \epsilon & \text{if } a = \arg\max_{a'} Q_t(s, a') \\ \epsilon & \text{otherwise.} \end{cases}$$

2.1.2 Policy Gradient Based Learning

Another approach to the problem of learning policies are gradient based algorithms. Here, the policy is an explicit function independent of the value function, defined by a certain parametrization. The policy is improved by updating its parameters proportionally to the gradient of a performance objective which is taken with respect to the parameters of the policy function.

Deterministic Policy Gradient

For the *deterministic policy gradient* we consider actions $a = \mu(s \mid \theta^\mu)$ provided by a function μ represented by a differentiable function approximator with parameters θ^μ. Furthermore, the expected discounted return under the current policy is our objective function. It can be written as the expectation

$$J(\mu) = \mathbb{E}\left[G_0^\gamma\right].$$

With an initial state distribution $p_1(s)$, a density $p(s \to s', t, \mu)$ at state s' after transitioning for t time steps from state s and a discounted state distribution

$$\rho^\mu(s) = \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} p_1(s) p(s \to s', t, \mu) \mathrm{d}s$$

the objective function can be reformulated [9] as

$$
\begin{aligned}
J(\mu) &= \int_{\mathcal{S}} \rho^{\mu}(s) q_{\mu}(s, \mu(s \mid \theta^{\mu})) \mathrm{d}s \\
&= \mathbb{E}_{s \sim \rho^{\mu}} \left[q_{\mu}\left(s, \mu(s \mid \theta^{\mu})\right) \right].
\end{aligned}
$$

The gradient is given by

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

which is the result of the chain rule applied to the gradient of the Q-function with respect to the parameters of the policy.

Off-Policy Deterministic Actor Critic

The deterministic policy gradient can be used to derive an *actor-critic* learning algorithm. The policy function is referred to as the *actor* since it is responsible for choosing actions based on a state input. The value function is the *critic* since it "criticizes" the actions taken by the actor. Since deterministic policies do not ensure adequate exploration of the state space, the policy is learned off-policy using a stochastic behavior policy $\beta(a|s)$. The performance objective is modified to be the value function of the target policy μ averaged over the state distribution of the behavior policy ρ^{β} [17]

$$
\begin{aligned}
J_{\beta}(\mu) &= \int_{\mathcal{S}} \rho^{\beta}(s) q_{\mu}(s, \mu(s \mid \theta^{\mu})) \mathrm{d}s \\
&= \mathbb{E}_{s \sim \rho^{\beta}} \left[q_{\mu}(s, \mu(s \mid \theta^{\mu})) \right].
\end{aligned}
$$

Furthermore, the true action-value function is substituted by a differentiable approximation $Q(s, a \mid \theta^{Q})$ with parameters θ^{Q}. The *off-policy deterministic policy gradient* is then given by

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

2.2 Deep Learning

In classical machine learning the data we want to draw conclusions from is preprocessed by hand. Information which the researcher thinks is vital in order to make decisions, a so-called *feature*, is extracted and an algorithm is fed with this representation of the data. The goal of *deep learning* is to instead let an algorithm find both, a representation and a decision, based on unprocessed data [18]. An often used class of function approximators are *neural networks*, which were inspired by how information is processed in biological central nervous systems. In the following, neural networks and how they are trained is explained.

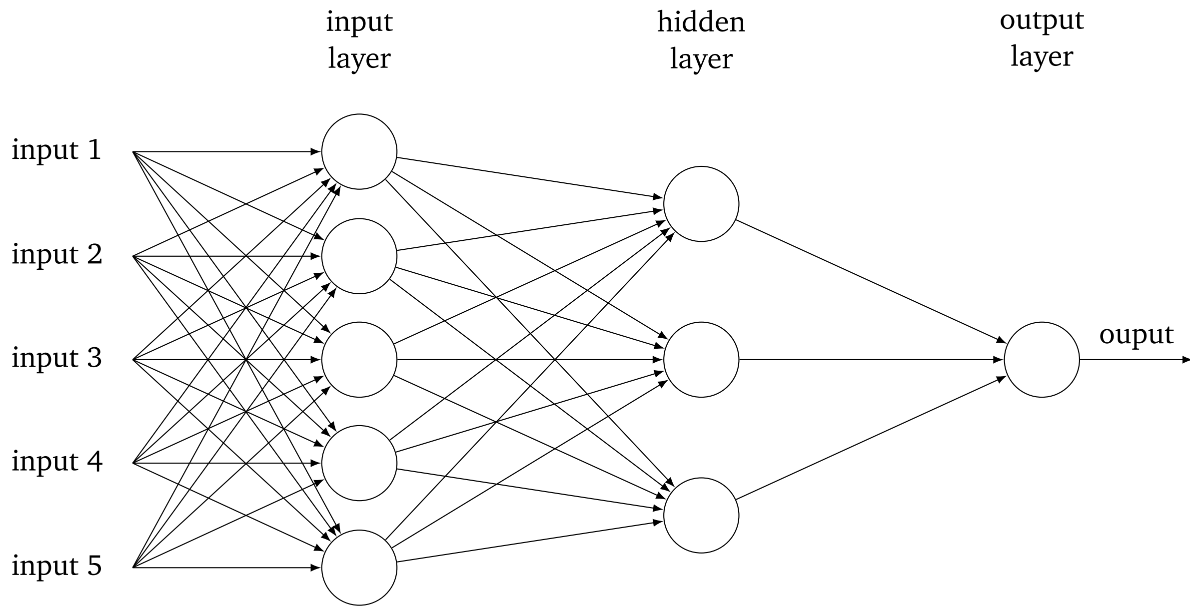


Figure 2.1.: Simple model of an artificial neural network.

2.2.1 Feed Forward Neural Networks

Figure 2.1 shows the diagram of a fully connected artificial feed-forward neural network. The circles represent units of simple computations, so-called *neurons*, which are arranged in *layers*. The shown network has one input, one output and one hidden layer. Data in the form of a vector $\boldsymbol{x}$ flows along the edges where it is weighted by a weighting matrix $\boldsymbol{W}$, summed and finally undergoes a non-linear transformation f called *activation*. Additionally a bias $\boldsymbol{b}$ can be added before the non-linearity is applied. A detailed view of one neuron with three inputs and one output can be found in Figure 2.2, which in the case of $j = 1$ is equal to the connection between the hidden layer and the output layer of the example. The output of a single *neuron* is given by

$$y_j = f\left(\sum_{i=1}^{I} w_{i,j} x_i + b_j\right),$$

and a single layer of neurons in matrix-vector notation is of the form

$$\boldsymbol{y} = f(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}),$$

introduced as the *perceptron* by Rosenblatt in 1958 [19]. Common activation functions are the hyperbolic tangent $f(x) = \tanh(x)$ and the logistic sigmoid function $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$. The most popular one nowadays is the *rectified linear unit* $f(x) = \max(0, x)$ (or Relu in short) [20, 21]. Several modifications of the Relu exist of which the *exponential linear unit (Elu)* is the one we will use in this thesis. It is defined by

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

with a hyperparameter α which controls the value to which an Elu saturates for negative net inputs [22]. An illustration of the different activation functions can be seen in Figure 2.3.

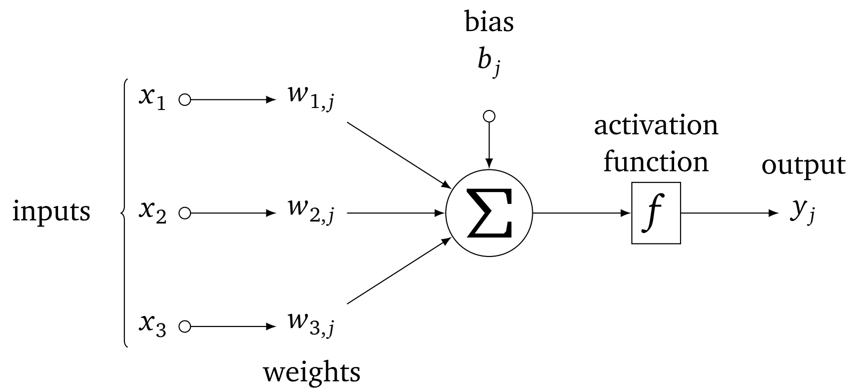
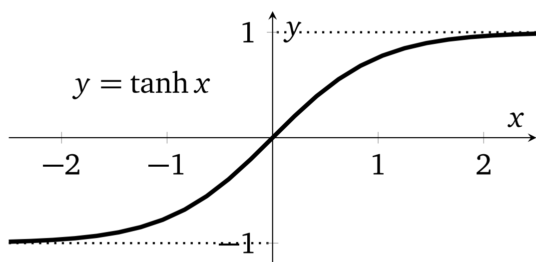
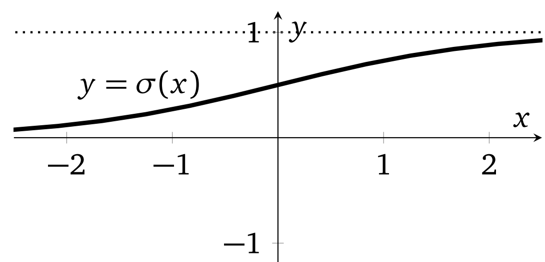


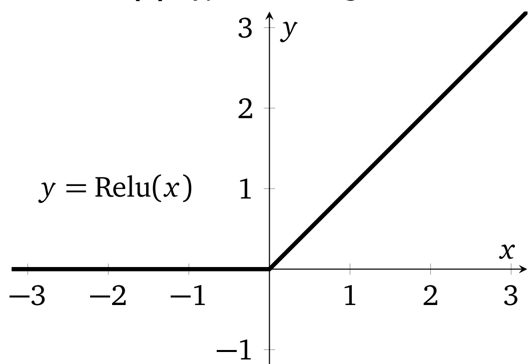
Figure 2.2.: One neuron with three inputs and one output.



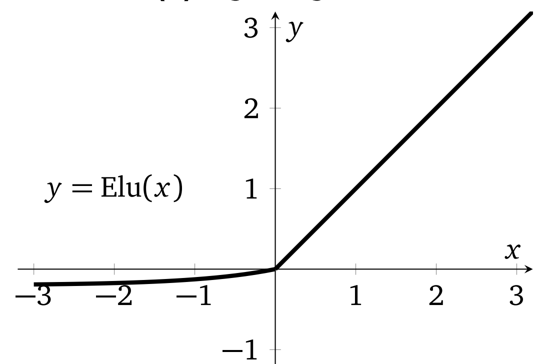
(a) Hyperbolic tangent



(b) Logistic sigmoid



(c) Rectified linear unit



(d) Exponential linear unit

Figure 2.3.: Examples of common activation functions.

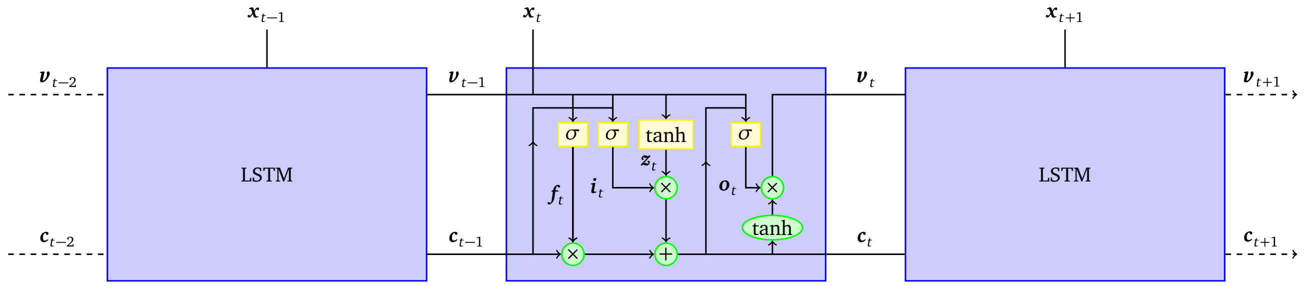


Figure 2.4.: Flow diagram of an LSTM cell.

Using a single layer network learning complex functions, for example the XOR-function [23], is often not possible. Thus, several layers of artificial networks are "stacked" together such that the output of one layer is fed into the input of a next layer. In theory, the more layers are stacked upon each other the higher the level of abstractions are such a *deep neural network* can find [24, 25, 26].

2.2.2 Recurrent Neural Networks

In contrast to feed-forward neural networks, *recurrent neural networks (RNNs)* allow for outputs to be fed back into previous layers. This mechanism enables the network to maintain an internal state using previous inputs and it may be used, but is not limited to the case, whenever data shows a sequential temporal aspect. A standard RNN can be described by weight matrices $\boldsymbol{W}_{vx}$, $\boldsymbol{W}_{vv}$ and $\boldsymbol{W}_{vo}$ and biases $\boldsymbol{b}_v$ and $\boldsymbol{b}_o$ for the input data, for the hidden state and for the output value. The equations describing the output at one time step t are given by

$$\begin{aligned}
\boldsymbol{v}_t &= f_v(\boldsymbol{W}_{vx}^T \boldsymbol{x}_t + \boldsymbol{W}_{vv}^T \boldsymbol{v}_{t-1} + \boldsymbol{b}_v), \\
\boldsymbol{y}_t &= f_o(\boldsymbol{W}_{vo}^T \boldsymbol{v}_t + \boldsymbol{b}_o),
\end{aligned}$$

with activation functions f_h and f_o. However, for sequences with many time steps this approach does not give satisfying results due to the *vanishing gradient problem* [27]. It is a difficulty in gradient based learning methods where, due to the chain rule applied during the update of deep networks, many small gradients are multiplied and the resulting error signal decreases exponentially. A popular class of RNNs overcoming this problem is the *long short-term memory (LSTM)* introduced by Hochreiter and Schmidhuber in 1997 [28]. LSTMs maintain two inner states, the cell state $\boldsymbol{c}_t$ and the hidden state $\boldsymbol{v}_t$. An LSTM cell is characterized by the equations

$$\begin{aligned}
\boldsymbol{z}_t &= f(\boldsymbol{W}_z^T \boldsymbol{x}_t + \boldsymbol{R}_z^T \boldsymbol{v}_{t-1} + \boldsymbol{b}_z), \\
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i^T \boldsymbol{x}_t + \boldsymbol{R}_i^T \boldsymbol{v}_{t-1} + \boldsymbol{b}_i), \\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f^T \boldsymbol{x}_t + \boldsymbol{R}_f^T \boldsymbol{v}_{t-1} + \boldsymbol{b}_f), \\
\boldsymbol{c}_t &= \boldsymbol{i}_t \odot \boldsymbol{z}_t + \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}, \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o^T \boldsymbol{x}_t + \boldsymbol{R}_o^T \boldsymbol{v}_{t-1} + \boldsymbol{b}_o), \\
\boldsymbol{v}_t &= \boldsymbol{o}_t \odot g(\boldsymbol{c}_t),
\end{aligned}$$

where $\boldsymbol{a} \odot \boldsymbol{b}$ denotes the component-wise product of two vectors of same dimension. The activation functions f and g are commonly chosen to be the hyperbolic tangent. A flow diagram including all computations can be found in Figure 2.4. Yellow boxes denote a single layer network with the according activation function while green nodes denote component-wise operations on vectors.

2.2.3 Training of Neural Networks

Depending on the task different output layers and *cost functions* are used to evaluate the performance of a neural network [29]. In classification tasks typically a softmax output layer is used which squashes all outputs into the range of 0 to 1, which can be interpreted as probabilities for each class. After the final layer, the cross entropy loss is applied to measure how accurate the prediction was. In this thesis, however, we only care about regression where we take linear output layers and define a mean squared error over the output values $\boldsymbol{y}_n$ given some target values $\boldsymbol{t}_n$. For N samples the *loss function* is given by

$$L = \frac{1}{N}\sum_{n=1}^{N} \|\boldsymbol{y}_n - \boldsymbol{t}_n\|^2.$$

The goal of *training* is to minimize the loss function by optimizing the set of parameters $\boldsymbol{\theta} := (\boldsymbol{W}_1, \boldsymbol{b}_1, \dots, \boldsymbol{W}_K, \boldsymbol{b}_K)$ for each of the K layers of the network. Since artificial neural networks are differentiable parametrized functions, we can accomplish this by using gradient descent on the loss function. The gradient $\nabla_{\boldsymbol{\theta}} L$ is obtained by *backpropagation*, an algorithm to iteratively calculate the gradients of each layer of a neural network using the chain rule [30]. A special case of this algorithm is *backpropagation through time (BPTT)* which is used to compute the gradient of loss functions if recurrent neural networks are used. The idea is to unfold the time dimension of the network such that we have one cell for each time point of a sequence. The result is a feed-forward neural network where all layers share the same parameters of the single RNN cell.

For computational reasons, *stochastic gradient descent (SGD)* is preferred over the full gradient. It is an iterative method which tries to minimize a differentiable function by approximating its gradient over mini-batches or even single samples to find a local optimum. Using an approximation $\Delta_\theta L$ for the true gradient the update rule for SGD is given by

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \Delta_{\boldsymbol{\theta}} L.$$

The *learning rate* α is a parameter defining the size of the step taken in the negative direction of the gradient and can be set by the designer of the algorithm. A more sophisticated approach to hand picking the learning rate are adaptive learning rate methods like ADAM [31].

2.3 Deep Reinforcement Learning

With the exception of LQR systems, the action-value function in environments with infinite state and action spaces (i.e. non-finite MDPs) can only be approximated. In deep reinforcement learning we use deep neural networks as function approximators for value functions and policies, which are well suited for continuous high dimensional input.

For the algorithms derived in this thesis we need two key ingredients. The first one is *deep Q-learning* introduced by Mnih et al. [6, 7] in the framework of *deep Q-networks* which connects the previously mentioned Q-learning algorithm with the concept of deep learning. The second one is the *deterministic policy gradient* [9] with the extension to the *deep deterministic policy gradient* algorithm [8] and the *recurrent deterministic policy gradient* algorithm [10]. It uses the concepts of deep Q-learning to represent the Q-function but in addition maintains a separate actor function which maps observed states to actions, also represented by a deep neural network. A comparison expressing the main differences between the presented algorithms is shown in Table 2.1.

2.3.1 Deep Q-Learning

The idea behind the deep Q-learning algorithm is to use a deep neural network with parameters θ^Q to approximate a Q-function for continuous state and discrete action spaces and apply Q-learning to learn a policy. First, we need to introduce a loss function,

$$L(\theta^Q) = \mathbb{E}_{s\sim\rho^\beta, a\sim\beta, r\sim E}\left[\left(y - Q(s, a \mid \theta^Q)\right)^2\right],$$

to iteratively optimize the parameters of the approximated Q-function. The behavior policy β to explore the state space is an ϵ-greedy strategy and ρ^β is the discounted distribution of visited states under this policy. The target values y are given by the Bellman equation, i.e.

$$y = r + \gamma \max_{a'} Q(s', a' \mid \theta^Q)$$

and the resulting gradient is given by [6]

$$\nabla_{\theta^Q} L(\theta^Q) = \mathbb{E}_{s\sim\rho^\beta, a\sim\beta, r\sim E}\left[\left(r + \gamma \max_{a'} Q(s', a' \mid \theta^Q) - Q(s, a \mid \theta^Q)\right)\nabla_{\theta^Q} Q(s, a \mid \theta^Q)\right]. \tag{2.4}$$

The DQN Algorithm

In practice, however, this approach needs several adjustments to reliably converge to meaningful solutions. Instead of computing the full gradient, the loss function is optimized using stochastic gradient descent based on randomly sampled mini-batches of previously experienced transitions $e = (s, a, r, s')$. These transitions are stored in a finite FIFO (first in, first out; oldest samples stored are the first to be deleted if the cache is full) cache $\mathcal{D}$ called *replay memory*. This technique is known as *experience replay* [32] and breaks correlations between subsequent samples to improve and speed up learning and also lets the algorithm reuse these samples. Furthermore, the parameters of the network are held fixed for a certain time-span and used in a *target network* Q' as parameters $\theta^{Q'}$ in order to avoid oscillations during the learning process. Including these ideas the loss function is changed to

$$L(\theta^Q) = \mathbb{E}_{e\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q'(s', a' \mid \theta^{Q'}) - Q(s, a \mid \theta^Q)\right)^2\right].$$

Another interesting property of these *deep Q-networks* is the fact, that instead of taking the action as an input it is part of the output of the network for each state. This reduces the computational complexity for each state to one forward pass but in return limits the algorithm to problems with a finite number of actions. The learned policy is the greedy strategy

$$\mu^*(s) = \arg\max_a Q(s, a \mid \theta^Q).$$

2.3.2 Deep Deterministic Policy Gradient

A major drawback of the deep Q-learning algorithm is its restriction to problems with finite action sets. Instead of letting the Q-function yield Q-values for all possible actions, an actor function $a = \mu(s \mid \theta^\mu)$ is introduced to provide deterministic actions. This function will also be represented by a deep neural network with parameters θ^μ. The loss function is given by

$$L(\theta^Q) = \mathbb{E}_{s\sim\rho^\beta, a\sim\beta, r\sim E}\left[\left(Q(s, a \mid \theta^Q) - y\right)^2\right]$$

with target values

$$y = r + \gamma \max_{a'} Q(s', a' \mid \theta^Q).$$

The objective function is the same as in the off-policy deterministic policy gradient.

Algorithm 1: Deep Q-learning with experience replay

Initialize replay memory $\mathcal{D}$ to capacity R
Initialize action-value network Q with random parameters θ^Q
Initialize target network Q' with parameters $\theta^{Q'} \leftarrow \theta^Q$
for episode = $1, \ldots, M$ **do**
  Receive initial observation state s_1
  for $t = 1, \ldots, T$ **do**
    With probability ϵ select a random action a_t otherwise select $a_t = \max_a Q(s_t, a \mid \theta^Q)$
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random mini-batch of N transitions $(s_i, a_i, r_i, s_{i+1})_{i=1,\ldots,N}$ from $\mathcal{D}$
    Set $y_i = \begin{cases} r_i & \text{for terminal } s_{i+1} \\ r_i + \gamma \max_{a'} Q(s, a' \mid \theta^Q) & \text{for non-terminal } s_{i+1} \end{cases}$
    Perform a gradient descent step on $\frac{1}{N}\sum_i (y_i - Q(s_i, a_i \mid \theta))^2$ according to equation (2.4)
    Every C steps set $\theta^{Q'} \leftarrow \theta^Q$
  end
end

The DDPG Algorithm

As in deep Q-learning we want to find the greedy policy $\mu(s \mid \theta_\mu) = \arg\max_a Q(s, a \mid \theta^Q)$ but since we are dealing with deep neural networks a global maximization over Q is not feasible. Using the off-policy deterministic actor critic this problem can be overcome. The actor function will be optimized to eventually yield the maximizing action. Like DQN this algorithm suffers from instabilities during the learning process. Therefore, a replay memory is used again to store past transitions and, in addition to the target critic network Q', a target actor network $\mu'(s|\theta^{\mu'})$ is kept to provide the targets,

$$y = r + \gamma Q'(s', \mu'(s' \mid \theta^{\mu'}) \mid \theta^{Q'})$$

for the loss function to be optimized. Instead of the hard assignments of the parameters of the target networks like in DQN, the authors of the paper [8] propose soft updates using a moving average over the original set of parameters, i.e.

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'},$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'},$$

with $\tau \ll 1$. With the help of target networks and a replay memory the loss function of the *deep deterministic policy gradient (DDPG)* algorithm is now given by

$$L(\theta^Q) = \mathbb{E}_{e\sim\mathcal{D}}\left[\left(Q(s, a \mid \theta^Q) - y\right)^2\right].$$

The objective function

$$J_{\text{DDPG}} = \mathbb{E}_{e\sim\mathcal{D}}\left[G_0^\gamma\right]$$

is formulated over transitions sampled from experience replay, too. The gradient is given by

$$\nabla_{\theta^\mu} J_{\text{DDPG}} = \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_{\theta^\mu}\mu(s \mid \theta^\mu)\nabla_a Q(s, a \mid \theta^Q)\big|_{a=\mu(s|\theta^\mu)}\right], \tag{2.5}$$

and the parameters of the function approximators are updated according to [8]

$$\theta^Q \leftarrow \theta^Q + \alpha_Q \nabla_{\theta^Q} L(\theta^Q)$$
$$\theta^\mu \leftarrow \theta^\mu + \alpha_\mu \nabla_{\theta^\mu} J_{\text{DDPG}}$$

using for example an adaptive learning rate method like ADAM [31]. Exploration of the state space is ensured by the behavior policy

$$\beta(a|s) = \mu(s|\theta^\mu) + \nu, \quad \nu \sim \mathcal{N}$$

where $\mathcal{N}$ is a noise process, for example an Ornstein-Uhlenbeck process [33] which generates temporally correlated noise.

Algorithm 2: DDPG algorithm

Initialize replay memory $\mathcal{D}$ to capacity R
Initialize critic network Q and actor network μ with random weights
Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
for episode = $1, \ldots, M$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state s_1
  for $t = 1, \ldots, T$ **do**
    Select action $a_t = \mu(s_t \mid \theta^\mu) + \nu$, $\nu \sim \mathcal{N}$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random mini-batch of N transitions $(s_i, a_i, r_i, s_{i+1})_{i=1,\ldots,N}$ from $\mathcal{D}$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q(s_i, a_i \mid \theta^Q))^2$
    Update the actor policy using the gradient (Equation 2.5) approximated by samples:

$$\nabla_{\theta^\mu} J_{\text{DDPG}} \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q(s, a \mid \theta^Q)\big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s \mid \theta^\mu)\big|_{s=s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end
end

2.3.3 Recurrent Deterministic Policy Gradient

The *recurrent deterministic policy gradient (RDPG)* extends DPG, which relies on the knowledge of the full state, to the domain of partially observable Markov decision processes (POMDPs). Here, the agent is unable to perceive the true state s and has to rely on observations o drawn from the set of observations $\mathcal{O}$. In order to make optimal decisions, the agent, in theory, requires the full history $h_t = (o_1, a_1, o_2, a_2, \ldots, a_{t-1}, o_t)$ on which it bases its actions $a = \mu(h_t)$, since the observations do not necessarily obey the Markov property anymore. The objective function,

$$J(\mu) = \mathbb{E}_\tau\left[\sum_{t=1}^{\infty} \gamma^{t-1} R(s_t, a_t)\right],$$

is written over whole trajectories $\tau = (s_1, o_1, a_1, s_2, \dots)$ which are generated by a trajectory distribution under the current policy. Analogously, the critic

$$q_\mu(h_t, a_t) = \mathbb{E}_{s_t|h_t}[R(s_t, a_t)] + \mathbb{E}_{\tau_{>t}|h_t,a_t}\left[\sum_{i=1}^{\infty} \gamma^i R(s_{t+i}, a_{t+i})\right]$$

evaluates pairs of history and actions where $\tau_{>t} = (s_{t+1}, o_{t+1}, a_{t+1}, \dots)$ are future trajectories and expectations are taken with respect to histories leading to the current state and the future trajectory. The update for the actor becomes

$$\nabla_{\theta^\mu} J(\mu) = \mathbb{E}_\tau\left[\sum_{t=1} \gamma^{t-1} \nabla_{\theta^\mu}\mu(h_t) \nabla_a \, q_\mu(h_t, a)\big|_{a=\mu(h_t)}\right].$$

The RDPG Algorithm

As in DDPG, both the critic and the actor are approximated by deep neural networks, but since we explicitly have to deal with sequences of observations recurrent neural networks like the LSTM are used. The usage allows us to only process current observations and actions instead of whole trajectories because information of past observations is implicitly stored in the hidden state of the recurrent neural network. Again, a replay memory and target networks are used. During the algorithm, mini-batches of full trajectories are sampled from the replay memory leading to the following loss function

$$L(\theta^Q) = \mathbb{E}_{\tau\sim\mathcal{D}}\left[\left(Q(h_t, a_t \mid \theta^Q) - y_t\right)^2\right],$$

with target values

$$y = r_t + \gamma Q'(h_{t+1}, \mu'(h_{t+1} \mid \theta^{\mu'}) \mid \theta^{Q'}).$$

The objective function

$$J_{\text{RDPG}} = \mathbb{E}_{\tau\sim\mathcal{D}}\left[\sum_{t=1}^{\infty} \gamma^{t-1} R(s_t, a_t)\right]$$

is based on trajectories sampled from experience replay and the gradient for the actor is given by

$$\nabla_{\theta^\mu} J_{\text{RDPG}} = \mathbb{E}_{\tau\sim\mathcal{D}}\left[\sum_{t=1} \gamma^{t-1} \nabla_{\theta^\mu}\mu(h_t) \nabla_a \, q_\mu(h_t, a)\big|_{a=\mu(h_t)}\right]. \tag{2.6}$$

The parameters of the networks and are updated analogously to the DDPG algorithm.

Algorithm 3: RDPG algorithm

Initialize replay memory $\mathcal{D}$ to capacity R
Initialize critic network Q and actor network μ with random weights
Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^{Q}$, $\theta^{\mu'} \leftarrow \theta^{\mu}$
for episode = 1, ..., M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Initialize empty history h_0
  for t = 1, ..., T **do**
    Receive observation o_t
    Append observation and previous action to history $h_t \leftarrow h_{t-1}, a_{t-1}, o_t$
    Select action $a_t = \mu(h_t \mid \theta^{\mu}) + \nu,\ \nu \sim \mathcal{N}$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t
  end
  Store the sequence $(o_1, a_1, r_1, \ldots, o_T, a_T, r_T)$ in $\mathcal{D}$
  Sample random minibatch of N episodes $(o_{1,i}, a_{1,i}, r_{1,i}, \ldots, o_{T,i}, a_{T,i}, r_{T,i})_{i=1,\ldots,N}$ from $\mathcal{D}$
  Construct histories $h_{t,i} = (o_{t,i}, a_{t,i}, r_{t,i}, \ldots, a_{t-1,i}, o_{t,i})$
  Compute target values for each sample episode $y_1^i, \ldots, y_T^i$ using the recurrent target networks

$$y_{t,i} = r_{t,i} + \gamma Q'(h_{t+1,i}, \mu'(h_{t+1,i} \mid \theta^{\mu'}) \mid \theta^{Q'})$$

  Compute critic update (using BPTT):

$$\nabla_{\theta^Q} L(\theta^Q) \approx \frac{1}{NT} \sum_i \sum_t (Q(h_{t,i}, a_{t,i} \mid \theta^Q) - y_{t,i})\, \nabla_{\theta^Q} Q(h_{t,i}, a_{t,i} \mid \theta^Q)$$

  Update the actor policy using the gradient (Equation 2.6) approximated by samples (using BPTT):

$$\nabla_{\theta^\mu} J \approx \frac{1}{NT} \sum_i \sum_t \nabla_a\, Q(h, a \mid \theta^Q)\big|_{h=h_{t,i}, a=\mu(h_{t,i})}\ \nabla_{\theta^\mu}\, \mu(h \mid \theta^\mu)|_{h=h_{t,i}}$$

  Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

end

Algorithm	Observability	Domain	Summary
DQN	full observability	Continuous states, discrete actions	Q-function approximated by feed forward neural network, policy is selected by taking action which maximizes Q-function for the state
DDPG	full observability	Continuous states, continuous actions	based on off-policy deterministic actor critic, Q-function and actor approximated by feed forward network
RDPG	partial observability	Continuous states, continuous actions	extension of DPG to partial observability, Q-function and actor approximated by recurrent neural networks

Table 2.1.: Comparison between deep reinforcement learning algorithms.

3 Guided Deep Reinforcement Learning for Homogeneous Multi-Agents Systems

The algorithms introduced in Chapter 2 are all designed for single-agent scenarios and cannot be transfered directly to the multi-agent case. Nevertheless, we can take the ideas behind the deterministic policy gradient and combine it with the *decentralized partially observable Markov decision process (Dec-POMDP)* framework. In particular, we consider cooperative homogeneous multi-agent systems where all agents are supposed to have the same set of actions and identical behavior. What separates them is their current local state and thus, their current view of the global system state differs. Although the behavior of the agents is solely based on their own perceptions, we assume to have knowledge of the full state which allows us to *guide* the system through the learning process.

3.1 The Decentralized POMDP Framework

A formal mathematical formulation of a cooperative MAS is provided by the Dec-POMDP framework [34], which is an extension of the MDP to cover partial observability and multiple agents in a collaborative environment. It is defined as a tuple $\langle \mathcal{B}, \mathcal{S}, \boldsymbol{\mathcal{A}}, P, R, \mathfrak{O}, O, z, I \rangle$, where

- $\mathcal{B} = \{1, \dots, N\}$ is the set of N agents,
- $\mathcal{S}$ is a set of states s in which the system of agents can be in,
- $\boldsymbol{\mathcal{A}}$ is a set of joint actions,
- P is a transition probability function,
- R is a reward function,
- $\mathfrak{O}$ is a set of joint observations,
- O is an observation probability function,
- z is the horizon of the problem, and
- I is the initial state distribution.

At each point in time a joint action $\boldsymbol{a} = \langle a^1, \dots, a^M \rangle$ is taken and the state s advances to state s' according to the transition function P with probability $\Pr(s'|s, \boldsymbol{a})$. Furthermore, a joint observation $\boldsymbol{o} = \langle o^1, \dots, o^M \rangle$ with local observations o^i for each agent i is determined by the observation function O with probability $\Pr(\boldsymbol{o}|\boldsymbol{a}, s')$. In an MDP, agents can take optimal actions based on the current state since the Markov property makes information from all past states superfluous. Observations, in contrast, lack the Markov property and, since each agent can base its actions only on this signal, they maintain a history $h^i_t = (o^i_1, a^i_1, \dots, a^i_{t-1}, o^i_t)$ at time t of past observations o^i and actions a^i. The agents are rewarded based on the collective behavior of the system of M agents and the reward function is given by $r = R(s, \boldsymbol{a})$. Communication between the agents is not made explicit, it is rather modeled implicitly through states, actions and observations.

3.2 Multi-Agent Guided Deterministic Policy Gradient

In order to formulate the deterministic policy gradient for multi-agent systems we require the action-value function

$$q_\mu(s, \boldsymbol{a}) = q_\mu(s, a^1, \dots, a^M)$$

to evaluate the joint action of all agents in the current state. The performance function is given by

$$J = \mathbb{E}_{s \sim \rho^\mu}[q_\mu(s, \boldsymbol{a})].$$

The joint action is composed of the individual actions $a^i = \mu(h^i \mid \theta^\mu)$ yielded by an actor function which evaluates the history of each agent, consisting of local observations and actions which led to the current state. The parameters θ^μ are shared between all agents, which ensures identical behavior. The derivative of the performance function with respect to the parameters θ^μ of the actor function for M agents is given by

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s \sim \rho^\mu}\left[\sum_{i=1}^{M} \nabla_{\theta^\mu} \mu(h^i \mid \theta^\mu) \, \nabla_{a^i} q_\mu(s, a^1, \dots, a^M)\Big|_{a^i = \mu(h^i \mid \theta^\mu)}\right].$$

A detailed derivation of this gradient can be found in Appendix A.1. An interesting question arises about how to deal with the histories of observations and actions on which the agents build their decisions on. An implementation using deep neural networks restricts us to a fixed size of the history's length and, in case we use non recurrent networks, a finite time horizon. In the following sections we will show two different representations. First, using standard feed-forward neural networks for both the critic and the actor, and second, a recurrent network for the actor is considered. The resulting algorithms will be referred to as *multi-agent guided deep deterministic policy gradient (MAGDDPG)* and *multi-agent guided recurrent deterministic policy gradient (MAGRDPG)*.

3.3 Multi-Agent Guided Deep Deterministic Policy Gradient

For the first algorithm we consider standard feed-forward neural networks with parameters θ^Q for the critic and parameters θ^μ shared by all actors. The history $h_t^i = (a^i_{T_{\text{prev}}-1}, o^i_{T_{\text{prev}}}, \dots, a^i_{t-1}, o^i_t)$ of each agent i is a vector of current and past observations, and actions within a fixed time horizon T_{prev}. This means that the agent can only remember events as far as T_{prev} time steps in the past. The loss function for the critic update is given by

$$L(\theta^Q) = \mathbb{E}_{s \sim \rho^\mu, \boldsymbol{a} \sim \beta, r \sim E}\left[(Q(s, \boldsymbol{a} \mid \theta^Q) - y)^2\right],$$

with target values

$$y = r + \gamma \max_{\boldsymbol{a}'} Q\left(s', \boldsymbol{a}' \mid \theta^Q\right)$$

and consecutive states s and s'. The actor is updated along the gradient of the performance function $J = \mathbb{E}[Q(s, \boldsymbol{a} \mid \theta^Q)]$ using the action value function:

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s \sim \rho^\mu}\left[\sum_{i=1}^{M} \nabla_{\theta^\mu} \mu(h^i \mid \theta^\mu) \, \nabla_{a^i} Q(s, a^1, \dots, a^M \mid \theta^Q)\Big|_{a^i = \mu(h^i \mid \theta^\mu)}\right].$$

The MAGDDPG Algorithm

For the algorithm we apply the same methods as before. Tuples $e = \langle s, \boldsymbol{h}, \boldsymbol{a}, r, s', \boldsymbol{h}' \rangle$ are stored in a replay memory and then sampled from during the updates, where $\boldsymbol{h}$ denotes a joint history $\langle h^1, \dots, h^M \rangle$ for a given time point. The loss function is modified to

$$L(\theta^Q) = \mathbb{E}_{e\sim\mathcal{D}}\left[(Q(s, \boldsymbol{a} \mid \theta^Q) - y)^2\right],$$

with target values

$$y = r + \gamma Q'\left(s', \mu'(h'^1 \mid \theta^{\mu'}), \mu'(h'^2 \mid \theta^{\mu'}), \dots, \mu'(h'^M \mid \theta^{\mu'}) \mid \theta^{Q'}\right).$$

The actor too is updated based on experiences:

$$\nabla_{\theta^\mu} J = \mathbb{E}_{e\sim\mathcal{D}}\left[\sum_{i=1}^{M} \nabla_{\theta^\mu}\mu(h^i \mid \theta^\mu)\, \nabla_{a^i} Q^{\theta^Q}(s, a^1, \dots, a^M \mid \theta^Q)\Big|_{a^i=\mu(h^i \mid \theta^\mu)}\right]. \tag{3.1}$$

The parameters of the target networks follow the set of original parameters with soft updates like in DDPG. Before learning starts a *warmup phase* is run where we use the initial policy with random parameters to create samples to fill the replay memory with. The full procedure is displayed in Algorithm 4.

3.4 Multi-Agent Guided Recurrent Deterministic Policy Gradient

The second algorithm uses recurrent neural networks, such as LSTMs, to handle the histories of the actors, similar to the RDPG algorithm (Algorithm 3). Since we assume to have knowledge of the true state, the Q-function will still be represented by a standard feed-forward neural network.

The MAGRDPG Algorithm

In the literature one can find different ways how to train recurrent neural networks in the context of deep reinforcement learning. While Heess et al [10] propose to store full episodes and carry the hidden state of the recurrent network forward until the end of the episode, Hausknecht et al [35] propose *random updates* which start at a random point within a sampled episode for only T_{unroll} time steps. The second approach is computationally more efficient with the drawback that the hidden state has to be re-initialized by zeros for each update and, therefore, it is possibly harder to learn long term dependencies. In both cases we store whole episode trajectories $\tau = \langle s_1, \boldsymbol{o}_1, \boldsymbol{a}_0, r_1, \dots, s_{T+1}, \boldsymbol{o}_{T+1}, \boldsymbol{a}_T, r_T \rangle$ in a replay memory $\mathcal{D}$. In the case of random updates we sample minibatches starting at a random time point t_{start} of the form $e = \langle s_{t_{\text{start}}:t_{\text{start}}+T_{\text{unroll}}+1}, \boldsymbol{o}_{t_{\text{start}}:t_{\text{start}}+T_{\text{unroll}}+1}, \boldsymbol{a}_{t_{\text{start}}-1:t_{\text{start}}+T_{\text{unroll}}}, r_{t_{\text{start}}+T_{\text{unroll}}} \rangle$. Using samples like these we can define a loss function where we evaluate the last state of the minibatch sequence, i.e.

$$L(\theta^Q) = \mathbb{E}_{e\sim\mathcal{D}}\left[(Q(s_{t_{\text{start}}+T_{\text{unroll}}}, \boldsymbol{a}_{t_{\text{start}}+T_{\text{unroll}}} \mid \theta^Q) - y)^2\right],$$

with target values

$$\begin{aligned} y = r_{t_{\text{start}}+T_{\text{unroll}}} + \gamma Q'\big(s_{t_{\text{start}}+T_{\text{unroll}}+1}, &\mu'(o^1_{t_{\text{start}}+T_{\text{unroll}}+1}, a^1_{t_{\text{start}}+T_{\text{unroll}}} \mid \theta^{\mu'}),\\ &\mu'(o^2_{t_{\text{start}}+T_{\text{unroll}}+1}, a^2_{t_{\text{start}}+T_{\text{unroll}}} \mid \theta^{\mu'}), \dots,\\ &\mu'(o^M_{t_{\text{start}}+T_{\text{unroll}}+1}, a^M_{t_{\text{start}}+T_{\text{unroll}}} \mid \theta^{\mu'}) \mid \theta^{Q'}\big), \end{aligned}$$

Algorithm 4: MAGDDPG algorithm

Initialize replay memory $\mathcal{D}$ to capacity R
Initialize critic network Q and actor network μ with random weights
Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^{Q}$, $\theta^{\mu'} \leftarrow \theta^{\mu}$
for episode = *1, . . . , E* **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state s_1
  for each agent i **do**
    Set initial action $a_0^i = 0$
    Receive initial observation o_1^i based on s_1
    Set initial history h_1^i with the tuple (a_0^i, o_1^i) repeated t_{prev} times
  end
  for t = *1, . . . , T* **do**
    for each agent i **do**
      Select action $a_t^i = \mu(h_t^i \mid \theta^{\mu}) + \nu,\ \nu \sim \mathcal{N}$ according to the current policy and exploration noise
      Execute action a_t^i
    end
    Observe reward r_t and new state s_{t+1}
    for each agent i **do**
      Receive observation o_{t+1}^i based on s_{t+1}
      Update history h_{t+1}^i by deleting the oldest tuple $(a_{t-m_{\text{prev}}}^i, o_{t-m_{\text{prev}}+1}^i)$ and append new tuple (a_t^i, o_{t+1}^i)
    end
    Concatenate all h_t^i to joint history $\boldsymbol{h}_t$ (and h_{t+1}^i to $\boldsymbol{h}_{t+1}$), all actions a_t^i to joint action $\boldsymbol{a}_t$ (and a_{t+1}^i to $\boldsymbol{a}_{t+1}$)
    Store transition $(s_t, \boldsymbol{h}_t, \boldsymbol{a}_t, r_t, s_{t+1}, \boldsymbol{h}_{t+1})$ in $\mathcal{D}$
    if episode > warmup **then**
      Sample random minibatch of N transitions $(s_j, \boldsymbol{h}_j, \boldsymbol{a}_j, r_j, \boldsymbol{h}_{j+1}, s_{j+1})_{j=1,\ldots,N}$ from $\mathcal{D}$
      Set $y_j = r_j + \gamma Q'(s_{j+1}, \mu'(h_{j+1}^1 \mid \theta^{\mu'}), \ldots, \mu'(h_{t+1}^M \mid \theta^{\mu'}) \mid \theta^{Q'})$
      Update critic by minimizing the loss: $L = \frac{1}{N}\sum_j (y_j - Q(s_j, a_j^1, \ldots, a_j^M \mid \theta^Q))^2$
      Update the actor policy using the policy gradient (Equation 3.1) approximated by mini-batch samples:

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{M} \nabla_{\theta^{\mu}} \mu(h_j^i \mid \theta^{\mu})\, \nabla_{a^i} Q^{\mu_\theta}(s_j, a^1, \ldots, a^M)\Big|_{a^i = \mu(h_j^i \mid \theta^{\mu})}$$

      Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

    end
  end
end

where the input to the actor function is the current observation and the last action of agent i given the hidden state from the previous time step. For the update of the actor function we have to consider the whole sequence so that the gradient of the performance function with respect to the parameters of the actor is given by

$$\nabla_{\theta^\mu} J = \mathbb{E}_{e\sim\mathcal{D}}\left[\sum_{t=t_{\text{start}}}^{t_{\text{start}}+T_{\text{unroll}}} \sum_{i=1}^{M} \nabla_{\theta^\mu}\mu(o_t^i, a_{t-1}^i \mid \theta^\mu)\, \nabla_{a_t^i} Q^{\theta^Q}(s_t, a_t^1, \ldots, a_t^M \mid \theta^Q)\Big|_{a_t^i=\mu(o_t^i, a_{t-1}^i \mid \theta^\mu)}\right]. \tag{3.2}$$

The full procedure can be found in Algorithm 5.

Algorithm 5: MAGRDPG algorithm

Initialize replay memory $\mathcal{D}$ to capacity R
Initialize critic network Q and actor network μ with random weights
Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^{Q}$, $\theta^{\mu'} \leftarrow \theta^{\mu}$
for episode = 1, ..., M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state s_1
  for each agent i **do**
    Set initial action $a_0^i = 0$
    Receive initial observation o_1^i based on s_1
    Set initial hidden state to zero
  end
  for t = 1, ..., T **do**
    for each agent i **do**
      Select action $a_t^i = \mu(o_t^i, a_{t-1}^i \mid \theta^\mu) + \nu,\ \nu \sim \mathcal{N}$ Execute action a_t^i
    end
    Observe reward r_t and new state s_{t+1}
    if episode > warmup **then**
      Sample random minibatch of N episodes $\langle s_{1,j}, \boldsymbol{o}_{1,j}, \boldsymbol{a}_{0,j}, r_{1,j}, \ldots, s_{T+1,j}, \boldsymbol{o}_{T+1,j}, \boldsymbol{a}_{T,j}, r_{T,j} \rangle_{j=1,\ldots,N}$ from $\mathcal{D}$
      Choose random start point t_{start} within the episodes and set experiences e accordingly
      Set target values y_j

$$
\begin{aligned}
y_j = r_{t_{\text{start}}+T_{\text{unroll}},j} + \gamma Q'\Big(s_{t_{\text{start}}+T_{\text{unroll}}+1,j}\ , &\mu'(o^1_{t_{\text{start}}+T_{\text{unroll}}+1,j}, a^1_{t_{\text{start}}+T_{\text{unroll}},j} \mid \theta^{\mu'}), \\
&\ldots, \\
&\mu'(o^M_{t_{\text{start}}+T_{\text{unroll}}+1,j}, a^M_{t_{\text{start}}+T_{\text{unroll}},j} \mid \theta^{\mu'}) \mid \theta^{Q'}\Big)
\end{aligned}
$$

      Update critic by minimizing the loss: $L = \frac{1}{N}\sum_j (y_j - Q(s_{t_{\text{start}}+T_{\text{unroll}},j}, \boldsymbol{a}_{t_{\text{start}}+T_{\text{unroll}},j} \mid \theta^Q))^2$
      Update the actor policy using the policy gradient (Equation 3.2) approximated by mini-batch samples:

$$
\nabla_{\theta^\mu} J \approx \frac{1}{N T_{\text{unroll}}} \sum_{j=1}^{N} \sum_{t=t_{\text{start}}}^{t_{\text{start}}+T_{\text{unroll}}} \sum_{i=1}^{M} \nabla_{\theta^\mu} \mu(o^i_{t,j}, a^i_{t-1,j} \mid \theta^\mu)\, \nabla_{a^i_{t,j}} Q^{\theta^Q}(s_{t,j}, a^1_{t,j}, \ldots, a^M_{t,j} \mid \theta^Q)\Big|_{a^i_{t,j}=\mu(o^i_{t,j}, a^i_{t-1,j} \mid \theta^\mu)}
$$

      Update the target networks:

$$
\begin{aligned}
\theta^{Q'} &\leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\
\theta^{\mu'} &\leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}
\end{aligned}
$$

    end
  end
  Concatenate all o_t^i to joint history $\boldsymbol{o}_t$ (and o_{t+1}^i to $\boldsymbol{o}_{t+1}$), all actions a_t^i to joint action $\boldsymbol{a}_t$ (and a_{t+1}^i to $\boldsymbol{a}_{t+1}$)
  Store the sequence $\langle s_1, \boldsymbol{o}_1, \boldsymbol{a}_0, r_1, \ldots, s_{T+1}, \boldsymbol{o}_{T+1}, \boldsymbol{a}_T, r_T \rangle$ in $\mathcal{D}$
end

4 Application in Simulated Robot Swarms

In this chapter we introduce a simulation environment for multi-agent systems. The agents are designed to have capabilities similar to those of the Kilobot robots shown in Section 1. Subsequently, a task is formulated on which the performance of the learning algorithm is evaluated.

4.1 Implementation

Depending on the size of the scenario learning can take up to millions of gradient descent steps to achieve decent policies. Since learning on a real system would be infeasible an environment with agents similar to the Kilobots was implemented using *Python* to simulate all interactions. The robot swarm here is defined as a simple point cloud without collision detection. Actions are simulated with commands u_{left} and u_{right} for a virtual left and right motor, similar to the Kilobots' movement. A command between 0 and 1 for each motor is composed to a linear and an angular velocity factor of the form $k_{\text{linear}} = \frac{u_{\text{left}}+u_{\text{right}}}{2}$ and $k_{\text{angular}} = (u_{\text{left}} - u_{\text{right}})$. Given the current local state s_t^i of an agent i consisting of its coordinates (x_t^i, y_t^i) and orientation ϕ_t^i the next state s_{t+1}^i is determined by

$$\phi_{t+1}^i = \phi_t^i + k_{\text{angular}} v_{\text{angular, max}} \Delta T$$
$$(x_{t+1}^i, y_{t+1}^i) = (x_t^i, y_t^i) + k_{\text{linear}}(\cos(\phi_{t+1}^i), \sin(\phi_{t+1}^i)) v_{\text{linear, max}} \Delta T,$$

with $v_{\text{linear, max}} = 1\,\text{cm/s}$ and $v_{\text{angular, max}} = 45\,\text{deg/s}$. The length of one time step is chosen to be $\Delta T = 1\,\text{s}$. The full system state representation s_t is composed of the coordinates of the target and the local states of each moving agent. In the fashion of other deep reinforcement learning algorithms a representation in form of camera images of the scene could be possible but is not considered in the scope of this thesis. We argue that the presence of a wall indicating the borders of the reachable space can be sensed by a simulated ambient light sensor if the walls give some kind of illumination feedback.

Communication is not explicitly implemented but the basic reasoning which can be deducted from a message is modeled. This includes the distance to a neighboring agent within a radius of 4 cm, the ability to identify the target and information whether a neighbor is in the vicinity of the target. Since we want to feed all information into a neural network we need a fixed size representation of the distances. Identification of the target, sensing of the wall and the presence of a neighbor in the vicinity of the target are all single values, but the number of agents within one agent's communication radius is changing during episodes. Instead of collecting raw distances in a vector this can be solved, for example, with an (improper) distribution over sensed distances. In our case, we take 11 Gaussian radial basis functions

$$\psi_k(d_m^i, \mu_k) = \exp\left(\frac{(d_m^i - \mu_k)^2}{2\sigma^2}\right), \; k = 1, 2, \ldots, 11$$

equally spaced in the range of the communication radius (see Figure 4.1) and sum up the contributions each distance d_m^i between agent i and agent m makes with each basis function. The distribution over distances for agent i is given by a vector $\boldsymbol{u}^i$ consisting of entries

$$u_k^i = \sum_{m=1}^{M} \delta_{m,\text{dist}}^i \psi_k(d_m^i, \mu_k)$$

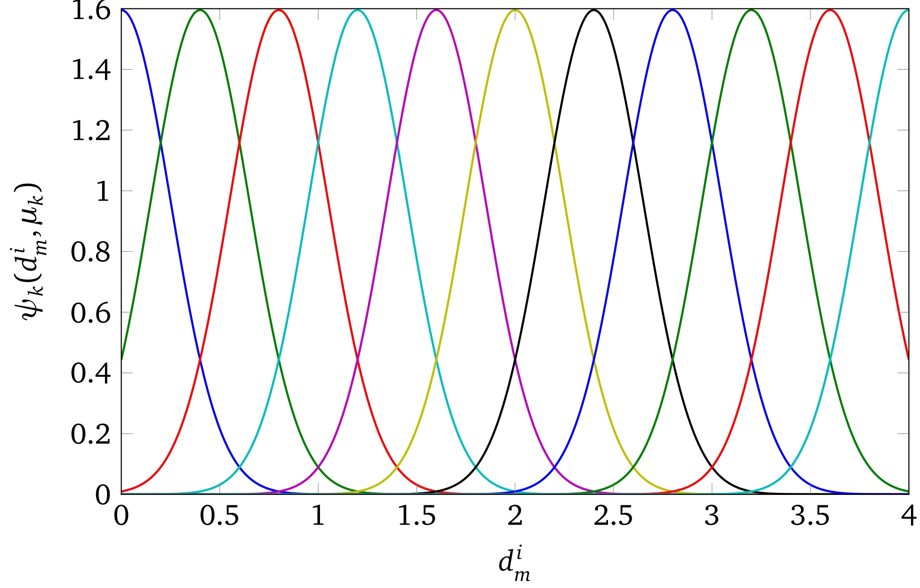


Figure 4.1.: 11 basis functions with standard deviation $\sigma = 0.25$ equally spread over the communication radius.

with

$$\delta^i_{m,\text{dist}} = \begin{cases} 1 & \text{if } d^i_m \leq d_{\text{comm}} \\ 0 & \text{else.} \end{cases}$$

The observation representation o^i_t of an agent is a combination of all mentioned features:

$$o^i_t = \begin{bmatrix} \text{signal if wall is present} \\ \text{distance to target if in range, else } -1 \\ \text{signal if neighboring agent seeing the target is present} \\ \text{distribution over distances to agents in communication range} \end{bmatrix}.$$

For better processing in a neural network the observation representation as well as the state representation are normalized by their maximum values.

4.2 Task: Cooperative Localization

In the single-agent case an often used scenario to measure the performance of an agent is the *Morris water navigation task* or *Morris water maze* [11, 36]. It was originally designed to study spatial learning and memory of rodents. A mouse is put into a circular pool of water and is supposed to find a platform that is hidden just under the water surface somewhere in the pool which allows it to escape.

In the context of a group of Kilobot-like agents this task can be formulated as follows. A designated target robot is placed somewhere in a limited field sending out a message signature which identifies it as such. The rest of the robots move around searching for the target. In order to receive a reward they to find it in a limited time frame. The following reward function, which rewards the behavior of all agents combined, will be evaluated:

$$R(s_t, a^1_t, \dots, a^M_t) = 5\left(\frac{1}{M^2}\Big(\sum_{i=1}^{M} \delta^i_{\text{target}}\Big)^2 - \frac{1}{M}\sum_{i=1}^{M} 0.01 {a^i_t}^2\right)$$

Scaling with a factor of 5 seems some kind of arbitrary but has shown to improve the stability of the learning algorithm. The reward term includes a small action penalty to slightly enforce smooth action trajectories and the δ's are defined by

$$\delta^i_{\text{target}} = \begin{cases} 1 & \text{if agent } i \text{ is within the communication radius of the target} \\ 0 & \text{else.} \end{cases}$$

While cooperation is not a necessity to fulfill the task (each agent could, in theory, find the goal on its own), we expect that policies learned by a higher amount of agents should achieve a higher reward quicker.

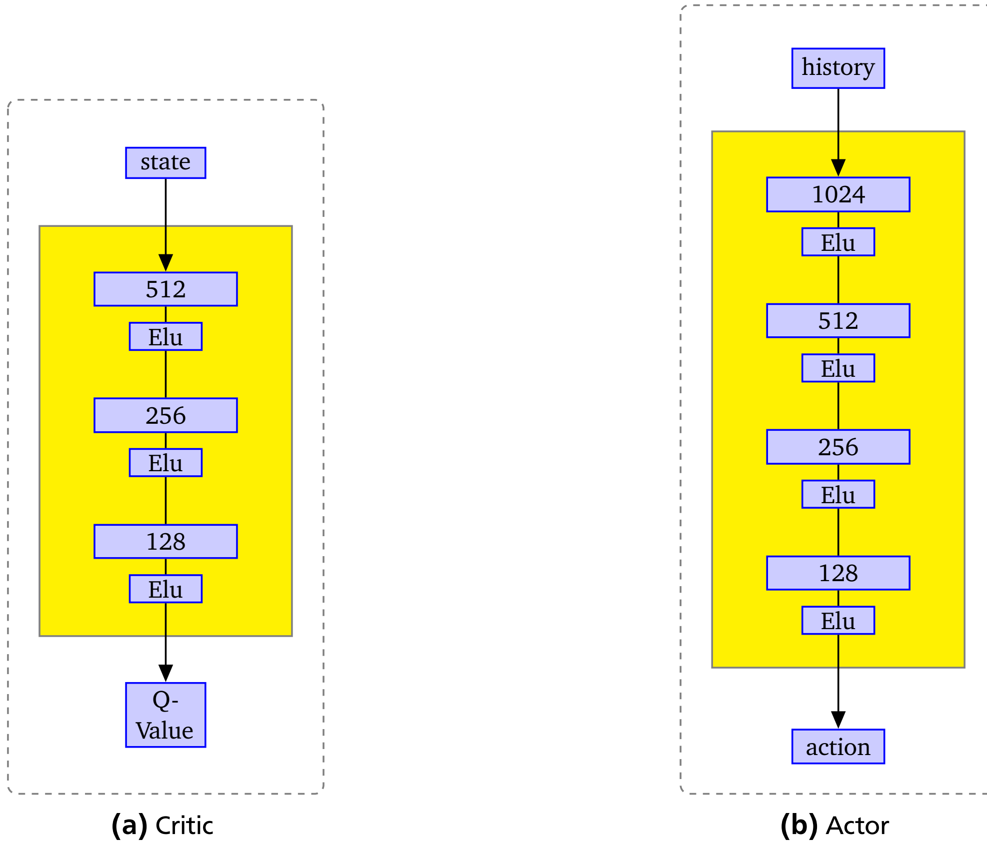
4.3 Experimental Setup

The implementation of the multi-agent environment is embedded into a learning framework which follows Algorithms 4 and 5. Experiments with different numbers of agents and different time horizons for the histories of the agents are simulated multiple times. Policies (the weights and biases of the actor network) are stored at different time points during learning. Also, at these points the current performance of the policy is evaluated. The following parameters apply throughout the learning experiments if not stated differently:

- Environmental parameters:
  - Local state space: $\{(x, y, \phi) \in \mathbb{R}^3 : |x| \leq 10,\ |y| \leq 10,\ |\phi| \leq \pi\}$
  - Communication radius: 4
  - $v_{\text{linear, max}} = 1$ and $v_{\text{angular, max}} = \frac{\pi}{4}$
  - 11 basis functions for distribution over distances
- Learning related parameters:
  - Number of time steps per Episode: 200
  - Number of warmup episodes: 500
  - Number of learning episodes: 10 000
  - Number of evaluation episodes: 20
  - Replay size: 500 000
  - Mini-batch size: 32
  - Decay rate τ for soft target updates: 0.001
  - Base learning rate for critic: 0.001
  - Base learning rate for actor: 0.0001

4.3.1 MAGDDPG Parameter Setup

For the non-recurrent MAGDDPG algorithm we initialize the feed-forward neural networks with the following architectures and parameters. The critic has three and the actor four hidden layers. The input data to each layer is processed by a fully connected layer followed by exponential linear units as activation functions. Figure 4.2 shows both architectures with the corresponding numbers of neurons in the hidden layers. The parameters of the hidden layers are initialized from a uniform distribution



(a) Critic **(b)** Actor

Figure 4.2.: Network architectures for the MAGDDPG algorithm.

$[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the dimensionality of the input of each layer. The output layer of the critic is also a fully connected layer with a linear activation function, initialized by a uniform distribution $[-3 \cdot 10^{-4}, 3 \cdot 10^{-4}]$. The outputs of the actor function u_{left} and u_{right} are bounded in the range 0 and 1 due to the constraints of the simulated agent. While it is possible to use an activation function like the sigmoid function to squash an infinite range of numbers into the interval of 0 and 1, Hausknecht et al. [37] argue that this approach leads to poor results, since the update of the parameters of the actor function depends on the gradient of the Q-function with respect to the action taken. If, for example, the actor output is close to 1 but the critic suggests to increase the action even further due to a higher expected reward, the sigmoid more and more saturates and, in case the action value has to decrease again in a future learning iteration step, it may take many update steps for the sigmoid function to leave the saturated regions. Instead they propose a method called *inverting gradient* which uses a linear output layer and applies the following transformation to the gradient of the critic:

$$\nabla_{a^i} Q(s, \boldsymbol{a}) = \begin{cases} (a^i_{\text{max}} - a^i)/(a^i_{\text{max}} - a^i_{\text{min}}) & \text{if } \nabla_{a^i} \text{ suggests increasing } a^i \\ (a^i_{a^i - \text{min}})/(a^i_{\text{max}} - a^i_{\text{min}}) & \text{otherwise.} \end{cases}$$

Gradients are downscaled as soon as the current activation approaches the boundaries and the direction is inverted if it exceeds the boundaries. As soon as the gradient suggests moving away from the boundary it happens immediately. The parameters of the output layer of the actor are initialized randomly from a uniform distribution $[-3 \cdot 10^{-3}, 3 \cdot 10^{-3}]$

4.3.2 MAGRDPG Parameter Setup

In the MAGRDPG algorithm the feed forward neural network of the actor is substituted with an LSTM recurrent neural network with an internal representation of 128 units, followed by a fully connected

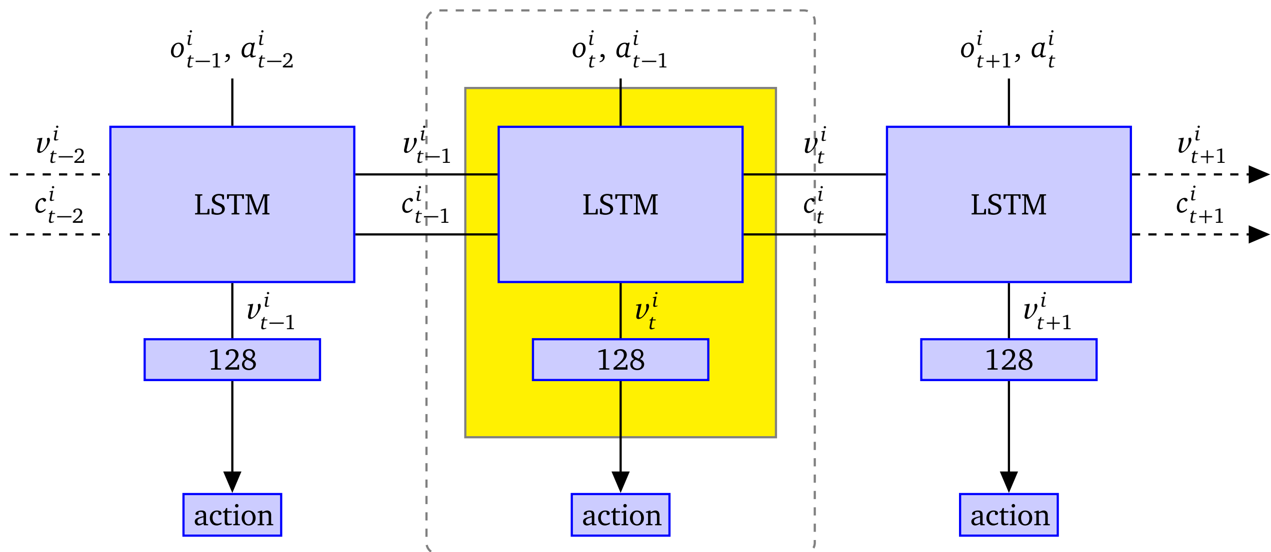


Figure 4.3.: Actor using recurrent LSTM.

layer to match the two-dimensional action space. Its temporal structure is depicted in Figure 4.3. The parameters of the LSTM are initialized by a normal distribution $\mathcal{N}(0, 0.01)$ and the output layer again by a uniform distribution $[-3 \cdot 10^{-3}, 3 \cdot 10^{-3}]$. The network for the critic is identical to the non-recurrent case. Also, the inverting gradient method is applied to the gradients for the actor update.

5 Results and Evaluation

In order to test the performance of the algorithms different experiments were conducted in the environment described in Section 4.3. The first experiment compares the impact of increasing numbers of agents during the learning phase on the average return received for an episode. In order to evaluate the performance of the resulting policies of experiment 1, the learned policies are further evaluated on scenarios with higher numbers of agents. The second experiment investigates the influence of varying time horizons of the agents' histories.

5.1 Evaluation of the MAGDDPG Algorithm

In this section we show the results and evaluation of the experiments using the MAGDDPG algorithm.

5.1.1 Experiment 1: Varying Number of Agents

In the first experiment we investigate how well the algorithm can handle increasing numbers of agents. Starting with the baseline of one agent we further evaluate the learning procedure on two, three, four, six and eight agents. Each case is executed ten times and the results are averaged to display the algorithm's performance. The horizon of the agents' history is 20 time steps throughout all trials.

Discussion

The plots in Figures 5.1 to 5.6 show the averaged undiscounted return at different stages during the learning phase with the parameter setup described in Section 4.3. The first 100 000 time steps show the return of the initial policy with random parameters during the warmup phase where learning has not yet started. 20 evaluation episodes of the policy were executed every 20 000 time steps to generate the data for each plot.

The first observation is that it takes longer to learn a decent policy the more agents learn simultaneously. While in the one-agent and two-agent scenario a successful policy is learned almost immediately after learning starts, the scenario with six agents takes approximately one million update steps until the policy ceases to improve significantly. Furthermore, the learning process is not as stable anymore, which can be seen even more clearly in the case of eight agents where hardly decent policies are learned. A reason for this behavior may be found in the values of the loss function during the update steps. With increasing numbers of agents the loss becomes larger and the resulting gradients eventually lead the algorithm to diverge, despite the use of an adaptive learning rate optimizer. Decreasing the base learning rate for the critic to 0.0001 alleviates this issue with the drawback of an overall slower learning speed. The results with the changed parameter can be found in Figures 5.7 and 5.8 for six and eight agents, respectively. From the learning curve it looks like the policies with eight agents have not yet converged, but at some point the simulations were aborted due to time constraints.

Although the reward function is the same for all experiments, the rewards between scenarios with different numbers of agents cannot be directly compared, due to the particular reward structure being used. For example, in the scenario of two agents the reward for one agent near the target is equal to the reward of four agents near the target in the eight agent scenario. Given a certain probability of an agent finding the target, it is much more likely to achieve a high reward in a low agent count scenario but the resulting

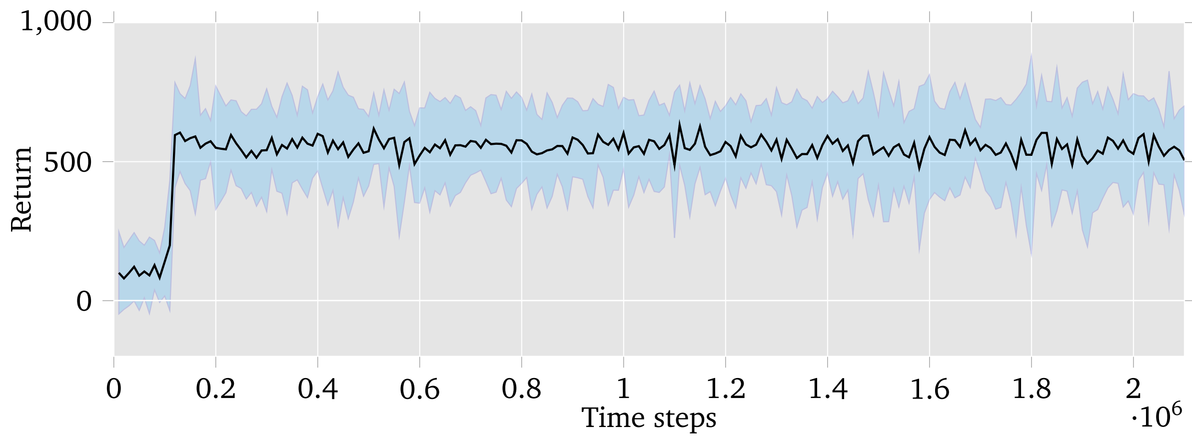


Figure 5.1.: Mean learning curve for one-agent learning scenario, shown with two times standard deviation.

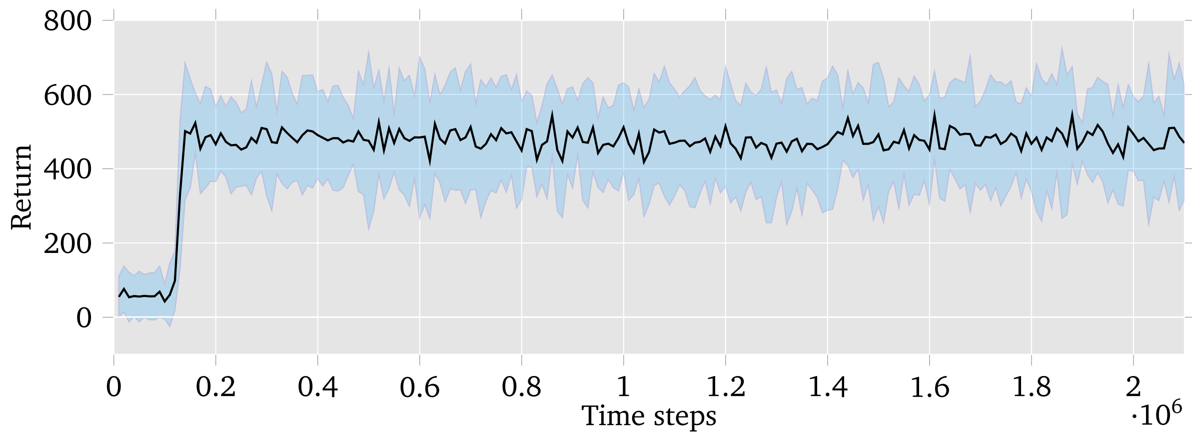


Figure 5.2.: Mean learning curve for two-agent learning scenario, shown with two times standard deviation.

policy is not necessarily superior to one learned with more agents. This can be seen in the evaluation of the policies in Section 5.1.2.

While the learning curves tell us something about the overall performance of the algorithm, we can also take a closer look at the behavior the agents learn. Figures 5.9 and 5.10 show the trajectories of four agents using a policy which was learned on the same number. A cross marks the starting point and a circle the final point after 200 time steps. The target is marked by a diamond shape. The circle around the target describes the communication radius in which an agent needs to be to sense the target. This radius also applies for the moving agents but is not plotted.
The parameters of the networks are initialized to be close to zero which results in almost no motion, as can be seen in Figure 5.9a. After 40 000 updates the agents move in circular patterns without the ability to distinguish between good and bad states. Agents enter and leave the target area or reach the border of the limited space without knowing how to react. At iteration 80 000 the agents start to recognize the target but are unable to stay close to it. After 120 000 iterations the agents have already an idea of how to stay close to the target and also how to react if they reach the state border. This behavior of staying close to the target and avoiding walls is now improved and the trajectory of a policy after 2 000 000 update steps is shown in Figure 5.10d.

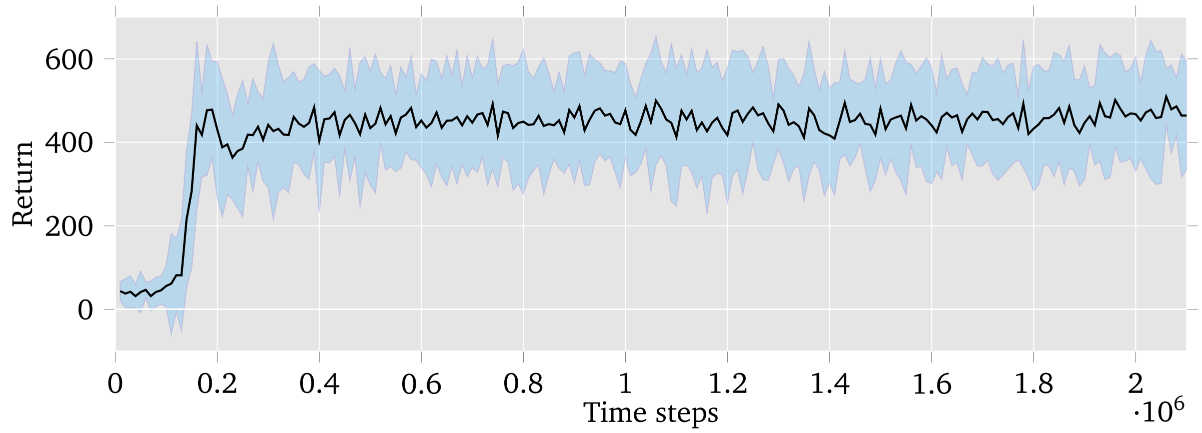


Figure 5.3.: Mean learning curve for three-agent learning scenario, shown with two times standard deviation.

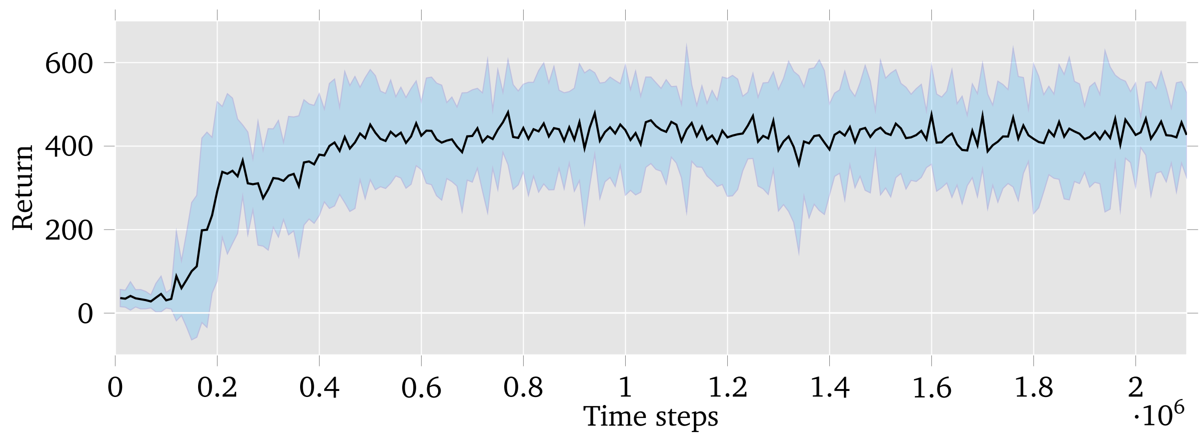


Figure 5.4.: Mean learning curve for four-agent learning scenario, shown with two times standard deviation.

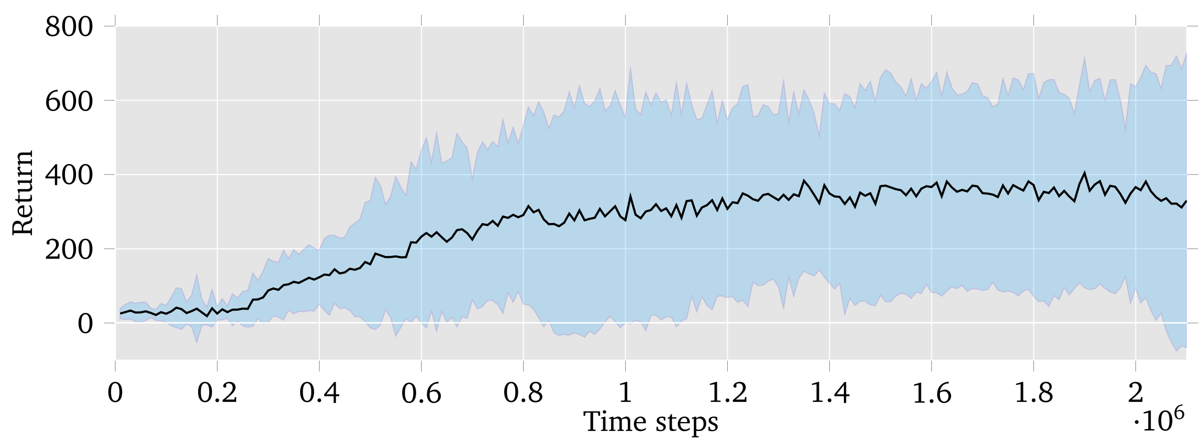


Figure 5.5.: Mean learning curve for six-agent learning scenario, shown with two times standard deviation.

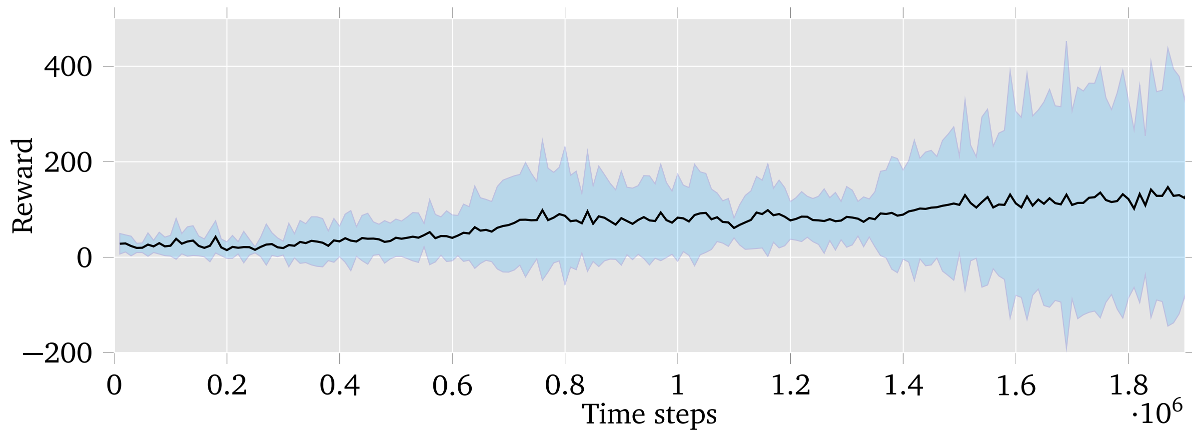


Figure 5.6.: Mean learning curve for eight-agent learning scenario, shown with two times standard deviation.

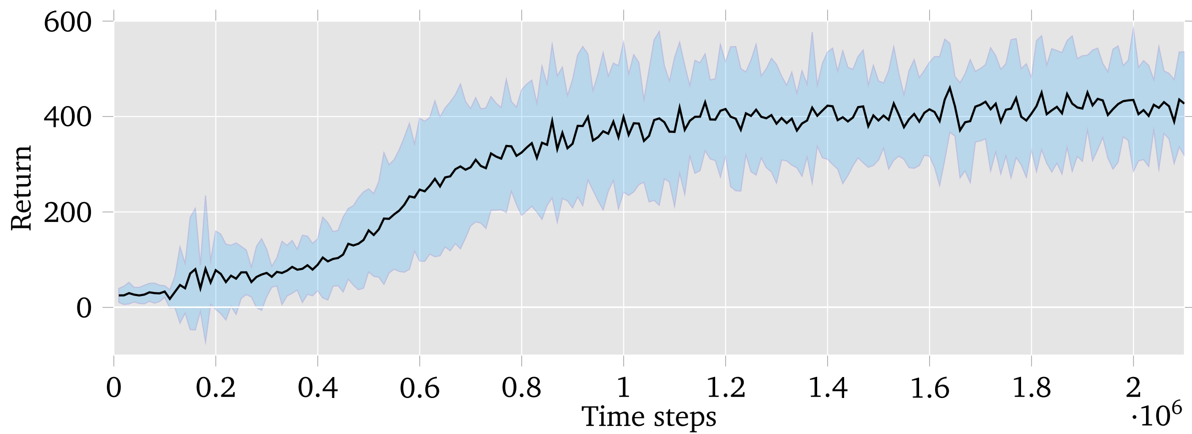


Figure 5.7.: Mean learning curve for six-agent learning scenario, shown with two times standard deviation (decreased base learning rate).

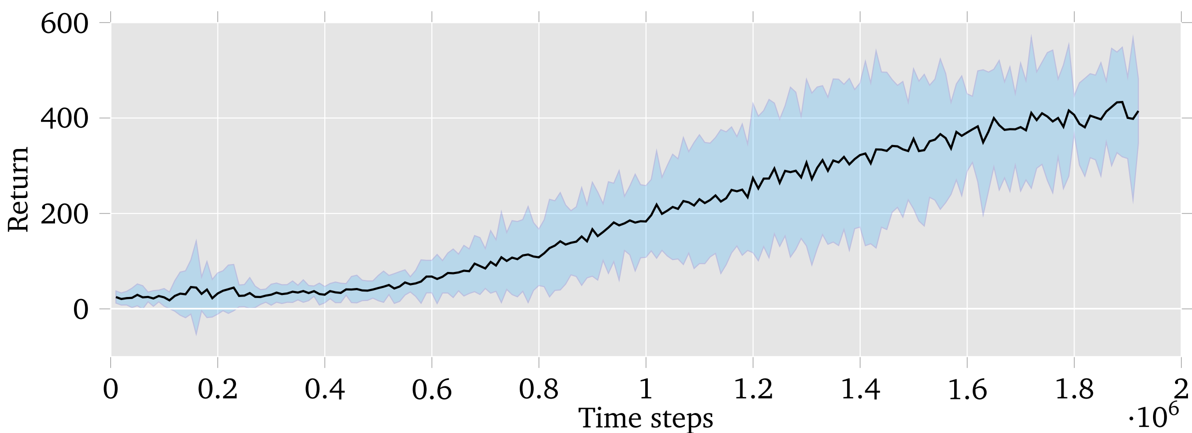
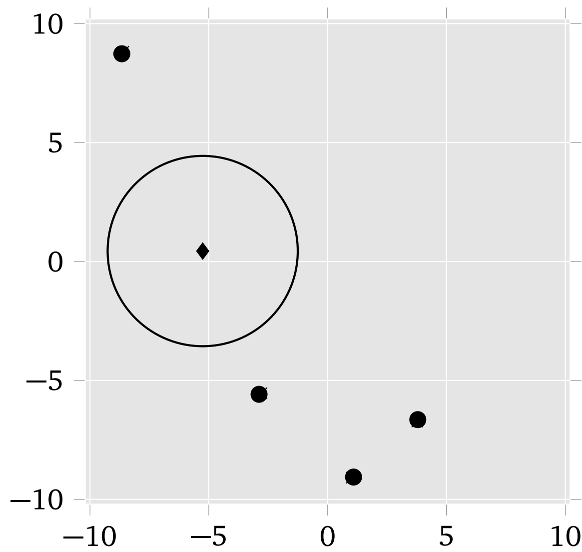
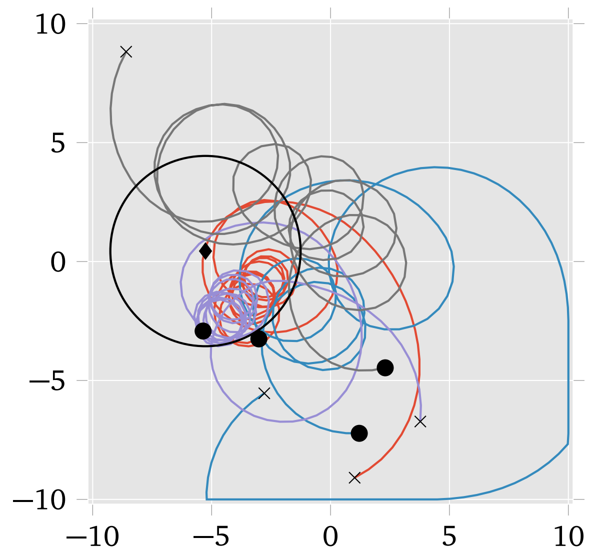


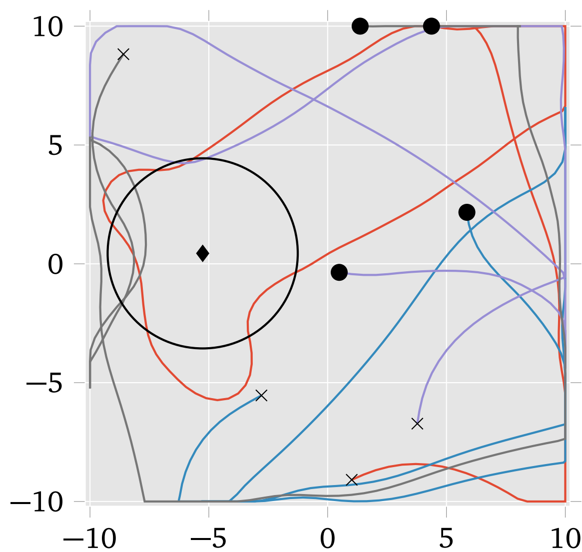
Figure 5.8.: Mean learning curve for eight-agent learning scenario, shown with two times standard deviation (decreased base learning rate).



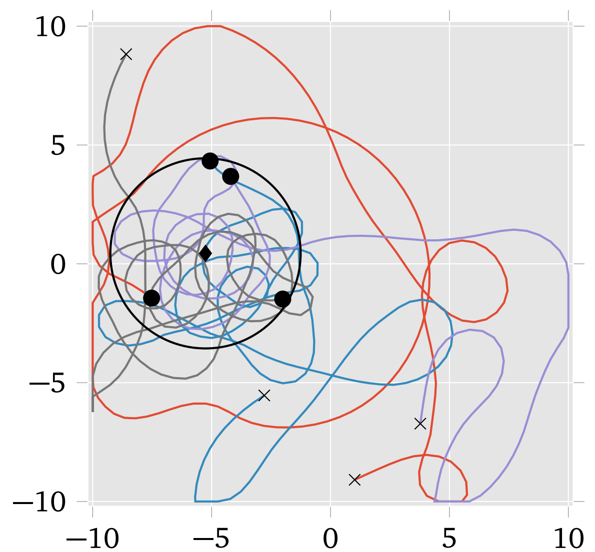
(a) Trajectory using initial policy.



(b) Trajectory after 40 000 iterations.

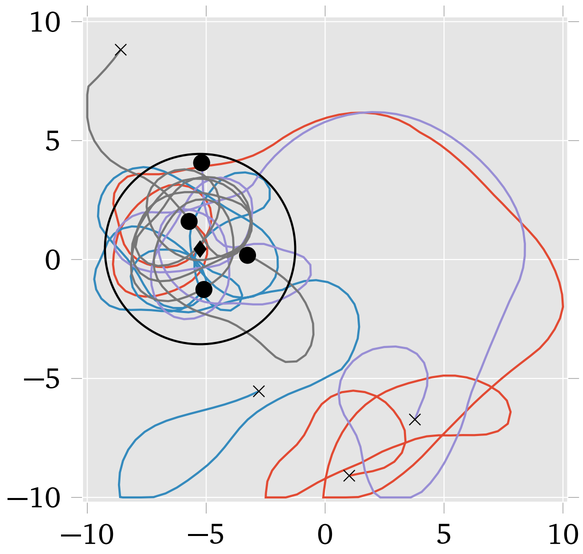


(c) Trajectory after 80 000 iterations.

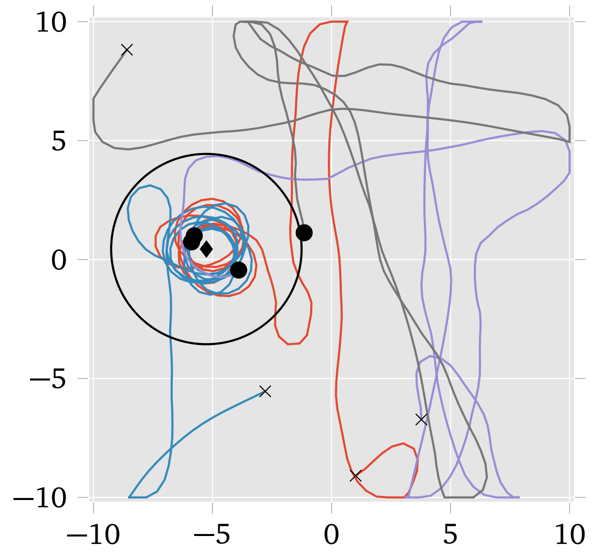


(d) Trajectory after 120 000 iterations.

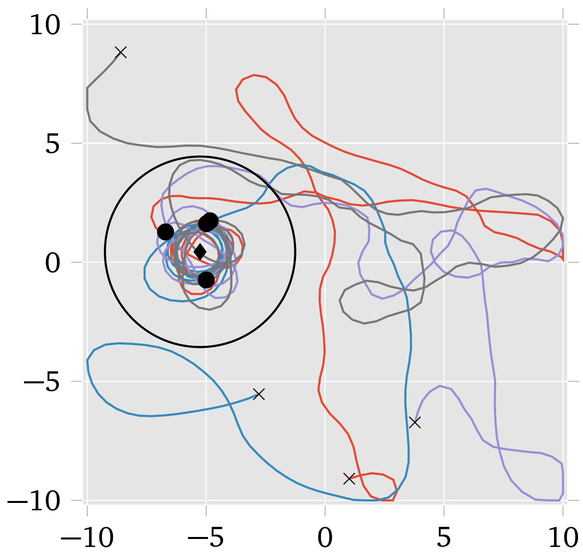
Figure 5.9.: Trajectories of policies learned and executed by four agents after different numbers of update iterations.



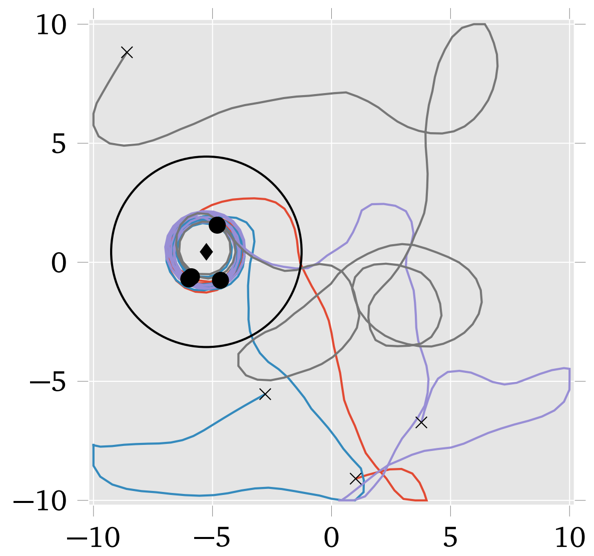
(a) Trajectory after 200 000 iterations.



(b) Trajectory after 500 000 iterations.



(c) Trajectory after 1 000 000 iterations.



(d) Trajectory after 2 000 000 iterations.

Figure 5.10.: Trajectories of policies learned and executed by four agents after different numbers of update iterations.

5.1.2 Evaluation of Policies

In order to compare the performance of policies learned on different numbers of agents, and also to test how well the learned policies can generalize, we take all policies of experiment 1 and evaluate them on scenarios with more agents. For the case where six and eight agents were involved during the learning process we take the policies learned with decreased learning rate.

Discussion

Figures 5.11 and 5.13 show the results of evaluations in scenarios with 16 and 32 agents and an episode length of 500 time steps. The instantaneous reward at each time step is averaged over 500 episodes per policy. The result of the evaluation indicates a benefit of learning with more agents.

In both evaluations, a policy learned with one agent has the lowest expected reward throughout the whole episode and the expected reward increases when using policies that were learned with more agents. Policies learned by more than one agent not only achieve a higher overall reward but also achieve it faster, indicating better communication capabilities. For example, in the scenario of 16 agents the reward earned by policies learned by three and four agents at approximately 200 time steps is already achieved after around 170 time steps by policies learned by six and eight agents. The difference is even bigger in the case of 32 agents where a policy learned by six agents achieves a reward equal to the highest reward of a policy learned by three agents after 250 time steps on average.

Also of interest are the policies learned by eight agents which especially in the case of 32 agents cannot compete with the other policies. This supports the idea that learning of the policies has not yet finished.

The boxplots shown in Figures 5.12 and 5.14 show the undiscounted return of each episode versus the number of agents involved during training for the evaluation with 16 and 32 agents, respectively. The red bar indicates the median return and the blue box the upper and lower quartile. The end of the whiskers include all samples within 1.5 times the upper and lower interquartile range. Values outside of this range are considered as outliers.

Variance of the return is high on all policies but the median confirms the findings by the instantaneous reward with the biggest improvement between policies learned by one, two and three agents. Although a policy learned by one or two agents can show returns as high as returns achieved by policies learned by more than two agents in the evaluation on 32 agents, the boxplot suggests that these should be considered as outliers.

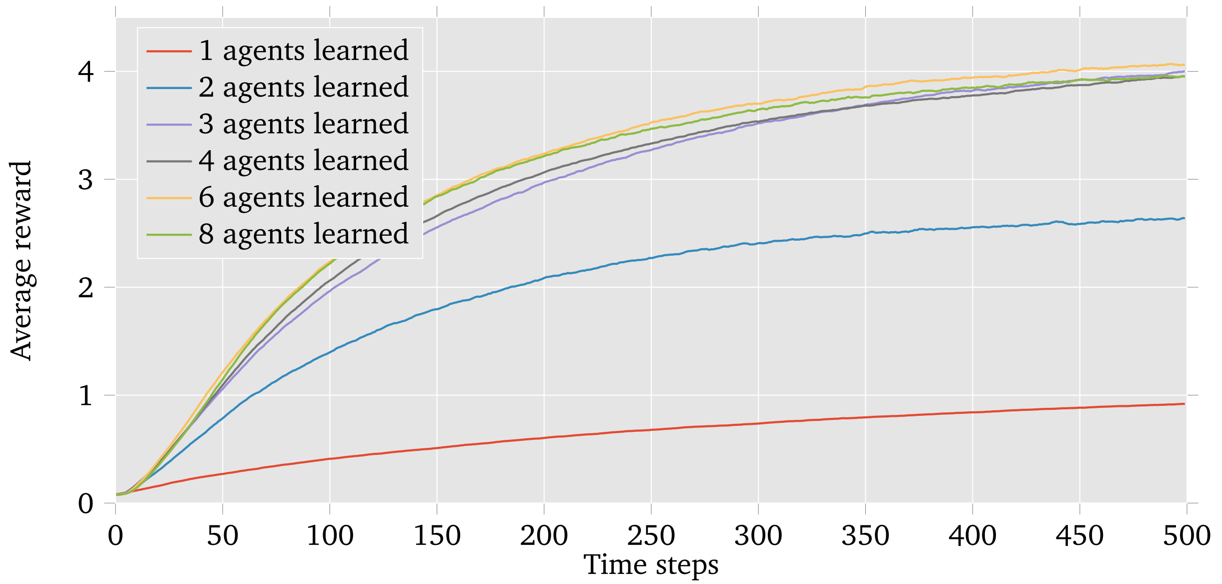


Figure 5.11.: Mean reward over time in a scenario of 16 agents using policies learned by 1, 2, 3, 4, 6 and 8 agents.

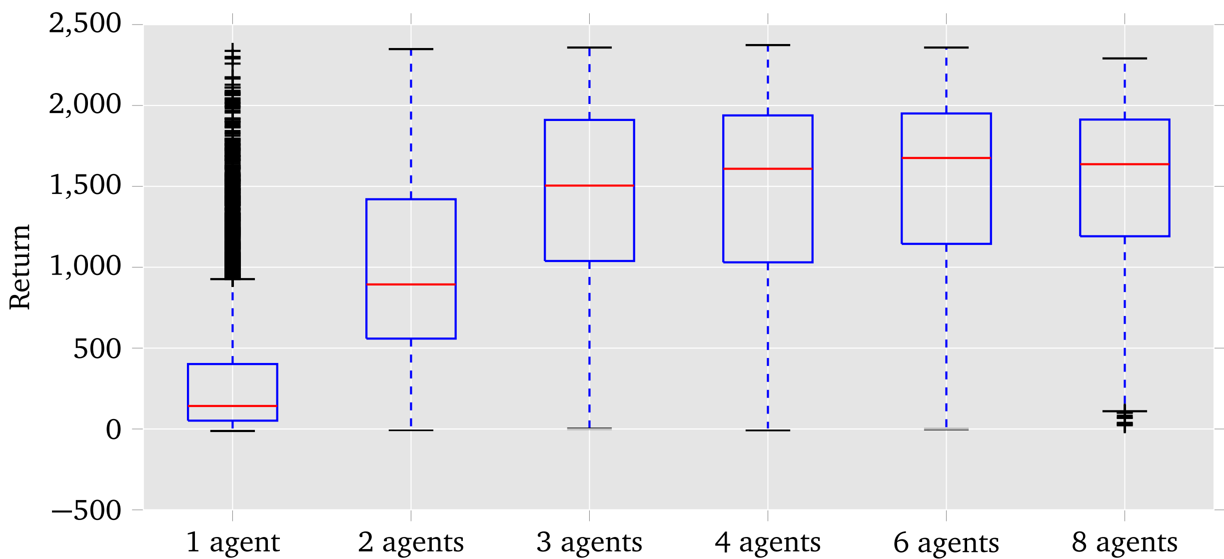


Figure 5.12.: Boxplots of returns in a scenario of 16 agents using policies learned by 1, 2, 3, 4, 6 and 8 agents.

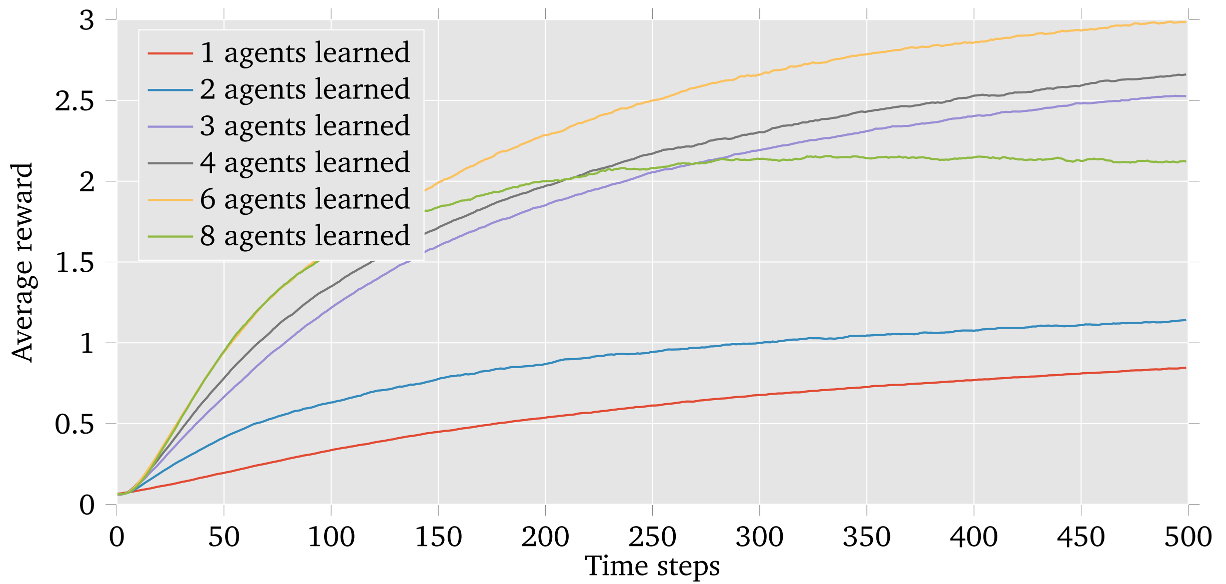


Figure 5.13.: Mean reward over time in a scenario of 32 agents using policies learned by 1, 2, 3, 4, 6 and 8 agents.

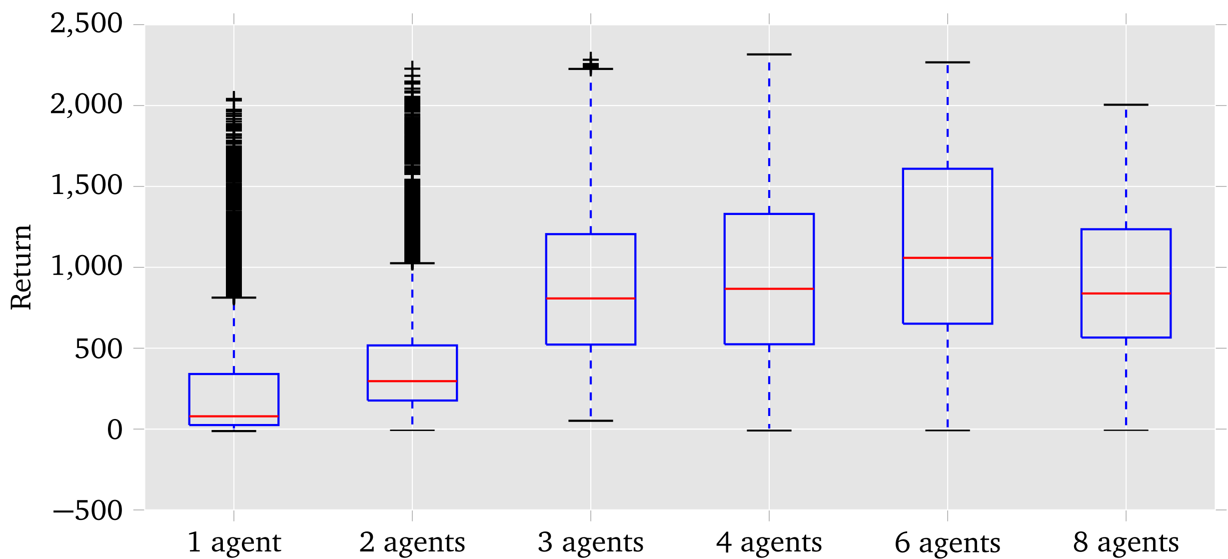


Figure 5.14.: Boxplots of returns in a scenario of 32 agents using policies learned by 1, 2, 3, 4, 6 and 8 agents.

5.1.3 Experiment 2

In the second experiment we investigate how well the task can be executed in the two-agent scenario with varying horizons of the agents' histories. Figures 5.15 to 5.20 show the performance of the algorithm for horizons T_{prev} of the last 1, 2, 3, 4, 5, and 10 time steps. After inspection of the learning curve seen in Figure 5.2 the number of learning episodes was reduced to 2000 (or 400 000 gradient update steps, equivalently) plus 500 warmup episodes.

Discussion

A horizon of $T_{prev} = 1$ is equivalent to a reflex agent who only acts according to its current perception. In a partially observable environment this usually does not lead to a successful behavior which is proven by the learning curve shown in Figure 5.15 where no progress over the course of 400 000 steps is made. Interestingly, the problem can already be solved with a horizon of $T_{prev} = 2$ as can be seen in Figure 5.16. Increasing the horizon further hardly improves performance. This may be due the fact that communication in a two-agent scenario is rather rare and thus, each agent more or less acts as in a single agent scenario.

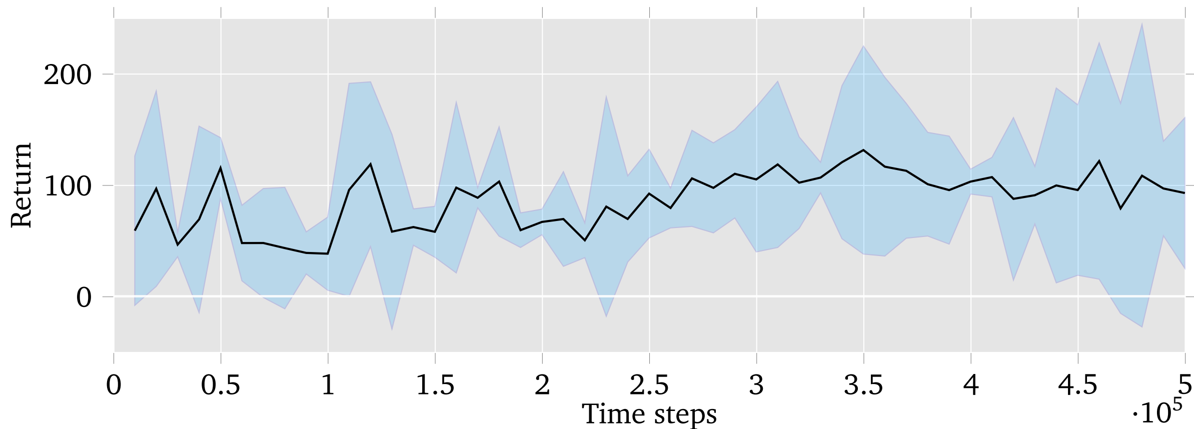


Figure 5.15.: Mean learning curve for two-agent learning scenario with a horizon of 1, shown with two times standard deviation.

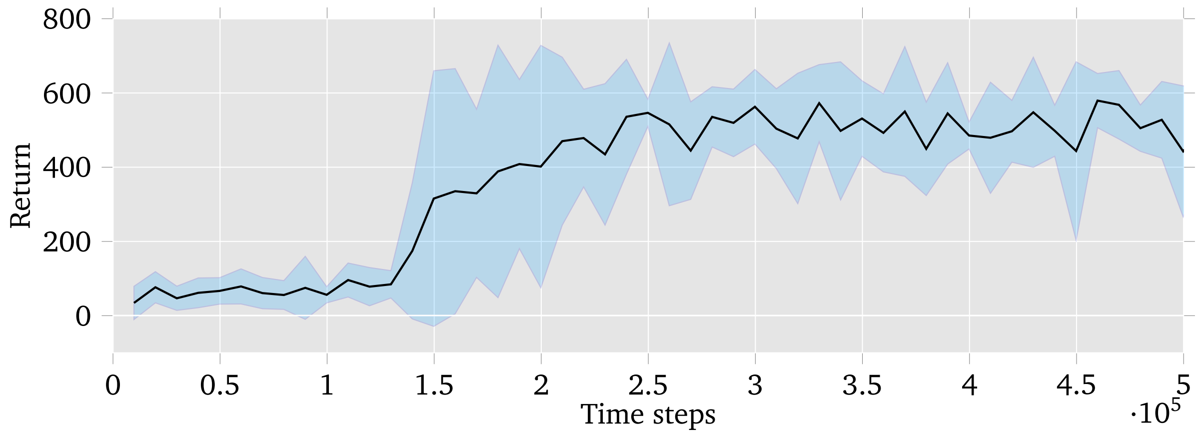


Figure 5.16.: Mean learning curve for two-agent learning scenario with a horizon of 2, shown with two times standard deviation.

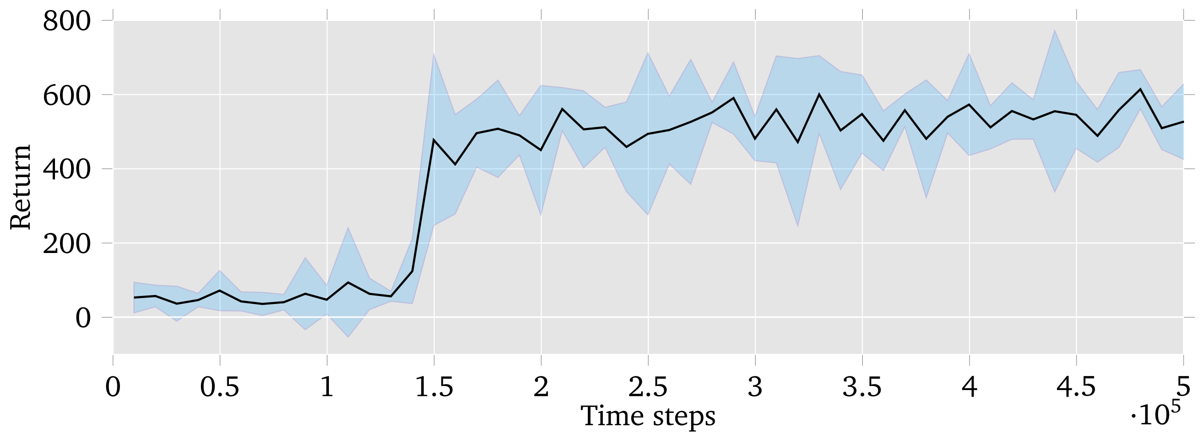


Figure 5.17.: Mean learning curve for two-agent learning scenario with a horizon of 3, shown with two times standard deviation.

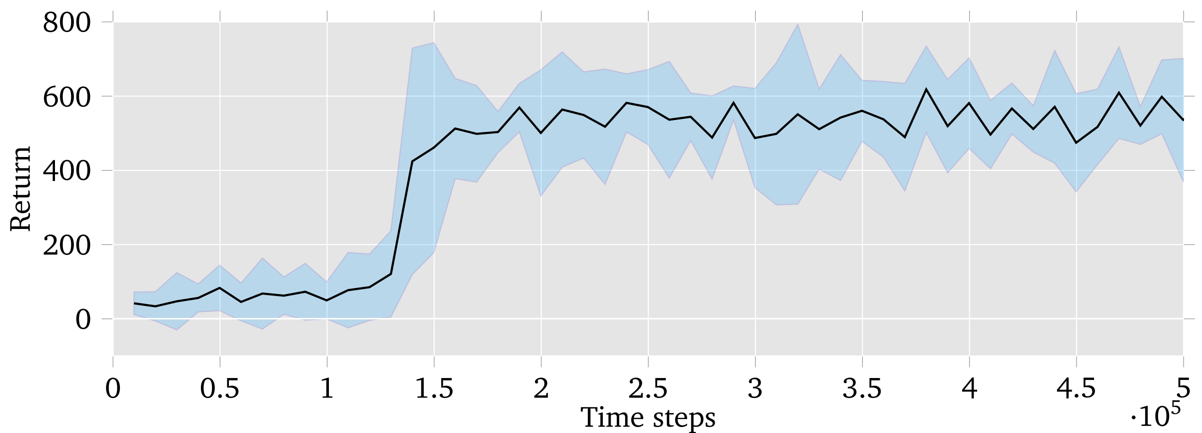


Figure 5.18.: Mean learning curve for two-agent learning scenario with a horizon of 4, shown with two times standard deviation.

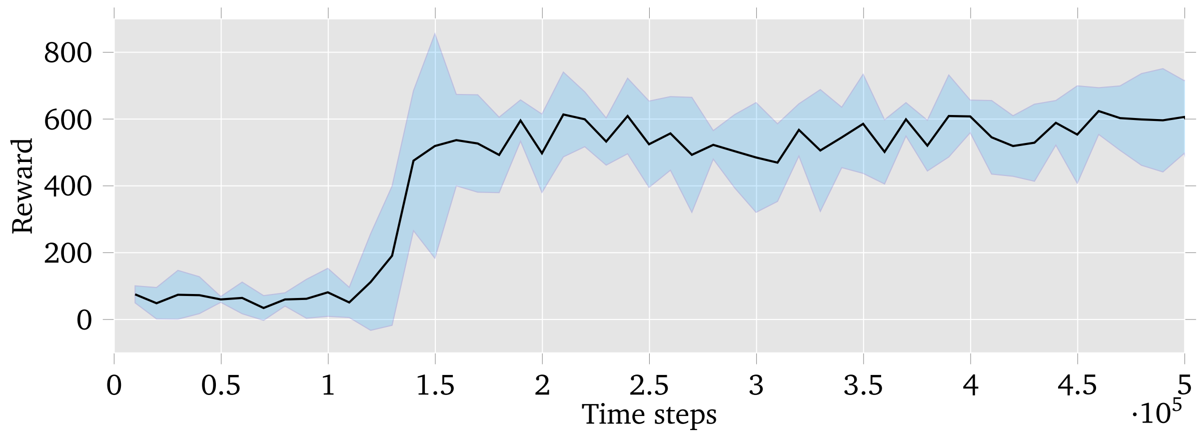


Figure 5.19.: Mean learning curve for two-agent learning scenario with a horizon of 5, shown with two times standard deviation.

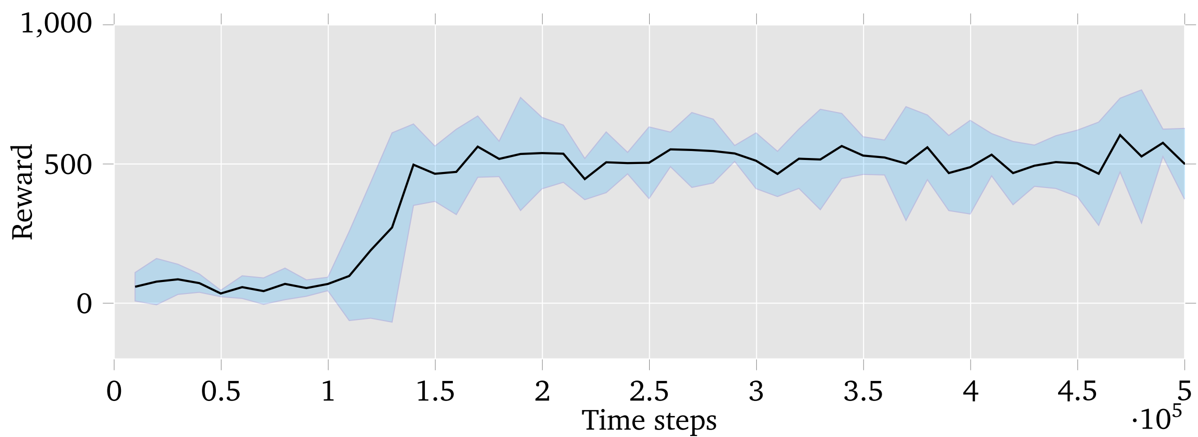


Figure 5.20.: Mean learning curve for two-agent learning scenario with a horizon of 10, shown with two times standard deviation.

5.2 Evaluation of the MAGRDPG Algorithm

Multiple experiments with the MAGRDPG algorithm were conducted including different numbers of agents, full episode horizons and truncated horizons during the learning phase. Unfortunately, non of these experiments yielded reliable or even meaningful results. It is not quite clear what is the reason for this behavior, since the theory behind the algorithm is identical to the non-recurrent algorithm. Possible explanations are the heavily increased learning time, especially during full episode horizon trials where the unfolded network consists of 200 layers or the different representation structure using LSTMs. Finally, mistakes in the implementation cannot be ruled out either.

6 Conclusion and Future Work

Conclusion

In this thesis, we showed that recent developments in the field of deep reinforcement learning, which until now mostly assume a single learning agent, are applicable to multi-agent scenarios including partial observability and continuous state and action spaces. We provide the framework of multi-agent guided deterministic policy gradient, an extension of the deterministic policy gradient, that can learn a policy based on a Q-function of states and joint actions of cooperative homogeneous agents, while the agents execute actions based on their local perception of the system state.

Two different implementations are presented which differ in the way the history of local perceptions is represented, the first one being the MAGDDPG algorithm, which uses feed-forward neural networks and a fixed horizon for the agents' histories, the second one being the MAGRDPG algorithm, which relies on the internal representation of recurrent neural networks, such as LSTMs, to represent the history.

The algorithms were tested in an environment with agents whose capabilities are based on a small-scale robot platform called Kilobots. For this we implemented a simulation environment using transition models according to the movement of and communication between these. The task, for which a policy should be learned, was to locate a target agent, based on the limited information provided to the agents.

The MAGDDPG algorithm successfully solved the task producing reliable policies learned by up to eight agents. The performance of the policies was further evaluated on scenarios with higher numbers of agents. It could be shown that policies learned by more agents benefit from better communication abilities, achieving, on average, a higher reward earlier compared to policies learned by less agents. Unfortunately, in the course of this thesis we were unable to produce meaningful results using the MAGRDPG algorithm for reasons which are not completely clear yet. One problem certainly is the increased computational load, since the depths of the unfolded recurrent network increases with the horizon the agents are trained on. Combined with multiple agents we were unable to achieve the same amount of training steps used in the non-recurrent case.

Future Work

The task, on which the algorithm was tested on in this thesis, does not necessarily require cooperation to be fulfilled so a next step is to find a task which explicitly needs cooperation. This could be, for example, the task of cooperative box pushing, where objects, which are too "heavy" to be moved by a single agent, have to be moved from a starting place to a target destination.

Furthermore, the stability of the algorithm at the moment suffers if more than 8 agents are present during learning. For true swarm behavior it would be interesting to find solutions to scale the algorithm to much higher numbers of learning agents.

Finally, a deeper investigation on why the recurrent algorithm failed to provide meaningful results is necessary.

Bibliography

[1] C. R. Kube and E. Bonabeau, "Cooperative transport by ants and robots," *Robotics and Autonomous Systems*, vol. 30, p. 37, 1998.

[2] A. Martinoli, K. Easton, and W. Agassounon, "Modeling swarm robotic systems: a case study in collaborative distributed manipulation," *The International Journal of Robotics Research*, vol. 23, no. 4, pp. 415–436, 2004.

[3] N. R. Hoff, A. Sagoff, R. J. Wood, and R. Nagpal, "Two foraging algorithms for robot swarms using only local communication," *2010 IEEE International Conference on Robotics and Biomimetics, ROBIO 2010*, pp. 123–130, 2010.

[4] S. Nouyan, R. Gross, M. Bonani, F. Mondada, and M. Dorigo, "Teamwork in self-organized robot colonies," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 4, pp. 695–711, 2009.

[5] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434, 2005.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[8] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[9] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 387–395, 2014.

[10] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," *arXiv*, pp. 1–11, 2015.

[11] R. Morris, "Developments of a water-maze procedure for studying spatial learning in the rat," *Journal of Neuroscience Methods*, vol. 11, no. 1, pp. 47–60, 1984.

[12] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *arXiv*, pp. 1–12, 2015.

[13] J. N. Foerster, Y. M. Assael, N. de Freitas, and S. Whiteson, "Learning to communicate to solve riddles with deep distributed recurrent q-networks," *arXiv:1602.02672v1 [cs.AI]*, 2016.

[14] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed., 1998.

[15] C. J. C. H. Watkins, *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.

[16] C. J. C. H. Watkins and P. Dayan, "Technical note: q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[17] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," *Proceedings of the Twenty-Ninth International Conference on Machine Learning (ICML)*, pp. 457–464, 2012.

[18] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning." Book in preparation for MIT Press, 2016.

[19] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, pp. 386–408, 1958.

[20] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," *Proceedings of the 27th International Conference on Machine Learning*, no. 3, pp. 807–814, 2010.

[21] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," *Aistats*, vol. 15, pp. 315–323, 2011.

[22] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *Under review of ICLR2016*, no. 1997, pp. 1–13, 2015.

[23] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, vol. 165. MIT Press, 1969.

[24] N. Cohen, O. Sharir, and A. Shashua, "On the expressive power of deep learning: A tensor analysis," in *Conference on Learning Theory*, 2016.

[25] Y. Bengio and O. Delalleau, "On the expressive power of deep architectures," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6925 LNAI, pp. 18–36, 2011.

[26] M. Raghu, B. Poole, J. M. Kleinberg, S. Ganguli, and J. Sohl-Dickstein, "On the expressive power of deep neural networks," *CoRR*, vol. abs/1606.05336, 2016.

[27] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, p. 91, 1991.

[28] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[29] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, vol. 4. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[31] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, pp. 1–13, 2014.

[32] L.-J. Lin, *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, CMU, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

[33] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Phys. Rev.*, vol. 36, pp. 823–841, sep 1930.

[34] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein, "The complexity of decentralized control of markov decision processes," *Mathematics of Operations Research*, vol. 27, no. 4, pp. 819–840, 2002.

[35] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *arXiv preprint arXiv:1507.06527*, 2015.

[36] R. D'Hooge and P. P. De Deyn, "Applications of the morris water maze in the study of learning and memory," 2001.

[37] M. Hausknecht and P. Stone, "Deep reinforcement learning in parameterized action space," *arXiv*, pp. 1–12, 2016.

A Appendix

A.1 Multi-Agent Guided Deterministic Policy Gradient

Setting:

- A State s
- Action-observation histories for M agents: h^i for $i = 1, \dots, M$
- A deterministic policy $a = \mu(h^i \mid \theta^\mu)$
- A global reward function $R(s, a^1, a^2, \dots, a^M)$
- An initial state distribution $p_1(s_1)$
- A density at state s' after transitioning for t time steps from state s: $p(s \to s', t, \mu)$
- A discounted state distribution: $\rho^\mu(s') := \int_S \sum_{t=1}^{\infty} \gamma^{t-1} p_1(s) p(s \to s', t, \mu) \mathrm{d}s$
- A Q-function $Q^\mu(s, a^1, \dots, a^M)$, $a^i = \mu(h^i \mid \theta^\mu)$ rating the actions of all agents in the current state and then following policy μ

Performance objective:

$$
\begin{aligned}
J(\mu) &= \int_S \rho^\mu(s) V^\mu(s) \mathrm{d}s \\
&= \int_S \rho^\mu(s) Q^\mu(s, \mu(h^1 \mid \theta^\mu), \dots, \mu(h^M \mid \theta^\mu)) \mathrm{d}s
\end{aligned}
$$

Gradient for two agents:

$$
\begin{aligned}
\nabla_\theta V^\mu(s) =& \nabla_\theta Q^\mu(s, a_1, a_2) \\
=& \nabla_\theta Q^\mu(s, \mu(h^1 \mid \theta^\mu), \mu(h^2 \mid \theta^\mu)) \\
=& \nabla_\theta \left(R(s, \mu(h^1 \mid \theta^\mu), \mu(h^2 \mid \theta^\mu)) + \int_S \gamma p(s' | s, \mu(h^1 \mid \theta^\mu), \mu(h^2 \mid \theta^\mu)) V^\mu(s') \mathrm{d}s' \right) \\
=& \nabla_\theta \mu(h^1 \mid \theta^\mu) \nabla_a R(s, a, \mu(h^2 \mid \theta^\mu)) \Big|_{a=\mu(h^1|\theta^\mu)} \\
&+ \nabla_\theta \mu(h^2 \mid \theta^\mu) \nabla_a R(s, \mu(h^1 \mid \theta^\mu), a) \Big|_{a=\mu(h^2|\theta^\mu)} \\
&+ \nabla_\theta \int_S \gamma p(s' | s, \mu(h^1 \mid \theta^\mu), \mu(h^2 \mid \theta^\mu)) V^\mu(s') \mathrm{d}s' \\
=& \nabla_\theta \mu(h^1 \mid \theta^\mu) \nabla_a R(s, a, \mu(h^2 \mid \theta^\mu)) \Big|_{a=\mu(h^1|\theta^\mu)} \\
&+ \nabla_\theta \mu(h^2 \mid \theta^\mu) \nabla_a R(s, \mu(h^1 \mid \theta^\mu), a) \Big|_{a=\mu(h^2|\theta^\mu)} \\
&+ \int_S \gamma p(s' | s, \mu(h^1 \mid \theta^\mu), \mu(h^2 \mid \theta^\mu)) \nabla_\theta V^\mu(s') \\
&+ \Big(\nabla_\theta \mu(h^1 \mid \theta^\mu) \nabla_a p(s' | s, a, \mu(h^2 \mid \theta^\mu)) \Big|_{a=\mu(h^1|\theta^\mu)}
\end{aligned}
$$

$$
\begin{aligned}
&+ \nabla_\theta \mu(h^2 \mid \theta^\mu) \nabla_a \, p(s'|s, \mu(h^1 \mid \theta^\mu), a)\big|_{a=\mu(h^2|\theta^\mu)} \Big) V^\mu(s')\mathrm{d}s' \\
=& \nabla_\theta \mu(h^1 \mid \theta^\mu) \; \nabla_a \left(R(s, a, \mu(h^2 \mid \theta^\mu)) + \int_{\mathcal{S}} \gamma p(s'|s, a, \mu(h^2 \mid \theta^\mu)) V^\mu(s')\mathrm{d}s' \right)\Bigg|_{a=\mu(h^1|\theta^\mu)} \\
&+ \nabla_\theta \mu(h^2 \mid \theta^\mu) \; \nabla_a \left(R(s, \mu(h^1 \mid \theta^\mu), a) + \int_{\mathcal{S}} \gamma p(s'|s, \mu(h^1 \mid \theta^\mu), a) V^\mu(s')\mathrm{d}s' \right)\Bigg|_{a=\mu(h^2|\theta^\mu)} \\
&+ \int_{\mathcal{S}} \gamma p(s'|s, \mu(h^1 \mid \theta^\mu), \mu(h^2 \mid \theta^\mu)) \nabla_\theta V^\mu(s')\mathrm{d}s' \\
=& \nabla_\theta \mu(h^1 \mid \theta^\mu) \; \nabla_a Q^\mu(s, a, \mu(h^2 \mid \theta^\mu))\big|_{a=\mu(h^1|\theta^\mu)} \\
&+ \nabla_\theta \mu(h^2 \mid \theta^\mu) \; \nabla_a Q^\mu(s, \mu(h^1 \mid \theta^\mu), a)\big|_{a=\mu(h^2|\theta^\mu)} \\
&+ \int_{\mathcal{S}} \gamma p(s \to s', 1, \mu) \nabla_\theta V^\mu(s')\mathrm{d}s'
\end{aligned}
$$

Iterating the formula:

$$
\begin{aligned}
=& \nabla_\theta \mu(h^1 \mid \theta^\mu) \; \nabla_a Q^\mu(s, a, \mu(h^2 \mid \theta^\mu))\big|_{a=\mu(h^1|\theta^\mu)} \\
&+ \nabla_\theta \mu(h^2 \mid \theta^\mu) \; \nabla_a Q^\mu(s, \mu(h^1 \mid \theta^\mu), a)\big|_{a=\mu(h^2|\theta^\mu)} \\
&+ \int_{\mathcal{S}} \gamma p(s \to s', 1, \mu) \nabla_\theta \mu(h'^1 \mid \theta^\mu) \; \nabla_a Q^\mu(s', a, \mu(h'^2 \mid \theta^\mu))\big|_{a=\mu(h'^1|\theta^\mu)} \mathrm{d}s' \\
&+ \int_{\mathcal{S}} \gamma p(s \to s', 1, \mu) \nabla_\theta \mu(h'^2 \mid \theta^\mu) \; \nabla_a Q^\mu(s', \mu(h'^1 \mid \theta^\mu), a)\big|_{a=\mu(h'^2|\theta^\mu)} \mathrm{d}s' \\
&+ \int_{\mathcal{S}} \gamma p(s \to s', 1, \mu) \int_{\mathcal{S}} \gamma p(s' \to s'', 1, \mu) \nabla_\theta V^\mu(s'')\mathrm{d}s''\mathrm{d}s' \\
=& \nabla_\theta \mu(h^1 \mid \theta^\mu) \; \nabla_a Q^\mu(s, a, \mu(h^2 \mid \theta^\mu))\big|_{a=\mu(h^1|\theta^\mu)} \\
&+ \nabla_\theta \mu(h^2 \mid \theta^\mu) \; \nabla_a Q^\mu(s, \mu(h^1 \mid \theta^\mu), a)\big|_{a=\mu(h^2|\theta^\mu)} \\
&+ \int_{\mathcal{S}} \gamma p(s \to s', 1, \mu) \nabla_\theta \mu(h'^1 \mid \theta^\mu) \; \nabla_a Q^\mu(s', a, \mu(h'^2 \mid \theta^\mu))\big|_{a=\mu(h'^1|\theta^\mu)} \mathrm{d}s' \\
&+ \int_{\mathcal{S}} \gamma p(s \to s', 1, \mu) \nabla_\theta \mu(h'^2 \mid \theta^\mu) \; \nabla_a Q^\mu(s', \mu(h'^1 \mid \theta^\mu), a)\big|_{a=\mu(h'^2|\theta^\mu)} \mathrm{d}s' \\
&+ \int_{\mathcal{S}} \gamma^2 p(s \to s', 2, \mu) \nabla_\theta V^\mu(s')\mathrm{d}s' \\
&\vdots \\
=& \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t p(s \to s', t, \mu) \nabla_\theta \mu(h'^1 \mid \theta^\mu) \; \nabla_a Q^\mu(s', a, \mu(h'^2 \mid \theta^\mu))\big|_{a=\mu(h'^1|\theta^\mu)} \mathrm{d}s' \\
&+ \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t p(s \to s', t, \mu) \nabla_\theta \mu(h'^2 \mid \theta^\mu) \; \nabla_a Q^\mu(s', \mu(h'^2 \mid \theta^\mu), a)\big|_{a=\mu(h'^2|\theta^\mu)} \mathrm{d}s'
\end{aligned}
$$

Now taking expectation over S_1:

$$
\begin{aligned}
\nabla_\theta J(\mu) =& \nabla_\theta \int_{\mathcal{S}} p_1(s) V^\mu(s) \mathrm{d}s \\
=& \int_{\mathcal{S}} p_1(s) \nabla_\theta V^\mu(s) \mathrm{d}s \\
=& \int_{\mathcal{S}} p_1(s) \Bigg(\int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t p(s \to s', t, \mu) \nabla_\theta \mu(h'^1 \mid \theta^\mu)\ \nabla_a Q^\mu(s', a, \mu(h'^2 \mid \theta^\mu))\big|_{a=\mu(h'^1|\theta^\mu)} \mathrm{d}s' \\
& + \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t p(s \to s', t, \mu) \nabla_\theta \mu(h'^2 \mid \theta^\mu)\ \nabla_a Q^\mu(s', \mu(h'^2 \mid \theta^\mu), a)\big|_{a=\mu(h'^2|\theta^\mu)} \mathrm{d}s' \Bigg) \mathrm{d}s \\
=& \int_{\mathcal{S}} \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t p_1(s) p(s \to s', t, \mu) \nabla_\theta \mu(h'^1 \mid \theta^\mu)\ \nabla_a Q^\mu(s', a, \mu(h'^2 \mid \theta^\mu))\big|_{a=\mu(h'^1|\theta^\mu)} \mathrm{d}s' \mathrm{d}s \\
& + \int_{\mathcal{S}} \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t p_1(s) p(s \to s', t, \mu) \nabla_\theta \mu(h'^2 \mid \theta^\mu)\ \nabla_a Q^\mu(s', \mu(h'^1 \mid \theta^\mu), a)\big|_{a=\mu(h'^2|\theta^\mu)} \mathrm{d}s' \mathrm{d}s \\
=& \int_{\mathcal{S}} \rho^\mu(s) \nabla_\theta \mu(h^1 \mid \theta^\mu)\ \nabla_a Q^\mu(s, a, \mu(h^2 \mid \theta^\mu))\big|_{a=\mu(h^1|\theta^\mu)} \mathrm{d}s \\
& + \int_{\mathcal{S}} \rho^\mu(s) \nabla_\theta \mu(h^2 \mid \theta^\mu)\ \nabla_a Q^\mu(s, \mu(h^1 \mid \theta^\mu), a)\big|_{a=\mu(h^2|\theta^\mu)} \mathrm{d}s \\
=& \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu(h^1 \mid \theta^\mu)\ \nabla_a Q^\mu(s, a, \mu(h^2 \mid \theta^\mu))\big|_{a=\mu(h^1|\theta^\mu)}] \\
& + \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu(h^2 \mid \theta^\mu)\ \nabla_a Q^\mu(s, \mu(h^1 \mid \theta^\mu), a)\big|_{a=\mu(h^2|\theta^\mu)}] \\
=& \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu(h^1 \mid \theta^\mu)\ \nabla_a Q^\mu(s, a, \mu(h^2 \mid \theta^\mu))\big|_{a=\mu(h^1|\theta^\mu)} \\
& + \nabla_\theta \mu(h^2 \mid \theta^\mu)\ \nabla_a Q^\mu(s, \mu(h^1 \mid \theta^\mu), a)\big|_{a=\mu(h^2|\theta^\mu)}]
\end{aligned}
$$

For M agents:

$$
\nabla_\theta J(\mu) = \mathbb{E}_{s\sim\rho^\mu}\left[\sum_{n=1}^{M} \nabla_\theta \mu(h^i \mid \theta^\mu)\ \nabla_{a^i} Q^\mu(s, a^1, \ldots, a^M)\big|_{a^i=\mu(h^i|\theta^\mu)} \right]
$$