Communication and Cooperation in Decentralized Multi-Agent Reinforcement Learning

Nico Ring

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh

2018

Abstract

Many real-world problems can only be solved when multiple agents work together. Each agent often has only a partial view of the whole environment. Therefore, communication and cooperation are important abilities for agents in these environments. In this thesis, we test the ability of an existing deep multi-agent actor-critic algorithm to cope with partially observable scenarios. In addition, we propose to adapt this algorithm to use recurrent neural networks, which enables using information from past steps to improve action-value predictions. We evaluate the algorithms in cooperative environments that require cooperation and communication. Furthermore, we simulate varying degrees of partial observability of actions and observations from other agents. Our experiments show that our proposed recurrent adaptation achieves equally good or significantly better success rates compared to the original method when only actions of other agents are partially observed. However, for other settings our adaptation also can perform worse than the original method, especially when the observations of other agents are only partially observed.

Acknowledgements

I would like to thank my supervisor Dr Stefano V. Albrecht for his time and advice during the project as well as for his comments on this thesis. Furthermore, I want to thank my family and friends for their support throughout my whole studies.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Nico Ring*)

To my parents.

Contents

Chapter 1

Introduction

Cooperation and communication are important abilities for intelligent and autonomous agents that are deployed in complex environments including multiple agents. In many real-world problems, each agent has only partial observability of the environment. In these settings, communication with other agents enables them to exchange information such that each agent gets a better picture of the situation. Communication can also be necessary to coordinate the behavior of the agents, where the goals or states of the agents can be seen as a special case of partial observability.

It is possible to design the communication and cooperation of agents in advance, such as sharing certain state information with other agents [2, 47, 49, 53]. However, agents might be required to optimize their behavior online, because environments can be so complex that a priori design of policies is difficult [44, 48]. This is especially the case when the environment changes over time. Therefore, we are interested in algorithms that learn a policy autonomously for a given task.

Reinforcement learning (RL) algorithms are able to optimize a policy for an environment, while only being given a numerical reward signal [51]. Recently, single-agent RL methods combined with neural networks have proven to exceed human performance in tasks with huge state spaces such as the game of Go [46] and Atari games [35], as well as tasks with high dimensional action spaces such as locomotive control [29].

There also has been a lot of recent work on using deep RL for learning to cooperate and communicate in multi-agent settings [10, 12, 13, 11, 16, 31, 36, 40, 50].

These approaches show that deep RL is a promising approach, as the experiments showed the emergence of cooperation and simple communication in various environments. However, these approaches rely on *centralized training*, which means that more information is available during training than they have access to at test time. An example are robots in a lab setting that are connected to a central computer that shares information between all robots, e.g. their positions [24]. Other examples are differentiable communication channels [10, 50] and access to the policies of other agents [31].

The assumption of a centralized training is problematic, because it is often not possible to share additional information during training. For example, robots that are deployed in real-world scenarios have only a limited communication bandwidth. Therefore, it might not be feasible to share information such as their policies. There also may be no central entity which has full knowledge about the environment. Thus, algorithms for these scenarios must be able to deal with partial observability and limited communication during training and execution.

Approaches to deep decentralized multi-agent RL exist. For example, Omidshafiei et al. [39] proposed an algorithm based on recurrent neural networks that is able to train cooperative agents in decentralized settings, which means agents have no knowledge of each other. However, their approach does not scale to large action spaces and we think that the assumption of having no information about other agents at all is too restrictive.

Therefore, we want to investigate whether the *multi-agent deep deterministic policy gradients* (MADDPG) algorithm proposed by Lowe et al. [31] applies to decentralized settings. This algorithm scales well for high-dimensional continuous action and observation spaces, but assumes centralized training with access to other agents' policies, actions and observations. They already proposed the possibility to model the policies of other agents, which removes one assumption of the centralized training. During the centralized training the actions and observations of all agents are used to train critic networks, these are then used to train policies that only depend on local observations. Thus, the question that remains is how much the algorithm depends on having access to other agents' actions and observations to train the critic.

The contribution of this thesis is twofold: First, we examine how much the MAD-

DPG method depends on other agents' actions and observations about the environment, by making these partially observable for the individual agents during the learning process. We simulate partial observability by adding noise to observed actions and observations. Second, we propose an adaption to MADDPG named Rec-MADDPG that uses recurrent neural networks as critics to deal with the partial observability. Using recurrent neural networks in partially observable environments has already been proposed for the single-agent case [15, 17] and also for multi-agent scenarios [39], as mentioned above. However, to the best of our knowledge, this is the first multi-agent actor-critic algorithm for continuous action and state spaces for partially observable environments. Furthermore, we introduce a way to simulate and experiment with different degrees of partial observability of other agents in the cooperative environments of Mordatch and Abbeel [36].

This thesis is structured as follows: In the next chapter, we give background information about deep neural networks, single-agent as well as multi-agent RL, including the MADDPG algorithm. The second chapter first formalizes the problem of partial observability of other agents in decentralized settings, proposes a recurrent multi-agent learning algorithm for partially observed decentralized environments and introduces the multi-agent environments in which we test and compare the proposed algorithm with MADDPG. In Chapter 4, we present and discuss the results of our experiments. Then we compare our approach to related work, in Chapter 5 and finally propose future directions and conclude our work in the last chapter.

Chapter 2

Background

This chapter gives the necessary background information for the rest of this thesis. We first introduce deep neural networks, including feedforward and recurrent networks. After that, in the second section, we introduce the standard single-agent RL setting and traditional learning algorithms. Then, we present an actor-critic method and describe how to combine it with deep neural networks. In the third section, we introduce the multi-agent RL problem. We describe the used theoretical multi-agent framework and discuss approaches to the multi-agent learning problem. Finally, we present the multi-agent deep actor-critic method that is examined and extended in this work.

2.1 Deep Neural Networks

Deep neural networks [27] (DNN) have recently outperformed other machine learning techniques on many tasks, for example image recognition [25] and machine translation [56]. In the domain of RL, deep neural networks were the driving force behind the success of playing Atari games [35] with super-human performance in some games. Neural networks also already have been successfully applied to multi-agent problems, for example, beating professional players in the game of Go [46] or learning to communicate [10, 50]

Representation-learning methods are able to automatically discover the representation of raw input data that is needed for the task at hand, e.g. regression or classification. Deep neural networks are representation-learning methods that

compute multiple levels of representation before computing the final output [27]. Each representation is computed by a simple non-linear layer that takes the representation from the previous layer and transforms it into a slightly more abstract one. By combining enough of such transformations, it is possible to approximate very complex functions without the requirement of any feature engineering.

The remainder of this section gives a brief overview about the structure of neural networks, optimization techniques and also introduces recurrent neural networks that are able to store information over time.

2.1.1 Feedforward Networks

Feedforward networks consist of a stack of non-linear layers, where each layer is a linear transformation of the input features to a number of output features, denoted as hidden units. Then, a non-linear function is applied to each of these output features. The following equation describes a forward computation of a layer in a neural network. The input into the layer is a d-dimensional vector $\mathbf{x}$, the output is the h-dimensional vector $\mathbf{y}$ and the layer has weights w_{ij} and bias b_j:

$$z_j = \sum_{i=1}^{d} w_{ij} x_i + b_j, \quad y_j = f(z_j), \tag{2.1}$$

where $\mathbf{z}$ are called the activations of the layer and $f(\cdot)$ is a non-linear function such as sigmoid $\sigma(x) = 1/(1+\exp(-x))$. The weights of each layer can be trained with the *backpropagation* method: given a derivative of the error of the prediction with respect to the outputs of the network we can apply the chain rule step by step and retrieve the error-derivative with respect to each weight and bias. More formally, for the layer described in equation 2.1, the derivatives can be computed as follows, given $\frac{\partial E}{\partial y_j}$, the error-derivative of the layer's output:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j}, \tag{2.2}$$

this gives us the error-derivatives of the activations, which can then be used to compute the error-derivatives of the weights and biases:

$$\frac{\partial E}{w_{ij}} = x_i \frac{\partial E}{\partial z_j}, \quad \frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial z_j} \tag{2.3}$$

and finally we compute the error-derivatives for the input of the layer which can then be used to compute the error-derivatives in the next layer:

$$\frac{\partial E}{\partial x_i} = \sum_{j=1}^{h} w_{ij} \frac{\partial E}{z_j}. \tag{2.4}$$

We are now able to learn the weights of a neural network, for example for a regression task, by providing input data and the target values. First, we compute a complete forward pass, i.e. starting from the input, all outputs of the layers are computed until we get the final prediction. Then, we compute the error of the prediction based on the target value and backpropagate the error. After that, we update the the weights to minimize the error by setting them to $w_{ij} \leftarrow w_{ij} - \alpha \frac{\partial E}{w_{ij}}$, where α is the learning rate that determines how much we change the weights.

Only using one example to compute the error-derivative of the weights is called *stochastic gradient descent*, because we estimate the true derivative over the whole dataset with a monte carlo approximation of the sum of all derivatives. It is common to sample mini-batches of several samples to estimate the derivatives instead of using single samples because it stabilizes the updates and is more efficient to compute. There are extensions of stochastic gradient descent that lead to a faster convergence. We use Adam [23] to train the neural networks in our experiments. Adam maintains per-parameter learning rates that improves performance on problems with sparse gradients and adapts the learning rates based on the average first and second momentum of the gradient that makes it perform well on online and non-stationary problems, which is especially appealing for RL problems.

The derivative of the sigmoid function approaches zero for very large or very small inputs, which can lead to the *vanishing gradient* problem [5], where layers can have a gradient close to zero which slows down learning for following layers. Therefore, it is common to use non-linear functions with a non-zero gradient. The most common used one and the function that we use for all our experiments, is the Rectified Linear Unit (ReLU) $f(x) = \max(0, x)$ [37], which has a non-zero gradient for positive activations.

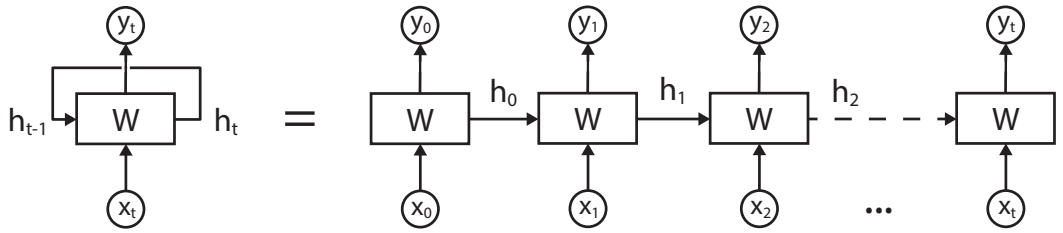


Figure 2.1: (left) structure of a simple recurrent network, (right) network unrolled over time [38].

2.1.2 Recurrent Networks

In contrast to feedforward networks, recurrent neural networks are able to use an internal memory to process sequential data and store information over multiple time-steps. This is done by passing hidden states to the next time-step, where they are treated as inputs to the network. When we unroll recurrent networks in time (as shown in Figure 2.1), we can see them as very deep neural networks where all layers share the same weights [27]. In this way, we can see that we can train a recurrent network with the same methods as feedforward networks. However, training of recurrent neural networks turned out to be very difficult, because "gradients either grow or shrink at each time-step" [27] which can lead to exploding or vanishing gradients over many time-steps [20, 5].

To tackle this problem Hochreiter and Schmidhuber [21] introduced long short-term memory (LSTM) networks. LSTMs have additional hidden units that act as a memory, whose natural behavior is to store information for a long time. This is done by copying them to the next time-step by multiplying them with a fixed weight of one. However, this connection is controlled by another unit that can decide to forget this information by setting copy-weight to a value smaller than one.

2.2 Reinforcement Learning

In this section, we describe the RL setting for the single agent case. We first introduce Markov decision processes for modeling stochastic environments and then show basic algorithms to compute value functions and policies for agents in these environments. After that, we describe how these methods can be combined

with deep learning by using neural networks as function approximators. The multi-agent learning method we apply and extend in this work is a deterministic actor-critic method. Therefore, we mainly describe methods for deterministic policies.

2.2.1 Markov Decision Processes

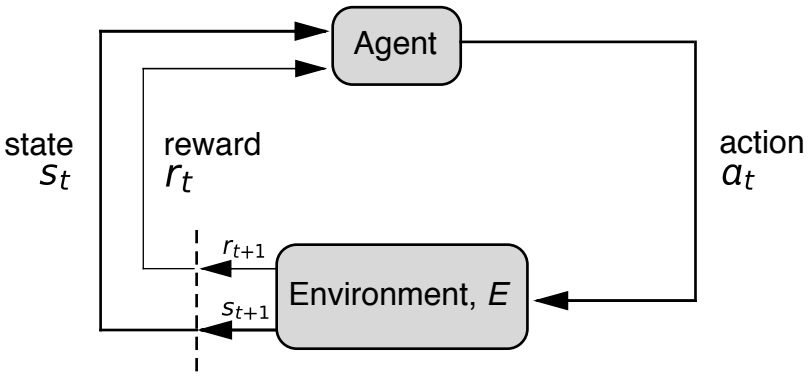


Figure 2.2: The reinforcement learning loop (taken from [51])

In a single-agent RL setting, as described by Sutton and Barto [51], an agent interacts with an environment E in discrete time-steps (as shown in Figure 2.2). At each time-step t the agent takes an action a_t, receives the current state s_t of the environment and a scalar reward r_t. The agent accumulates rewards while it moves through the state space. The behavior of the agents is defined by a policy π, which selects the agent's action in a state s based on a probability distribution $\pi(a|s)$. Such a policy is called a stochastic policy. If the policy is deterministic, i.e. for every state one action is selected with probability 1, we write: $\mu(s) = a$, thus a policy is a mapping from states to actions[1]. The overall goal of RL is to find a policy that maximizes the accumulated reward.

Most environments are stochastic, i.e. after taking an action there are multiple states in which the environment could end up with different probabilities. If the next state only depends on the current state and the action taken by the agent, then the environment satisfies the *Markov property*. Stochastic environments satisfying this property can be modeled using a *Markov decision process* (MDP). More formally, a MDP consists of the following elements: a set of states $s \in S$

[1]It is common in RL literature to refer to stochastic policies with π and to deterministic policies with μ.

with an initial state s_0 and a set of available actions $s \in A$. The dynamics of the MDP are defined by the transition function $p(s'|s,a)$, which defines the probability of the agent to end up in state s' after taking action a in state s. The expected immediate reward is given by $r(s,a)$. We call a MDP episodic if it terminates after a finite number of steps. Since all problems we consider in this thesis have a fixed number of maximal steps, we assume episodic MDPs from now on. Furthermore, we denote the last time-step in which the environment terminates by T. The return R_t is the discounted total reward that the agent accumulates after time-step t:

$$R_t = \sum_{k=t}^{T} \gamma^{k-t} r(s_k, a_k), \tag{2.5}$$

with $\gamma \in [0,1]$. The agent's goal is to maximize its accumulated discounted reward over time. Therefore, we are interested in the expected return when being in a particular state and following policy π. This is given by the value of a state, which is formally defined as the expected return after being in state s and following policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t | s_t = s], \tag{2.6}$$

where $\mathbb{E}_{\pi}$ denotes the expectation given the agent follows policy π. This function is called the *state-value function* for policy π. Similarly, we define the *action-value function* that gives the expected return, when taking action a in state s and thereafter following policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_t | s_t = s, a_t = a] \tag{2.7}$$

State-value and action-value functions play a central role throughout RL. For example, when the optimal action-value function is known, i.e. the highest return that can be achieved from the current state-action pair, selecting the action with the highest action-value in every state yields an optimal policy. For state-value functions the model of the environment is required to determine the optimal action for a given state.

A common framework for model-free RL is generalized policy iteration, with alternating steps of *policy evaluation* and *policy improvement* [51]. Policy evaluation methods estimate the value function of the policy, while the policy improvement step improves the policy with respect to the estimated value function. In the next sections, we first describe temporal difference learning for policy evaluation and

policy gradient methods that directly optimize parameterized policies. Finally, we introduce how these methods can be combined in actor-critic methods and how neural networks can be used to approximate actor and critic.

2.2.2 Temporal Difference Learning

A fundamental property of value functions is that they have a recursive relationship, which is used in RL and dynamic programming. Intuitively, we can compute the value of a state $V^\pi(s)$ based on the value of the next expected state and the reward received during the transition to this state [51]:

$$V^\pi(s) = \mathbb{E}_\pi\big[r(s,a) + \gamma V^\pi(s')\big] \tag{2.8}$$

$$= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\big[r(s,a) + \gamma V^\pi(s')\big] \tag{2.9}$$

This is referred to as the *Bellman equation* for state-values and can be used to compute the value of a policy if the transition function $p(\cdot)$ is known, however, this is not the case for most environments. Therefore, there are model-free algorithms to learn the value function from experiences that are collected while following the policy to evaluate. One of them is *temporal difference learning*, which uses an estimate of the expectation in equation 2.8, to estimate the state-values for the followed policy π. After transitioning from state s to s' and receiving reward r we update the value for state s as follows:

$$V(s) \leftarrow V(s) + \alpha\big[r + \gamma V(s') - V(s)\big], \tag{2.10}$$

here α is the learning rate at which we update the value estimations. Intuitively, we compute a new estimate for $V(s)$ by incorporating the received reward as new information in combination with value of the next state.

Similarly, with the Bellman equation for action-values it is possible to compute the action-values recursively:

$$Q^\pi(s,a) = \mathbb{E}_{r,s'\sim E}\big[r + \gamma \mathbb{E}_\pi[Q^\pi(s',a')]\big] \tag{2.11}$$

$$= \sum_{s'} p(s'|s,a)\big[r(s,a) + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s',a')\big] \tag{2.12}$$

here we have one expectation for the next state and received reward and another expectation over the action-values of the next state. The first expectation can be

estimated by taking transitions while following the evaluated policy π as samples, as we have done for equation 2.10. We could use the next chosen action to estimate the inner expectation, which would yield the *SARSA* algorithm [43, 51]. Another possibility is to use a deterministic policy $\mu(\cdot)$, which removes the expectation because always the same action is chosen:

$$Q^{\mu}(s,a) = \mathbb{E}_{r,s' \sim E}\big[r + \gamma Q^{\mu}(s', \mu(s'))\big] \tag{2.13}$$

Now, the expectation only depends on the environment and therefore it is possible to learn Q^{μ} off-policy, which means we can use samples that are generated with another behavior policy β. For example, if we set $\mu(s) = \arg\max_a Q(s,a)$ and use the ϵ-greedy policy for exploration, we get the *Q-Learning* algorithm [54].

2.2.3 Deterministic Policy Gradient

As mentioned above, the primary goal of RL is to find a policy that maximizes the expected return. This is achievable by estimating the optimal action-value function and then selecting actions greedily. Another way is to optimize a parameterized policy directly, based on the gradient of the policy. This enables learning policies for continuous high-dimensional action spaces, where it is difficult to select the action with the highest action-value.

Instead of improving the policy greedily with respect to the action-value function, policy gradient methods optimize the policy based on the gradient of the performance objective with respect to the policy's parameters. The performance objective is defined as $J(\pi) = \mathbb{E}[R_1|\pi]$ and represents the expected cumulative discounted reward from the start state when following policy π.

For deterministic policies the policy gradient has an appealing form: it is the expectation of the action-value function's gradient. More specifically, given a differentiable action-value function Q^{μ} for a deterministic parameterized policy μ_θ, we can improve the policy by updating its parameters in proportion to the gradient $\nabla_\theta Q^{\mu}(s, \mu_\theta(s))$. The *deterministic policy gradient theorem* [45] states that the gradient of the performance objective $\nabla_\theta J$ is the expectation of the gradient $\nabla_\theta Q^{\mu}$:

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s \sim \rho^{\mu}}\big[\nabla_\theta Q^{\mu}(s, \mu_\theta(s))\big] \tag{2.14}$$

$$= \mathbb{E}_{s \sim \rho^{\mu}}\big[\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu}(s,a)|_{a=\mu_\theta(s)}\big], \tag{2.15}$$

where ρ^μ denotes the discounted state visitation distribution and equation 2.15 can be derived by applying the chain rule to equation 2.14. Silver et al. [45] have also shown that above equality holds approximately for a state visitation distribution ρ^β from a behavior policy β. Thus, this expectation can be estimated off-policy with samples produced by a policy distinct to the optimized one.

2.2.4 Deterministic Actor-Critic

Actor-critic methods combine temporal difference learning and policy gradient methods. The actor is a parameterized policy that has to be optimized. The critic is the action-value function of the actor and is learned with temporal difference learning. The parameters of the actor get optimized with respect to the critic. So we use samples from a policy to learn a action-value function and then use these samples and the action-value function to optimize the actor.

For deterministic policies we can use the *off-policy deterministic actor-critic* algorithm [45]. This algorithm updates policy μ_θ in the direction of the off-policy deterministic policy gradient, which is given in equation 2.15 when a different policy is used to generate the samples. For the critic we replace the true action-value function of the policy with a parameterized differentiable approximation $Q_w \approx Q^\mu$. The critic is learned with off-policy temporal difference learning based on equation 2.13 with trajectories generated by a behavior policy β. The parameter updates for actor and critic can now be compute as follows:

$$\delta_t = r_t + \gamma Q_w(s_{t+1}, \mu_\theta(s_{t+1})) - Q_w(s_t, a_t) \tag{2.16}$$

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q_w(s_t, a_t) \tag{2.17}$$

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu_\theta(s_t) \nabla_a Q_w(s_t, a)|_{a=\mu(s_t)} \tag{2.18}$$

2.2.5 Deep Deterministic Policy Gradient

In this section, we describe how neural networks can be used to approximate actor and critic in the algorithm described in the section above. The use of neural networks enables us to train complex policies that generalize well in large state and action spaces. However, deep neural networks use non-linear activation functions, thus convergence is no longer guaranteed. In the following, we describe

how Lillicrap et al. [29] modified the off-policy deterministic actor-critic algorithm to allow the use of neural networks as function approximators for the actor and the critic.

Most optimization algorithms for neural networks, such as stochastic gradient descent, assume the training samples to be independently and identically distributed [4]. This is not the case when the samples are generated sequentially by exploring the environment and training is done online. Furthermore, it is common to learn in mini-batches to improve efficiency of learning. Mnih et al. [35] tackled this problem by employing *experience replay* [30]. With experience replay, a replay buffer R is used, which is a first-in and first-out queue with a fixed size to store tuples of previously experienced transitions (s_t, a_t, r_t, s_{t+1}). Instead of learning online, at each training step both the actor and the critic are trained on a mini-batch that has been sampled uniformly from the replay buffer. Sampling from a large replay buffer allows the networks to learn across a set of uncorrelated experiences.

Implementing the training of the critic directly from equation 2.17 with neural networks can be particularly unstable. The network $Q_w(s, a)$ that is being updated is also used to compute the action-value in the next state $Q_w(s_{t+1}, \mu_\theta(s_{t+1}))$ when calculating the temporal difference error in equation 2.16. The problem is that an update that increases $Q_w(s, a)$ often also increases $Q_w(s_{t+1}, \mu_\theta(s_{t+1}))$ and hence the temporal difference error would not increase, which possibly leads to divergence [35].

The solution proposed by Lillicrap et al. [29] is to perform 'soft' updates of target actor and critic networks, in contrast to doing a copy the weights after a fixed number of steps as done by Mnih et al. [35]. Therefore, we create copies of the actor and the critic networks, $\mu'_{\theta'}(s)$ and $Q'_{w'}(s, a)$ respectively. These networks are specifically used to calculate the target value in the temporal difference error:

$$\delta_t = r_t + \gamma Q'_{w'}(s_{t+1}, \mu'_{\theta'}(s_{t+1})) - Q_w(s_t, a_t) \tag{2.19}$$

However, when training neural networks it is more common to specify a loss function which then is differentiated by the learning framework to compute the gradients with respect to the parameters. Thus, we consider the training of the action-value function as a regression problem, where the mean squared error

between the prediction and a target value is most common as a loss function:

$$L(w) = \mathbb{E}_{s_t,a_t,r_t,s_{t+1}\sim U(R)}[(y_t - Q_w(s_t, a_t))^2] \tag{2.20}$$

where the target value is based on the temporal-difference target, as given in equation 2.13 and uses the target networks to avoid divergence:

$$y_t = r_t + \gamma Q'_{w'}(s_{t+1}, \mu'_{\theta'}(s_{t+1})) \tag{2.21}$$

The critic's neural network can now be updated by minimizing the loss function in equation 2.20, where the expectation can be computed by sampling transitions uniformly from the replay buffer. Analogously, the actor's network is updated by sampling transitions and minimizing $L(\theta) = -\mathbb{E}_{s\sim\rho^\mu}\big[Q_w(s, \mu_\theta(s))\big]$.

After updating the actor and the critic, the target networks are 'softly' updated as follows: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$, w is updated analogously. In this way, the target values are constrained to change slowly, which leads to greater stability of learning. One could argue that soft updates slow down learning, but Lillicrap et al. [29] found that the induced stability of learning greatly outweighs any other potential disadvantages.

With these modifications Lillicrap et al. [29] were able to successfully train policies for various environments with high-dimensional state and action spaces with an algorithm they call *deep deterministic policy gradients* (DDPG). The full algorithm is given in Algorithm 1 in the appendix.

2.3 Multi-Agent Reinforcement Learning

In standard RL as described in the last section, we have a single learning agent in a stationary environment. Multi-agent reinforcement learning (MARL), in contrast, considers multiple-agent in the same environment that optimize their policies through RL. This can lead to non-stationary environments as the behavior of other agents changes over time. Therefore, acquired knowledge from past experiences can become obsolete when the policy of the other agents changes.

In the simple two-player game rock-paper-scissors, for example, the optimal policy depends heavily on the policy of the other agent. If the other agent always play scissors, the optimal policy would be to play rock. But when the other agent

changes its policy, to always play paper, our policy that was optimal becomes the worst possible policy. These kinds of considerations are part of game theory, where the central concern is to find an equilibrium point where no agent wants to change its policy.

While rock-paper-scissors is a competitive game, we focus on cooperative games, where all agents have the same goal, because in these cases agents are most incentivized to cooperate. While the agents are not adversarial in cooperative games, they still change their policies to achieve the common goal which can lead to non-stationary environments. For example, when two cars that drive in opposite directions on the same road have to coordinate to avoid a collision, a change in policies can lead to worse performance for both agents. Therefore, the main goal for cooperative MARL is to coordinate the actions of the agents, while it is not necessary for the agents to consider adversarial actions of other agents.

In the next sections, we first introduce the *decentralized Markov decision processes* as a theoretical framework for cooperative MARL. After that we describe, how we can apply single-agent methods to multi-agent environments which serves as a baseline for multi-agent algorithms. Then, we describe an extension of the DDPG algorithm to enable it to cope with multi-agent environments, which includes modeling other agents.

2.3.1 Decentralized Markov Decision Processes

We model the interactions of multiple agents in a potentially stochastic environment as a *decentralized Markov decision process* (Dec-MDP), which is a special case of the *decentralized partially observable Markov decision process* (Dec-POMDP), where the observations of all agents jointly identify the true state of the environment [6]. This is the motivation for the use of communication for problems in this class. Agents that learn to communicate necessary information about their own observations, may be able to solve the problem without depending on the history of their observations, which is a common way to deal with partial observability.

An example for a Dec-MDP is a team of robots, where the state consists of the locations of all robots and each robot knows its position perfectly [24]. Therefore, the observations of all agents together can determine the state of the environment.

However, each agent still only partially observes the full state. Bernstein et al. [6] have shown that the worst-case complexity of finding an optimal policy for Dec-MDPs is in the same as of Dec-POMDPs. This implies that distributed control and not only hidden states make these problems hard to solve.

More formally, a Dec-MDP is defined as follows: n agents interact with the environment in discrete time-steps, the state space of the environment is $s \in S$, the joint action space of the agents is $\boldsymbol{A} = A_1 \times A_i \times \cdots \times A_n$ and $\boldsymbol{O} = O_1 \times O_i \times \cdots \times O_n$ is the joint observation space, for which exists a function $J : \boldsymbol{O} \to S$ with $J(o_1, \ldots, o_n) = s$. Each agent selects an action $a_i \in A_i$ and the joint action $\boldsymbol{a} = (a_1, \ldots, a_n)$ causes the environment to transition from s to the next state s' with probability $p(s'|s, \boldsymbol{a})$. At each time-step each agent receives a private observation $o_i \in O_i$ with joint probability $o(\boldsymbol{o}|s, \boldsymbol{a})$. Let $(o_{i,1}, \ldots, o_{i,t}) \in H_{i,t}$ be the observation history of agent i at time-step t. The behavior of each agent is then modeled with a policy $\mu_i : H_{i,t} \to A_i$ that maps from the history of observations of an agent to its actions and the joint policy is denoted $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_n)$. The group of agents receives a joint reward $r_t = r(s_t, \boldsymbol{a}_t)$ and the joint return is $R_t = \sum_{k=t}^{T} \gamma^{k-t} r(s_k, \boldsymbol{a}_k)$, with T being the time horizon of the task.

As for single-agent MDPs (cf. Section 2.2.1) we can define value functions that represent the expected cumulative discounted future reward or return when joint policy $\boldsymbol{\mu}$ is followed. The state-value function for Dec-MDPs is defined as:

$$V^{\boldsymbol{\mu}}(s) = \mathbb{E}_{\boldsymbol{\mu}}[R_t | s_t = s] \tag{2.22}$$

and the action-value function is:

$$Q^{\boldsymbol{\mu}}(s, \boldsymbol{a}) = \mathbb{E}_{\boldsymbol{\mu}}[R_t | s_t = s, \boldsymbol{a}_t = \boldsymbol{a}] \tag{2.23}$$

There are different approaches on how to learn action-value functions in multi-agent environments. As described in the next section, some algorithms only use an agent's own actions while others assume that every agent has access to the joint-actions.

2.3.2 Independent and Joint-Action Learning

For fully cooperative settings, Claus and Boutilier [9] pointed out that there are two distinct ways to learn action-values for multiple agents. They refer to the

two classes of agents as *independent learners* and *joint-action learners*.

Independent learners only learn action-values for their own actions, based on equation 2.13. Thus, these agents learn action-values without regard to actions of other agents. This is a very scalable approach, because the computational complexity is independent of the number of agents in the environment. However, learning action-values in this way assumes a stationary environment which is not the case here. Therefore, a convergence of learning is not guaranteed.

Joint-action learners learn action-values for joint-actions in contrast to individual actions. Thus, they learn the joint action-value function, as given in equation 2.23. However, this approach implies that each agents observes the actions the other agents have chosen, which is a strong assumption for partially observed domains. In addition, the approach is not as scalable as independent learners as the input for the action-value function grows linearly with the number of agents.

Both classes of agents are supported by Dec-MDPs, which we use to model the environment. The definition of Dec-MDPs only states that every agent receives a potentially partial observation of the environment. In Section 3.1 of the next chapter, we describe how we model the uncertainty about the observations and actions of other agents directly.

2.3.3 Centralized Training and Decentralized Execution

A lot of recently proposed algorithms for learning to cooperate and communicate in multi-agent settings [10, 50, 31], rely on a paradigm called *centralized training and decentralized execution*. This paradigm was first introduced by Kraemer and Banerjee [24]. They allow learning agents to access otherwise hidden information during the learning process, which will then not be available when executing the learned policy. This procedure is not applicable to all real-world problems, but Kraemer and Banerjee [24], for example, describe a scenario in which robots are trained in a lab setting and are connected to a central computer that has complete observability of the whole environment (e.g. with a overhead camera). Therefore, in this framework it is possible to use a joint-action learner in the training phase, as long as the learned policy does not depend on the joint-actions.

There are also other ways to make use of a centralized training. There is some recent work, for example, that uses differentiable communication channels between the agents during training [10, 50], such that the agents are able to exchange error signal with respect to their communication. To achieve this the communication during training must be continuous to be differentiable, but it is possible to discretize the communication at execution time, as done in [10].

2.3.4 Multi-Agent DDPG

In this thesis, we examine and extend the *multi-agent deep deterministic policy gradient* (MADDPG) [31] algorithm, which is an extension of DDPG to the multi-agent domain. In the following, we describe the original method, while we discuss our adaptions in the next chapter. The full MADDPG algorithm is given in Algorithm 2 in the appendix.

MADDPG makes use of centralized training with decentralized execution. Their proposed actor-critic algorithm uses the joint action-value function as a critic to learn a policy that only depends on the agent's private observations. Therefore, while training the agents, we assume that it is possible to violate the assumption of Dec-MDPs that agents only have knowledge of their own observations and chosen actions (cf. Section 2.3.1).

More specifically, for a game with n agents, MADDPG uses *local* policies $\boldsymbol{\mu} = \{\mu_1, \dots, \mu_n\}$ that are parameterized by $\boldsymbol{\theta} = \{\theta_1, \dots, \theta_n\}$ and a *centralized* action-value function $Q_i(\boldsymbol{o}, \boldsymbol{a}) \approx Q^{\boldsymbol{\mu}}(s, \boldsymbol{a})$ for each agent which are parameterized by $\boldsymbol{w} = \{w_1, \dots, w_n\}$. For brevity, we drop the parameters from the policies and action-values functions in the subscript, so $\mu_i = \mu_{\theta_i}$ and $Q_i = Q_{w_i}$. Each centralized action-value function takes the actions and observations of all agents as input and outputs the action-value for agent i. Thus, we train a separate action-value function for each agent. Now, we extend equation 2.15 for the multi-agent case:

$$\nabla_{\theta_i} J = \mathbb{E}_{\boldsymbol{o}, \boldsymbol{a} \sim \rho^{\beta}} \left[\nabla_{\theta_i} \mu_i(o_i) \nabla_{a_i} Q_i(\boldsymbol{o}, \boldsymbol{a}_{-i}, a_i)|_{a_i = \mu_i(o_i)} \right], \tag{2.24}$$

where $\boldsymbol{o} = (o_1, \dots, o_n)$, $\boldsymbol{a} = (a_1, \dots, a_n)$, $\boldsymbol{a}_{-i} = \boldsymbol{a} \setminus \{a_i\}$ and β is a stochastic behavior policy to generate the training samples to ensure sufficient exploration of the environment. The centralized action-value functions can be learned by minimizing the following loss function, which is an extension of equation 2.20 for

the multi-agent case:

$$L(w_i) = \mathbb{E}_{\boldsymbol{o}_t, \boldsymbol{a}, r_i, \boldsymbol{o}_{t+1} \sim U(R)} \left[(y_t - Q_i(\boldsymbol{o}_t, \boldsymbol{a}))^2 \right] \tag{2.25}$$

and the target value is:

$$y_t = r_t + \gamma Q_i'(\boldsymbol{o}_{t+1}, \boldsymbol{a'})|_{a'_j = \mu'_j(o_{j,t+1})} \tag{2.26}$$

where $\boldsymbol{\mu'} = \{\mu'_1, \ldots, \mu'_n\}$ is a set of target policies whose parameters are softly updated with the parameters of the current policies to stabilize training. The expectation of the loss is computed with samples from a replay buffer R that contains tuples $(\boldsymbol{o}_t, \boldsymbol{a}, r_t, \boldsymbol{o}_{t+1})$, which represent the observations, actions and received rewards of all agents in a transition from state s_t to s_{t+1}.

2.3.5 Modeling Other Agents

To train the critic, each agent requires the policies of all other agents to compute the temporal-difference target as described in equation 2.26. This is a very strong assumption even if we assume that additional information can be used, because policy representation can be very large and therefore it might be unfeasible for the agents to transfer them to other agents due to a limited bandwidth and therefore a training on the same machine is required. To remove the dependency on other agents' policies Lowe et al. [31] suggest that each agent i can maintain a model $\hat{\mu}_{\phi_i^j}$ of each other agent. Here, ϕ_i^j denotes the parameters maintained by agent i, which parameterize the model of agent j. For simplicity we drop the parameter subscript and write $\hat{\mu}_i^j = \hat{\mu}_{\phi_i^j}$. The models are trained with maximum likelihood estimation, by minimizing the negative log-likelihood of a parameterized probability distribution which is parameterized by ϕ_i^j:

$$L(\phi_i^j) = -\mathbb{E}_{o_j, a_j} \left[\log \hat{\mu}_i^j(a_j|o_j) + \lambda H(\hat{\mu}_i^j) \right], \tag{2.27}$$

where H is the entropy of the model's probability distribution and the second term serves as an entropy regularizer to prevent overfitting of the latest data. The models are used to replace the use of real policies in equation 2.26, by predicting the next action using the probability distribution. Then, the temporal difference target y_t can be computed as follows:

$$y_t = r_t + \gamma Q_i'(\boldsymbol{o}_{t+1}, \hat{\mu}_i^1(o_{1,t+1}), \ldots, \mu'_i(o_{i,t+1}), \ldots, \hat{\mu}_i^n(o_{n,t+1})) \tag{2.28}$$

Thus, when training the action-value function of agent i we use the target policy network of this agent and the agent model for all other agents. While Lowe et al. [31] suggested to use target networks for the modeled policies as well, we found that there is no significant advantage in using target networks for modeled agents. Furthermore, Lowe et al. [31] propose to train the models online, by minimizing the loss in equation 2.27 using stochastic gradient descent using the most recent observations and actions, before updating the actor and the critic in each training step.

Chapter 3

Problem and Approach

The goal of this thesis is to evaluate how multiple agents can learn to solve tasks in a decentralized setting. In particular, we are interested in how much information about other agents is required to successfully solve a task in a team. In Section 2.3.2, we described the distinction between joint-action learners that have full knowledge about the other agents and independent learners that only have access to their own actions and observations. Having full access to other agents' observations and actions is a very strong assumption in decentralized environments with partial observability. It is also not realistic in most scenarios that each agent is not aware at all of the other agents in the environment. We therefore propose to model the observability of other agents directly, which enables us to test the dependency of the learning procedure on the knowledge about other agents by simulating different degrees of partial observability.

In the last chapter, we have described the MADDPG algorithm as a multi-agent extension of DDPG. MADDPG assumes full access to observations and actions of other agents as well as their policies, while there is the possibility to model the policies of other agents. In this chapter, we introduce an extension of MADDPG to deal with partial observability of other agents using recurrent neural networks.

The remainder of this chapter is structured as follows: First, we describe how we explicitly model the partial observability of other agents' observations and actions. Then, we introduce Rec-MADDPG as an extension of MADDPG motivated by the partial observability of other agents. Finally, we describe the environments in which we conduct our experiments and how we use them to test different aspects of the algorithms.

3.1 Partial Observability of Other Agents

The Dec-MDP model, which we introduced in Section 2.3.1, models the observations $o_i \in O_i$ each agent receives with a joint probability distribution $o(\boldsymbol{o}|s, \boldsymbol{a})$. This probability distribution is different for each environment and thus the observability of other agents' actions and observations depends on the environment. However, we would like to model the uncertainty about actions and observations of other agents explicitly. Therefore, similar to [41], we assume that each agent makes potentially noisy observations of other agents' trajectories, modeled with probability distributions over the observed observations $\omega(\boldsymbol{o}_{-i}^{(obs)}|\boldsymbol{o})$ and observed actions $\alpha(\boldsymbol{a}_{-i}^{(obs)}|\boldsymbol{a})$ of the other agents[1].

The information for one state transition that agent i is able to use to optimize its policy are now: $(o_t^{(i)}, \boldsymbol{o}_t^{(obs)}, a_t^{(i)}, \boldsymbol{a}_t^{(obs)}, r_t, o_{t+1}^{(i)}, \boldsymbol{o}_{t+1}^{(obs)})$, where $o_t^{(i)}$, $o_{t+1}^{(i)}$ and $a_t^{(i)}$ are the original observations and chosen actions of agent i respectively. The observations of the other agents' observations and actions are drawn from $\boldsymbol{o}_t^{(obs)} \sim \omega(\boldsymbol{o}_t)$, $\boldsymbol{o}_{t+1}^{(obs)} \sim \omega(\boldsymbol{o}_{t+1})$ and $\boldsymbol{a}_t^{(obs)} \sim \alpha(\boldsymbol{a}_t)$.

In our experiments, we use a Gaussian distribution to simulate the uncertainty about observations of other agents: $\omega(\boldsymbol{o}_{-i}^{(obs)}|\boldsymbol{o}) = \mathcal{N}(\boldsymbol{o}_{-i}^{(obs)}|\boldsymbol{o}, \sigma^2)$ and a Gumbel-Softmax distribution [22, 32] to introduce uncertainty about the chosen actions $\alpha(\boldsymbol{a}_{-i}^{(obs)}|\boldsymbol{a}) = \text{Gumbel-Softmax}(\boldsymbol{a}_{-i}^{(obs)}|\boldsymbol{a}, \tau)$. The Gumbel-Softmax distribution is a continuous distribution that can approximate samples from a categorical distribution. In this way, we can gradually increase the uncertainty by increasing the standard distribution σ of the Gaussian distribution and the temperature τ of the Gumbel-Softmax distribution (increasing τ leads to more uniform samples). Thus, we are able to test how much the performance of different algorithms decreases when we increase the noise-level step by step.

The Gumbel-Softmax distribution is utilized to add noise to the actions, because the observed actions are used as targets when train the models of other agents and the actions are computed with the softmax function and have to sum to one. Therefore, we decided to sample noisy actions in a way in which they are still valid actions, which would not be the case when adding Gaussian noise.

[1] For better readability we drop the subscript $-i$ in the following, which denotes that these are the observations or actions of all other agents without agent i.

3.2 Recurrent Multi-Agent DDPG

In this section, we propose an extension of the MADDPG algorithm to use recurrent neural networks as critics to deal with the partial observability of other agents' actions and observations. As described in the section above, we assume a partial observability of the other agents while the observations of all agents together determine the state of the environment (cf. Section 2.3.1). This motivates the use of communication between agents to share necessary information enabling them to successfully complete the task.

The MADDPG algorithm, which has been described in Section 2.3.4, has been shown to enable agents to learn to cooperate and communicate [31]. However, this method assumes full access to the observations and actions of all other agents. Therefore, we want to extend the method to be able to work with partial observations of other agents. Hausknecht and Stone [15] have proposed *deep recurrent Q-networks* to address partial observability in single-agent settings. Instead of using feedforward networks to approximate the action-value function they use recurrent neural networks, which maintain an internal state and aggregate the observations over time. This approach has also been applied to actor-critic systems [17] for continuous control and to learn action-value functions in partial observable multi-agent settings [39] and learning to communicate [13].

To address the partial observability of the other agents' actions and observations we therefore propose to use deep recurrent neural networks to approximate the critics. This enables the estimation of action-values based on the history of observed actions and observations. We still use feedforward networks for the actors, because we assume a jointly observable state and that agents can exchange all necessary information in each step.

In the following, we describe the adaptions we have made to enable the MADDPG method to make use of recurrent critic networks. These include a replay buffer that stores episodes of transitions for each agent, recurrent critic networks and a modified training procedure that addresses learning with mini-batches of episodes.

3.2.1 Episodic Replay Buffer

In a single agent scenario, the replay buffer stores past transitions, consisting of observations, the chosen action and the received reward of one agent [30, 35] (cf. Section 2.2.5). For the training of critic networks with MADDPG each agent requires the transition data of all agents to train its networks. Since a centralized training is assumed anyway, a central replay buffer is used that stores transitions as tuples $(\boldsymbol{o}_t, \boldsymbol{a}, r_t, \boldsymbol{o}_{t+1})$, which include the observations and actions of all agents.

We aim for a decentralized setting where each agent has only partial observability of the observations and actions of other agents. Hence, we maintain a replay buffer R_i for each agent with transitions $(o_t^{(i)}, \boldsymbol{o}_t^{(obs)}, a_t^{(i)}, \boldsymbol{a}_t^{(obs)}, r_t, o_{t+1}^{(i)}, \boldsymbol{o}_{t+1}^{(obs)})$, which distinguish between own observations and actions and the data from other agents. Learning recurrent neural networks for the critics, necessitates sequential samples of transitions in each training step. Therefore, the transitions are stored along the axes episode and time-step. Thus, each agent stores the transitions sequentially along the time-step axis for each episode. After each episode a new sequence is started along the episode axis and if the capacity of the replay buffer is reached the first added episodes are removed in a first-in first-out manner. The sequences along the time-step axis can have varying length. However, we zero-pad all sequences to the maximal episode length, because this enables the usage of efficient optimizer implementations in standard deep learning libraries.

This replay buffer definition is similar to how Omidshafiei et al. [39] structured their *Concurrent Experience Replay Trajectories* to train recurrent Q-networks for multiple agents. However, they assume a fully decentralized setting where other agents are not observable. Therefore, they require that the training samples of private observations and actions are drawn concurrently from the local replay buffers, which means that the same time-steps in the same episodes are used. In this way, local policy updates are correlated and the agents tend to converge to the same equilibrium if there are multiple ones. To achieve this, synchronization of the agents' sampling methods is required. However, this is not possible for every scenario, for example when humans are involved. Since we assume partial observability of other agents, local policy updates are correlated because they are optimized based on partial observations about other agents, which are correlated with the real observations and actions. Therefore, each agent can update its networks independently.

3.2.2 Recurrent Critic Networks

To cope with partial observability of other agents, each agent maintains a deep recurrent critic network, which enables the critic to predict action-values based on the history of observed actions and observations. The recurrent neural network approximates the true action-value function of the current joint policy:

$$Q_i(o_t^{(i)}, \boldsymbol{o}_t^{(obs)}, a_t^{(i)}, \boldsymbol{a}_t^{(obs)}, h_{t-1}^{(i)}) \approx Q^{\boldsymbol{\mu}}(s_t, \boldsymbol{a}_t), \tag{3.1}$$

where $h_{t-1}^{(i)}$ is the hidden state of the recurrent network at the previous time-step, $o_t^{(i)}$ and $a_t^{(i)}$ are the local observations and action, while $\boldsymbol{o}_t^{(obs)}, a_t^{(i)}$ and $\boldsymbol{a}_t^{(obs)}$ are the observed actions and observations of other agents. If $t = 1$, an initial value for h_{t-1} is used that is learned by the optimization procedure. We found that using learned initial hidden states leads to a much better performance, than using zero vectors as initial hidden states. Furthermore, we use *long short-term memory* (LSTM) networks [21] to make the critic recurrent (cf. Section 2.1.2), as these kind of networks is able to learn long-term dependencies and has been successfully applied in other partial observable reinforcement learning settings [15, 17, 39].

3.2.3 Training Algorithm

In the following, we describe the Rec-MADDPG training algorithm for agents with recurrent critic networks and the full algorithm is given in Algorithm 3 in the appendix. Each agent i maintains a local policy $\mu_i(o^{(i)})$ parameterized with θ_i and an action value function $Q_i(\cdot)$ parameterized by w_i, which is based on local observations and actions as well as partially observed observations and actions from other agents, as described in the last section. Furthermore, each agents also maintains parameters θ_i' and w_i' for the actor and critic respectively. In order to compute the temporal difference target, predicting other agents' actions is necessary, because we do not assume access to the policies of other agents. Therefore, an agent models the other agents' behavior with a policy representation $\hat{\mu}_i^j$ that is parameterized by ϕ_i^j. In addition, each agent stores its experiences and the observed experiences of other agents in a episodic replay buffer R_i.

To train actor and critic networks, each agent samples episodes from the local episodic replay buffer. We decided to train the recurrent networks on complete episodes instead of partial trajectories, because in this way the hidden state is

always initialized at the beginning of the episode in contrast to random starting points, when using smaller subsets of an episode [15]. Thus, in every training step an agent i samples a mini-batch $\mathcal{B}$ consisting of B episodes, $\mathcal{B} = \{\mathcal{E}_1, \ldots, \mathcal{E}_B\}$, from R_i, where one episode is a sequence of state-action-reward tuples:

$$\begin{aligned}\mathcal{E}_b = \big((o_1^{(i)}, \boldsymbol{o}_1^{(obs)}, a_1^{(i)}, \boldsymbol{a}_1^{(obs)}, r_1, \ldots, \\ (o_T^{(i)}, \boldsymbol{o}_T^{(obs)}, a_T^{(i)}, \boldsymbol{a}_T^{(obs)}, r_T)\big),\end{aligned} \tag{3.2}$$

with T being the number of the episode's final step. Each episode $\mathcal{E}_b$ is used to compute a sequence of target values $(y_1^b, \ldots, y_T^b)$, where each target is computed as follows:

$$\begin{aligned} y_t^b &= r_t^b + \gamma Q_i'(o_{t+1}^{(i)}, \boldsymbol{o}_{t+1}^{(obs)}, \boldsymbol{a}', h_t^{(i)}), \\ \text{with} \quad a_j' &= \begin{cases} \mu_i'(o_{i,t+1}) & \text{if } j = i \\ \hat{\mu}_i^j(o_{j,t+1}^{(obs)}) & \text{else} \end{cases} \end{aligned} \tag{3.3}$$

where the hidden state of the previous step in the episode is used or a learned initial hidden state if $t = 1$. The target is computed with target networks μ_i' and Q_i' of actor and critic respectively to stabilize the updates. Furthermore, actions for the action-value of the next state are predicted using either learned models of the other agents $\hat{\mu}_i^j$ or the target network for the agent's own action. Based on the sequence of target values a sequence of temporal difference errors $(\delta_1^b, \ldots, \delta_T^b)$ is calculated for each episode, with:

$$\delta_t^b = y_t^b - Q_i(o_t^{(i)}, \boldsymbol{o}_t^{(obs)}, a_t^{(i)}, \boldsymbol{a}_t^{(obs)}, h_{t-1}^{(i)}), \tag{3.4}$$

where again the hidden state is carried forward throughout the episode. Given the sequences of temporal difference errors for all episodes of the mini-batch, we can compute the loss that is minimized with gradient descent to learn the critic networks:

$$L(w_i) = \mathbb{E}_{\mathcal{E}_b \sim U(R_i)}\big[(\delta_t^b)^2\big] \tag{3.5}$$

Thus, the loss function for the critic networks is the mean squared error between the predictions of the critic network and the temporal difference targets. The policy network's parameters are trained by gradient ascend on the policy gradient for agent i:

$$\nabla_{\theta_i} J = \mathbb{E}_{\mathcal{E}_b \sim U(R_i)}\left[\nabla_{\theta_i} \mu_i(o_t^{(i)}) \nabla_{a_i} Q_i(o_t^{(i)}, \boldsymbol{o}_t^{(obs)}, \mu_i(o_t^{(i)}), \boldsymbol{a}_t^{(obs)}, h_{t-1}^{(i)})\right], \tag{3.6}$$

where sequences of action-values $Q_i(\cdot)$ are computed for each episode, with replacing the chosen actions for agent i with the actions chosen by the current policy $a_i = \mu_i(o_t^{(i)})$ and also carrying forward the hidden state throughout the episode as for the critic networks.

We found that doing multiple training steps for the models for each training step of actor and critic as well as including transitions from recent time-steps, was more stable than learning the models completely online, as suggested by Lowe et al. [31]. Therefore, the models are learned with experiences collected to train actor and critic. Before conducting the training steps of actor and critic, we sample multiple mini-batches from recent transitions in the replay buffer, in contrast to episodes for actor and critic. Where recent means that only transitions from the last p time-steps are considered, where usually $p \ll \text{size}(R)$. This is especially beneficial when the observations of other agents are noisy. In this case, multiple training examples from a limited past seem to reduce the noise in the gradient to train the probability distribution of the model. We adapt equation 2.27 as follows:

$$L(\phi_i^j) = -\mathbb{E}_{a_j^{(obs)}, o_j^{(obs)} \sim U(R_i^p)} \big[\log \hat{\mu}_i^j(a_j^{(obs)} | o_j^{(obs)}) + \lambda H(\hat{\mu}_i^j) \big] \tag{3.7}$$

So now the model of agent j is learned based on the partially observed observations $o_j^{(obs)}$ and actions $a_j^{(obs)}$ that are sampled from R_i^p, which is agent i's replay buffer R_i limited to the most recent p transitions.

3.3 Cooperative Environments

For our experiments, we use two cooperative versions of the physically-simulated two-dimensional environment that was used for the experiments to evaluate MADDPG and was proposed by Mordatch and Abbeel [36]. Every environment consists of n agents and l landmarks, in a two-dimensional world with continuous space and discrete time. Landmarks can have descriptive characteristics such as a certain color, which are observable by agents. Each agent can move around the environment by taking continuous actions representing the acceleration. In some environments, agents are able to communicate by uttering symbols that are broadcasted and represented as one-hot encodings. The symbols have no predefined meaning, thus the agents can assign arbitrary concepts to symbols during

the learning process. As we only use the cooperative subset of the environments, all agents receive the same reward signal.

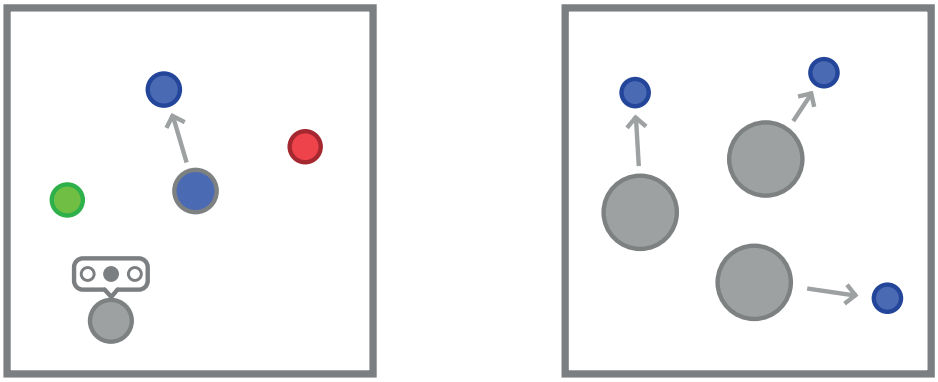
Every task in these environments includes target landmarks that have to be covered by agents. The task differ in the number of agents and targets as well as the observability of which landmark is the target. Each agent receives private observations that are different for each agent, thus there is a partial observability of the environment. Nevertheless, since we adopt a Dec-MDP model, all observations together determine the state of the environment.

The original environments, as used in [36, 31], were not goal-based but had a shaped reward for a certain goal. We modified the environments to have a set of terminating states, such that an episode ends when the desired state is reached, i.e. the task has been accomplished. We experimented with sparse rewards of 1 for reaching the goal and 0 in all other cases. Training in these environments turned out to difficult, because agents have to do a lot of exploration before receiving a positive reward. Therefore, we decided to keep the shaped rewards for each task, which is always based on the distance to the target locations, with sometimes including a penalty for collisions of agents. We noticed that the agents often come close to the target location, but not close enough to be in a terminal state. Similar, observations have been made by Lowe et al. [31], during their experiments with MADDPG. Because we believed that the reason is that the action-value gradients were too little close to the target location, we added a bonus reward for reaching the target area. This indeed solved the problem and we were able to consistently reach success-rates of 100%, as opposed to success rates with a high variance and a mean of approximately 75% before.

In the following, we describe the two environments that we use. While the first is more focused on the aspect of communication, the second focuses more on cooperation between agents.

3.3.1 Cooperative Communication

This environment contains two agents of which, in the basic task, one is the "speaker" and the other the "listener" (as illustrated in Figure 3.1a). Of these agents only the speaker knows the target location to which the listener has to



(a) Cooperative communication environment. The three smaller colored dots are landmarks. The grey larger dot is the speaker agent and the larger colored dot is the listener, the color matches the color of the target landmark for visualization purposes.

(b) Cooperative navigation environment. The three smaller dots are landmarks and the three larger dots are agents that have to cover all three landmarks.

Figure 3.1: Schematic illustrations of the two environments used for our experiments.

move. The speaker agent (grey in the figure) is not able to move but can communicate and its only observation is the color of the target landmark. The listener agent (blue in the figure, which represents its target) can move but cannot communicate and observes its velocity vector, the relative landmark positions and the symbols send by the speaker. The task is episodic and ends after 25 steps or when the listener reached the target landmark. This is the case when it is within distance ϵ to the center of the target landmark. The reward is the negative distance between the listener and the target landmark and a bonus reward of 10 is given for reaching the target.

In a more complex version of this task, both agents are speaker and listener at the same time. Thus, each agent knows the target of the other agent and both are able to move and communicate. In addition to the target color of the other agent, the observations of each agent are the same as of the listener described above. Both agents have to reach their target landmarks to successfully finish the task. The reward is the sum of the negative distances of each agent to its target location and a bonus reward of 10 for reaching the goal.

We refer to the first described task as *speaker-listener* and the second as *reference*. Both of these task require communication to successfully finish the task, as only

the other agents knows the target locations. Without this knowledge, the agents only can head to a random landmark and thus cannot not achieve a success rate of 100% over multiple runs.

3.3.2 Cooperative Navigation

This environment contains three landmarks and three agents whose task is to cover all three landmarks, while avoiding collisions (as illustrated in Figure 3.1b). All agents receive the same type of observations: their velocity vector, the relative landmark positions and the communication. In the original environment, used in the experiments in [31], the agents observed the positions of the other agents as well. We removed these observations and added the ability to communicate instead. In this way, this environment becomes partially observable for each agent and the agents are required to communicate to coordinate their actions.

The reward is the negative of the sum of the minimal distance for each landmark to any agent, therefore there is the incentive that each agent is as close as possible to a distinct landmark. In addition, there is a penalty of -1 for every time two agents collide and a bonus of 10 for fulfilling the task. An episode ends after 25 steps or when all landmarks are covered. This environment tests if the agents are able to coordinate through communication, such that they do not move towards the same landmark at the same time.

Chapter 4

Experiments

With our experiments, we want to assess the effect of partial observability of other agents' observations and actions on the ability of the MADDPG and Rec-MADDPg algorithms to train agents that are able to cooperate. All our experiments are conducted in the cooperative physical environments we have introduced in Section 3.3.

To simulate partial observability of the the agents, we add noise to the observations and actions, as described in Section 3.1. For the observations we add Gaussian noise, where we can vary the standard deviation σ to simulate different levels of uncertainty and for the actions we us the Gumbel-Softmax distribution, where we can vary the temperature τ. In the experiments, we vary σ and τ separately to be able to assess the influence of uncertainty about the actions and observations independently.

We describe the details of our implementation of the different algorithm in the next section, to allow for reproducibility of our results. After that, we describe how we evaluate the different algorithms and finally present and discuss the results of the experiments.

4.1 Implementation Details

We have implemented both MADDPG and Rec-MADDPG in PyTorch and the environments are implemented using the OpenAI gym [7] framework[1]. The implementations of both algorithms directly follow the implementation of Lowe et al. [31] and the environments are a modification version of the implementation of the environments introduced by Mordatch and Abbeel [36] provided by OpenAI[2]. In the following, we describe the most important details of our implementation.

Our feedforward policy and critic networks consist of 3 layers with 64 units each. The critic networks have one output unit that predict the action values for the given observations and actions. For the recurrent critic we replace the second layer with a LSTM layer with 64 memory units, such that there is one fully connected layer before the LSTM and one layer afterwards that computes the action value. The policy networks take the observations of an agent and output logits which are passed to a probability distribution. For all hidden layers we use ReLU activation functions, but for the LSTM tanh and sigmoid activations are used [21].

Following the reference implementation of MADDPG from Lowe et al. [31], we use the Gumbel-Softmax distribution [22, 32] as a differentiable approximation of the categorical distribution. For deterministic actions we take the action probabilities, which is are the softmax values of the logits. Throughout the experiments we use a temperature of $\tau = 1$ for the Gumbel-Softmax distribution.

In the experiments we use a discount factor of $\gamma = 0.95$. For every training run, we do 1.5 million simulation steps and train the agents every 100 steps. At every training step the networks are optimized using Adam [23] with base learning rates 0.01 and 0.001 for critic and actor respectively. The target network parameters are updated with a rate of $\tau = 0.01$. We use a replay buffer size of 1 million transitions or 50,000 episodes. The mini-batches sampled from the replay buffers have a size of 1024 and 256 for transitions and episodes respectively.

The agent models use the same network architecture as the actors and are also

[1] Our implementation of MADDPG and Rec-MADDPG: `https://github.com/nicoring/rec-maddpg`, implementation of the environments: `https://github.com/nicoring/multiagent-particle-envs`

[2] Reference implementation of MADDPG: `https://github.com/openai/maddpg` and code of the original environments: `https://github.com/openai/multiagent-particle-envs`

trained with Adam with a base learning rate of 0.0001. For every training step of actor and critic the models are optimized for 20 steps on mini-batches of 64 transitions sampled from the most recent 5000 transitions.

4.2 Evaluation Metrics

We evaluate the performance of the agents based on the received cumulative reward at the end of the episode and the success rate of reaching a goal over multiple evaluation runs. For both reward and success rate we are interested how they develop over time during training, which shows how fast the different approaches reach an optimal or good solution. In addition, we are interested in the final performance of the methods after a fixed number of training steps.

We use a stochastic behavior policy during training to ensure sufficient exploration, but learn a deterministic policy. Therefore, to evaluate the performance of the trained policy, we run 50 evaluation episodes every 1000 training steps. The reported reward is then the average final reward over these 50 evaluation runs and the success rate is the fractions of these runs that were successful.

Since reinforcement learning includes a lot of randomness, due to exploration and stochastic environments, the performance can vary a lot between different runs. Therefore, we average our results over 10 runs with different random seeds and determine the confidence interval of the real mean of the performance with bootstrapping[3] and determine if performances are significantly different with 2-sample unequal variances t-tests[4], as proposed in [19].

4.3 Results

In this section, we present the results of the experiments we have conducted. We wanted to assess how much partial observability impairs the MADDPG method and test if we can improve the performance with Rec-MADDPG. All agents are trained using our implementation, as described in Section 4.1. Adding noise to the

[3]Used bootstrapping implementation: `https://github.com/facebookincubator/bootstrapped`

[4]Used t-test implementation from Scipy: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html`

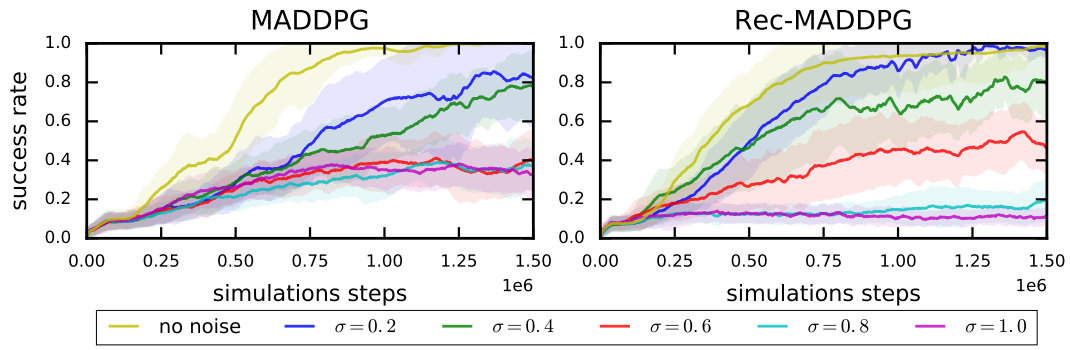


Figure 4.1: Learning curves for MADDPG on the left and Rec-MADDPG on the right for the speaker-listener task, showing the success rates for different values of the standard deviations for the observation noise.

observed actions and observations is tested separately to determine their impact independently. We report the rewards and success rates throughout the training, as discussed above. We first present the results from experiments in the cooperative communication environment and then the results from the cooperative navigation environment. After that we compare the final achieved success-rates and returns and analyze the predictions of the critic networks. We summarize and discuss conclusions from these results in the next section.

4.3.1 Cooperative Communication

We first examine the speaker-listener task in the cooperative communication environment, see Section 3.3.1 for a detailed description. The difficulty in this task is that the speaker, has to communicate the its private observation, the color of the target landmark, and concurrently the listener has to learn to understand the sent messages and move to the respective landmark. Independent learners, such as DDPG, that do not have knowledge about the other agents, are not able to solve this environment [31]. They only learn to move to the centroid of all three landmarks, which minimizes the distance to all of them. Thus, although, this task seems fairly simple, it has not been solved yet using traditional single-agent reinforcement learning methods. Therefore, it is interesting to test how much the performance decreases as we increase the noise-level, because if the uncertainty about the other agents' observations and actions is too high, the MADDPG method is basically equivalent to DDPG.

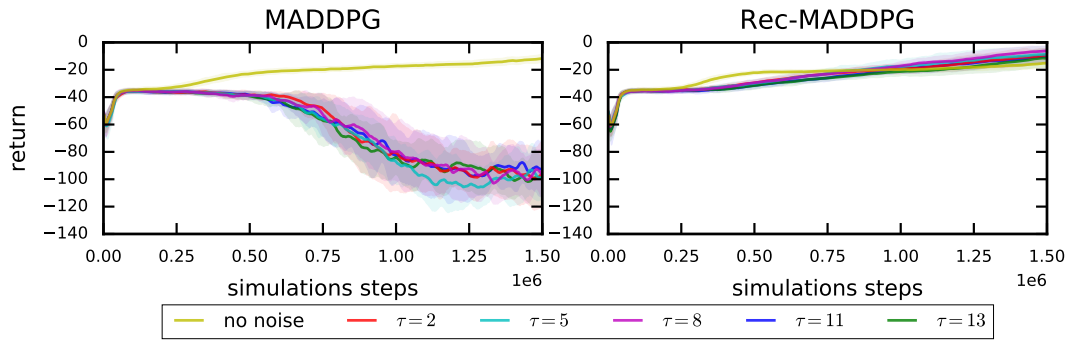


Figure 4.2: Learning curves for MADDPG on the left and Rec-MADDPG on the right for the reference task, showing the reward for different values of the temperature for the noise in observed actions.

Figure 4.1 compares the performance of MADDPG and Rec-MADDPG during training for varying noise-levels for the observations of the other agents. We can see that Rec-MADDPG converges faster to higher success rates for lower noise-levels in general and is even able to reach success rates of 100% up to a certain degree of uncertainty. However, it has worse success rates than MADDPG for higher noise-levels at the end of the training.

While the performance of both algorithms decreased for increasing uncertainty about other agents' observations, we found that both algorithms are robust against noise in the observed actions for this task (the results are given in Figure B.1). Both algorithms were able to achieve success rates of 100% for all noise-levels, without significant differences in mean performance.

For the reference task, however, there were significant differences between MADDP and Rec-MADDPG. As can be seen in Figure 4.2, the achieved reward of Rec-MADDPG is rising similarly for different levels of uncertainty, while achieving mean success rates around 70%, as shown in Figure B.2. For MADDPG, in contrast, the reward decreases over time for all noise-levels and the success rate does not exceed 0%. So it seems that using Rec-MADDPG poses a significant advantage for this task if the actions of other agents are only observed partially.

4.3.2 Cooperative Navigation

Next, we examine the ability of MADDPG and Rec-MADDPG to learn to coordinate in the spread task, while only getting noisy observations of the other agents to train the critic network. The difficulty in this task is that the agents have to coordinate to which landmarks they go, while not observing where the other agents are or which actions they take. Thus, to successfully complete the task the agents need to communicate in order to coordinate their actions.

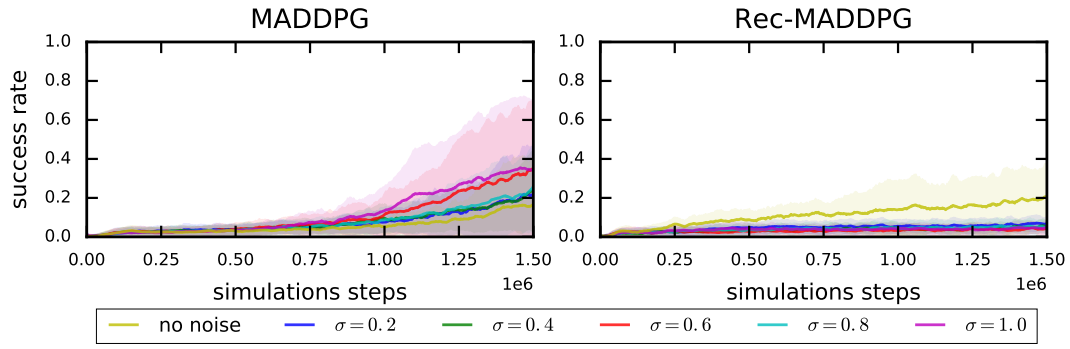


Figure 4.3: Learning curves for Rec-MADDPG on the left and MADDPG on the right for the cooperative navigation task. Both diagrams show the success rate for different values of the standard deviation of the noise in observed observations of other agents.

First of all, we observed that even without partial observability agents never consistently reached a success rate of 100% in the given training time. Therefore, this task seems to be harder than the cooperative communication tasks. When only partially observing other agents' observations (see Figure 4.3), the Rec-MADDPG algorithm is not able exceed success rates of 10%, in contrast to MADDPG that achieves mean success rates of up to 40%. When analyzing the learned policies of Rec-MADDPG more closely, we found that they sometimes move towards separate landmarks but only rarely reach them and quite often multiple agents move towards the same landmark.

Interestingly, when only the actions of other agents are partially observed, Rec-MADDPG performs better than MADDPG. Rec-MADDPG is able to train policies that achieve success-rates of up to approximately 60%, as shown in Figure 4.4. MADDP's success-rates, in contrast, are close to 0% for noisy observed actions throughout the whole training. Therefore, we think that Rec-MADDPG is able

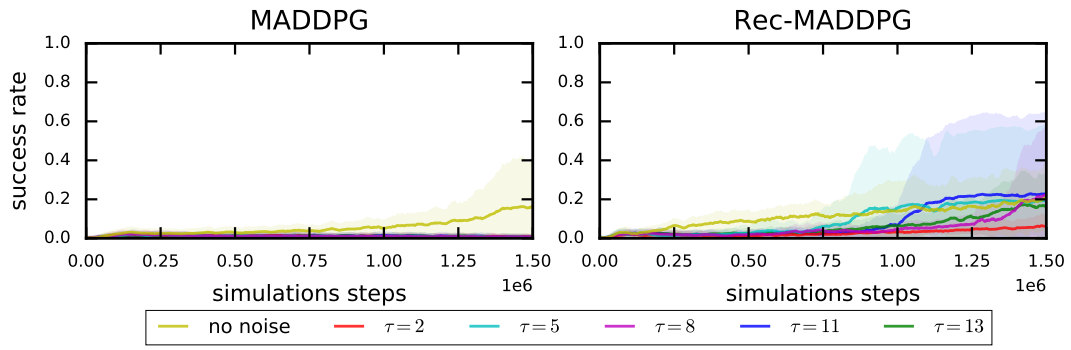


Figure 4.4: Learning curves for Rec-MADDPG on the left and MADDPG on the right for the cooperative navigation task. Both diagrams show the success rate for different values of the temperature for the noise in observed actions.

to infer the actions given the history of observation and noisy actions and thus the recurrent critic poses an advantage in this scenario.

4.3.3 Comparison of Final Performance

After examining the learning curves of MADDPG and Rec-MADDPG for all three tasks, we now compare the final performances across the tasks for different noise-levels for the observed actions and observations. This enables us to make more precise statement about which algorithms performed better for which task. Additionally, we can see how the performance changes for increasing noise-levels.

In Figure 4.5, we show the mean success rates and 95% confidence intervals for 10 runs. For the case of partial observability of actions of other agents, we can clearly see that both algorithms solve the speaker-listener task perfectly for all noise-levels and there are also no significant differences for in the returns (cf. Figure B.3). None of the algorithms solves the reference or spread tasks perfectly, but Rec-MADDPG is significantly better on the reference task than MADDPG, which only achieves success rates of 0% for all noise-levels. Although Rec-MADDPG does not achieve significantly higher success rates for the spread task, the final mean returns for this task of Rec-MADDPG are significantly better than of MADDPG (cf. Tables B.2 and B.4). Interestingly, there are no significant differences in the performance of Rec-MADDPG when the noise increases. This strengthens our assumption that Rec-MADDPG is able to infer the next action based on the observation history.

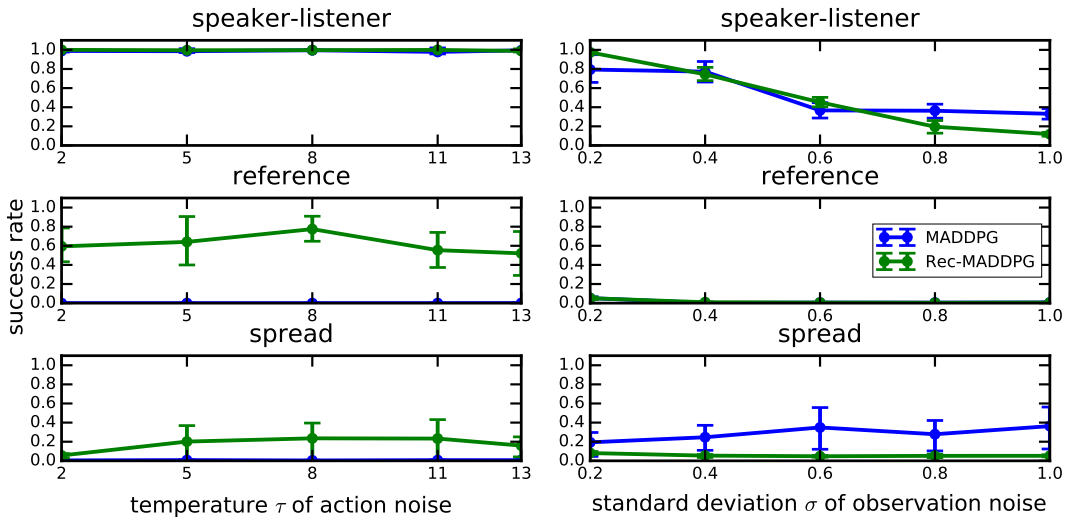


Figure 4.5: Final success rates achieved by MADDPG and Rec-MADDPG for different environments and noise-levels. The line plots show the mean performance and the bars represent the 95% confidence intervals. Significance values can be found in Tables B.1 and B.2.

In case of partial observability of other agents' observations, as shown on the right in Figure 4.5, the picture is not as clear as for partially observed actions. In our first experiment, it seemed to vary for different noise levels which of the algorithms performs better on the speaker-listener task. Therefore, we conducted a second experiment with more fine noise-levels. The results of this experiment are shown in Figure 4.6 and p-values are reported in Table B.5. There we can see that for $\sigma \leq 0.4$ none of the algorithms is significantly better. For $\sigma = 0.5$ and $\sigma = 0.6$ the confidence intervals do not overlap and the p-values are below

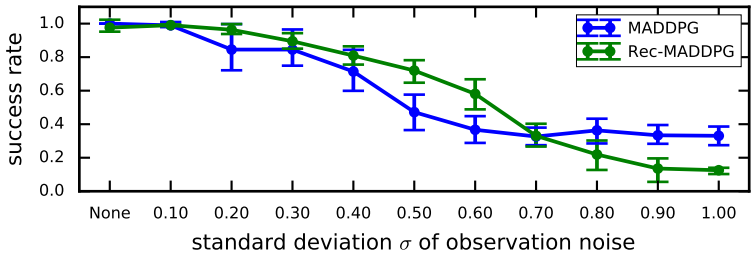


Figure 4.6: Final success rates for the speaker-listener task of MADDPG and Rec-MADDPG for increasing observation noise. The line plots show the mean performance and the bars represent the 95% confidence intervals. Significance values can be found in Table B.5.

the significance level of 0.05 and therefore we conclude that the differences in the performance are significant. For noise-levels of $\sigma \geq 0.8$, however, MADDPG has a significantly better performance than Rec-MADDPG.

For the reference task with partial observability of other agents' observations both of the algorithms achieve success rates not much higher than 0%, Rec-MADDPG achieves significantly better returns than MADDPG (see B.3 and Table B.3). Thus, while both algorithms have not yet learned to successfully solve the task completely, the agents trained with Rec-MADDPG seem to move closer to the correct target landmarks. For the spread task, in contrast, MADDPG achieves better success rates and significantly higher returns than Rec-MADDPG.

4.3.4 Action-Value Prediction Error

We now further investigate the quality of the critics and how the feedforward critic compares to the recurrent critic. We stored the trained neural networks of all ten returns. Then, after the training has finished we ran the final policy to create 100 episodes for each run. For each state-action pair in these episodes we computed the action-value using the final critic network and calculated the returns for each time-step. We are interested in how well the critic predicts the actual return. Thus, we computed the mean squared error $\text{MSE}_t = \frac{1}{N}\sum_t (R_t - Q_i(\boldsymbol{o}_t, \boldsymbol{a}_t))^2$ between return and predicted action-value for all episodes and time-steps.

We computed the MSE for every algorithm, task and noise combination and averaged the results over the 10 runs. The results are shown in Figure 4.7. The results are correlated to the final performance of the algorithms described in the previous section, i.e. when one algorithms has a better performance on a task then the error of the predicted action-values is smaller. Interesting insights are how much the errors differ. For the reference and spread tasks with partial observability of the actions the errors of MADDPG are extreme, which corresponds with the success rates of 0%. This shows that the feedforward critic is not able at all to predict the action values, when the actions of other agents are uncertain.

The prediction errors of the recurrent critic for the speaker-listener task with partial observed observation of other agents only partly matches the final performance of Rec-MADDPG on this task. While the prediction errors of Rec-

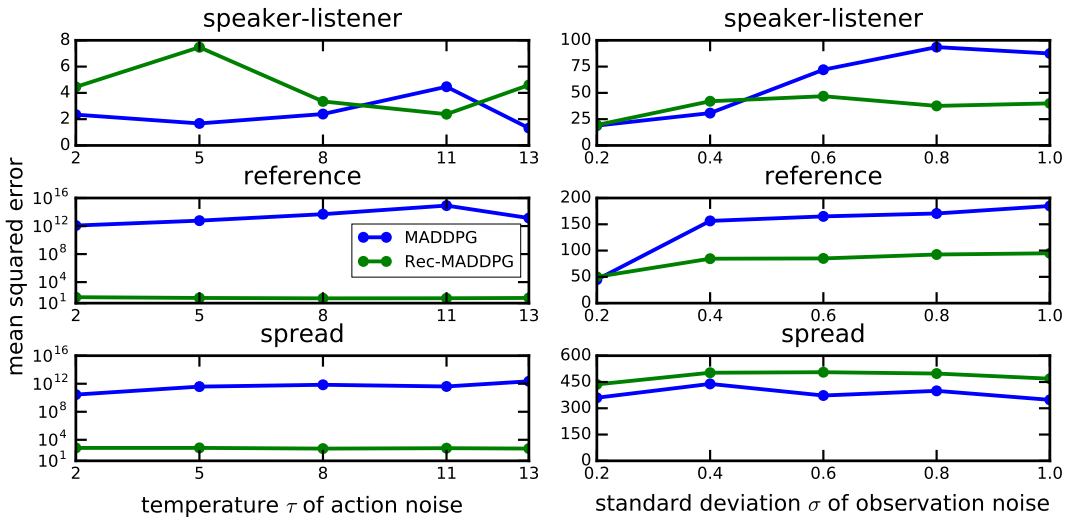


Figure 4.7: Comparison of mean squared error between predicted action-values and actual returns for MADDPG and Rec-MADDPG for different environments and noise-levels.

MADDPG are smaller than MADDPG's for higher noise-levels, it has a significant worse performance compared to MADDPG for observation noise with $\sigma \geq 0.8$.

Figure 4.8 shows the mean squared error against the steps until the episode's end for a noise-level of $\sigma = 0.8$. There we can see that Rec-MADDPG has higher errors for action-values close to the end of an episode. We believe the increase of the error at the end is caused by the fact that, when training the rolled-out recurrent neural network, earlier steps receive more information from later steps than steps in the end. Thus, the gradient for earlier steps is less noisy and

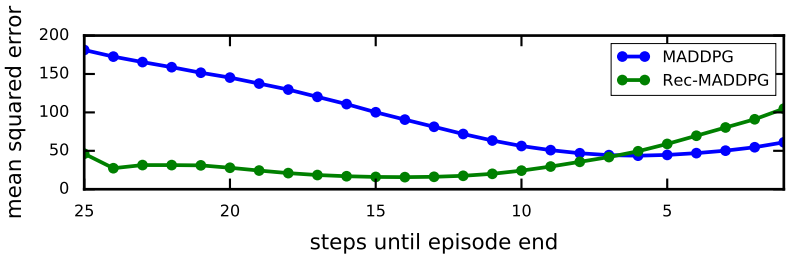


Figure 4.8: Mean squared error of the predicted action-values at different time-steps for MADDPG and Rec-MADDPG on the speaker-listener with noise-level $\sigma = 0.8$. Since episodes have varying length, we use the steps until the episode's end on the x-axis.

more accurate. Therefore, the policy improvement is done on worse action-value predictions, for the last steps of an episode, leading to a worse performing policy compared to MADDPG.

4.4 Discussion

We wanted to explore if the MADDPG algorithm and our recurrent adaption Rec-MADDPG can deal with noisy observations and actions of other agents in a decentralized multi-agent scenario. In this section, we summarize and discuss the results of our experiments.

The experiments have shown that neither of MADDPG and Rec-MADDPG are significantly better in all tested scenarios. However, we found that the recurrent critic of Rec-MADDPG can alleviate the challenges MADDPG has with partial observability of other agents in some scenarios. When actions were only partially observed, Rec-MADDPG was still able to learn reasonable policies for the reference and spread task, which solved the task in approximately 70% and 20% of the cases respectively. For these two tasks, however, MADDPG was not able to learn successful policies with success rates not much larger than 0%. In these situations, Rec-MADDPG seems to be able to compensate the uncertainty by predicting the action-values based on the history of observations. Thus, we think that given the history of observations and partially observed actions recurrent critics are able to infer the taken actions of other agents in these physical environments. This observation might not generalize to all partially observed environments, because here the movement directly results in a change of the position and thus action can be easily reconstructed given the history of observations.

For partially observed observations of other agents in the cooperative communication environment, Rec-MADDPG had a similar and sometimes even significantly better performance than MADDPG for noise with a standard deviation of up to 0.7. In the experiments with more observation noise, Rec-MADDPG had a significantly worse performance than MADDPG. In the cooperative communication environment, agents are required to communicate private observations to successfully complete the task. Thus, this shows that a recurrent critic is able to infer the private information for lower noise-levels, but fails to do so for too much

noise. We believe that in the latter case the complexity of training recurrent neural networks poses a disadvantage compared to MADDPG, which uses easier to train feedforward networks.

In addition, the Rec-MADDPG algorithm failed to train successful policies for the cooperative navigation task, when given noisy observations of other agents. MADDPG, in contrast, was able to train policies successfully, but none of the learned policies achieved success rates of 100%. This shows that Rec-MADDPG can be advantageous if the observations of other agents are stationary. In the cooperative communication environment, the the critic of the listener only partially observes which color of the target landmark is observed by the speaker. Thus, when the critic is recurrent it can learn to estimate the true observation based on multiple noisy observations. For the cooperative navigation task, however, the positions of the other agents are of most interest for the critic to predict the action-values and the distributions of the noisy observations change as the policies of the other agents change. In this case, Rec-MADDPG seems to not be able to generate good action-value predictions as the learned policies show a poor performance. Thus, we assume that recurrent critics are mainly beneficial for partially observed observations of other agents that are stationary.

While it was interesting to analyze the impact of noise in observed actions and observations independently, it would also have been interesting to do more experiments where we apply both types of noise. On the other hand, it would also be interesting to apply the noise more specifically. For example, only applying noise for the movement part of the actions, because one could argue that we could keep track of received messages and therefore know what messages other agents have send. The same is true for the observations, as the incoming messages are part of the observations.

Chapter 5

Related Work

Multi-agent reinforcement learning (MARL) sits at the intersection of multi-agent systems [55, 48] and reinforcement learning (RL) [51]. The survey of Busoniu et al. [8] provides an overview and a taxonomy of algorithms in this field. Standard RL is mainly concerned with finding the optimal policy or getting close to an optimal policy that maximizes the return for a given problem. Multi-agent learning is more complex because the environment is non-stationary due to concurrently learning agents. While maximizing the reward is the clear goal of single-agent RL, different learning goals have been proposed for MARL. Examples of learning goals are to find policies that form an equilibrium, Pareto-optimality or achieving maximal social welfare.

In this thesis, we focused on decentralized and cooperative environments, where all agents have the same reward function and only can partially observe other agents. The learning goal for these tasks is to find a joint policy that maximizes the expected shared cumulative reward. One of the first algorithms for decentralized MARL was *independent Q-learning* [53], where each agent learns a action-value function using the single-agent Q-learning algorithm. The *distributed Q-learning* [26] accounts for concurrent learning with multiple agents by ignoring low returns, which are assumed to be caused by exploratory actions of other agents. Both algorithms, however, are based on tabular Q-functions and therefore do not scale to large problems.

Within the last few years, there has been great success with applying deep neural networks in single-agent RL settings with large state and/or action spaces [35, 46,

29, 18]. Recently, deep Q-networks (DQN) have been combined with *independent Q-learning* [53] to study emergence of cooperative and competitive behavior in the game of pong [52] and sequential social dilemmas [28]. Since these approaches are based on *independent learning* they are applicable in decentralized settings but they assume that the agents are completely independent and have no knowledge about each other. Additionally, although DQN can deal with large state-spaces it is infeasible for large action-spaces.

There also has been a lot of recent work for learning to communicate and cooperate with centralized learning and decentralized execution. Foerster et al. [13] use an adaption of recurrent DQN [15] with shared weights between all agents and without experience replay to learn simple communication protocols to solve riddle games. Other approaches use differentiable communication channels such that gradients can be passed between agents during training [10, 50]. Mordatch and Abbeel [36] proposed a model-based and centralized method that uses the same policy for every agent and thus it is not suitable for decentralized environments.

Our work is based on the multi-agent actor-critic MADDPG algorithm from Lowe et al. [31], which uses a centralized critic for every agent, where the critic has access to the actions and observations of other agents. In this thesis, we assume that the other agents are partially observable. Since MADDPG has a separate critic for every agent it applies to decentralized settings when modeling other agents' policies. Foerster et al. [12] also proposed an actor-critic algorithm that uses a specialized policy gradient to address the credit assignment problem, but they use a single centralized critic for all agents and thus there is no simple modification to enable this method for decentralized environments.

Another method by Foerster et al. [11] proposes a policy gradient method that accounts for anticipated policy changes of other agents when updating an agent's policy. They propose the same agent modeling method as we use and has been proposed by Lowe et al. [31], but their work is focused on competitive games while we focus on cooperative environments.

Other recent work has specifically addressed the problem of modeling other agents for deep MARL. The following approaches could be used instead of learning a policy model for every other agent in the environment. Rabinowitz et al. [41] and Grover et al. [14] propose similar approaches that combine agent identification and

policy reconstruction, by learning to generate agent embeddings from trajectories and to predict actions given an embedding. Raileanu et al. [42] propose to use the agent's own policy to predict other agents' actions. They assume that each agent has some hidden state based on which they choose their actions. Thus, they propose to infer the hidden state and then use the agent's own policy in combination with the inferred state to predict actions of the other agent. These methods are specifically designed for deep RL, but there has been done a lot more research on agent modeling in the field of multi-agent systems. The survey from Albrecht and Stone [1] gives a good overview on this subject.

Our approach is also related to independent learner algorithms as these naturally are decentralized. One recent proposal that is similar to our work is the deep decentralized multi-task and MARL algorithm from Omidshafiei et al. [39]. Similar to one proposed model in [10] they use recurrent DQN, without sharing the weights between the agents. Furthermore, they apply hysteretic Q-learning [33] which addresses the problem that an agent can receive a poor reward due to bad action choices of other agents and has been shown to work well empirically [57, 34, 3]. In hysteretic Q-learning, a smaller learning rate is used when the temporal difference error for updating the action-values is negative. While the approach of Omidshafiei et al. [39], would not scale for the the environments we did our experiments in, it could be interesting for future work if using hysteretic Q-learning to train the critics would be beneficial.

Chapter 6

Conclusion and Future Work

In this thesis, we proposed the Rec-MADDPG algorithm as a recurrent extension of MADDPG [31] and compared their performance in partially observable cooperative environments. To simulate different degrees of partial observability of other agents, we introduced adding noise to the observed actions and observations. We conducted experiments with different simulated noise levels for observed actions and observations.

When actions were only partially observed, Rec-MADDPG was still able to learn policies that were able to solve the reference and spread tasks in approximately 70% and 20% of the cases respectively. In contrast, MADDPG was not able to achieve success rates much larger than 0% for two tasks. Therefore, we concluded that the recurrent critic seems to be able to reconstruct the taken action based on the history of the observations.

Rec-MADDPG can be advantageous over MADDPG when inferring stationary observations of other agents, if the noise-level is not too high. However, for too high noise levels or non-stationary observations, such as the positions of the agents, Rec-MADDPG had a significantly worse performance compared to MADDPG.

We only analyzed partial observability of actions and observations separately. Therefore, next steps for future work could be to compare Rec-MADDPG and MADDPG with partially observability of actions and observations combined. It would also be interesting to test if our findings hold for competitive and mixed environments. Additionally, more experiments should be conducted in more com-

plex and realistic environments that have a natural partial observability of other agents to test if our simulated partial observability is appropriate. In more realistic environment, the state of the environment is not any more jointly observable, thus these experiments would also investigate if our approach generalizes to Dec-POMDPs.

Rec-MADDPG seems to be able to infer taken actions of other agents given their observations and partially observed actions. Usually, it is not possible to observe actions directly, for example we can observe how a robot moves around but we do not directly observe which internal actions it chose. Therefore, future research could examine if Rec-MADDPG still works when no actions are observed at all. If this does not work, another possibility would be to use models to infer actions from environment changes and then use these action estimates for Rec-MADDPG's training procedure.

We only made the critic recurrent, because we assumed a jointly observable state and that it is possible to communicate all necessary state information at each step. If this is not the case, however, the actor might need to be able to maintain an internal state too. Thus, future work could also use recurrent neural networks for the actors and agents models.

In summary, this thesis has shown that Rec-MADDPG can be significantly better than MADDPG, especially when the actions of other agents are only partially observed. However, in some environments MADDPG has a better performance than Rec-MADDPG, thus which algorithms performs better is highly dependent on the specifics of an environment.

Appendix A

Algorithms

For completeness, we provide the algorithms for DDPG, MADDPG and Rec-MADDPG below.

Algorithm 1 Deep Deterministic Policy Gradient (DDPG) [29]

Randomly initialize critic $Q_w(s, a)$ and actor $\mu_\theta(s)$ with weights w and θ
Initialize target network Q' and μ' with weights $\theta' \leftarrow \theta$, $w' \leftarrow w$
Initialize replay buffer $\mathcal{R}$
for episode $= 1, 2, \ldots, n$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation o_1
  for $t = 1, 2, \ldots, T$ **do**
    Selection action $a_t = \mu_\theta(o_t) + \mathcal{N}_t$ according to current policy and exploration noise
    Execute action a_t and receive reward r_t and new observation o_{t+1}
    Store transition (o_t, a_t, r_t, o_{t+1}) in $\mathcal{R}$
    Sample a random minibatch of B transitions (o_b, a_b, r_b, o_{b+1}) from $\mathcal{R}$
    Set $y_b = r_b + \gamma Q'_{w'}(o_{b+1}, \mu'_{\theta'}(o_{b+1}))$
    Update critic by minimizing loss: $L(w) = \frac{1}{N}\sum_b (y_b - Q_w(o_b, a_b))^2$
    Update the actor's policy using the sampled policy gradient:

$$\nabla_\theta J \approx \frac{1}{N} \sum_b \nabla_\theta \mu_\theta(o_b) \nabla_a Q_w(o_b, \mu_\theta(o_b))$$

    Update the target networks:

$$w' \leftarrow \tau w + (1 - \tau) w'$$
$$\theta' \leftarrow \tau \theta + (1 - \tau) \theta'$$

  end for
end for

Algorithm 2 Multi-Agent Deep Deterministic Policy Gradient for n agents [31]

for agent $i = 1$ to n **do**
  Randomly initialize critic $Q_{w_i}(s, a)$ and actor $\mu_{\theta_i}(s)$ with weights w_i and θ_i
  Initialize target network Q'_i and μ'_i with weights $\theta'_i \leftarrow \theta_i$, $w'_i \leftarrow w_i$
end for
Initialize replay buffer $\mathcal{R}$
for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observations $\boldsymbol{o}_1$
  for $t = 1$ to T **do**
    select actions for each agent i: $a_t^{(i)} = \mu_i(o_t^{(i)}) + \mathcal{N}_t$, w.r.t. current policy and exploration noise process
    Execute joint action $\boldsymbol{a}_t = (a_t^{(1)}, \ldots, a_t^{(n)})$ and receive r_t and $\boldsymbol{o}_{t+1}$
    Store transition $(\boldsymbol{o}_t, \boldsymbol{a}_t, r_t, \boldsymbol{o}_{t+1})$ in $\mathcal{R}$
    for agent $i = 1$ to n **do**
      Sample a random minibatch of B transitions $(\boldsymbol{o}_b, \boldsymbol{a}_b, r_b, \boldsymbol{o}_{b+1})$ from $\mathcal{R}$
      Set $y_b = r_b + \gamma Q'_i(\boldsymbol{o}_{b+1}, \boldsymbol{a'})|_{a'_j = \mu'_j(o_{j,b+1})}$
      Update critic by minimizing loss: $L(w_i) = \frac{1}{B}\sum_b (y_b - Q_i(\boldsymbol{o}_b, \boldsymbol{a}_b))^2$
      Update the actor's policy using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{B} \sum_b \nabla_{\theta_i} \mu_i(o_{b,i}) \nabla_{a_i} Q_i(\boldsymbol{o}_b, \boldsymbol{a}_b)|_{a_i = \mu_i(o_{b,i})}$$

    end for
    Update the target networks for each agent i:

$$w'_i \leftarrow \tau w_i + (1 - \tau) w'_i$$
$$\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$$

  end for
end for

Algorithm 3 Recurrent Multi-Agent Deep Deterministic Policy Gradient for n agents

for agent $i = 1$ to n **do**
  Randomly initialize critic $Q_{w_i}(s, a)$ and actor $\mu_{\theta_i}(s)$ with weights w_i and θ_i
  Initialize target network Q'_i and μ'_i with weights $\theta'_i \leftarrow \theta_i$, $w'_i \leftarrow w_i$
  Initialize replay buffer $\mathcal{R}_i$
  For each other agent j initialize agent model $\hat{\mu}_i^j$ with parameters ϕ_i^j
end for
for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observations $\boldsymbol{o}_1$
  for $t = 1$ to T **do**
    select actions for each agent i: $a_t^{(i)} = \mu_i(o_t^{(i)}) + \mathcal{N}_t$, w.r.t. current policy and exploration noise process
    Execute joint action $\boldsymbol{a}_t = (a_t^{(1)}, \ldots, a_t^{(n)})$ and receive r_t and $\boldsymbol{o}_{t+1}$
    Store observations, actions and reward $(\boldsymbol{o}_t, \boldsymbol{a}_t, r_t)$ in $\mathcal{R}_i$, where $\boldsymbol{o}_{-i} \sim \omega(\boldsymbol{o}_t)$ and $\boldsymbol{a}_{-i} \sim \alpha(\boldsymbol{a}_t)$ are partially observed.
    for agent $i = 1$ to n **do**
      Train agent models: TrainModels$(R_i, \hat{\mu}_i^1, \ldots, \hat{\mu}_i^n)$
      Sample a random minibatch of B episodes from R_i:

$$(\boldsymbol{o}_1^b, \boldsymbol{a}_1^b, r_1^b, \ldots, \boldsymbol{o}_T^b, \boldsymbol{a}_T^b, r_T^b)_{b=1,\ldots,B}$$

      Compute target values $(y_1^b, \ldots, y_T^b)$ for each episode in the minibatch:

$$y_t^b = r_t^b + \gamma Q'_i(\boldsymbol{o}_{t+1}^b, \boldsymbol{a}', h_t^b),$$
$$\text{with} \quad a'_j = \begin{cases} \mu'_i(o_{i,t+1}^b) & \text{if } j = i \\ \hat{\mu}_i^j(o_{j,t+1}^b) & \text{else} \end{cases}$$

      Update critic by minimizing loss:

$$L(w_i) = \frac{1}{BT} \sum_b \sum_t (y_t^b - Q_i(\boldsymbol{o}_t^b, \boldsymbol{a}_t^b, h_t^b))^2$$

      Update the actor's policy using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{BT} \sum_b \sum_t \nabla_{\theta_i} \mu_i(o_{t,i}^b) \nabla_{a_i^b} Q_i(\boldsymbol{o}_t^b, \boldsymbol{a}_t^b, h_t^b)|_{a_i^b = \mu_i(o_{t,i}^b)}$$

    end for
    Update the target networks for each agent i:

$$w'_i \leftarrow \tau w_i + (1 - \tau) w'_i$$
$$\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$$

  end for
end for

Algorithm 4 TrainModels procedure to train agent models.

Input: replay buffer R_i and agent models to train $(\hat{\mu}_i^1, \ldots, \hat{\mu}_i^n)$
for training steps **do**
  for $\hat{\mu}_i^j$ in $(\hat{\mu}_i^1, \ldots, \hat{\mu}_i^n)$ **do**
    Sample a random minibatch of B observation-action pairs $(o_j^b, a_j^b)_{b=1,\ldots,B}$ from the latest the p transitions in R_i.
    Update agent model by minimizing loss:

$$L(\phi_i^j) = -\frac{1}{B}\sum_b \log \hat{\mu}_i^j(a_j^b|o_j^b) + \lambda H(\hat{\mu}_i^j)$$

  end for
end for

Appendix B

Supplementary Results

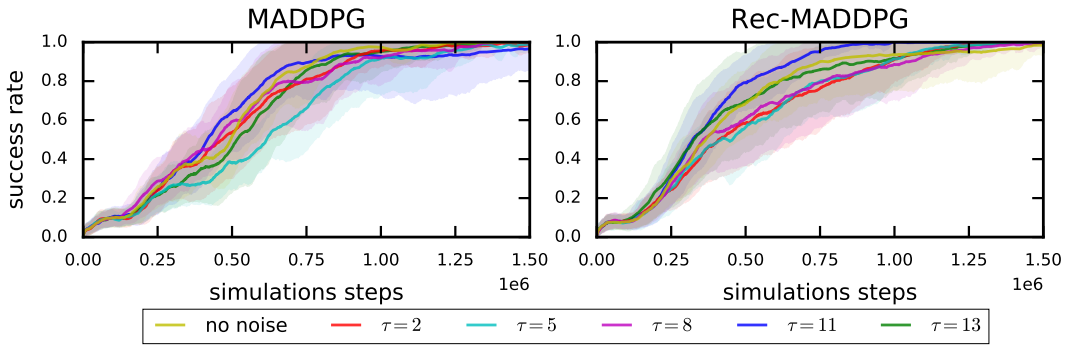


Figure B.1: Learning curves for MADDPG on the left and Rec-MADDPG on the right for the speaker-listener task, showing the success rates for different values of the temperature for the action noise.

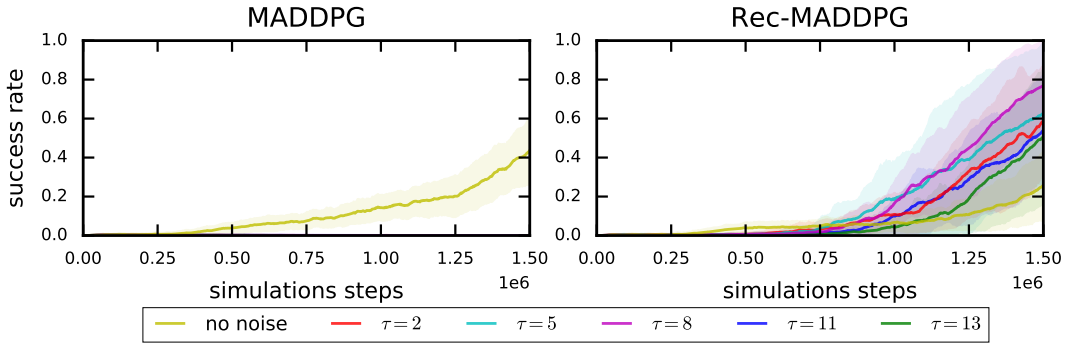


Figure B.2: Learning curves for MADDPG on the left and Rec-MADDPG on the right for the reference task, showing the success rates for different values of the temperature for the action noise.

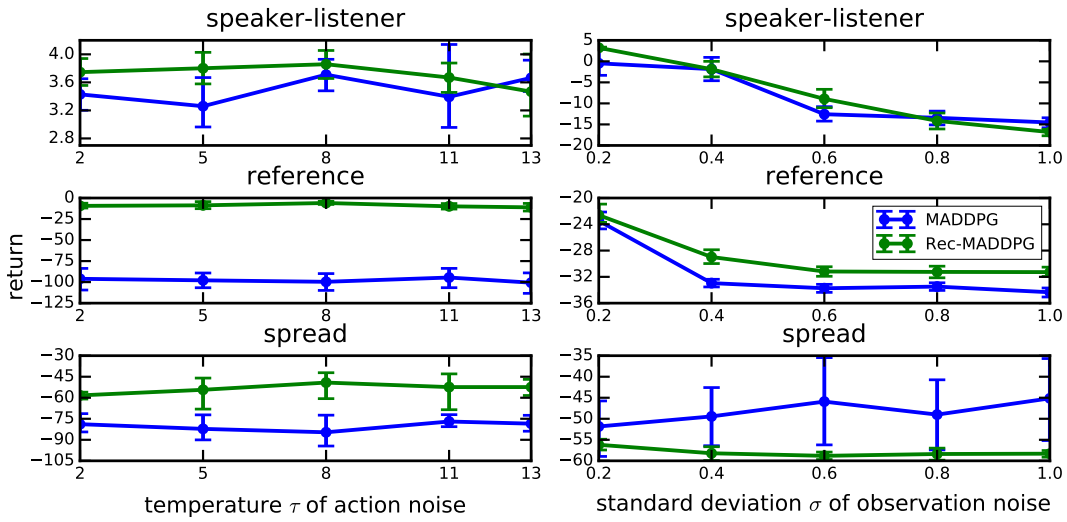


Figure B.3: Final returns achieved by MADDPG and Rec-MADDPG for different environments and noise-levels. Significance values can be found in Tables B.3 and B.4.

task name	$\sigma = 0.2$	$\sigma = 0.4$	$\sigma = 0.6$	$\sigma = 0.8$	$\sigma = 1.0$
speaker-listener	0.049	0.690	0.116	0.006	0.000
reference	0.932	0.038	0.871	0.445	0.695
spread	0.154	0.027	0.034	0.032	0.032

Table B.1: Significance, represented with p-values, of the difference between mean success rates of both algorithm for different observation noise-levels. Based on the same data as Figure 4.5.

task name	$\tau = 2$	$\tau = 5$	$\tau = 8$	$\tau = 11$	$\tau = 13$
speaker-listener	0.209	0.473	0.765	0.354	0.599
reference	0.000	0.007	0.000	0.002	0.005
spread	0.010	0.261	0.163	0.287	0.055

Table B.2: Significance, represented with p-values, of the difference between mean success rates of both algorithm for different action noise-levels. Based on the same data as Figure 4.5.

task name	$\sigma = 0.2$	$\sigma = 0.4$	$\sigma = 0.6$	$\sigma = 0.8$	$\sigma = 1.0$
speaker-listener	0.052	0.991	0.027	0.593	0.014
reference	0.428	0.000	0.000	0.003	0.000
spread	0.269	0.048	0.052	0.070	0.039

Table B.3: Significance, represented with p-values, of the difference between mean returns of both algorithm for different observation noise-levels. Based on the same data as Figure B.3.

task name	$\tau = 2$	$\tau = 5$	$\tau = 8$	$\tau = 11$	$\tau = 13$
speaker-listener	0.067	0.034	0.374	0.475	0.494
reference	0.000	0.000	0.000	0.000	0.000
spread	0.000	0.008	0.001	0.034	0.000

Table B.4: Significance, represented with p-values, of the difference between mean returns of both algorithm for different action noise-levels. Based on the same data as Figure B.3.

no noise	$\sigma = 0.1$	$\sigma = 0.2$	$\sigma = 0.3$	$\sigma = 0.4$	$\sigma = 0.5$
0.292	0.921	0.142	0.154	0.197	0.021

$\sigma = 0.6$	$\sigma = 0.7$	$\sigma = 0.8$	$\sigma = 0.9$	$\sigma = 1.0$
0.016	0.580	0.023	0.010	0.000

Table B.5: Significance, represented with p-values, of the difference between mean success rate of both algorithm for different observation noise-levels for the speaker-listener task. Based on same data as Figure 4.6.

Bibliography

[1] S.V. Albrecht and P. Stone. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence*, 258:66–95, 2018.

[2] Tucker Balch and Ronald C Arkin. Communication in reactive multiagent robotic systems. *Autonomous robots*, 1(1):27–52, 1994.

[3] Nikos Barbalios and Panagiotis Tzionas. A robust approach for multi-agent natural resource allocation based on stochastic optimization algorithms. *Applied Soft Computing*, 18:12–24, 2014.

[4] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[5] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[6] Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[8] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, And Cybernetics-Part C: Applications and Reviews, 38 (2), 2008*, 2008.

[9] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. *Proceedings of National Conference on Artificial Intelligence AAAI/IAAI*, 1998:746–752, 1998.

[10] Jakob Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2137–2145, 2016.

[11] Jakob Foerster, Richard Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning aware-

ness. In *Proceedings of the Seventeenth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2018.

[12] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[13] Jakob N Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *arXiv preprint arXiv:1602.02672*, 2016.

[14] Aditya Grover, Maruan Al-Shedivat, Jayesh Gupta, Yuri Burda, and Harrison Edwards. Learning policy representations in multiagent systems. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1802–1811, 2018.

[15] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents*, 2015.

[16] Serhii Havrylov and Ivan Titov. Emergence of language with multi-agent games: Learning to communicate with sequences of symbols. In *Advances in Neural Information Processing Systems*, pages 2149–2159, 2017.

[17] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

[18] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[19] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*, 2017.

[20] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

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

[22] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *Proceedings of the International Conference on Learning Representations*, 2017.

[23] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[24] Landon Kraemer and Bikramjit Banerjee. Rehearsal based multi-agent re-

inforcment learning of decentralized plans. In *The 8th Workshop Multiagent Sequential Decision Making Under Uncertainty*, page 24, 2013.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

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

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[28] Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 464–473. International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[29] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *Proceedings of the International Conference on Learning Representations*, 2016.

[30] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[31] Ryan Lowe, YI WU, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6382–6393, 2017.

[32] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *Proceedings of the International Conference on Learning Representations*, 2017.

[33] Laëtitia Matignon, Guillaume Laurent, and Nadine Le Fort-Piat. Hysteretic q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 64–69, 2007.

[34] Laetitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. Independent reinforcement learners in cooperative markov games: a survey regarding coordination problems. *The Knowledge Engineering Review*, 27(1):1–31, 2012.

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

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

[37] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning*, pages 807–814, 2010.

[38] Christopher Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs`, 2015. Accessed: 2018-08-12.

[39] Shayegan Omidshafiei, Jason Pazis, Christopher Amato, Jonathan P. How, and John Vian. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2681–2690, 2017.

[40] Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *CoRR*, abs/1703.10069, 2017.

[41] Neil Rabinowitz, Frank Perbet, Francis Song, Chiyuan Zhang, S. M. Ali Eslami, and Matthew Botvinick. Machine theory of mind. In *Proceedings of the 35th International Conference on Machine Learning*, pages 4218–4227, 2018.

[42] Roberta Raileanu, Emily Denton, Arthur Szlam, and Rob Fergus. Modeling others using oneself in multi-agent reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, pages 4257–4266, 2018.

[43] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.

[44] Sandip Sen and Gerhard Weiss. Learning in multiagent systems. *Multiagent systems: A modern approach to distributed artificial intelligence*, pages 259–298, 1999.

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

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

[47] Larry M Stephens and Matthias B Merx. The effect of agent control strategy on the performance of a dai pursuit problem. In *Proceedings of the 10th International Workshop on Distributed Artificial Intelligence*, 1990.

[48] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[49] Peter Stone, Manuela Veloso, and Patrick Riley. The cmunited-98 champion simulator team. In *Robot Soccer World Cup*, pages 61–76. Springer, 1998.

[50] Sainbayar Sukhbaatar, Rob Fergus, et al. Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems*, pages 2244–2252, 2016.

[51] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[52] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395, 2017.

[53] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[54] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[55] Gerhard Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

[56] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[57] Yinliang Xu, Wei Zhang, Wenxin Liu, and Frank Ferrese. Multiagent-based reinforcement learning for optimal reactive power dispatch. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1742–1751, 2012.