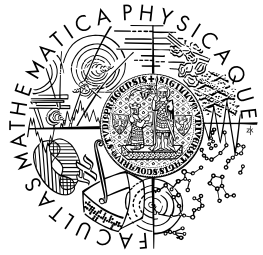


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jan Uhlík

Cooperative Multi-Agent Reinforcement Learning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Firstly, I thank my parents, Jana and Jan, for their all-life support. I also owe Káťa for maintaining my everyday sanity.

From academia, I thank my supervisor Martin for the fantastic opportunity to work on this thesis and for all his support. During the studies, I have also met marvelous friends, Jonáš Kulhánek, Patrik Valkovič and Martin Vastl who helped me go through the whole master's degree. I also want to thank Milan Straka, who directed me toward Deep (Reinforcement) Learning with his remarkable courses.

Lastly, I thank my aunts Lucka and Miluška for their English lessons. Without them, I would hardly have managed to write this thesis.

Title: Cooperative Multi-Agent Reinforcement Learning

Author: Bc. Jan Uhlík

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Deep Reinforcement Learning has achieved a plenty of breakthroughs in the past decade. Motivated by these successes, many publications extend the most prosperous algorithms to multi-agent systems. In this work, we firstly build solid theoretical foundations of Multi-Agent Reinforcement Learning (MARL), along with unified notations. Thereafter, we give a brief review of the most influential algorithms for Single-Agent and Multi-Agent RL. Our attention is focused mainly on Actor-Critic architectures with centralized training and decentralized execution. We propose a new model architecture called MATD3-FORK, which is a combination of MATD3 and TD3-FORK. Finally, we provide thorough comparative experiments of these algorithms on various tasks with unified implementation.

Keywords: Reinforcement Learning Multi-Agent Systems Deep Learning

Contents

Introduction

Since ancient times people have dreamed of constructions of Artificial Intelligence (AI), a non-biological equivalent of our minds. But only the progress in the field of computer machines during the 20th century enabled us to start fulfilling this age-long dream. One of the biggest breakthroughs is TD-Gammon [Tesauro, 1995] originally published in 1992. It is able to be a dignified opponent to an average player and moreover, it helps human experts to come up with correct backgammon strategy. However, it is not able to beat the top level a human players. We thought it would take decades of development before AI would defeat human world champion in such a sophisticated game. But only five years later in 1997 the Deep Blue [Campbell et al., 2002] misleads us. The machine beats the recent world chess champion Garry Kasparov in a six-game match. But should we really assign the credit to the AI or was Garry fighting against group of chess grandmasters equipped with a superpower computer machine? Either way, people will remember that match as a breakthrough.

In last the two decades we have witnessed rapid progress in this field. The majority of state-of-the-art (SOTA) algorithms are based on Deep Artificial Neural Network (DANN), which in sense is nothing more than composition of affine and non-linear functions. But there are two key things behind this success. Firstly, we come up with an efficient training algorithm known as Back-Propagation [Rumelhart et al., 1986]. Secondly, thanks to the hardware engineers, we have a sufficient computer power for training models with millions of parameters.

We can illustrate this rapid progress in computer vision. In 2012 first Deep Learning (DL) model called AlexNet [Krizhevsky et al., 2017] wins the image recognition contest and sets a baseline for further development. Later, the ResNet [He et al., 2016] significantly improves the results and with EfficientNet [Tan and Le, 2019] we can say that image recognition is already solved. The same can be said about object detection, where DL starts dominating with a two-stage detector Fast R-CNN [Girshick, 2015]. And again with a single-staged detector EfficientDet [Tan et al., 2020] we can mark object detection as solved. Other example is Natural Language Processing (NLP), which contains disciplines such as language translation, dialogue systems or sentiment analysis. As the main influential publication we need to remind BERT [Devlin et al., 2019], which employs attention architecture Transformer [Vaswani et al., 2017].

Up until now we have discussed so called *supervised learning*, where we have a training dataset and our goal is to come up with a model which extracts and generalizes useful patterns within data. However, in this work, we will discuss Reinforcement Learning (RL). In RL we do not have any training datasets, rather, we have an environment model which we can interact with. We usually think of this setting as an agent that performs one of the predefined actions in particular state and receiving the information about any change of this state, together with some *reward* information.

We have already mentioned TD-Gammon, which is one of the first attempts in combining RL and NN, but in 2013 Deep Q Network (DQN) [Mnih et al., 2015] amazes the world. This model is able to learn how to play Atari-2600 games [Bellemare et al., 2013] from raw image inputs. Moreover, in many of

these games it surpasses the human players. It is a remarkable and unexpected achievement as the DQN fulfills all assumptions of *the deadly triad* (function approximation, bootstrapping and off-policy training) described in [Sutton and Barto, 2018]. For these assumptions we have a *Baird's* counterexample, which shows the possible divergence of the training.

This unhoped-for success raises a wave of new interest in RL and only three years later in 2016 we watch a Go match between AlphaGo [Silver et al., 2016] and 9-dan professional, Lee Sedol. History repeats itself. As well as in the match between Garry and DeepBlue, people favour the human champion and as in the previous match the AI unequivocally wins. With publishing of its successor AlphaZero [Silver et al., 2018] which is able to defeat the AlphaGo with no additional human knowledge, there is no doubt that people are surpassed by AI in board games such as Chess and Go.

Strengthened by these previous successes in Single-Agent RL (SARL), the recent publications discuss the possible generalization to the Multi-Agent RL (MARL), where the environment hosts multiple agents. There are many scenarios in which the agents interact with each other. In fully cooperative settings all agents share one goal and usually they must learn how to collaborate in order to accomplish the task. A nice example of this scenario can be self-driven cars, when each car needs to cooperate in order to avoid traffic accidents. On the other hand, in adversary settings we have two or more groups of agents which oppose each other. Real world analogy is stock exchange agents who buy and sell goods and each agent wants to maximize her own profit. By introduction of MARL we yet again face new problems and challenges. The main issue being the environment becoming non-stationary from the agent point of view.

In every young scientific discipline, the notation varies from publication to publication. Also the theoretical background is often omitted in the interests of brevity. These facts make the reading of them extremely difficult. Therefore, we dedicate the whole Chapter 1 to theoretical formalization of both SARL and MARL. We come up with a unified notation, which, in our opinion, has been the best combination of proposed notations so far. In Chapter 2 we give a brief overview of the most influential algorithms for SARL. We focus on SOTA algorithms based on Actor-Critic architecture. We start the Chapter 3 by discussing how we can directly use these algorithms in MARL. Furthermore, we describe the Centralized Training with Decentralized Execution (CT-DE) approach, along with its successful implementations. In Chapter 4 we propose a new model architecture called MATD3-FORK, which is a combination of MATD3 and TD3-FORK. The whole Chapter 5 is allocated to a review of a wide range of RL frameworks. There we deal with the possible frameworks with algorithms as well as with environments. And finally, in Chapter 6 we present thorough comparative experiments with algorithms and environments discussed in previous chapters.

1. Notation and Formulation

In this chapter we formalize the standard RL problem and present the notation which we use in the rest of this work. The chapter is split into two main sections. In Section 1.1 we give the fundamentals for Single-Agent RL (SARL). We mainly follow the first part of [Sutton and Barto, 2018], however, we obey the notation introduced by [Achiam, 2018] as it is easier to extend for Multi-Agent RL (MARL) setting, which is presented in Section 1.2.

1.1 Single-Agent RL Formulation

The SARL tries to capture the relation between the *environment* and the *agent*. We can imagine the environment as a world which can be fully described by its current *state*. The agent interacts with this world by executing one of the predefined actions. In the simplest case the agent is aware of the environment state so she can choose the action with respect to this knowledge. Based on the current state of the environment and the chosen action the world changes its state. Flowingly, the agent receives the information about the new state together with a *reward*, which signals whether it was a wise decision. The agent's ultimate goal is to maximize a cumulative reward of these feedbacks. In the following subsections we transform these thoughts into proper mathematical definitions.

1.1.1 Models

As a foundation stone for the further description we use Markov Decision Process as presented in [Boutilier, 1996]. All other models are in some sense only extensions of this model.

Definition 1.1 (MDP)**.** A Markov Decision Process (MDP) is a 6-tuple $\langle \mathcal{S}, \mathcal{A}, R, P, \rho_0, \gamma \rangle$, where[1,2]

- $\mathcal{S}$ is the set of all possible *states*,
- $\mathcal{A}$ is the set of all possible *actions*,
- $R\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function*, with $r_t = R(s_t, a_t, s_{t+1})$ being reward obtained by executing action $a_t \in \mathcal{A}$ in state $s_t \in \mathcal{S}$ which leads to state $s_{t+1} \in \mathcal{S}$,
- $P\colon \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ is the *transition model*, with $P(s_{t+1} \mid s_t, a_t)$ being the probability of transitioning into state $s_{t+1} \in \mathcal{S}$ from state $s_t \in \mathcal{S}$ by taking action $a_t \in \mathcal{A}$,
- ρ_0 is the *initial state distribution*,
- $\gamma \in [0, 1]$ is a *discount factor*.

[1]Note that in our definition we assume that it is possible to execute every single action from set $\mathcal{A}$ in every state. In situation when it does not hold, we would need to generalize the definition by introducing multiple sets $\mathcal{A}^s$ consisting of actions which can be executed in state $s \in \mathcal{S}$.

[2]In many publications the reward function is further simplified to the form $R\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $r_t = R(s_t, a_t)$. Or even just to the form $R\colon \mathcal{S} \to \mathbb{R}$, where $r_t = R(s_t)$.

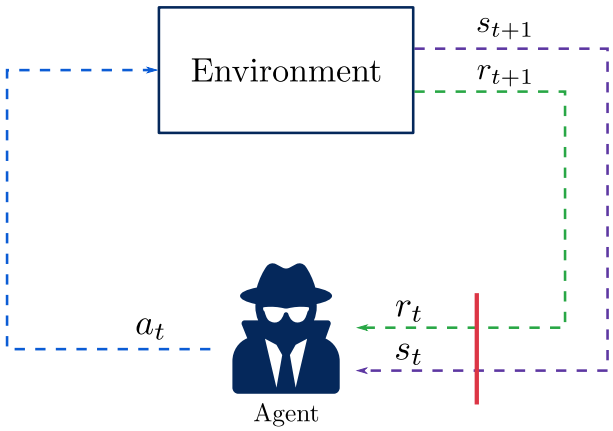


Figure 1.1: Markov Decision Process (MDP)

The word *Markov* in the process name refers to the *Markov property* known from Markov chain theory, which says that the next state depends only on the current one and nothing else. In our case the next state depends only on the pair — current state and action.

In this work we assume that we do not have access to the transition model P, unless we say otherwise. This kind of RL is known as *model-free*, whereas the opposite assumption is called *model-based*.

In some situations it can be better to presuppose that the agent is not fully aware of the environment state. The typical real world example is a robot which obtains the information through its sensors. This *observation* contains typically far less information compared to the full description of the world. Rather, we think of it as only being based on this state. In the next definition we formalize this extension into the Partially Observable Markov Decision Process as presented in [Pineau et al., 2006].

Definition 1.2 (POMDP)**.** A Partially Observable Markov Decision Process (POMDP) is an 8-tuple $\langle \mathcal{S}, \mathcal{A}, \Omega, R, P, O, \rho_0, \gamma \rangle$, where

- $\mathcal{S}$ is the set of all possible *states*,
- $\mathcal{A}$ is the set of all possible *actions*,
- Ω is the set of all possible *observations*,
- $R\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function*, with $r_t = R(s_t, a_t, s_{t+1})$ being reward obtained by executing action $a_t \in \mathcal{A}$ in state $s_t \in \mathcal{S}$ which leads to state $s_{t+1} \in \mathcal{S}$,
- $O\colon \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\Omega)$ is the *observation model*, with $O(\omega_{t+1} \mid a_t, s_{t+1})$ being the probability of observing observation $\omega_{t+1} \in \Omega$ after action $a_t \in \mathcal{A}$ which leads to state $s_{t+1} \in \mathcal{S}$,
- $P\colon \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ is the *transition model*, with $P(s_{t+1} \mid s_t, a_t)$ being the probability of transitioning into state $s_{t+1} \in \mathcal{S}$ from state $s_t \in \mathcal{S}$ by taking action $a_t \in \mathcal{A}$,
- ρ_0 is the *initial state distribution*,
- $\gamma \in [0, 1]$ is a *discount factor*.

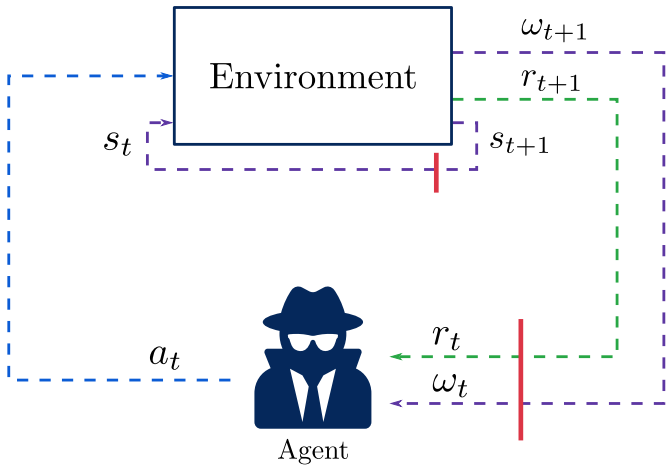


Figure 1.2: Partially Observable Markov Decision Process (POMDP)

In the following sections we take MDP as our reference model, so all other definitions are formulated with respect to it. However, the generalization to POMDP is straightforward as we mostly only substitute the state $s \in \mathcal{S}$ by the observation $\omega \in \Omega$.

1.1.2 Policy

Let's remember the classical planning problem, where actions have *deterministic* results. This means that by choosing the action we also choose the subsequent state, in which the environment will be after performing this action. The desired goal in this case is to come up with the *sequence of actions* which the agent should take.

Contrary to the classical planning problem, the MDP has *stochastic* transition model. For each action there is some uncertainty where it will lead us. Therefore, the desired goal in this case is mapping from the current state to the action that agent shall perform. This mapping is known as *policy* and we think of it as an agent's strategy.

For better distinction we use a different notation for deterministic and stochastic policies, namely[1]

$$\mu \colon S \to A \qquad a_t = \mu(s_t), \tag{1.1}$$
$$\pi \colon S \to \mathcal{P}(A) \qquad a_t \sim \pi(\cdot \mid s_t). \tag{1.2}$$

We often work with functions which are parameterized by the set of parameters θ. To highlight this fact we use the subscript notation:

$$a_t = \mu_\theta(s_t)$$
$$a_t \sim \pi_\theta(\cdot \mid s_t)$$

[1]The notation $x \sim \mathcal{N}(\mu, \sigma^2)$ means that x is a sample from probability distribution on the right hand side (in this case normal distribution with mean μ and variance σ).

1.1.3 The SARL Problem

In order to formalize the agent goal, we need to first define the term *trajectory* also known as *episode*.

Definition 1.3 (Trajectory)**.** A trajectory τ is a sequence of states and actions

$$\tau \stackrel{\text{def}}{=} (s_0, a_0, s_1, a_1, s_2, \ldots),$$

where

$$\begin{aligned} & & s_0 &\sim \rho_0(\cdot) \\ \forall t\colon \in \mathbb{N} & & a_t &\sim \pi(\cdot \mid s_t) \\ \forall t\colon \in \mathbb{N} & & s_{t+1} &\sim P(\cdot \mid s_t, a_t) \end{aligned}$$

By the *length of trajectory* we mean the maximal index value associated with action, denoted by $|\tau|$. The set of all possible trajectories for given environment is denoted by $\mathcal{T}$. With $\tau_{m:n}$ we indicate the *truncated trajectory* starting in timestep m and ending in n *e.g.,*[1]

$$\tau_{m:n} \stackrel{\text{def}}{=} (s_m, a_m, s_{m+1}, a_{m+1}, \ldots, a_{n-1}, s_n).$$

Now, when we know what the trajectory is, we can define *cumulative reward* obtained by the agent through the trajectory. This is also known as *return*.

Definition 1.4 (Return)**.** Let τ be a trajectory. We define *return* as

$$R(\tau) \stackrel{\text{def}}{=} \sum_{k=0}^{|\tau|} \gamma^k r_k.$$

When we are dealing with *truncated trajectory* we define *reward-to-go* analogously.

Definition 1.5 (Reward-to-go)**.** Let τ be a trajectory and $t \in [|\tau|]$.[2] We define *reward-to-go* as

$$R_t(\tau) \stackrel{\text{def}}{=} R(\tau_{t:}) = \sum_{k=0}^{|\tau_{t:}|} \gamma^k r_{t+k}.$$

Usually, the literature tells the difference between *finite-horizon* trajectory, where its length is upper-bounded by some constant $H \in \mathbb{N}$, and *infinite-horizon* trajectory, where it is infinite. Previous definitions are applicable to both of these cases, only in infinite case we demand $\gamma < 1$, so the infinite sum converges.

Agent's goal is to come up with a policy which would maximize agent's *return*. However, the environment is not deterministic and so we need to take into account the stochasticity of the transition model. This leads us to the definition of *expected return* which the agent wants to maximize.

[1] Similar to Python syntax, we often denote $\tau_{0:n}$ by $\tau_{:n}$ (respectively $\tau_{m:|\tau|}$ by $\tau_{m:}$).

[2] By $[n]$, where $n \in \mathbb{N}$, we mean a set of natural numbers $\{1, 2, \ldots, n\}$.

Definition 1.6 (Expected Return)**.** Let τ be a trajectory and π a policy. We define *expected return* as

$$J(\pi) \stackrel{\text{def}}{=} \int_{\tau \in \mathcal{T}} P(\tau \mid \pi) R(\tau) \, \mathrm{d}\tau = \mathop{\mathbb{E}}_{\tau \sim \pi} [R(\tau)] \,,$$

where

$$P(\tau \mid \pi) \stackrel{\text{def}}{=} \rho_0(s_0) \prod_{t=0}^{|\tau|} P(s_{t+1} \mid s_t, a_t) \pi(a_t \mid s_t)$$

is a probability of the trajectory τ.

The central optimization problem in RL is to find the *optimal policy*.

Definition 1.7 (Optimal Policy)**.** Let $J(\cdot)$ is an expected return. We define *optimal policy* as

$$\pi^* = \arg\max_{\pi} J(\pi).$$

1.1.4 Value Functions

Assume that we already have a policy π and we would like to know whether it is a good idea to follow it. For this purpose it is useful to define *state-value* and *action-value* functions.

Definition 1.8 (State-Value Function)**.** Let π be a policy and $s \in \mathcal{S}$ is arbitrary state. We define *state-value function* as[1]

$$V_\pi(s) \stackrel{\text{def}}{=} \mathop{\mathbb{E}}_{\tau \sim \pi} [R(\tau) \mid s_0 = s] \,. \tag{1.3}$$

Optimal state-value function, which returns expected return for optimal policy, is defined as

$$V_{\pi^*}(s) \stackrel{\text{def}}{=} \max_{\pi} V_\pi(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi} [R(\tau) \mid s_0 = s] \,.$$

Definition 1.9 (Action-Value Function)**.** Let π be a policy, $s \in \mathcal{S}$ a state and $a \in \mathcal{A}$ an action. We define *action-value function* as

$$Q_\pi(s, a) \stackrel{\text{def}}{=} \mathop{\mathbb{E}}_{\tau \sim \pi} [R(\tau) \mid s_0 = s, a_0 = a] \,. \tag{1.4}$$

Optimal action-value function, which returns expected return for optimal policy, is defined as

$$Q_{\pi^*}(s, a) \stackrel{\text{def}}{=} \max_{\pi} Q_\pi(s, a) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi} [R(\tau) \mid s_0 = s, a_0 = a] \,.$$

Lastly, in many algorithms it is beneficial to reflect how much better it is to perform a particular action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$, rather than absolute value given by $Q_\pi(s, a)$. For this purpose we define *advantage function*.

Definition 1.10 (Advantage Function)**.** Let $s \in \mathcal{S}$ be a state, $a \in \mathcal{A}$ an action and π a policy. We define *advantage function* as

$$A_\pi(s, a) \stackrel{\text{def}}{=} Q_\pi(s, a) - V_\pi(s).$$

[1] Strictly speaking this definition is valid only for *infinite-horizon return*. For *finite-horizon return* it matters whether we are visiting the state at the beginning of the episode or at the end of it. In this case we can simply add the time information to the state, alternatively we can extend this definition so the value function accepts time as an argument $V_\pi(s, t)$.

Bellman Equations

There is an intimate relation between state-value and action-value functions. This relation can be immediately derived from the definitions. In particular, we can express the first named with the second named and vice versa. Furthermore, we can expand the right hand side and get recursive equations known as *Bellman equations* [Bellman, 1957].

Theorem 1.1 (Bellman Equations)**.** *Let π be a policy, $s \in \mathcal{S}$ a state and $a \in \mathcal{A}$ an action. Then it holds*

$$\begin{aligned} V_\pi(s) &= \mathop{\mathbb{E}}_{a\sim\pi(s)} [Q_\pi(s,a)] \\ &= \mathop{\mathbb{E}}_{a\sim\pi(s)} \left[\mathop{\mathbb{E}}_{s'\sim P(\cdot \mid s,a)} [R(s,a,s') + \gamma V_\pi(s')] \right] \qquad (1.5) \\ Q_\pi(s,a) &= \mathop{\mathbb{E}}_{s'\sim P(\cdot \mid s,a)} [R(s,a,s') + \gamma V_\pi(s')] \\ &= \mathop{\mathbb{E}}_{s'\sim P(\cdot \mid s,a)} \left[R(s,a,s') + \gamma \mathop{\mathbb{E}}_{a'\sim\pi(s')} [Q_\pi(s',a')] \right] \qquad (1.6) \end{aligned}$$

Often, we simplify the notation for $s' \sim P(\cdot \mid s,a)$ to $s' \sim P$ and $a \sim \pi(s)$ to $a \sim \pi$.[1]

For the optimal value functions there is a special form of these equations.

Theorem 1.2 (Bellman Optimality Equations)**.** *Let $s \in \mathcal{S}$ be a state and $a \in \mathcal{A}$ an action. Then it holds*

$$V_{\pi^*}(s) = \max_a \mathop{\mathbb{E}}_{s'\sim P} [R(s,a,s') + \gamma V_{\pi^*}(s')] \qquad (1.7)$$

$$Q_{\pi^*}(s,a) = \mathop{\mathbb{E}}_{s'\sim P} \left[R(s,a,s') + \gamma \max_{a'} Q_{\pi^*}(s',a') \right] \qquad (1.8)$$

These relations are important theoretical results which stand in the background of many algorithms. In a situation when we have access to the transition model P of the MDP and the sets $\mathcal{S}, \mathcal{A}$ are finite, we can use algorithms like *Policy Iteration* or *Value Iteration*. For these algorithms we have proofs that the returned policy is the optimal one.

We can use the optimal value functions and formulate the optimal policy with them.

Theorem 1.3. *There always exists a unique optimal state-value function V_{π^*}, a unique optimal action-value function Q_{π^*} and a (not necessarily unique) deterministic optimal policy π. Furthermore, we can express it element-wise as*

$$\begin{aligned} \pi^*(s) &= \arg\max_a Q_{\pi^*}(s,a) \qquad (1.9) \\ &= \arg\max_a \mathop{\mathbb{E}}_{s'\sim P} [R(s,a,s') + \gamma V_{\pi^*}(s')]. \end{aligned}$$

Proof. Proof can be found in [Sutton and Barto, 2018]. □

[1] Let's notice that the second simplification brings the ambiguity as we have already used a similar notation $\tau \sim \pi$ in Definition 1.6. The first usage means that we are sampling the trajectory with respect to the policy π (and implicitly also with respect to transition model). The second usage means sampling single action from our policy π in a specific state.

	Fully Observable	Partially Observable
Shared Reward	MMDP	dec-POMDP
Individual Reward	SG	POSG

Table 1.1: Models for MARL

1.2 Multi-Agent RL Formulation

In many situations we face the problem where the environment contains multiple agents. These agents are typically completely independent and autonomous. It can be also the case that each agent performs a different set of actions and moreover, has a different goal as the reward feedback can vary from agent to agent. Analogous to the previous section, we first introduce suitable models for MARL. Later, we extend the notation known from the previous section. We draw knowledge mainly from [Terry et al., 2020b], [OroojlooyJadid and Hajinezhad, 2019] and [Foerster, 2018].

1.2.1 Models

The straightforward extension of MDP is Multi-Agent Markov Decision Process (MMDP) [Boutilier, 1996], where the agents perform actions in parallel and receive a single reward feedback. In case that we are working with partial observability, we can in the same way extend POMDP to the Decentralized Partially Observable Markov Decision Process (dec-POMDP) [Bernstein et al., 2002]. The drawback of these models is the impossibility to have dissimilar goals for each agent. This limitation is overcome by Stochastic Game (SG) [Shapley, 1953], which introduces the individual reward feedback for each agent. Yet again, we can incorporate the partial observability, so we get Partially Observable Stochastic Game (POSG) [Hansen et al., 2004].[1] To improve clarity we add the summary Table 1.1.

Definition 1.11 (POSG)**.** A Partially Observable Stochastic Game (POSG) is a 9-tuple $\left\langle \mathcal{S}, N, \{\mathcal{A}^{i\in[N]}\}, \{\Omega^{i\in[N]}\}, \{R^{i\in[N]}\}, P, \{O^{i\in[N]}\}, \rho_0, \gamma \right\rangle$, where[2]

- $\mathcal{S}$ is the set of all possible *states*,
- N is the *number of agents*,
- $\mathcal{A}^i$ is the set of all possible *actions* for agent i. We denote Cartesian product of these by $\mathcal{A}^{\Pi} \stackrel{\text{def}}{=} \prod_{i\in[N]} \mathcal{A}^i$.
- Ω^i is the set of all possible *observations* for agent i,
- $R^i \colon \mathcal{S} \times \mathcal{A}^{\Pi} \times \mathcal{S} \to \mathbb{R}$ is the *reward function* for agent i, with $r_t^i = R^i(s_t, \boldsymbol{a}_t, s_{t+1})$ being reward obtained by executing actions $\boldsymbol{a}_t \in \mathcal{A}^{\Pi}$ in state $s_t \in \mathcal{S}$ which leads to state $s_{t+1} \in \mathcal{S}$,

[1] As with MDP, we can further extend this model and specify which actions are possible to perform in each state.

[2] As highlighted in [Terry et al., 2020b], the observation model O^i for a single agent i does not directly depend on the actions of other agents. However, this dependency is implicitly given by the new state $s_{t+1} \in \mathcal{S}$, which is already dependent on all actions $\boldsymbol{a} \in \mathcal{A}^{\Pi}$.

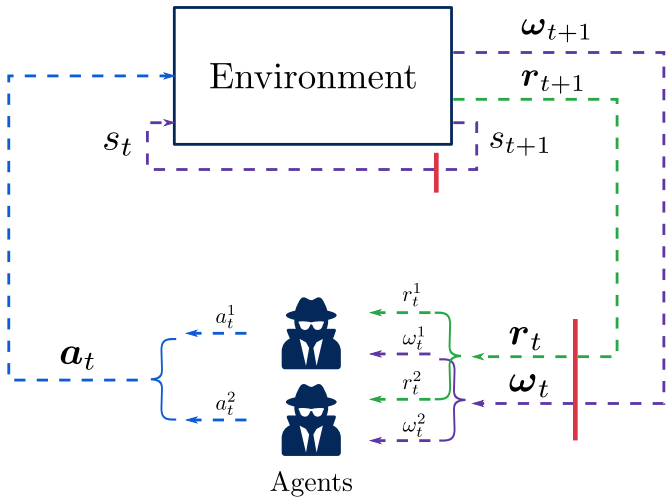


Figure 1.3: Partially Observable Stochastic Game (POSG)

- $O^i \colon \mathcal{S} \times \mathcal{A}^i \to \mathcal{P}(\Omega^i)$ is the *observation model* for agent i, furthermore $O^i(\omega_{t+1}^i \mid a_t^i, s_{t+1})$ is the probability of observing observation $\omega_{t+1}^i \in \Omega^i$ after action $a_t^i \in \mathcal{A}^i$ which leads to state $s_{t+1} \in \mathcal{S}$,
- $P \colon \mathcal{S} \times \mathcal{A}^\Pi \to \mathcal{P}(\mathcal{S})$ is the *transition model*, with $P(s_{t+1} \mid s_t, \boldsymbol{a}_t)$ being the probability of transitioning into state $s_{t+1} \in \mathcal{S}$ from state $s_t \in \mathcal{S}$ by taking actions $\boldsymbol{a}_t \in \mathcal{A}^\Pi$,
- ρ_0 is the *initial state distribution*,
- $\gamma \in [0, 1]$ is a *discount factor*.

In the previous definition we have used some not yet explained notations. In order to denote joint-(action, policy, observation)[1] across all agents we use a bold math case *e.g.,*

$$\boldsymbol{a} \stackrel{\text{def}}{=} \{a^1, \dots, a^N\} \in \mathcal{A}^\Pi.$$

It is also handy to use $\boldsymbol{a}^{-i}$ to signify a joint-action except the action for agent i, so

$$\boldsymbol{a}^{-i} \stackrel{\text{def}}{=} \{a^1, \dots, a^{i-1}, a^{i+1}, \dots a^N\} \in \prod_{i \in [N] \setminus \{i\}} \mathcal{A}^i.$$

We use the same notation also for joint-policy $\boldsymbol{\pi}$ and joint-observation $\boldsymbol{\omega}$.

In a special case, where

$$\forall i, j \in [N] : A^i = A^j \wedge \Omega^i = \Omega^j,$$

we say that agents are *homogeneous*. Otherwise, we call them *heterogeneous*.

The POSG is strong enough to meet all our requirements, so from the theoretic point of view we could stop here. Nevertheless, it can be quite challenging to implement an environment correctly, as one needs to be careful with the parallel action execution. Because of that potential issue, the Agent-Environment Cycle Game (AECG) is proposed in [Terry et al., 2020b].

[1]When we work with these models in theory there is no need to strictly specify the order and therefore we represent joint-(action, policy, observation) as ordinary sets. On the other hand, we need to be at most careful in specifying the order during the implementation.

Definition 1.12 (AECG)**.** A Agent-Environment Cycle Game (AECG) is a 12-tuple $\left\langle \mathcal{S}, N, \{\mathcal{A}^{i\in[N]}\}, \{\Omega^{i\in[N]}\}, \{R^{i\in[N]}\}, \{T^{i\in[N]}\}, P, \{O^{i\in[N]}\}, V, s_0, i_0, \gamma \right\rangle$, where

- $\mathcal{S}$ is the set of all possible *states*,
- N is the *number of agents*. There is an additional "environment" agent with index 0, denote $[N^\cup] \stackrel{\text{def}}{=} [N] \cup \{0\}$.
- $\mathcal{A}^i$ is the set of all possible *actions* for agent i. We set $\mathcal{A}^0 \stackrel{\text{def}}{=} \{\varnothing\}$ and denote $\mathcal{A}^\cup \stackrel{\text{def}}{=} \bigcup_{i\in[N^\cup]} \mathcal{A}^i$.
- Ω^i is the set of all possible *observations* for agent i,
- $R^i \colon \mathcal{S} \times N^\cup \times \mathcal{A}^\cup \times \mathcal{S} \to \mathcal{P}(\mathcal{R}^i)$ is the *reward function* for agent i. Let $\mathcal{R}^i \subsetneq \mathbb{R}$ denote a *finite* subset of all possible rewards for agent i. By $R^i(r_t \mid s_t, j, a_t^j, s_{t+1})$ we mean the probability of agent i obtaining reward $r_t \in \mathcal{R}^i$, after agent j executes action $a_t^j \in \mathcal{A}^j$ in state $s_t \in \mathcal{S}$ which leads to state $s_{t+1} \in \mathcal{S}$.
- $T^i \colon \mathcal{S} \times \mathcal{A}^i \to \mathcal{S}$ is the deterministic *transition function for agent* i,
- $P \colon \mathcal{S} \to \mathcal{P}(\mathcal{S})$ is the *transition function for the environment*, with $P(s' \mid s)$ being the probability of transitioning into state $s' \in \mathcal{S}$ from state $s \in \mathcal{S}$.
- $O^i \colon \mathcal{S} \to \mathcal{P}(\Omega^i)$ is the *observation model* for agent i, with $O^i(\omega^i \mid s)$ being the probability of observing observation $\omega^i \in \Omega^i$ while in state $s \in \mathcal{S}$,
- $V \colon \mathcal{S} \times N^\cup \times \mathcal{A}^\cup \to \mathcal{P}(N^\cup)$ is the *next agent function*, with $V(j \mid s, i, a^i)$ the probability that agent j will be the next agent permitted to act given that agent i has just taken action $a^i \in \mathcal{A}^i$ in current state $s \in \mathcal{S}$.
- $s_0 \in \mathcal{S}$ is the *initial state*,
- $i_0 \in [N^\cup]$ is the *initial agent* to act,
- $\gamma \in [0, 1]$ is a *discount factor*.

In order to explain how this particular model works we quote here[1] the original publication [Terry et al., 2020b, p.3-4]:

> The game then evolves in "turns" where in each turn the game starts in some state $s \in \mathcal{S}$ with agent $i \in N^\cup$ to act; agent i receives an observation $\omega^i \in \Omega^i$ and then chooses an action $a^i \in \mathcal{A}^i$, and the game transitions into a new state $s' \in \mathcal{S}$; then, a new agent $i' \in N^\cup$ is determined who will be the one to act in the next turn. The observation ω^i that is received is random, occurring with probability $O^i(\omega^i \mid s)$. The new state s' is determined by one of the transition functions:
>
> - if $i = 0$, then the current turn is an "environment step" and the next state s' is random, occurring with probability $P(s' \mid s)$;
> - if $i > 0$, then the new state is deterministically $s' = T^i(s, a^i)$.
>
> The next agent i' is determined randomly by the next-agent function, with i' being chosen with probability $V(i' \mid s, i, a^i)$. Finally, at every turn, every agent receives some (possibly negative) reward. In the preceding example, every agent j will receive a random reward r' , with probability

[1] With our modified notation.

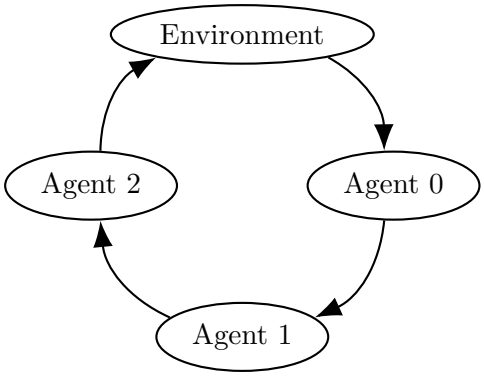


Figure 1.4: AEC diagram for MPE Simple Spread environment

$R^j(r' \mid s, i, a^i, s')$. In order for the next-agent and reward functions to be well defined in environment steps, we use the special action $\varnothing$. For example, agent i' will be next to act after an environment step in state s with probability $V(s, 0, \varnothing, i')$.

It is very important that the Theorem 1.4 holds, as the POSG is easier to work with from the theoretical point of view. On the other hand, when we want to examine a proposed algorithm on a new environment it is more comfortable to code it in AECG model.

Theorem 1.4. *For every POSG, there is an equivalent AECG and vice versa.*

Proof. Proof can be found in [Terry et al., 2020b].

□

In many environments the order of the agents turns is predefined and fixed, hence we can sketch out the *AEC Diagram* as you can see in Figure 1.4.

1.2.2 Game Settings

With the different choice of *reward functions* we get diverse game settings. As the simplest example we can choose[1,2]

$$\forall i, j \in [N]: \qquad R^i(s_t, \boldsymbol{a}_t, s_{t+1}) = R^j(s_t, \boldsymbol{a}_t, s_{t+1}).$$

We call this setting *fully cooperative* as all agents share their interests.

Another option, well known from game theory, is *zero-sum* game where

$$\sum_{i \in [N]} R^i(s_t, \boldsymbol{a}_t, s_{t+1}) = 0.$$

In such a setting in order to increase one agent's gain, the others must be penalized.

The most general setting is named *general-sum* as it covers both previous settings. As the name implicates, we are not limited to fulfill any restrictions.

[1]Note that this is actually a step back from SG to MMDP.

[2]We omit the quantifiers for states $\forall s_t, s_{t+1} \in \mathcal{S}$ and actions $\forall \boldsymbol{a} \in \mathcal{A}^{\Pi}$.

By *cooperative* setting we mean a general-sum game where the agents need to coordinate together in order to meet the requirements. Standard implementations split the reward function into two parts: *global reward* shared among all agents; *individual reward* associated only with the single agent. Be careful that every agent seeks to maximize her own reward and so the environment can also introduce hostile mind. Formally, we say

$$\forall i, j \in [N], i \neq j, \exists k \in \mathbb{R}, k > 0\colon \qquad R^i(s_t, \boldsymbol{a}_t, s_{t+1}) = kR^j(s_t, \boldsymbol{a}_t, s_{t+1}).$$

Finally, we use the term *competitive* setting as a reference to the general-sum game, where agents are split into two or more teams. Within these teams we can find (fully) cooperative settings. Frequently, there is a rivalry among these teams. Formally, we say

$$\exists i, j \in [N], i \neq j, \exists k \in \mathbb{R}, k < 0\colon \qquad R^i(s_t, \boldsymbol{a}_t, s_{t+1}) = kR^j(s_t, \boldsymbol{a}_t, s_{t+1}).$$

1.2.3 The MARL Problem

Analogous to Section 1.1 we generalize the definitions of *trajectory*, *return* and *expected return* by incorporating multiple agents. As before, we use fully observable SG as our ground model.

Definition 1.13 (Trajectory)**.** A trajectory $\boldsymbol{\tau}$ is a sequence of states and actions

$$\boldsymbol{\tau} \stackrel{\text{def}}{=} (s_0, \boldsymbol{a}_0, s_1, \boldsymbol{a}_1, s_2, \ldots),$$

where

$$\begin{aligned} & s_0 \sim \rho_0(\cdot) \\ \forall t\colon \in \mathbb{N} \qquad & \boldsymbol{a}_t \sim \boldsymbol{\pi}(\cdot \mid s_t) \\ \forall t\colon \in \mathbb{N} \qquad & s_{t+1} \sim P(\cdot \mid s_t, \boldsymbol{a}_t) \end{aligned}$$

By the *length of trajectory* we mean the maximal index value associated with action, denoted by $|\boldsymbol{\tau}|$. The set of all possible trajectories for given environment is denoted by $\mathcal{T}$. With $\boldsymbol{\tau}_{m:n}$ we indicate the *truncated trajectory* starting in timestep m and ending in n *e.g.,*

$$\boldsymbol{\tau}_{m:n} \stackrel{\text{def}}{=} (s_m, \boldsymbol{a}_m, s_{m+1}, \boldsymbol{a}_{m+1}, \ldots, \boldsymbol{a}_{n-1}, s_n).$$

Definition 1.14 (Return)**.** Let $\boldsymbol{\tau}$ be a trajectory and $i \in [N]$ an agent. We define *return for agent* i as

$$R^i(\boldsymbol{\tau}) \stackrel{\text{def}}{=} \sum_{k=0}^{|\boldsymbol{\tau}|} \gamma^k r_k^i.$$

Definition 1.15 (Reward-to-go)**.** Let $\boldsymbol{\tau}$ be a trajectory, $t \in [|\boldsymbol{\tau}|]$ and $i \in [N]$ an agent. We define *reward-to-go* as

$$R_t^i(\boldsymbol{\tau}) \stackrel{\text{def}}{=} R^i(\boldsymbol{\tau}_{t:}) = \sum_{k=0}^{|\boldsymbol{\tau}_{t:}|} \gamma^k r_{t+k}^i.$$

Definition 1.16 (Expected Return)**.** Let $\boldsymbol{\tau}$ be a trajectory, $\boldsymbol{\pi}$ a policy and $i \in [N]$ an agent. We define *expected return for agent* i as

$$J^i(\boldsymbol{\pi}) \stackrel{\text{def}}{=} \int_{\boldsymbol{\tau} \in \boldsymbol{\mathcal{T}}} P(\boldsymbol{\tau} \mid \boldsymbol{\pi}) R^i(\boldsymbol{\tau}) \, \mathrm{d}\boldsymbol{\tau} = \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \boldsymbol{\pi}} \left[R^i(\boldsymbol{\tau}) \right],$$

where

$$P(\boldsymbol{\tau} \mid \boldsymbol{\pi}) \stackrel{\text{def}}{=} \rho_0(s_0) \prod_{t=0}^{|\boldsymbol{\tau}|} P(s_{t+1} \mid s_t, \boldsymbol{a}_t) \boldsymbol{\pi}(\boldsymbol{a}_t \mid s_t)$$

is a probability of the trajectory $\boldsymbol{\tau}$.

Nash Equilibrium

Contrary to SARL, where we formalize the central optimization problem with the Definition 1.7, it is unclear what behavior we are looking for in MARL. Indeed, we face a multi-criteria optimization problem, where it is not possible to easily combine each criterion. To see this, let's remember *zero-sum* games where sum of all rewards is alway zero. Therefore, one of the most important concepts from the game theory is *Nash-equilibrium* [Foerster, 2018].

Definition 1.17 (Nash Equilibrium)**.** We say that joint-policy $\boldsymbol{\pi}^*$ is a *Nash-equilibrium* if and only if[1]

$$\forall i \in [N], \forall \pi^i: \qquad J^i(\boldsymbol{\pi}^*) \geq J^i(\pi^i, \boldsymbol{\pi}^{*-i}).$$

Unfortunately, an environment can possibly contain multiple *Nash-equilibrium* policies, moreover, each of them may have a different payoff. This brings an option to get stuck in a local optima as there is no environment pressure to change agent behavior. Even if we force to have only equilibria with identical payoffs, finding the solution can be problematic, inasmuch each agent can head to a different Nash strategy.

Finally, it is useful to define *best response* concept.

Definition 1.18 (Best Response)**.** Let $i \in [N]$ be an agent and $\boldsymbol{\pi}^{-i}$ a joint-policy. We define *best response* as

$$\hat{\pi}^i(\boldsymbol{\pi}^{-i}) \stackrel{\text{def}}{=} \arg\max_{\pi^i} J^i(\pi^i, \boldsymbol{\pi}^{-i}).$$

1.2.4 Value Functions

For the sake of completness we introduce a generalized version of value functions as well.

Definition 1.19 (State-Value Function)**.** Let $\boldsymbol{\pi}$ be a policy, $s \in \mathcal{S}$ arbitrary state and $i \in [N]$ an agent. We define *state-value function for agent* i as

$$V^i_{\boldsymbol{\pi}}(s) \stackrel{\text{def}}{=} \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \boldsymbol{\pi}} \left[R(\boldsymbol{\tau}) \mid s_0 = s \right].$$

[1] Strictly speaking we should write $J^i(\boldsymbol{\pi}^*) \geq J^i(\{\pi^i, \boldsymbol{\pi}^{*-i}\})$, but yet again we simplify the notation as there is no space for misunderstanding.

Definition 1.20 (Action-Value Function)**.** Let $\boldsymbol{\pi}$ be a policy, $s \in \mathcal{S}$ arbitrary state, $\boldsymbol{a} \in \mathcal{A}^{\Pi}$ actions and $i \in [N]$ an agent. We define *action-value function for agent* i as

$$Q^i_{\boldsymbol{\pi}}(s, \boldsymbol{a}) \stackrel{\text{def}}{=} \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \boldsymbol{\pi}} \left[R(\boldsymbol{\tau}) \mid s_0 = s, \boldsymbol{a}_0 = \boldsymbol{a} \right].$$

Definition 1.21 (Advantage Function)**.** Let $s \in \mathcal{S}$ be a state, $\boldsymbol{a} \in \mathcal{A}^{\Pi}$ actions, $\boldsymbol{\pi}$ a policy and $i \in [N]$ an agent. We define *advantage function for agent* i as

$$A^i_{\boldsymbol{\pi}}(s, \boldsymbol{a}) \stackrel{\text{def}}{=} Q^i_{\boldsymbol{\pi}}(s, \boldsymbol{a}) - V^i_{\boldsymbol{\pi}}(s).$$

Note that we omit *optimal value functions* in the definitions. The reason is that an agent cannot force other agents to follow a particular policy. This is closely related to other phenomena which arise by incorporating multiple agents, namely *non-stationarity* [OroojlooyJadid and Hajinezhad, 2019]. From the perspective of any individual agent the environment is constantly changing as each agent performs training. Formally, we say[1]

$$\forall i \in [N]: \qquad \boldsymbol{\pi}^{-i} \neq \boldsymbol{\pi'}^{-i} \implies P(s' \mid s, a^i, \boldsymbol{a}^{-i}) \neq P(s' \mid s, a^i, \boldsymbol{a'}^{-i}),$$

where

$$\begin{aligned} \boldsymbol{a}^{-i} &\sim \boldsymbol{\pi}^{-i}(s) \\ \boldsymbol{a'}^{-i} &\sim \boldsymbol{\pi'}^{-i}(s). \end{aligned}$$

Note that this non-stationarity is not present in MDP. Because the $\boldsymbol{\pi}^{-i}$ changes over time as the policies of other agents change, the usage of the expanded version of *Bellman equations* [Foerster et al., 2017]

$$\begin{aligned} V^i_{\boldsymbol{\pi}}(s) &= \mathop{\mathbb{E}}_{\boldsymbol{a} \sim \boldsymbol{\pi}} \left[\mathop{\mathbb{E}}_{s' \sim P} \left[R(s, \boldsymbol{a}, s') + \gamma V^i_{\boldsymbol{\pi}}(s') \right] \right] \\ Q^i_{\boldsymbol{\pi}}(s, \boldsymbol{a}) &= \mathop{\mathbb{E}}_{s' \sim P} \left[R(s, \boldsymbol{a}, s') + \gamma \mathop{\mathbb{E}}_{\boldsymbol{a'} \sim \boldsymbol{\pi}} \left[Q^i_{\boldsymbol{\pi}}(s', \boldsymbol{a'}) \right] \right], \end{aligned}$$

is problematic even in a situation where $\mathcal{S}$ and $\{\mathcal{A}^{i \in [N]}\}$ are finite and we know the transition model P.

[1] Again we abuse the notation by using $P(s' \mid s, a^i, \boldsymbol{a}^{-i})$ rather than $P(s' \mid s, \{a^i, \boldsymbol{a}^{-i}\})$.

2. Single-Agent RL Algorithms

In this chapter we provide a brief review of the most influential algorithms for Single-Agent RL. We try to sort them through but it is quite hard to come up with a complete and exhaustive classification of RL algorithms. In many cases, an algorithm is on the border between two classes and so we cannot unambiguously place it within one or another bin. Despite this, we obey the standard taxonomy, which is well described in [Achiam, 2018].

2.1 RL Concepts and Taxonomy

This section is indented to by a brief refreshment of the key concepts in RL. As we will see, these are applicable to both SARL as well as MARL.

2.1.1 Exploration vs Exploitation

The most fundamental concept in whole RL is *exploration vs exploitation*. As we have already said, an agent wants to choose an action in order to maximize her expected return given by the Definition 1.6. But knowledge about what is good or bad is based on the interaction with the environment. So we need to reach balance between trying new things (*exploration*) and using this knowledge to make the best possible move (*exploitation*). Habitually, we are decreasing the *exploration* as the training proceeds. You can find more details on this topic in [Sutton and Barto, 2018].

2.1.2 Value Functions vs Policy Approximation

If we explicitly model a policy $\pi_\theta(\cdot|s)$ as a function with parameters θ (typically by NN), we are talking about *policy approximation*. Usually, we use a kind of Policy Gradient Theorem (PGT), which tells us how we shall modify the parameters so that we get a higher expected return.

On the other hand, we can approximate the *value function* from Definition 1.8 (respectively 1.9)[1] and, later on, derive the policy with the equation[2]

$$\pi(s) = \arg\max_a Q_\theta(s, a).$$

The drawback of this approach is that it is quite hard to generalize for environments with continuous actions. It is also a question how to estimate the target values. On the one hand, we can include information from the whole episode, in which case we are talking about Monte Carlo (MC) approach.[3] Alternatively, we can employ the Bellman equations from Theorem 1.1 and calculate Temporal Difference (TD) error. Naturally, we can continuously pass between these

[1] As has already been mentioned, we do not expect knowledge of a transition model, so we are forced to approximate *action-value function*. In cases that we have a transition model it is more convenient to approximate *state-value function*.

[2] Note that we are using π rather than μ. In case that there are two or more actions with the same value we can choose an arbitrary probability distribution among these actions.

[3] This is only possible if we are working with *finite-horizon episodic* environment.

approaches with n-step returns [Sutton and Barto, 2018] or even better with Generalized Advantage Estimation (GAE) [Schulman et al., 2016].

2.1.3 Model-free vs Model-based

We have already mentioned these terms in Chapter 1. The vast majority of recent algorithms use the *model-free* approach simply because the transition model is unknown or hard to get.

In case that we do have the transition model, such as in various board games like Chess or Go, we can use this knowledge and utilize some kind of planning. This is known as *model-based* method. A nice example is AlphaZero [Silver et al., 2018], which utilizes Monte Carlo Tree Search (MCTS) for enhancement the chosen action. If we do not have this model, we can try to learn it. One example of this technique is MuZero [Schrittwieser et al., 2020], which is a direct extension of AlphaZero.

2.1.4 Off-policy vs On-policy

By the *off-policy* setup, we mean that within each update step we can use data collected at any previous point during training. This approach is typical for value functions approximation. Contrary, the *on-policy* setup uses only data collected while acting according to the most recent version of the policy. Unsurprisingly, this is standard for policy approximation approach.

2.2 Q-learning

We cannot start our algorithm enumeration by nothing else than description of Q-learning [Watkins and Dayan, 1992]. It is a classical algorithm based on the Temporal Difference (TD) and according to our taxonomy we classify it as *value function approximation*, *model-free* and *off-policy*. The original algorithm is tabular, which means that it stores the data action-value function in an ordinary table. This implies that it can only be applicable to environments with finite $\mathcal{S}$ and $\mathcal{A}$ sets. The update of this table is performed as

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)).$$

See Algorithm 2.1 for full algorithm description.

The important phenomenon, which arises by using the previous described Q-learning, is *maximization bias*. We explain it on the borrowed example from the [Sutton and Barto, 2018]. Imagine an environment with four states A, B, C and D. The state A is the initial, whereas C, D are the terminal. The transition and reward models are captured in Figure 2.1. At first sight, the solution is to go right in the state A as the expectation return after going right is -0.1. However, it can happen that during the exploration we are "lucky" and we sample the positive reward when leaving state B. This transition is highlighted with green color. This will mislead the algorithm to suppose that there is actually a way how to achieve a positive expected return, even though it is not true. After all,

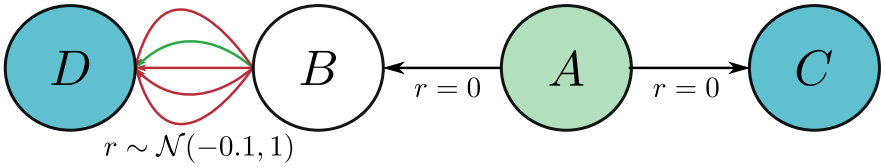


Figure 2.1: Maximization bias example

the reward estimate associated with this edge will decrease and so the correct solution will be found. But this process significantly slows down the training.

In order to overcome this problem authors of Double Q-learning [Hasselt et al., 2016] propose to train two Q-functions Q_1, Q_2. Then, the update of the first one is formulated with the usage of the second one and vice versa

$$Q_1(s,a) = Q_1(s,a) + \alpha(r + \gamma Q_2(s', \arg\max_{a'} Q_1(s',a')) - Q_1(s,a)),$$
$$Q_2(s,a) = Q_2(s,a) + \alpha(r + \gamma Q_1(s', \arg\max_{a'} Q_2(s',a')) - Q_2(s,a)).$$

Finally, when choosing the action we use policy ϵ-greedy *w.r.t.* Q_1 and Q_2. See Algorithm 2.2 for all details.

2.3 Deep Q Network

Probably, the most influential RL publication in the past decade has been [Mnih et al., 2015]. In this work, the authors propose using the NN as an action-value function approximator for the Q-learning algorithm. By itself, this combination fulfills the *deadly triad* assumptions (function approximation, bootstrapping and off-policy training) known from [Sutton and Barto, 2018]. Unfortunately, the *Baird's counterexample* published in [Williams and Baird, 1993] shows the possible training divergent in this setting. Therefore, the researchers have believed for many years that this is the barren direction.

Authors of the DQN have been aware of these issues and therefore they propose two important improvements. Firstly, the *target network* is the identical copy of the original network with frozen parameters. By the *original network* we mean the network which servers as the action-value approximator. These parameters are updated once in a while. The target network is then utilized for the estimation of the targets values during the updates of the original network. This effectively breaks the *deadly triad* and enables us the successful training.

Secondly, the authors come up with the concept of *replay buffer*. Simply speaking we exploit the *off-policy* nature of the Q-learning algorithm by saving all transitions to the buffer. The information saved there is valid not only for that very moment but also in the future. Therefore, we can learn from them multiple times. For practical reasons we often restrict the buffer size to some reasonable scale (based on the sizes of the observations and actions).

Together with Convolutional Neural Network (CNN) and additional reward clipping, this architecture is able to surpass human-level performances in the Atari-2600 games [Bellemare et al., 2013] only from the raw pixels inputs. Authors want to show the model capability to generalize among the various environments and therefore some kind of the reward modification is inevitable as the reward

Algorithm 2.1: Q-learning

Input: initial Q-function table where $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$: $Q(s,a) = 0$.

repeat

  Observe state s and select action a according to ϵ-greedy *w.r.t.* Q *e.g.,*

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon, \\ \arg\max_a Q(s,a), & \text{otherwise.} \end{cases}$$

  Execute a in the environment.

  Observe next state s', reward r and done signal d to indicate whether s' is terminal.

  if d is true **then**

    Reset environment state.

  end if

  Update Q-function

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)).$$

until convergence

Algorithm 2.2: Double Q-learning

Input: initial Q-function tables where $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$: $Q_1(s,a) = Q_2(s,a) = 0$.

repeat

  Observe state s and select action a according to ϵ-greedy *w.r.t.* Q_1 and Q_2 *e.g.,*

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon, \\ \arg\max_a Q_1(s,a) + Q_2(s,a), & \text{otherwise.} \end{cases}$$

  Execute a in the environment.

  Observe next state s', reward r and done signal d to indicate whether s' is terminal.

  if d is true **then**

    Reset environment state.

  end if

  Update Q-function

$$Q_1(s,a) = Q_1(s,a) + \alpha(r + \gamma Q_2(s', \arg\max_{a'} Q_1(s',a')) - Q_1(s,a)),$$
$$Q_2(s,a) = Q_2(s,a) + \alpha(r + \gamma Q_1(s', \arg\max_{a'} Q_2(s',a')) - Q_2(s,a)).$$

until convergence

Algorithm 2.3: Deep Q Network

Input: initial Q-function parameters ϕ, empty replay buffer $\mathcal{D}$.

Set target parameters equal to main parameters $\phi_{\text{targ}} \leftarrow \phi$.

repeat

Observe state s and select action a according to ϵ-greedy *w.r.t.* Q_ϕ *e.g.,*

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon, \\ \arg\max_a Q_\phi(s,a), & \text{otherwise.} \end{cases}$$

Execute a in the environment.

Observe next state s', reward r and done signal d to indicate whether s' is terminal.

Store (s, a, r, s', d) in replay buffer $\mathcal{D}$.

if d is true **then**

Reset environment state.

end if

Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$.

Compute targets

$$y(r, s', d) = r + \gamma(1-d) \max_{a'} Q_{\phi_{\text{targ}}}(s', a').$$

Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s,a) - y(r,s',d)\right)^2.$$

if time to update target network **then**

$$\phi_{\text{targ}} \leftarrow \phi.$$

end if

until convergence

can vary by order of magnitude from environment to environment. We provide the full Algorithm 2.3 for DQN with our experienced notation.

Since the first DQN publication in 2013, many improvements have been proposed. The most successful ones are picked and analyzed in publication [Hessel et al., 2018] and based on their combination the authors present a new architecture called *Rainbow*. The improvements are

- Double Q-learning,
- Dueling Deep Q Network,
- Prioritized Replay Buffer,
- Noise Nets,
- Distributional Reinforcement Learning,
- n-step return.

Similarly to the classical Q-learning, even the DQN suffers from the maximization bias. However, in this case we already have the second network (namely the *target network*) and so the target for the update can be formulated as

$$y(r, s', d) = r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \arg\max_{a'} Q_\phi(s', a')).$$

2.4 REINFORCE

REINFORCE [Williams, 1992] is the fundamental *policy approximation*, *model-free* and *on-policy* algorithm. Remember that we want to find parameters θ of the policy π_θ, which maximizes the expected return from Definition 1.6

$$J(\pi_\theta) \stackrel{\text{def}}{=} \mathop{\mathbb{E}}_{\tau\sim\pi}\left[R(\tau)\right].$$

In order to do that, REINFORCE employs gradient ascent *e.g.,*

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\pi_\theta)|_{\theta_k},$$

where α is a learning rate. Policy Gradient Theorem (PGT) is the key result as it tells us how to compute the $\nabla_\theta J(\pi_\theta)$. We formulate the policy π_θ so it returns the full distribution over all actions. Therefore, we can consider it as a stochastic policy and in order to preserve exploration we sample from this distribution, rather than using ϵ-greedy strategy. In environments with discrete action space we utilize *softmax* function to get the final probabilities, whereas with continuous action space we mostly use *diagonal Gaussian distribution.*

We formulate PGT in two different ways. The advantage of the first version is that it is almost identical to the final implementation. The other one is useful for its similarity with Deterministic Policy Gradient Theorem (DPGT), which we cover in the following section. It is easy to see that these two formulations are identical.

Theorem 2.1 (Policy Gradient Theorem)**.** *It holds*[1]

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &\propto \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{|\tau|}\Phi_t\nabla_\theta\log\pi_\theta(a_t\mid s_t)\right]\\ &\propto \mathop{\mathbb{E}}_{s_t\sim\eta_{\pi_\theta}}\mathop{\mathbb{E}}_{a_t\sim\pi_\theta}\left[\Phi_t\nabla_\theta\log\pi_\theta(a_t\mid s_t)\right],\end{aligned}$$

where Φ_t *could be any of*

$$\begin{aligned}\Phi_t &= R(\tau) && \textit{(return)},\\ \Phi_t &= R_t(\tau) && \textit{(reward-to-go)},\\ \Phi_t &= R_t(\tau) - b(s_t) && \textit{(return with baseline)},\\ \Phi_t &= Q_{\pi_\theta}(s_t, a_t) && \textit{(action-value function)},\\ \Phi_t &= A_{\pi_\theta}(s_t, a_t) && \textit{(advantage function)}\end{aligned}$$

and η_{π_θ} *is an on-policy distribution under policy* π_θ *e.g.,*[2]

$$\eta_{\pi_\theta}(s) = \int_{s'\in\mathcal{S}}\eta_{\pi_\theta}(s')\int_{a\in\mathcal{A}}\pi_\theta(a\mid s')P(s\mid s',a)\,\mathrm{d}a\,\mathrm{d}s'. \tag{2.1}$$

[1]The $\propto$ means *proportional to*, in other words the equality holds except for the multiplicative constant.

[2]For the episodic task the definition is more complicated as we also need to consider the initial state distribution ρ_0. More details can be found in [Sutton and Barto, 2018, p.199].

Algorithm 2.4: REINFORCE with Baseline

Input: initial policy parameters θ_0, initial V-function parameters ϕ_0.

for k in $0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy π_{θ_k} in the environment.

Compute rewards-to-go R_t for every timestep.

Compute advantage estimate A_t *w.r.t.* R_t and V_{ϕ_k} for every timestep.

Compute updated V-function parameters ϕ_{k+1} by one step of gradient descent using

$$\nabla_\phi \frac{1}{|\mathcal{D}_k|\,|\tau|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{|\tau|} (V_{\phi_k}(s_t) - R_t).$$

Compute updated policy parameters θ_{k+1} with gradient ascent algorithm in direction of policy gradient estimation

$$\theta_{k+1} \leftarrow \theta_k + \alpha_\theta g_k \qquad g_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{|\tau|} A_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)|_{\theta_k}.$$

end for

Proof. We show only the idea of the proof for the first listed case where we assume the *finite-horizon episodic* return.[1]

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [R(\tau)] \\
&= \nabla_\theta \int_{\tau \in \mathcal{T}} P(\tau \mid \pi_\theta) R(\tau)\, \mathrm{d}\tau && \text{(Definition 1.6)} \\
&= \int_{\tau \in \mathcal{T}} \nabla_\theta P(\tau \mid \pi_\theta) R(\tau)\, \mathrm{d}\tau && \text{(swap gradient and integral)} \\
&= \int_{\tau \in \mathcal{T}} P(\tau \mid \pi_\theta) \nabla_\theta \log P(\tau \mid \pi_\theta) R(\tau)\, \mathrm{d}\tau && \text{(log-derivative trick)} \\
&= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [\nabla_\theta \log P(\tau \mid \pi_\theta) R(\tau)] && \text{(expectation form)} \\
&= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{|\tau|} \nabla_\theta \log \pi_\theta(a_t \mid s_t) R(\tau) \right] && \text{(propagate gradient).} \\
&= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{|\tau|} R(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] && \text{(independence).}
\end{aligned}$$

□

In Algorithm 2.4 we approximate the last expectation with a sample mean calculated from the set of observed trajectories $\mathcal{D} = \{\tau_i\}$. This can be seen as the Monte Carlo approach as we need to observe the whole episode first. Due to this fact, we are limited only to the *finite-horizon episodic* environment.

[1] Note that third step is an unjustified operation as we cannot simply swap gradient with integral without appropriate assumptions.

Algorithm 2.5: 1-step Actor-Critic

Input: initial policy parameters θ, initial V-function parameters ϕ.

1 **repeat**

2 Observe state s and select action $a \sim \pi_\theta(\cdot \mid s)$.

3 Execute a in the environment.

4 Observe next state s', reward r and done signal d to indicate whether s' is terminal.

5 **if** d is true **then**

6 Reset environment state.

7 **end if**

8 Compute advantage

$$A = (r + \gamma(1 - d)V_\phi(s')) - V_\phi(s).$$

9 Update V-function by one step of semi-gradient descent algorithm in direction of gradient estimation

$$\phi_{k+1} \leftarrow \phi_k + \alpha_\phi g_\phi \qquad g_\phi = A\nabla_\phi V_\phi(s).$$

10 Update policy function with policy gradient estimation

$$\theta_{k+1} \leftarrow \theta_k + \alpha_\theta g_\theta \qquad g_\theta = A\nabla_\theta \log \pi_\theta(a \mid s).$$

11 **until** convergence

2.5 Actor-Critic

Actor-Critic (AC) is a combination of REINFORCE algorithm and TD method, contrary to MC method used in pure REINFORCE. With this generalization we are able to "backpropagate" the reward information immediately and so we are not forced to wait until the end of the episode. Therefore, we can overcome the previous limitation and apply this algorithm even on an *infinite-horizon* environment. The simplest one-step variant is listed in Algorithm 2.5.

The name is derived from the name similarity, where *actor* (*a.k.a.* policy network) "acts" (more precisely choosing an action to perform) and *critic* (*a.k.a.* state-value function) evaluates how good or bad the state is.

Nowadays, there are two main implementations of this algorithm. Firstly, there is Asynchronous Advantage AC (A3C) [Mnih et al., 2016], which uses asynchronous updates. Every thread shares the access to one single model and it does not care about the possible race conditions. This enables us to use as many CPUs as possible, but it is questionable how to properly incorporate the GPU or TPU.

Alternatively, we can use Parallel Advantage AC (PAAC) [Clemente et al., 2017], which utilizes one thread with the access to the model and the rest of them as workers interacting with the environment.[1] Here we can simply assign the GPU/TPU to the main thread as it is the only one which can perform the network update.

There are several improvements such as incorporating n-step return and/or RNN. Also, in the original publication, the authors firstly propose an *entropy regularization term*: $-\beta H(\pi_\theta(s))$, which prevents the premature convergence. This is a critical part because the premature convergence causes the exploration vanishment.

[1] This algorithm is also known as Advantage AC (A2C).

2.6 Deep Deterministic Policy Gradient

One of the most important algorithms, which is used as the cornerstone in many other algorithms, is Deep Deterministic Policy Gradient (DDPG) [Lillicrap et al., 2016]. It is quite difficult to classify it according to our taxonomy and there are at least two possible ways how to look at it. First and foremost, we can see it as a generalization of DQN for environments with continuous action space. The problem with DQN is the $\max_a Q(s, a)$ operation, calculation of which is straightforward in case of discrete action space, but problematic with continuous one.

Alternatively, we can say that DDPG is the combination of DQN and PGT, but contrary to REINFORCE algorithm, here we contemplate the deterministic policy $\mu(\cdot \mid s)$. By this choice we lose the theoretical support of PGT stated in Theorem 2.1. Fortunately, we are able to derive a similar result known as Deterministic Policy Gradient Theorem (DPGT) [Silver et al., 2014], which we state without the proof.

Either way, the algorithm is *off-policy* and we can take advantage of *replay buffer*. We use the *target network* concept also known from DQN for critic and even for actor. However, for updating target networks we use polyak averaging

$$\theta_{\text{targ}} \leftarrow \alpha\theta_{\text{targ}} + (1 - \alpha)\theta,$$

typically with large parameter $\alpha \approx 0.99$. All details are in Algorithm 2.6 and for illustration we add Figure 2.2.

Theorem 2.2 (Deterministic Policy Gradient Theorem)**.** *Under several assumptions about continuousness, it holds:*

$$\nabla_\theta J(\mu_\theta) \propto \mathop{\mathbb{E}}_{s\sim\eta_{\mu_\theta}} \left[\nabla_\theta \mu_\theta(s) \nabla_a Q_{\mu_\theta}(s, a)|_{a=\mu_\theta(s)}\right].$$

The Theorem 2.2 is not really surprising because we can reformulate the *expected return* as

$$J(\mu_\theta) \propto \mathop{\mathbb{E}}_{s\sim\eta_{\mu_\theta}} \left[Q_{\mu_\theta}(s, \mu_\theta(s))\right], \tag{2.2}$$

from which we get the DPGT by application of the chain rule.

It is important to note that here the action-value function has only *one output* and the evaluative action is given to it as an input. We also need to ensure that this function is differentiable *w.r.t.* action a. By using NN, this is assumption is fulfilled.

We must not forget to somehow involve exploration because the policy function is deterministic. Authors suggest using Ornstein-Uhlenbeck proccess to sample time-correlated noise. This is reasonable when we try to apply the algorithm to the real world robotics. For our purpose we employ ordinary mean-zero Gaussian noise.

Opposite to DQN, it is not clear how to use DDPG in a discrete environment. One option is to use Straight-Through Gumbel-Softmax estimator [Jang et al., 2016], which is attached as an actor head. We use this head during the action sampling on Line 3 and during the policy update on Line 13. Otherwise (Line 11), we use greedy action according to the raw actor output. By increasing the temperature parameter during the sampling, we can encourage the *exploration*.

Algorithm 2.6: Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$.

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$.

repeat

Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{\text{low}}, a_{\text{high}})$, where $\epsilon \sim \mathcal{N}$.

Execute a in the environment.

Observe next state s', reward r and done signal d to indicate whether s' is terminal.

Store (s, a, r, s', d) in replay buffer $\mathcal{D}$.

if d is true **then**

Reset environment state.

end if

Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$.

Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')).$$

Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2.$$

Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s)).$$

Update target networks with

$$\phi_{\text{targ}} \leftarrow \alpha\phi_{\text{targ}} + (1 - \alpha)\phi$$
$$\theta_{\text{targ}} \leftarrow \alpha\theta_{\text{targ}} + (1 - \alpha)\theta.$$

until convergence

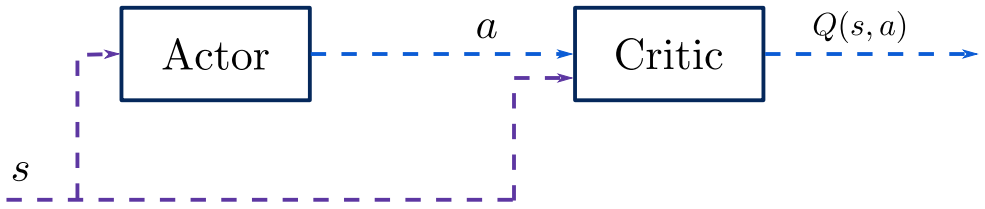


Figure 2.2: Deep Deterministic Policy Gradient (DDPG)

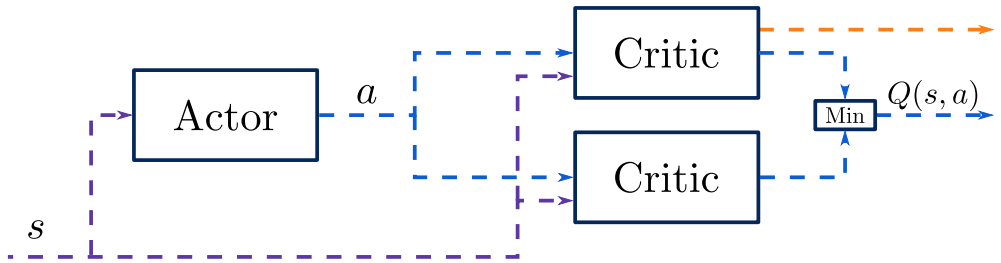


Figure 2.3: Twin Delayed DDPG (TD3)

2.7 Twin Delayed DDPG

Twin Delayed DDPG (TD3) [Fujimoto et al., 2018] is a direct amplification of DDPG, which employs three tricks to stabilize training. Firstly, it uses two critics and chooses the minimum of their predictions *e.g.,*

$$y(r, s', d) = r + \gamma(1 - d) \min_{j=\{1,2\}} Q_{\phi_{\text{targ},j}}(s', a').$$

The purpose of this change is to suppress the maximization bias and it is motivated by the Double Q-Learning known from Section 2.2.

Authors also show that it is essential to update the policy network *w.r.t.* quality critic. Therefore, they propose *delayed policy* updates, where they postpone the actor update by d steps beside the critic update. This results in reducing the training variance.

Finally, the *target policy smoothing* is employed, where they add a noise to the target actions during critic training *e.g.,*

$$a' = \text{clip}(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{\text{low}}, a_{\text{high}}), \qquad \epsilon \sim \mathcal{N}(0, \sigma).$$

This has a similar effect as delayed policy updates and it reduces the training variance. As usual, the full algorithm description is given in Algorithm 2.7. In Figure 2.3 we highlight the critic output which is used during the policy update with an orange color.

As of today, the TD3 is considered to be the state-of-the-art algorithm for off-policy continuous-actions RL. On this place, we would like to mention also algorithm Soft Actor-Critic (SAC) [Haarnoja et al., 2018], which shows comparable results to TD3.

Recently, the DDPG++ [Fakoor et al., 2020] suggests, along with other modifications, employing the minimum of two critics also during the policy update.[1] We can do it simply by changing Line 15 to the form

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \min_{j \in \{1,2\}} Q_{\phi_j}(s, \mu_\theta(s)).$$

In our experiments we call this Twin Delayed DDPG++ (TD4).

[1] There are other tricks which DDPG++ uses but this is highlighted by the authors as the crucial one. In order not to confuse readers, we use term TD4 for TD3 algorithm with only this enhancement.

Algorithm 2.7: Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$.

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$.

repeat

Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{\text{low}}, a_{\text{high}})$, where $\epsilon \sim \mathcal{N}$.

Execute a in the environment.

Observe next state s', reward r and done signal d to indicate whether s' is terminal.

Store (s, a, r, s', d) in replay buffer $\mathcal{D}$.

if d is true **then**

Reset environment state.

end if

Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$.

Compute target actions

$$a' = \text{clip}(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{\text{low}}, a_{\text{high}}), \qquad \epsilon \sim \mathcal{N}(0, \sigma).$$

Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{j=\{1,2\}} Q_{\phi_{\text{targ},j}}(s', a').$$

Update Q-functions by one step of gradient descent using

$$\forall j \in \{1, 2\}: \qquad \nabla_{\phi_j} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_j}(s, a) - y(r, s', d)\right)^2.$$

if time to update policy function **then**

Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s)).$$

Update target networks with

$$\forall j \in \{1, 2\}: \qquad \phi_{\text{targ},j} \leftarrow \alpha\phi_{\text{targ},j} + (1 - \alpha)\phi_j$$
$$\theta_{\text{targ}} \leftarrow \alpha\theta_{\text{targ}} + (1 - \alpha)\theta.$$

end if

until convergence

2.8 TD3 Forward-Looking Actor

Freshly, TD3 Forward-looking Actor (TD3-FORK) [Wei and Ying, 2020] proposes the combination of TD3 and a *model-based* approach. Authors, with reference to the human decision process, propose improving the action-value estimate used during the actor update by the "looking to the future" (hence the name forward-looking). In order to foretell the future the authors acquire the environment model approximators.

Remember the alternative expected return formulation from Equation 2.2

$$J(\mu_\theta) \propto \mathop{\mathbb{E}}_{s\sim\eta_{\mu_\theta}} \left[Q_{\mu_\theta}(s, \mu_\theta(s))\right].$$

We extend this by the environment model approximators as follows

$$J(\mu_\theta) \propto \mathop{\mathbb{E}}_{s\sim\eta_{\mu_\theta}} \left[Q_{\mu_\theta}\left(s, \mu_\theta(s)\right) + \lambda F\left(s, \mu_\theta(s)\right)\right], \tag{2.3}$$

where[1]

$$F(s_t, \mu_\theta(s_t)) = R_\psi(s_t, \mu_\theta(s_t), \tilde{s}_{t+1}) + \sum_{k=1}^{K} \left[\beta^k R_\psi(\tilde{s}_{t+k}, \mu_\theta(\tilde{s}_{t+k}), \tilde{s}_{t+k+1})\right] + \beta^{K+1} Q_{\mu_\theta}(\tilde{s}_{t+K+1}, \mu_\theta(\tilde{s}_{t+K+1})).$$

The *reward* approximator $R_\psi(s_t, a_t, s_{t+1})$ is used for the prediction of further rewards, whereas

$$\forall k \in [K]: \qquad \begin{aligned} \tilde{s}_{t+1} &= P_\kappa(s_t, \mu_\theta(s_t)) \\ \tilde{s}_{t+k+1} &= P_\kappa(\tilde{s}_{t+k}, \mu_\theta(\tilde{s}_{t+k})), \end{aligned}$$

are estimates of following states given by *system* approximator $P_\kappa(s_t, a_t)$. Note that the definitions of these additional approximators are identical to the reward function and transition model from the MDP Definition 1.1. The additional term $F(s, \mu_\theta(s))$ can be viewed as K-steps expansion of Bellman Equations 1.1 and the whole term as a combination of two estimators. Finally, once the term $\beta < 1$ is used we talk about the exponential decay.[2]

The training of system and reward approximators is, thanks to the replay buffer $\mathcal{D}$, a supervised learning. For their training, authors use smooth-L1 and MSE loss function, respectively. Finally, the parameter λ is called *adaptive weight* and it changes dynamically as the training proceeds. For more details see the original publication.

Algorithms 2.8 and 2.9 are taken from the original publication, however, we have made a lot of modifications as the original contains inconsistencies and typos.[3]

Finally, we can utilize the DDPG++ trick and substitute the Line 18 by

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} \left[\min_{j\in\{1,2\}} Q_{\phi_j}\left(s, \mu_\theta(s)\right) + \lambda F\left(s, \mu_\theta(s)\right)\right].$$

In our experiments we call this TD4 Forward-looking Actor (TD4-FORK).

[1] In original publication authors discuss only the variant with $K = 1$.

[2] The motivation behind is an assumption that the farther estimates are noisier.

[3] Surprisingly, the publication does not discuss how to deal with the ends of episodes. We believe that a suitable solution would be a simple cut of $F(s, a)$ expansion as soon as the episode ends. This is similar to what DDPG does when it computes targets on Line 11 in Algorithm 2.6.

Algorithm 2.8: TD3 Forward-looking Actor

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, system parameters κ, reward parameters ψ, empty replay buffer $\mathcal{D}$.

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$.
repeat
    Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{\text{low}}, a_{\text{high}})$, where $\epsilon \sim \mathcal{N}$.
    Execute a in the environment.
    Observe next state s', reward r and done signal d to indicate whether s' is terminal.
    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$.
    if d is true **then**
        Reset environment state.
    end if
    Update agent as described in Algorithm 2.9.
until convergence

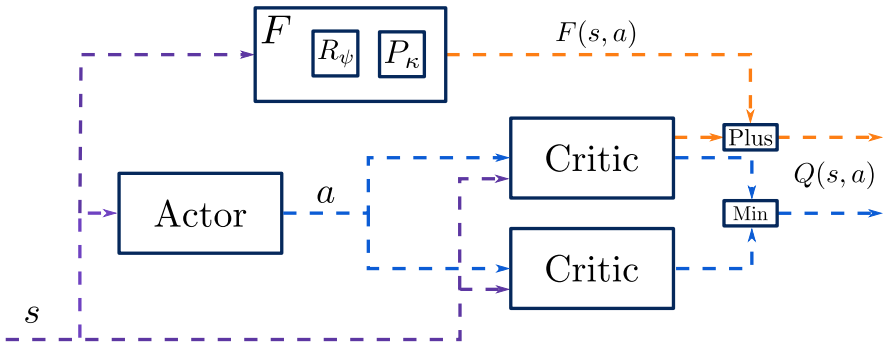


Figure 2.4: TD3 Forward-looking Actor (TD3-FORK)

Algorithm 2.9: TD3 Forward-looking Actor Update Agent

Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$.

Compute target actions

$$a' = \text{clip}(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{\text{low}}, a_{\text{high}}), \qquad \epsilon \sim \mathcal{N}(0, \sigma).$$

Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{j=\{1,2\}} Q_{\phi_{\text{targ},j}}(s', a').$$

Update Q-functions by one step of gradient descent using

$$\forall j \in \{1, 2\}: \qquad \nabla_{\phi_j} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_j}(s, a) - y(r, s', d)\right)^2.$$

Update system and reward network by one step of gradient descent using

$$\nabla_\kappa \frac{1}{|B|} \sum_{(s,a,s') \in B} \text{smooth-L1}(P_\kappa(s, a) - s') \qquad \nabla_\psi \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(R_\psi(s, a, s') - r\right)^2.$$

if time to update policy function **then**

Use reward and system network to compute

$$F\left(s_t, \mu_\theta(s_t)\right) = R_\psi\left(s_t, \mu_\theta(s_t), \tilde{s}_{t+1}\right) + \sum_{k=1}^{K} \left[\beta^k R_\psi\left(\tilde{s}_{t+k}, \mu_\theta(\tilde{s}_{t+k}), \tilde{s}_{t+k+1}\right)\right] + \beta^{K+1} Q_{\phi_1}\left(\tilde{s}_{t+K+1}, \mu_\theta(\tilde{s}_{t+K+1})\right),$$

where

$$\begin{aligned} \tilde{s}_{t+1} &= \text{clip}(P_\kappa(s_t, \mu_\theta(s_t)), s_{\text{low}}, s_{\text{high}}) \\ \forall k \in [K]: \qquad \tilde{s}_{t+k+1} &= \text{clip}(P_\kappa(\tilde{s}_{t+k}, \mu_\theta(\tilde{s}_{t+k})), s_{\text{low}}, s_{\text{high}}), \end{aligned}$$

Compute adaptive weight λ.

Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left[Q_{\phi_1}\left(s, \mu_\theta(s)\right) + \lambda F\left(s, \mu_\theta(s)\right)\right]$$

Update target networks with

$$\begin{aligned} \forall j \in \{1, 2\}: \qquad \phi_{\text{targ},j} &\leftarrow \alpha\phi_{\text{targ},j} + (1 - \alpha)\phi_j \\ \theta_{\text{targ}} &\leftarrow \alpha\theta_{\text{targ}} + (1 - \alpha)\theta. \end{aligned}$$

end if

3. Multi-Agent RL Algorithms

Equipped with knowledge of successful algorithms for SARL, we continue our explanation by generalizing these to the MARL setting. In Section 3.1 we show how to directly employ algorithms which we have discussed in the previous chapter for environments with multiple agents. Later on, in Section 3.2 we discuss the CTDE setting, which we consider as the most promising scheme for further research. Lastly, we describe several successful algorithms which are built exclusivelly for multi-agent setting.

Unlike the previous chapters, in which we worked with a fully observable model (namely MDP), we use the POSG as our foundation stone in MARL. There are several reasons for this decision. Firstly, the majority of SARL algorithms are described with an assumption of full observability and so it would be counterproductive to do it otherwise. Also, the generalization is still really straightforward because there is only one state and one observation at each timestep. With MARL we still have only one state but each agent has different observation and so we cannot simply interchange it as there is no bijection between these two sets.

3.1 Utilizing SARL Algorithms in MARL

In this section we provide an intermediate step between SARL and MARL. There are several options how to employ algorithms known from the previous chapter in environments with multiple agents. Widely quoted article is [Gupta et al., 2017], where authors suggest three possible training schemes.

3.1.1 Centralized Scheme

The most obvious and easiest approach is to create a joint model for all agents. This model takes as an input a joint-observation $\boldsymbol{\omega}$ and as an output presents a joint-action $\boldsymbol{a}$. The main downside is that we are conditioned on the existence of *central element*. In the real world application this implies the intensive communication between each agent and this central model. The agent herself just observes the environment and sends a request to the central. The central evaluates all observations and divides the tasks. In many situations, this communication is problematic as it adds a communication delay, which is in many cases intractable.

Moreover, this scheme does not scale very well. The naive implementation causes an exponential growth in these spaces with a number of agents, precisely $\prod_{i\in[N]} |\Omega_i|$ and $\prod_{i\in[N]} |\mathcal{A}_i|$ for discrete spaces. On the action side, it is possible to reduce this overhead by considering the factorized centralized controller which assumes the independence among agents sub-policies.[1] By this we get $\sum_{i\in[N]} |\mathcal{A}_i|$. Unfortunately, this is unthinkable on the observation side and so their concatenation will always result in exponential growth.

[1]This can be easily done for continuous action spaces, as we only concatenate outputs of NN. However, for discrete action spaces we are forced to perform softmax on each policy individually. This usually causes a significant intervention to model architecture.

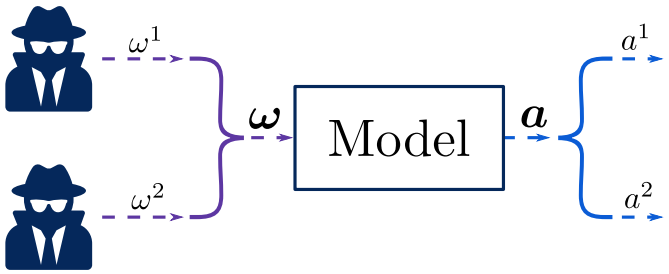


Figure 3.1: Centralized scheme

3.1.2 Concurrent Scheme

Alternatively, we can train each agent completely independently. So each of them has her own policy, which is in no way associated with policies of the others. For these reasons, we call this setting *concurrent scheme*. The implementation is straightforward and especially the extension to the environment with heterogeneous agents is for free. However, we face one main drawback. As we have already discussed in Subsection 1.2.4 the environment is not stationary from the agent point of view. This causes an instability of the training and is problematic with algorithms with replay buffer as the information saved there is out-dated.

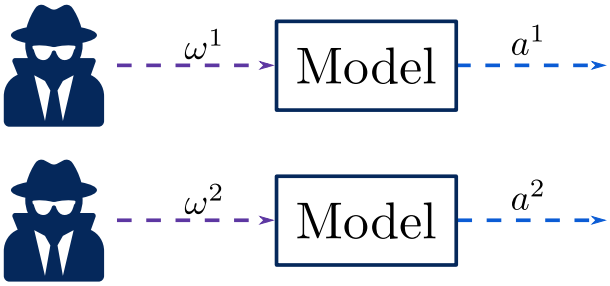


Figure 3.2: Concurrent scheme

3.1.3 Parameter Sharing Scheme

Lastly, in case that all agents are *homogeneous*, we can use only one model for all of them. Therefore, we talk about *parameter sharing scheme*. Note that this does not mean that all agents will behave in the same way, as each of them receives potentially different observations from the *observation model*.

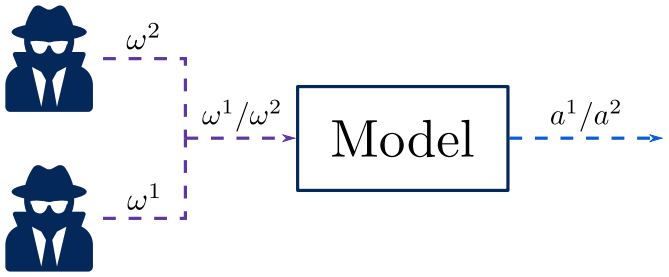


Figure 3.3: Parameter Sharing Scheme

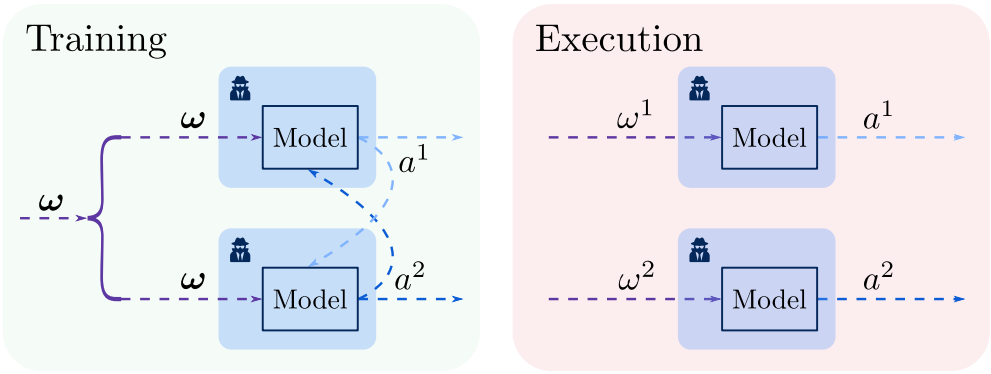


Figure 3.4: Centralized Training with Decentralized Execution (CT-DE)

3.2 Centralized Training with Decentralized Execution

As we have already said, in many situation the presence of the central element during the execution is problematic. With the aim of the decentralized execution in Actor-Critic model, we can fix the actor's input only to the local agent's observation. Note that in this work we strictly prohibit any kind of communication among the agents during the execution. On the other side of the coin, the training is typically centralized and so the presence of some centralized element is not problematic. Therefore, we can bring forward more information to our model such as observations of other agents, performing actions of other agents or even the Markov state of the environment if available. Similarly, when employing the AC architecture we can introduce this information to the critic's input, as the critic is not prototypically used during the inference. We can see this relaxation of restrictions as an implicit communication among the agents during the training.

This approach can be widely seen in the real world applications. In these, we typically train agents in a simulator, in which we have a full control of all kinds of communication. Moreover, we can often separate the Markov state of the environment relatively easily (or at least some part of it). Accordingly to these ideas, the name *centralized* is derived. Later on, after the models are deployed to the real situation, the trained agents must rely only on themselves.

Recently, the automotive industry has widely made use of this approach in autonomous vehicles. Each vehicle is represented as an agent in the traffic environment. In order to train these agents satisfactorily we are forced to drive several millions of kilometers. This is certainly not possible in real conditions, regardless of legislative view, which is often the brake on progress. Therefore, the common approach is to employ the simulator, where we can evoke the realistic conditions. Obviously, no simulator is one hundred percent accurate and so final phase of the training is unconditionally on the real roads. In this work we omit this last phase as we are more interested in theoretical and formal results.

For previously mentioned advantages, this approach has been found to be the most promising among all other suggested schemes. More details are available in [Foerster, 2018]. Finally, from now on, all following algorithms incorporate this scheme.

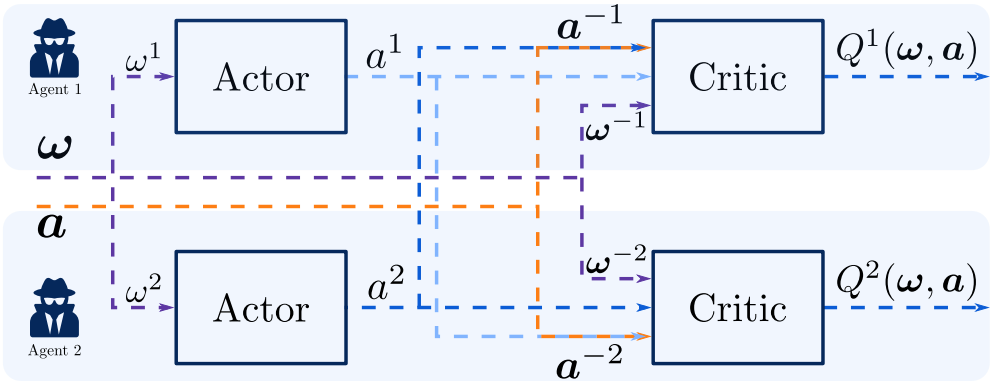


Figure 3.5: Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

3.3 Multi-Agent DDPG

Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [Lowe et al., 2017] is a compounding of DDPG and CT-DE. The actor for each agent necessarily receives only the agent's observation. On the other hand, the critic is used only during the training and therefore, it can receive some shared information among the other agents. Typically, the input forms a concatenation of joint-observation $\boldsymbol{\omega}$ and joint-action $\boldsymbol{a}$. In case that we have access to the environment Markov state s, it is convenient to add this to the critic input as well.

When we work with dec-POMDP, in which the one reward is shared among all agents, it is possible to have only one central critic. This critic is then shared among all agents. Contrary, in POSG, we are forced to implement a separate critic for each agent, because the rewards are potentially different.[1] This approach is described in Algorithm 3.1 with all other details. For better overview we obey notation

$$Q^i_\phi = Q_{\phi^i} \qquad \mu^i_\theta = \mu_{\theta^i} \qquad \nabla^i_\theta = \nabla_{\theta^i}.$$

In the publication authors also discuss the possible inference of other agents' policies. However, this is not necessary in the CT-DE setting. Furthermore, in order to train the robust policy which is resistant to changes in the policies of other agents, authors propose to train a collection of different sub-policies. At the beginning of each episode we sample a policy, which is going to be used during that episode.[2]

Recently, the experiments [Wei et al., 2018] have shown that it is beneficial to estimate the gradient for the policy update based on the current agents' policies rather than using actions from the replay buffer. This can be done by the substitution of Line 14 by

$$\nabla^i_\theta \frac{1}{|B|} \sum_{\boldsymbol{\omega} \in B} Q^i_\phi(\boldsymbol{\omega}, \boldsymbol{\mu_\theta}(\boldsymbol{\omega})).$$

In our experiments we call this Soft MADDPG (soft-MADDPG).

[1] In the implementation we use only this variant.
[2] We omit this trick in our implementation.

Algorithm 3.1: Multi-Agent Deep Deterministic Policy Gradient

Input: initial policy parameters $\boldsymbol{\theta}$, Q-function parameters $\boldsymbol{\phi}$, empty replay buffer $\mathcal{D}$.

Set target parameters equal to main parameters $\boldsymbol{\theta}_{\text{targ}} \leftarrow \boldsymbol{\theta}$, $\boldsymbol{\phi}_{\text{targ}} \leftarrow \boldsymbol{\phi}$.

repeat

  Observe joint-observation $\boldsymbol{\omega}$ and select joint-action $\boldsymbol{a} = \text{clip}(\boldsymbol{\mu_\theta}(\boldsymbol{\omega}) + \boldsymbol{\epsilon}, \boldsymbol{a}_{\text{low}}, \boldsymbol{a}_{\text{high}})$, where $\boldsymbol{\epsilon} \sim \mathcal{N}$.

  Execute $\boldsymbol{a}$ in the environment.

  Observe next observations $\boldsymbol{\omega'}$, rewards $\boldsymbol{r}$ and done signal for each agent $\boldsymbol{d}$.

  Store $(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega'}, \boldsymbol{d})$ in replay buffer $\mathcal{D}$.

  if all($\boldsymbol{d}$) is true **then**

    Reset environment state.

  end if

  Randomly sample a batch of transitions, $B = \{(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega'}, \boldsymbol{d})\}$ from $\mathcal{D}$.

  for agent i in $[N]$ **do**

    Compute targets

$$y(r^i, \boldsymbol{\omega'}, d^i) = r^i + \gamma(1 - d^i) Q^i_{\phi_{\text{targ}}}(\boldsymbol{\omega'}, \boldsymbol{\mu}_{\boldsymbol{\theta}_{\text{targ}}}(\boldsymbol{\omega'}))).$$

    Update Q-function by one step of gradient descent using

$$\nabla^i_\phi \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega'}, \boldsymbol{d}) \in B} \left(Q^i_\phi(\boldsymbol{\omega}, \boldsymbol{a}) - y(r^i, \boldsymbol{\omega'}, d^i) \right)^2.$$

    Update policy by one step of gradient ascent using

$$\nabla^i_\theta \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}) \in B} Q^i_\phi(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i)).$$

    Update target networks with

$$\phi^i_{\text{targ}} \leftarrow \alpha \phi^i_{\text{targ}} + (1 - \alpha)\phi^i$$
$$\theta^i_{\text{targ}} \leftarrow \alpha \theta^i_{\text{targ}} + (1 - \alpha)\theta^i.$$

  end for

until convergence

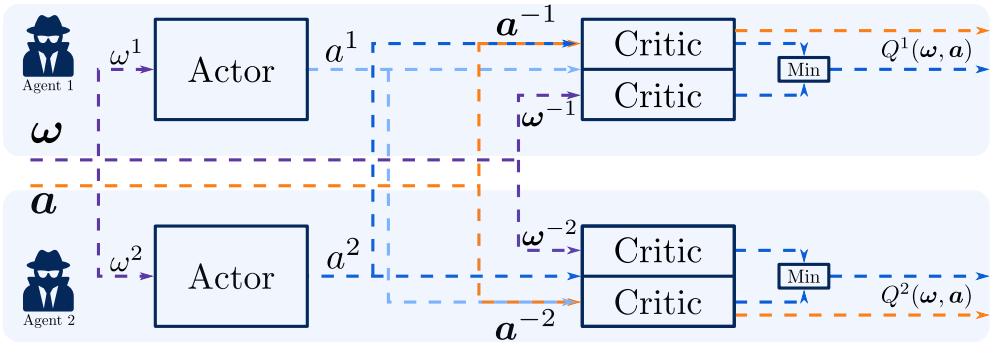


Figure 3.6: Multi-Agent Twin Delayed DDPG (MATD3)

3.4 Multi-Agent TD3

In publication [Ackermann et al., 2019] authors show that MADDPG suffers from the maximization bias in the very same way as the pure DDPG.

Therefore, authors adapt the extensions from TD3 and show that by adding one more extra critic and taking the minimum of both of them during the estimation of the target values *e.g.*,

$$y(r^i, \boldsymbol{\omega}', d^i) = r^i + \gamma(1 - d^i) \min_{j \in \{1,2\}} Q^i_{\phi_{\text{targ},j}}(\boldsymbol{\omega}', \boldsymbol{a}'),$$

we can significantly reduce the maximization bias. Moreover, authors utilize delayed policy updates and target policy smoothing for reducing variance as well. This modified version is called (unsurprisingly) Multi-Agent Twin Delayed DDPG (MATD3). For more details see Algorithm 3.2.

Naturally, we can combine the possible variations of TD3 and MADDPG. As far as we know, there is no publication which proposes, investigates and compares these combinations. In the interest of maintaining integrity, we include these variations here, rather than in the Chapter 4 with our contribution.

Firstly, we can use the *soft version* of the policy update by substitution Line 16 with

$$\nabla^i_\theta \frac{1}{|B|} \sum_{\boldsymbol{\omega} \in B} Q^i_{\phi_1}(\boldsymbol{\omega}, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{\omega})).$$

In our experiments we call this Soft MATD3 (soft-MATD3).

Alternatively, we can do the same trick as DDPG++ and take the minimum of both critics

$$\nabla^i_\theta \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}) \in B} \min_{j \in \{1,2\}} Q^i_{\phi_j}(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i)).$$

In our experiments we call this Multi-Agent TD4 (MATD4).

Finally, it is possible to combine both of these tweaks together, so we get

$$\nabla^i_\theta \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}) \in B} \min_{j \in \{1,2\}} Q^i_{\phi_j}(\boldsymbol{\omega}, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{\omega})).$$

In our experiments we call this Soft MATD4 (soft-MATD4).

Algorithm 3.2: Multi-Agent Twin Delayed DDPG

Input: initial policy parameters $\boldsymbol{\theta}$, Q-function parameters $\boldsymbol{\phi}_1, \boldsymbol{\phi}_2$, empty replay buffer $\mathcal{D}$.

Set target parameters equal to main parameters $\boldsymbol{\theta}_{\text{targ}} \leftarrow \boldsymbol{\theta}$, $\boldsymbol{\phi}_{\text{targ},1} \leftarrow \boldsymbol{\phi}_1$, $\boldsymbol{\phi}_{\text{targ},2} \leftarrow \boldsymbol{\phi}_2$.

repeat

  Observe joint-observation $\boldsymbol{\omega}$ and select joint-action $\boldsymbol{a} = \text{clip}(\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{\omega}) + \boldsymbol{\epsilon}, \boldsymbol{a}_{\text{low}}, \boldsymbol{a}_{\text{high}})$, where $\boldsymbol{\epsilon} \sim \mathcal{N}$.

  Execute $\boldsymbol{a}$ in the environment.

  Observe next observations $\boldsymbol{\omega}'$, rewards $\boldsymbol{r}$ and done signal for each agent $\boldsymbol{d}$.

  Store $(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega}', \boldsymbol{d})$ in replay buffer $\mathcal{D}$.

  if all($\boldsymbol{d}$) is true **then**

    Reset environment state.

  end if

  Randomly sample a batch of transitions, $B = \{(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega}', \boldsymbol{d})\}$ from $\mathcal{D}$.

  for agent i in $[N]$ **do**

    Compute target actions

$$\boldsymbol{a}' = \text{clip}(\boldsymbol{\mu}_{\boldsymbol{\theta}_{\text{targ}}}(\boldsymbol{\omega}') + \text{clip}(\boldsymbol{\epsilon}, -c, c), \boldsymbol{a}_{\text{low}}, \boldsymbol{a}_{\text{high}}), \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \sigma \boldsymbol{I}).$$

    Compute targets

$$y(r^i, \boldsymbol{\omega}', d^i) = r^i + \gamma(1 - d^i) \min_{j \in \{1,2\}} Q^i_{\phi_{\text{targ},j}}(\boldsymbol{\omega}', \boldsymbol{a}').$$

    Update Q-function by one step of gradient descent using

$$\forall j \in \{1, 2\}: \qquad \nabla^i_{\phi_j} \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega}', \boldsymbol{d}) \in B} \left(Q^i_{\phi_j}(\boldsymbol{\omega}, \boldsymbol{a}) - y(r^i, \boldsymbol{\omega}', d^i) \right)^2.$$

    if time to update policy function **then**

      Update policy by one step of gradient ascent using

$$\nabla^i_{\theta} \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}) \in B} Q^i_{\phi_1}(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_{\theta}(\omega^i)).$$

      Update target networks with

$$\phi^i_{\text{targ}} \leftarrow \alpha \phi^i_{\text{targ}} + (1 - \alpha)\phi^i$$
$$\theta^i_{\text{targ}} \leftarrow \alpha \theta^i_{\text{targ}} + (1 - \alpha)\theta^i.$$

    end if

  end for

until convergence

4. Our Contribution

We propose MATD3 Forward-looking Actor (MATD3-FORK) as our theoretical contribution. It is a natural combination of several previously discussed algorithms. As the basis we use MADDPG, which employs CT-DE scheme, and hence also the MATD3-FORK utilizes CT-DE. This is an essential property, as we have discussed in Chapter 3. In order to eliminate the maximization bias which causes the training instability we engage the two critics as proposed in TD3 and MATD3. Finally, we learn the environment transition and reward model as proposed in TD3-FORK. The information from these models is then utilized in a policy update for each agent. According to our taxonomy we talk about *off-policy*, *model-based* algorithm which utilizes both *value function* and *policy approximation*.

Similarly to Equation 2.3 we formulate the expected return for agent $i \in [N]$ as

$$J^i(\boldsymbol{\mu_\theta}) \propto \mathop{\mathbb{E}}_{\boldsymbol{\omega}\sim\eta_{\boldsymbol{\mu_\theta}}} \left[\mathop{\mathbb{E}}_{\boldsymbol{a}^{-i}\sim\boldsymbol{\mu}_{\boldsymbol{\theta}}^{-i}(\boldsymbol{\omega}^{-i})} \left[Q^i_{\boldsymbol{\mu_\theta}} \left(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i) \right) + \lambda F^i \left(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i) \right) \right] \right].$$

The additional term $F^i\left(\boldsymbol{\omega}_t, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i)\right)$ is again calculated *w.r.t.* *system* and *reward approximators* as

$$\begin{aligned} F^i \left(\boldsymbol{\omega}_t, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i) \right) &= R^i_\psi \left(\boldsymbol{\omega}_t, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i), \tilde{\boldsymbol{\omega}}_{t+1} \right) \\ &\quad + \sum_{k=1}^{K} \left[\beta^k R^i_\psi \left(\tilde{\boldsymbol{\omega}}_{t+k}, \boldsymbol{\mu_\theta}(\tilde{\boldsymbol{\omega}}_{t+k}), \tilde{\boldsymbol{\omega}}_{t+k+1} \right) \right] \\ &\quad + \beta^{K+1} Q^i_{\phi_1} \left(\tilde{\boldsymbol{\omega}}_{t+K+1}, \boldsymbol{\mu_\theta}(\tilde{\boldsymbol{\omega}}_{t+K+1}) \right), \end{aligned}$$

where

$$\begin{aligned} \tilde{\boldsymbol{\omega}}_{t+1} &= \mathrm{clip} \left(P_\kappa \left(\boldsymbol{\omega}_t, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i_t) \right), \boldsymbol{\omega}_{\mathrm{low}}, \boldsymbol{\omega}_{\mathrm{high}} \right) \\ \forall k \in [K]: \qquad \tilde{\boldsymbol{\omega}}_{t+k+1} &= \mathrm{clip} \left(P_\kappa \left(\tilde{\boldsymbol{\omega}}_{t+k}, \boldsymbol{\mu_\theta}(\tilde{\boldsymbol{\omega}}_{t+k}) \right), \boldsymbol{\omega}_{\mathrm{low}}, \boldsymbol{\omega}_{\mathrm{high}} \right), \end{aligned}$$

Note that we are forced to have the reward approximator R^i_ϕ for each agent $i \in [N]$ as we assume POSG. If the reward is the same for each agent[1] we could use only one common reward approximator R_ϕ. This simplification is actually used in case of the system approximator. Instead of training multiple approximators P^i_κ we utilize the joint approximator P_κ. This is possible thanks to the CT-DE approach. By this we provide more information to the system approximator and in return we get estimates of better quality.

The complete algorithm can be seen in Algorithms 4.1 and 4.2. Moreover, we add a Figure 3.6 for a better intuition. Again we use the orange color to highlight which part is utilized during the policy update.

[1]This is true for dec-POMDP or fully cooperative setting known from Section 1.2.2.

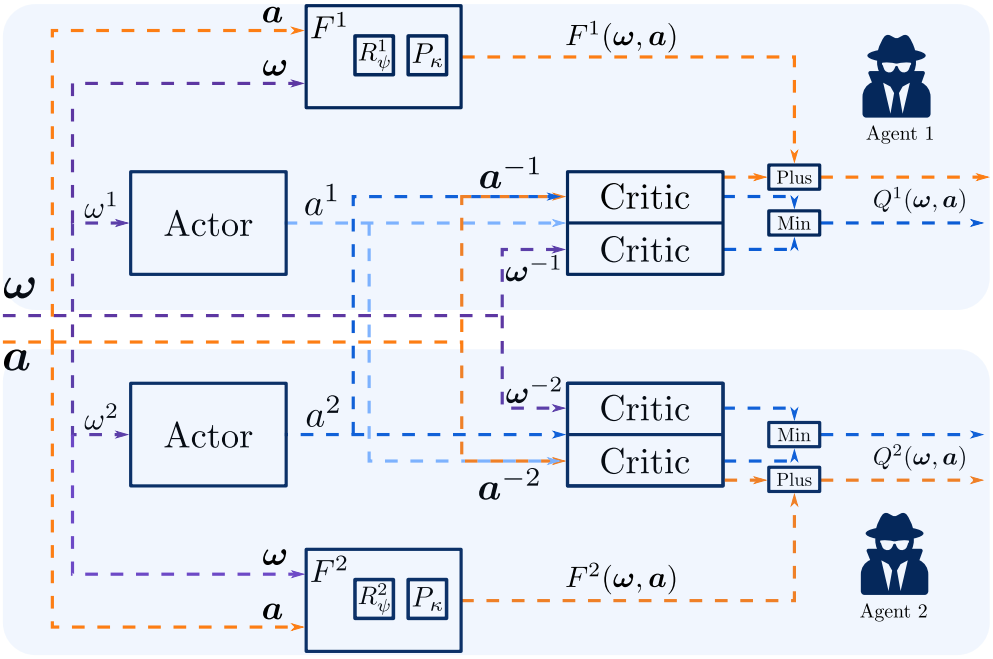


Figure 4.1: MATD3 Forward-looking Actor (MATD3-FORK)

As usual, we can tweak this basic algorithm by several tricks. Firstly, we can consider the *soft version* and rely on the current agent's policies rather than information from the replay buffer. This can be done by rewriting Line 20 into

$$\nabla^i_\theta \frac{1}{|B|} \sum_{\boldsymbol{\omega} \in B} \left[Q^i_{\phi_1} \left(\boldsymbol{\omega}, \boldsymbol{\mu_\theta}(\boldsymbol{\omega}) \right) + \lambda F^i \left(\boldsymbol{\omega}, \boldsymbol{\mu_\theta}(\boldsymbol{\omega}) \right) \right].$$

In our experiments we call this Soft MATD3-FORK (soft-MATD3-FORK).

Alternatively, we can employ the DDPG++ trick and calculate the gradient as

$$\nabla^i_\theta \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}) \in B} \left[\min_{j \in \{1,2\}} Q^i_{\phi_j} \left(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i) \right) + \lambda F^i \left(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_\theta(\omega^i) \right) \right].$$

In our experiments we call this MATD4 Forward-looking Actor (MATD4-FORK).

Altogether we can rewrite the update as

$$\nabla^i_\theta \frac{1}{|B|} \sum_{\boldsymbol{\omega} \in B} \left[\min_{j \in \{1,2\}} Q^i_{\phi_j} \left(\boldsymbol{\omega}, \boldsymbol{\mu_\theta}(\boldsymbol{\omega}) \right) + \lambda F^i \left(\boldsymbol{\omega}, \boldsymbol{\mu_\theta}(\boldsymbol{\omega}) \right) \right].$$

In our experiments we call this Soft MATD4-FORK (soft-MATD4-FORK).

Algorithm 4.1: MATD3 Forward-looking Actor

Input: initial policy parameters $\boldsymbol{\theta}$, Q-function parameters $\boldsymbol{\phi}_1, \boldsymbol{\phi}_2$, system parameters κ, reward parameters $\boldsymbol{\phi}$, empty replay buffer $\mathcal{D}$.

Set target parameters equal to main parameters $\boldsymbol{\theta}_{\text{targ}} \leftarrow \boldsymbol{\theta}$, $\boldsymbol{\phi}_{\text{targ},1} \leftarrow \boldsymbol{\phi}_1$, $\boldsymbol{\phi}_{\text{targ},2} \leftarrow \boldsymbol{\phi}_2$.

repeat

  Observe joint-observation $\boldsymbol{\omega}$ and select joint-action $\boldsymbol{a} = \text{clip}(\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{\omega}) + \boldsymbol{\epsilon}, \boldsymbol{a}_{\text{low}}, \boldsymbol{a}_{\text{high}})$, where $\boldsymbol{\epsilon} \sim \mathcal{N}$.

  Execute $\boldsymbol{a}$ in the environment.

  Observe next observations $\boldsymbol{\omega}'$, rewards $\boldsymbol{r}$ and done signal for each agent $\boldsymbol{d}$.

  Store $(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega}', \boldsymbol{d})$ in replay buffer $\mathcal{D}$.

  if all($\boldsymbol{d}$) is true **then**

    Reset environment state.

  end if

  Randomly sample a batch of transitions, $B = \{(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega}', \boldsymbol{d})\}$ from $\mathcal{D}$.

  Update system network by one step of gradient descent using

$$\nabla_{\kappa} \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{\omega}') \in B} \text{smooth-L1}(P_{\kappa}(\boldsymbol{\omega}, \boldsymbol{a}) - \boldsymbol{\omega}').$$

  for agent i in $[N]$ **do**

    Update agent i as described in Algorithm 4.2.

  end for

until convergence

Algorithm 4.2: MATD3 Forward-looking Actor — Update Agent i

13 Compute target actions

$$\boldsymbol{a}' = \text{clip}(\boldsymbol{\mu}_{\boldsymbol{\theta}_{\text{targ}}}(\boldsymbol{\omega}') + \text{clip}(\boldsymbol{\epsilon}, -c, c), \boldsymbol{a}_{\text{low}}, \boldsymbol{a}_{\text{high}}), \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma \boldsymbol{I}).$$

14 Compute targets

$$y(r^i, \boldsymbol{\omega}', d^i) = r^i + \gamma(1 - d^i) \min_{j \in \{1,2\}} Q^i_{\phi_{\text{targ},j}}(\boldsymbol{\omega}', \boldsymbol{a}').$$

15 Update Q-function by one step of gradient descent using

$$\forall j \in \{1, 2\}: \qquad \nabla^i_{\phi_j} \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{\omega}', \boldsymbol{d}) \in B} \left(Q^i_{\phi_j}(\boldsymbol{\omega}, \boldsymbol{a}) - y(r^i, \boldsymbol{\omega}', d^i) \right)^2.$$

16 Update reward network by one step of gradient descent using

$$\nabla^i_{\psi} \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}, r^i, \boldsymbol{\omega}') \in B} \left(R^i_{\psi}(\boldsymbol{\omega}, \boldsymbol{a}, \boldsymbol{\omega}') - r^i \right)^2.$$

17 **if** time to update policy function **then**

18 Use reward and system network to compute

$$\begin{aligned} F^i\left(\boldsymbol{\omega}_t, \boldsymbol{a}^{-i}, \mu^i_{\theta}(\omega^i)\right) = R^i_{\psi}\left(\boldsymbol{\omega}_t, \boldsymbol{a}^{-i}, \mu^i_{\theta}(\omega^i), \tilde{\boldsymbol{\omega}}_{t+1}\right) \\ + \sum_{k=1}^{K} \left[\beta^k R^i_{\psi}\left(\tilde{\boldsymbol{\omega}}_{t+k}, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{\omega}}_{t+k}), \tilde{\boldsymbol{\omega}}_{t+k+1}\right)\right] \\ + \beta^{K+1} Q^i_{\phi_1}\left(\tilde{\boldsymbol{\omega}}_{t+K+1}, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{\omega}}_{t+K+1})\right), \end{aligned}$$

where

$$\begin{aligned} \tilde{\boldsymbol{\omega}}_{t+1} &= \text{clip}\left(P_{\kappa}\left(\boldsymbol{\omega}_t, \boldsymbol{a}^{-i}, \mu^i_{\theta}(\omega^i_t)\right), \boldsymbol{\omega}_{\text{low}}, \boldsymbol{\omega}_{\text{high}}\right) \\ \forall k \in [K]: \qquad \tilde{\boldsymbol{\omega}}_{t+k+1} &= \text{clip}\left(P_{\kappa}\left(\tilde{\boldsymbol{\omega}}_{t+k}, \boldsymbol{\mu}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{\omega}}_{t+k})\right), \boldsymbol{\omega}_{\text{low}}, \boldsymbol{\omega}_{\text{high}}\right), \end{aligned}$$

19 Compute adaptive weight λ.

20 Update policy by one step of gradient ascent using

$$\nabla^i_{\theta} \frac{1}{|B|} \sum_{(\boldsymbol{\omega}, \boldsymbol{a}) \in B} \left[Q^i_{\phi_1}\left(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_{\theta}(\omega^i)\right) + \lambda F^i\left(\boldsymbol{\omega}, \boldsymbol{a}^{-i}, \mu^i_{\theta}(\omega^i)\right) \right].$$

21 Update target networks with

$$\begin{aligned} \phi^i_{\text{targ}} &\leftarrow \alpha \phi^i_{\text{targ}} + (1 - \alpha)\phi^i \\ \theta^i_{\text{targ}} &\leftarrow \alpha \theta^i_{\text{targ}} + (1 - \alpha)\theta^i. \end{aligned}$$

22 **end if**

5. Frameworks

In this chapter we provide a brief overview of frameworks which we consider the most relevant to MARL. We split our enumeration into two parts. Firstly, we list frameworks and libraries with RL algorithms. Secondly, we discuss environments in which we can test these algorithms.

If you are only interested in our implementation and the experiments results, you can fearlessly skip this chapter. Be only aware that we build our implementation on `SpinningUp` framework (described in Section 5.1.3) and `PettingZoo` library (described in Section 5.2.3).

5.1 Algorithm Frameworks

The RL has become the phenomenon of the past decade and therefore we can find almost an endless variety of libraries and repositories with algorithms for SARL. The quality of these implementations varies quite a lot and the most reliable source is often the author's implementation (if provided). On the other hand, the support for MARL is dismal and frequently limited only to MADDPG.

Each framework aims to capture a different quality. Whilst some of them try to be as optimized as possible, others take advantage of modularity and emphasize the extensibility. The large group of these frameworks try to be simple and easy to read, so the newcomer can run experiments without unnecessary overhead.

5.1.1 RLib

Link: `https://docs.ray.io`.

The `RLib` is a superstructure over the `Ray` library [Liang et al., 2017], which takes care of the calculations distribution over a computer cluster. The algorithms which are implemented in `RLib` can be easily scaled up over the multiple CPUs and GPUs. This framework also contains the most comprehensive list of already implemented algorithms as far as we know. For MARL there is an implementation of PS and concurrent schemes. Moreover, we can find there the MADDPG implementation as well.

Our original intention was to extend this library with other multi-agent algorithms, however, the library has been going through the intensive development. Accordingly, the API has been changing rapidly from release to release. Nevertheless, we highly recommend studing this framework as we expect, that it will be a further standard for RL.

5.1.2 Baselines

Link: `https://github.com/openai/baselines`.

OpenAI stands behind the `Baselines` repository [Dhariwal et al., 2017]. This framework contains standalone optimized implementations of the most important algorithms. It is an excellent source of various tricks and tips, however, there is no support for MARL and so it is unsuitable for us.

5.1.3 SpinningUp

Link: `https://spinningup.openai.com`.

Another project from OpenAI is the `SpinningUp` [Achiam, 2018]. Contrary to `Baselines`, the code here does not contain the optimization tricks and so the code is easy to read (and easy to extend). Moreover, the documentation contains the thorough theoretical foundations, which we have already mentioned in Chapter 1.

Our implementation is based on this framework as it contains the DDPG and the TD3 algorithm, which are the cornerstones for this work.

5.1.4 Machin

Link: `https://github.com/iffiX/machin`.

Lastly, we would like to mention an interesting project called `Machin` [Li, 2020]. It is a flexible RL framework based on PyTorch DL framework. This architecture is clean and well documented. In our opinion, the framework is a nice balance between the optimization, modularity and complexity.

5.2 Environment Frameworks

Unsurprisingly, there are notably fewer frameworks containing implemented environments. These serve as the training simulators for testing and debugging algorithms, just before the real world application.

5.2.1 Gym

Link: `https://gym.openai.com`.

Yet another fantastic project from OpenAI is `gym` library [Brockman et al., 2016]. One can find there an extensive collection of SARL environments with unified API. As of today, this API is de facto the standard in RL. There are various environments such as `BipedalWalker`, `CarRacing` or `Pong` (known from Atari-2600). Unfortunately, there is no support for MARL. Let us present a short code example in Listing A.1, taken from the official website.

5.2.2 Multi-Agent Gym

Link: `https://github.com/koulanurag/ma-gym`.

One of the available extensions of `gym` for MARL is `ma-gym` [Koul, 2019], which preserves the observation/action spaces classes from `gym`.[1] The usage is very similar as you can see in the example A.2. Note that it uses the `gym.envs.register` function so we can create environment as usual.

The library contains several multi-agent environments proposed in publication [Sunehag et al., 2017]. However, all of these contain only discrete observation/action spaces.

[1] As the work on this thesis started, our original intention was to use this library. During the preparation phase, we extended this library with `Lumberjack` environment proposed in publication [Albrecht and Ramamoorthy, 2013]. The patch was already merged into upstream, however, as the work on this thesis continued I decided not to use this library in favor of `PettingZoo`.

5.2.3 PettingZoo

Link: `https://www.pettingzoo.ml`.

Freshly, the `PettingZoo` [Terry et al., 2020a] framework comes up with the largest collection of MARL environments. There are two possible API which we can use. The first one follows AEC model described in Chapter 1. The second one is called *parallel* and it is similar to `ma-gym` with an exception that here we use dictionaries rather than lists. You can find the code example for both of them in Listings A.3 and A.4, respectively.

This library[1] is used for all of our experiments from Chapter 6.

[1] We use this library in version `2.6.1`.

6. Experiments

In this chapter we provide the experiment results for algorithms which we have described in Chapters 2 and 3. We also include their non-previously discussed modifications, namely TD4-FORK, soft-MATD3, MATD4 and soft-MATD4. Successively, we compare them to MATD3-FORK from Chapter 4.

6.1 Implementation

Despite the list of algorithm frameworks described in Chapter 5, we have chosen to reimplement these algorithms on our own. The goal of this work is not to come up with the next brand new framework, but rather, to investigate the algorithms and compare them. Therefore, we need a full control of all parts so we can see what does work and what does not.

Our implementation can be found in the attachment of this thesis.[1] This code contains a powerful CLI for running various training scenarios. We also provide a complete documentation with default values used within our experiments. Therefore, it is easy to reproduce our experiments and/or add new ones.

Due to the computational requirements, we run all experiments in MetaCentrum (see acknowledgment in Section 6.5). The overhead of running experiments there forced us to code many useful auxiliary scripts as a side project. These can be found in auxiliary repository.[2] We believe that many other researchers find this repository helpful as there was an unseen interest among many students during my lecture on this topic within InstallFest 2021.

6.2 Environment Descriptions

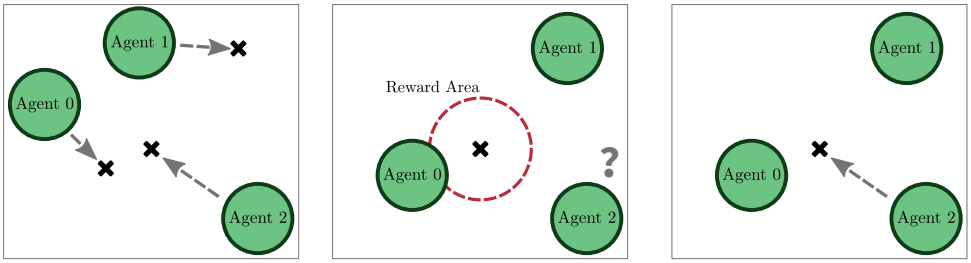
As the environment framework we have chosen `PettingZoo` from Section 5.2.3. More specifically, we have focused on Multi-Particle Environments (MPE), which were proposed by [Lowe et al., 2017]. This choice is not arbitrary. Among these environments we can find various scenarios for both fully cooperative and competitive settings. Moreover, the whole ecosystem can be relatively easily extended with a brand new scenario. Last but not least, the environments are simple to solve. This fact enables us to run comprehensive experiments with available academic computational resources as well as compare the results to handcrafted heuristics.

Unfortunately, the `PettingZoo` library only contains a version with discrete action spaces for MPE. More precisely, the agent movement is limited to five actions: NoOp, move left, move right, move upward, move downward. This is inappropriate for us as the majority of our algorithms expect continuous actions. Therefore, we have extended the library with a version incorporating continuous action spaces.[3] Also, we have modified our code with Straight-Through Gumbel-Softmax estimator head. For more details see Subsection 6.4.3.

[1] Alternatively, see repository `https://git.uhlik.me/mff/marl`.

[2] See repository `https://git.uhlik.me/juhlik/metacentrum`.

[3] See repository `https://git.uhlik.me/mff/PettingZoo`.



(a) Cooperative navigation (b) MPE Simple Spread (c) MPE Simple Target

Figure 6.1: Cooperative navigation

6.2.1 Simple Target

As a first *fully cooperative environment* we have used a variation of *Cooperative navigation*, which is also called *Simple Spread*. In order to explain it we quote here [Lowe et al., 2017, p.6]:

> In this environment, agents must cooperate through physical actions to reach a set of N landmarks. Agents observe the relative positions of other agents and landmarks, and are collectively rewarded based on the proximity of any agent to each landmark. In other words, the agents have to "cover" all of the landmarks. Further, the agents occupy significant physical space and are penalized when colliding with each other. Our agents learn to infer the landmark they must cover, and move there while avoiding other agents.

The Figure 6.1a shows this environment for three agents.

The initial experiments showed the incongruity of the *reward function* for this environment. The majority of algorithms failed to solve this environment in reasonable time as only a small subset of agents was able to reach target landmarks. The more detailed investigation of the reward function has shown that after the agent reaches the landmark, she can still be considered as the closest agent also for other landmarks. In other words, the reward area in which agents receive any reward feedback is shrinked significantly as can be seen in Figure 6.1b. This causes vanishing of reward information for other agents and makes the training difficult.

Another found bug was the missing if-condition, which resulted in incorrect collision detection. Simply speaking, agent was penalized because of the collision with herself.

For these reasons, we have proposed an alternative definition of the reward function in which agents receive a global reward for touching a so far unoccupied landmark. However, the agents who are already touching a landmark are omitted from calculating the shortest distance.[1] Again, this situation is captured in Figure 6.1c. We have called this *Simple Target*[2] and we have used it in following experiments.

[1] This change can be seen as a reward shaping of the original environment.

[2] Precisely `simple_target_v0`.

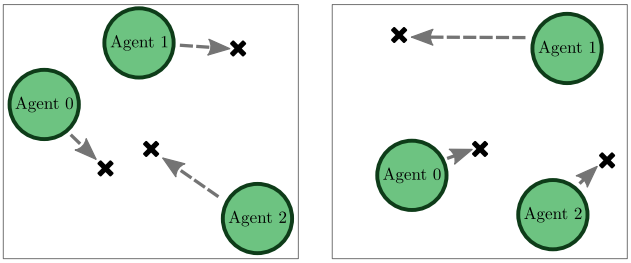


Figure 6.2: MPE Simple Collect

6.2.2 Simple Collect

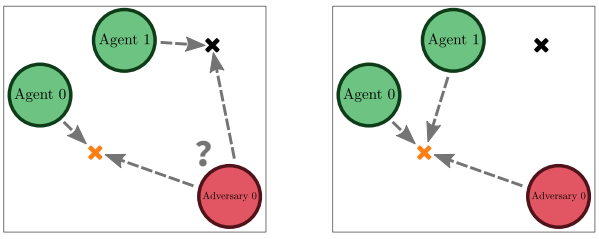
Simple Collect[1] is as an adaptation of Simple Target, where after the agents touch a target landmark, the landmark is relocated to a different position. Therefore, agents' goal is to touch/collect as many landmarks as possible. See Figure 6.2 to gain a visual intuition. Note that we have increased an episode length from 25 to 100 steps in order to highlight the capability differences among the models.

6.2.3 Simple Confuse

In order to represent *competitive* environments we have utilized a *Physical deception* also known as *Simple Adversary*. Again, in order to explain it we quote here [Lowe et al., 2017, p.7]:

> Here, N agents cooperate to reach a single target landmark from a total of N landmarks. They are rewarded based on the minimum distance of any agent to the target (so only one agent needs to reach the target landmark). However, a lone adversary also desires to reach the target landmark; the catch is that the adversary does not know which of the landmarks is the correct one. Thus the cooperating agents, who are penalized based on the adversary distance to the target, learn to spread out and cover all landmarks so as to deceive the adversary.

The Figure 6.3a shows this environment for two agents and one adversary. Similarly to the *Simple Target*, we have modified the reward function so agents get a bonus for touching the landmark. On the other hand, agents are penalized significantly in the case when the adversary touches the correct one. Our modified version used within the experiments is called *Simple Confuse*.[2]



(a) Optimal strategy (b) Suboptimal strategy

Figure 6.3: Physical Deception

[1] Precisely `simply_collect_v0`.
[2] Precisely `simple_confuse_v0`.

6.3 Hyperparameters

We have used the hyperparameters proposed in MATD3 [Ackermann et al., 2019] for all training schemes. In order to set the parameters for Forward-looking Actor we have run a grid-search.[1] The complete list of hyperparameters is in Table 6.1.

We have trained each model for a period of 2000 epochs. Each epoch consists of 250 steps within the environment. As the one episode takes 25 steps, this implies that one epoch contains 10 episodes and in total we run 20000 episodes per the whole training. This is comparable with training setting in MADDPG [Lowe et al., 2017] article, where authors run 25000 episodes. At the beginning of training, rather than using the noisy actor, we sampled a random action for 10000 steps. Finally, we updated our networks every 50 steps.

Name	**Single-Agent Schemes**	**CT-DE**
DDPG hyperparameters		
Hidden layers	$[64, 64]$	—
Activation function	ReLU	—
Discount factor γ	0.95	—
Polyak α	0.99	—
Actor learning rate	10^{-2}	—
Critic learning rate	10^{-2}	—
Replay buffer size	10^6	—
Noise scale	10^{-1}	—
Batch size	1024	—
TD3 hyperparameters		
Target noise	0.2	—
Noise clip	0.5	—
Policy delay	2	—
TD3-FORK hyperparameters		
Reward network learning rate	0.1	—
System network learning rate	0.1	—
Forward ratio λ	$\{0.25, 0.5, \mathbf{0.75}, 1\}$	$\{0.25, \mathbf{0.5}, 0.75, 1\}$
Forward steps K	$\{\mathbf{1}, 2, 3, 4\}$	$\{\mathbf{1}, 2, 3, 4\}$
Forward discount β	$\{\mathbf{0.25}, 0.5, 0.75, 1\}$	$\{\mathbf{0.25}, 0.5, 0.75, 1\}$

Table 6.1: Hyperparameters for algorithms

Name	**Target**	**Collect**	**Confuse**
Number of agents	3	3	2
Number of adversaries	0	0	1
Episode length	25	100	25

Table 6.2: Parameters of environments

[1] The values in grid-search are in curly brackets and the chosen ones are bold.

6.4 Results and Discussion

In order to obtain statistically significant data, we have run the training with 10 different initial random seeds for each algorithm. The Figures 6.4 (respectively 6.5) and 6.6 (respectively 6.7) show the mean training progress for each algorithm. The legend is sorted by the algorithm performance at the end of the training. For more details see Tables 6.3 (respectively 6.4) and 6.5 (respectively 6.6).[1]

6.4.1 Fully Cooperative Environments

For fully cooperative environments, we have compared the experiment results to two handcrafted heuristics. Firstly, *Closest heuristic* simulates the selfish agents who are heading to the closest targets no matter what the others are doing. This is a certainly suboptimal strategy as there is no guarantee of covering all landmarks.

Secondly, *Optimal heuristic* simulates the cooperation among agents. Specifically, each agent navigates herself to the landmark so that all of them are covered and the cumulative distance is minimized.[2]

Simple Target

The Figure 6.4 (respectively 6.5) has shown the mean performance for each training scheme. We can see that in general the CT-DE scheme surpasses SARL schemes. This result has been unsurprising as it only reinforces our intuition that centralized training, which brings more information, helps significantly.

The more detailed Figure A.2 (respectively A.5) has shown the results for SARL algorithms. We can notice that centralized DDPG diverges due to the unsuppressed maximization bias. It is also interesting that FORK version fails to learn when it is mixed with concurrent scheme. We have interpreted these results as the incompetence of the system and reward approximators to predict the environment models accurately because of the limited input information. We need to keep in mind that in this setting these approximators have only access to single agents' observations. Finally, the Parameter Sharing scheme shows poor results in this environment.

In Figure A.3 (respectively A.6) we have presented the results for utilizing the CT-DE algorithms. Yet again, we can see that unmanaged maximization bias reduces the MADDPG performance. This is only highlighted when employing the *soft version*. Beyond this problem, there is no statistically significant performance difference which arises within this environment and all other CT-DE algorithms have solved it.

The detailed look at Table 6.3 (respectively 6.4) has revealed that the CT-DE scheme combined with FORK has more stable results at the beginning of the training. On the other hand, at the end the FORK can deteriorate the training and prevent the smooth environment solving. This is in line with the results in publication [Wei and Ying, 2020] and so we have confirmed their result in importance of λ decay.

[1] In the digital attachment, one can find videos and animations for the best-trained model.

[2] Despite the name, this policy is not truly optimal as it completely omits to incorporate the collision penalty. However, the possible difference is negligible.

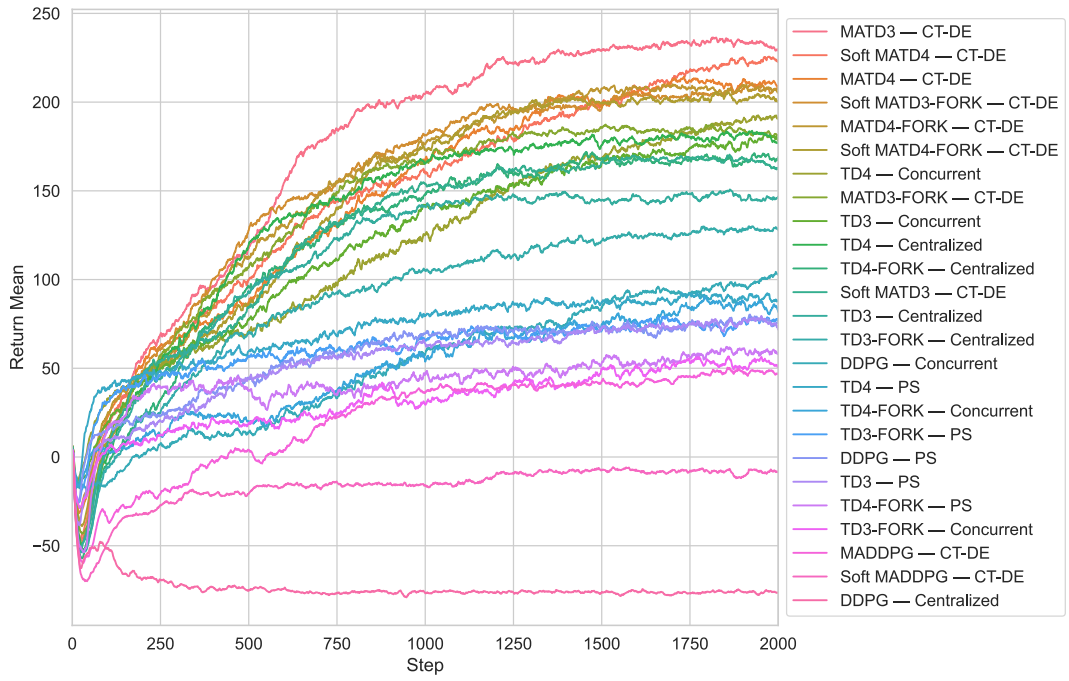


Figure 6.4: Training of MPE Simple Target Continuous

Algorithm — Scheme	**Step 1000**	**Step 1500**	**Step 2000**
MATD3 — CT-DE	**204.55 ± 75.20**	**229.22 ± 61.24**	**229.13 ± 60.49**
Soft MATD4 — CT-DE	160.05 ± 91.58	198.95 ± 60.09	222.70 ± 63.45
MATD4 — CT-DE	167.36 ± 93.70	197.96 ± 83.78	209.31 ± 73.73
Soft MATD3-FORK — CT-DE	181.43 ± 120.70	196.62 ± 115.77	207.12 ± 120.15
MATD4-FORK — CT-DE	176.10 ± 103.08	207.62 ± 63.07	205.87 ± 54.62
Soft MATD4-FORK — CT-DE	173.04 ± 121.90	199.92 ± 115.03	200.56 ± 112.56
TD4 — Concurrent	124.86 ± 56.81	168.19 ± 57.67	191.26 ± 66.95
MATD3-FORK — CT-DE	165.90 ± 109.04	185.39 ± 121.30	181.14 ± 117.56
TD3 — Concurrent	141.68 ± 95.89	166.04 ± 94.70	178.08 ± 70.62
TD4 — Centralized	165.42 ± 99.01	175.77 ± 91.21	177.23 ± 94.04
TD4-FORK — Centralized	147.53 ± 158.98	168.67 ± 162.29	169.09 ± 161.63
Soft MATD3 — CT-DE	153.94 ± 90.68	166.21 ± 118.72	162.48 ± 111.84
TD3 — Centralized	140.86 ± 172.41	144.93 ± 173.38	146.84 ± 173.46
TD3-FORK — Centralized	105.11 ± 155.83	121.86 ± 128.41	127.91 ± 128.45
DDPG — Concurrent	57.89 ± 158.75	84.37 ± 156.04	102.91 ± 163.00
TD4 — PS	79.51 ± 31.69	86.90 ± 24.38	88.14 ± 25.81
TD4-FORK — Concurrent	53.72 ± 52.19	74.92 ± 71.39	81.69 ± 76.66
TD3-FORK — PS	66.37 ± 36.13	71.32 ± 41.40	76.44 ± 31.63
DDPG — PS	69.52 ± 38.57	73.21 ± 29.49	75.89 ± 32.83
TD3 — PS	64.14 ± 99.31	72.93 ± 110.74	71.72 ± 107.94
TD4-FORK — PS	46.26 ± 44.39	52.97 ± 61.03	57.93 ± 46.61
TD3-FORK — Concurrent	29.50 ± 58.41	49.89 ± 69.70	51.17 ± 79.36
MADDPG — CT-DE	36.53 ± 121.71	42.91 ± 125.80	46.68 ± 130.79
Soft MADDPG — CT-DE	-16.09 ± 68.93	-7.41 ± 69.93	-8.69 ± 71.27
DDPG — Centralized	-76.77 ± 43.85	-76.46 ± 43.56	-76.39 ± 40.30

Table 6.3: Training of MPE Simple Target Continuous

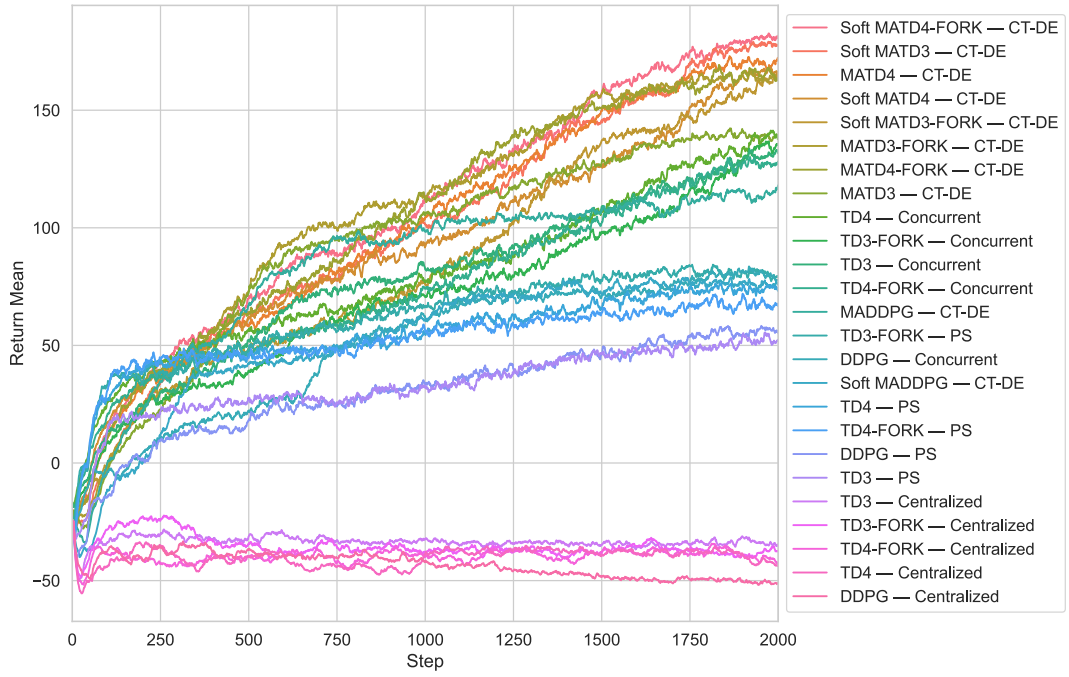


Figure 6.5: Training of MPE Simple Target Discrete

Algorithm — Scheme	**Step 1000**	**Step 1500**	**Step 2000**
Soft MATD4-FORK — CT-DE	111.75 ± 77.96	**157.47 ± 75.75**	**181.75 ± 53.02**
Soft MATD3 — CT-DE	101.32 ± 83.43	145.85 ± 88.63	177.61 ± 89.56
MATD4 — CT-DE	104.05 ± 66.35	150.05 ± 87.94	172.31 ± 46.00
Soft MATD4 — CT-DE	92.24 ± 75.62	126.30 ± 84.29	166.52 ± 98.42
Soft MATD3-FORK — CT-DE	75.79 ± 70.77	137.09 ± 104.59	165.54 ± 84.61
MATD3-FORK — CT-DE	**112.65 ± 65.49**	156.95 ± 78.50	163.67 ± 53.54
MATD4-FORK — CT-DE	111.99 ± 64.69	150.89 ± 85.10	163.32 ± 65.31
MATD3 — CT-DE	105.28 ± 69.32	127.97 ± 67.64	138.51 ± 72.62
TD4 — Concurrent	76.77 ± 42.61	106.56 ± 50.66	137.84 ± 47.28
TD3-FORK — Concurrent	70.46 ± 61.78	98.35 ± 98.07	135.50 ± 84.69
TD3 — Concurrent	82.29 ± 59.64	109.08 ± 57.14	131.59 ± 47.93
TD4-FORK — Concurrent	74.34 ± 50.74	102.12 ± 70.92	128.22 ± 73.07
MADDPG — CT-DE	98.26 ± 92.25	107.20 ± 112.55	118.11 ± 95.47
TD3-FORK — PS	66.57 ± 36.22	77.96 ± 34.42	79.53 ± 29.01
DDPG — Concurrent	61.63 ± 67.56	71.28 ± 70.27	77.90 ± 67.64
Soft MADDPG — CT-DE	60.59 ± 100.51	71.92 ± 106.74	75.72 ± 108.28
TD4 — PS	54.58 ± 41.65	65.69 ± 36.96	74.00 ± 40.12
TD4-FORK — PS	56.13 ± 37.18	63.85 ± 44.07	66.86 ± 34.37
DDPG — PS	33.35 ± 69.91	46.41 ± 78.70	55.19 ± 82.42
TD3 — PS	32.00 ± 73.67	46.71 ± 85.10	52.29 ± 87.22
TD3 — Centralized	-32.49 ± 32.23	-34.73 ± 28.95	-34.45 ± 32.21
TD3-FORK — Centralized	-38.20 ± 34.78	-38.26 ± 31.54	-37.41 ± 32.86
TD4-FORK — Centralized	-39.79 ± 27.42	-37.66 ± 28.74	-42.60 ± 23.18
TD4 — Centralized	-38.51 ± 30.20	-38.95 ± 30.45	-42.84 ± 36.64
DDPG — Centralized	-41.75 ± 29.24	-48.40 ± 31.10	-50.95 ± 31.73

Table 6.4: Training of MPE Simple Target Discrete

6.4.2 Competitive Environments

Besides the previous described heuristics, in case of Simple Confuse we have proposed an additional *Target heuristic*. It describes the situation in which each agent blindly heads to the target location. Obviously, this is suboptimal as the adversary can simply follow these agents.

During the training we fixed the adversary algorithm to the TD3. As the adversary is alone we talk about an ordinary usage of SARL algorithm. Unfortunately, the interpretability of the results is significantly harder in competitive environments because there are at least two (in our case precisely two) models which influence them. It can easily happen that the results are better because the adversary model is not capable of training properly and conversely. In order to deal with this issue, we propose three *adversary heuristics*.

1. *Adversary optimal heuristic*, in which adversary decides what to do based on the number of covered landmarks. If no landmark is occupied, the adversary goes to the landmark, whose sum of distances from the agents is minimal. Otherwise, the adversary goes to the closest covered landmark.
2. *Adversary middle target heuristic*, in which adversary goes to the middle of all landmarks.
3. *Adversary middle agents heuristic*, in which adversary goes to the middle of all agents.

After the training we selected the best trained model for each algorithm and interchanged the adversary policy with these handcraft heuristics. The results can be seen in Table A.1 (respectively A.2). The table is sorted out by the last columns.

Note that the best response to the *Adversary middle target heuristic* is *Target heuristic* as it minimizes the time in which any agent touches the target. Therefore, the models which fail to cover all landmarks and rather cover only the target landmark achieve better results in this column.

Simple Confuse

The Figure 6.6 we have shown that combination of SARL scheme and FORK works poorly in competitive environments. Similarly to the previous case, we blame the deficiency of provided information to the system and reward approximators as the key problem.[1] However, these findings have not been confirmed in discrete case. The Figure 6.7 have shown that in this case the combination reports the average results. In our opinion, the accuracy loss caused by the Gumbel-Softmax paradoxically works as training stabilization and therefore the results are better (see Section 6.4.3).

The Table A.1 (respectively A.2) have confirmed the domination of CT-DE. Quite surprisingly, the combination of TD3 and TD4 with PS scheme works really well. We explain this phenomenon by the low number of agents within the environment.

[1]Moreover, in our environment the goal of adversary goes against the agents goal and so from the perspective of our agents the environment changes even more rapidly. This results in chaotic gradient changes during the training of approximators.

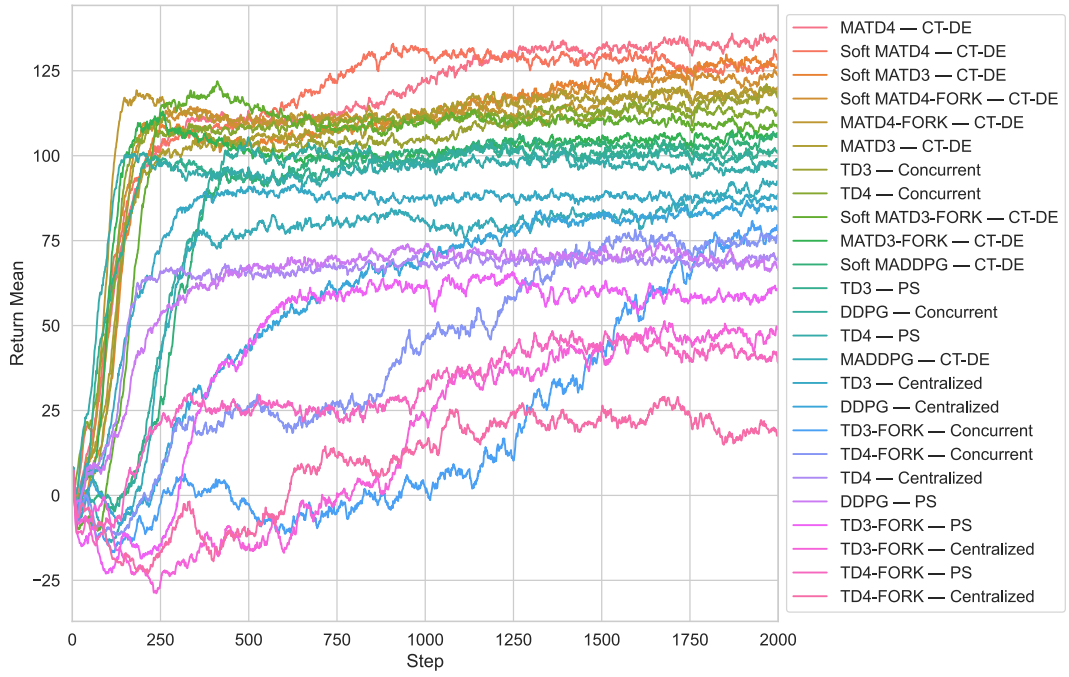


Figure 6.6: Training of MPE Simple Confuse Continuous

Algorithm — Scheme	**Step 1000**	**Step 1500**	**Step 2000**
MATD4 — CT-DE	122.18 ± 33.17	**131.30 ± 30.50**	**133.66 ± 27.11**
Soft MATD4 — CT-DE	**131.06 ± 37.29**	129.13 ± 31.19	127.84 ± 28.16
Soft MATD3 — CT-DE	112.32 ± 29.91	121.77 ± 28.70	126.46 ± 35.92
Soft MATD4-FORK — CT-DE	111.54 ± 21.17	120.62 ± 33.04	123.30 ± 33.84
MATD4-FORK — CT-DE	112.46 ± 31.82	115.03 ± 32.07	119.04 ± 36.35
MATD3 — CT-DE	113.12 ± 30.99	116.09 ± 30.69	118.65 ± 29.63
TD3 — Concurrent	105.01 ± 22.77	111.73 ± 31.89	116.86 ± 33.29
TD4 — Concurrent	111.86 ± 21.07	112.31 ± 21.51	112.78 ± 24.54
Soft MATD3-FORK — CT-DE	111.02 ± 27.48	110.57 ± 32.56	107.96 ± 25.47
MATD3-FORK — CT-DE	101.94 ± 19.59	105.07 ± 22.73	106.14 ± 31.57
Soft MADDPG — CT-DE	100.63 ± 20.79	103.23 ± 24.06	105.31 ± 21.57
TD3 — PS	98.97 ± 128.10	100.34 ± 123.74	100.67 ± 127.73
DDPG — Concurrent	97.02 ± 32.39	100.35 ± 26.67	98.67 ± 24.01
TD4 — PS	96.92 ± 123.85	98.33 ± 120.15	96.89 ± 124.90
MADDPG — CT-DE	80.88 ± 111.29	82.77 ± 105.02	91.45 ± 119.99
TD3 — Centralized	87.51 ± 109.69	87.82 ± 108.53	88.24 ± 102.05
DDPG — Centralized	70.40 ± 109.99	82.32 ± 109.73	83.46 ± 115.35
TD3-FORK — Concurrent	2.43 ± 85.80	42.99 ± 69.73	77.41 ± 37.24
TD4-FORK — Concurrent	45.92 ± 84.69	72.06 ± 90.34	76.21 ± 74.87
TD4 — Centralized	68.44 ± 118.04	68.00 ± 120.47	70.01 ± 116.58
DDPG — PS	73.56 ± 118.83	72.27 ± 124.91	66.27 ± 125.84
TD3-FORK — PS	62.00 ± 114.66	61.09 ± 113.96	60.19 ± 119.57
TD3-FORK — Centralized	22.97 ± 98.41	43.69 ± 93.67	49.01 ± 102.98
TD4-FORK — PS	31.90 ± 118.33	44.34 ± 122.45	38.96 ± 115.48
TD4-FORK — Centralized	14.61 ± 101.07	24.73 ± 104.00	18.10 ± 105.13

Table 6.5: Training of MPE Simple Confuse Continuous

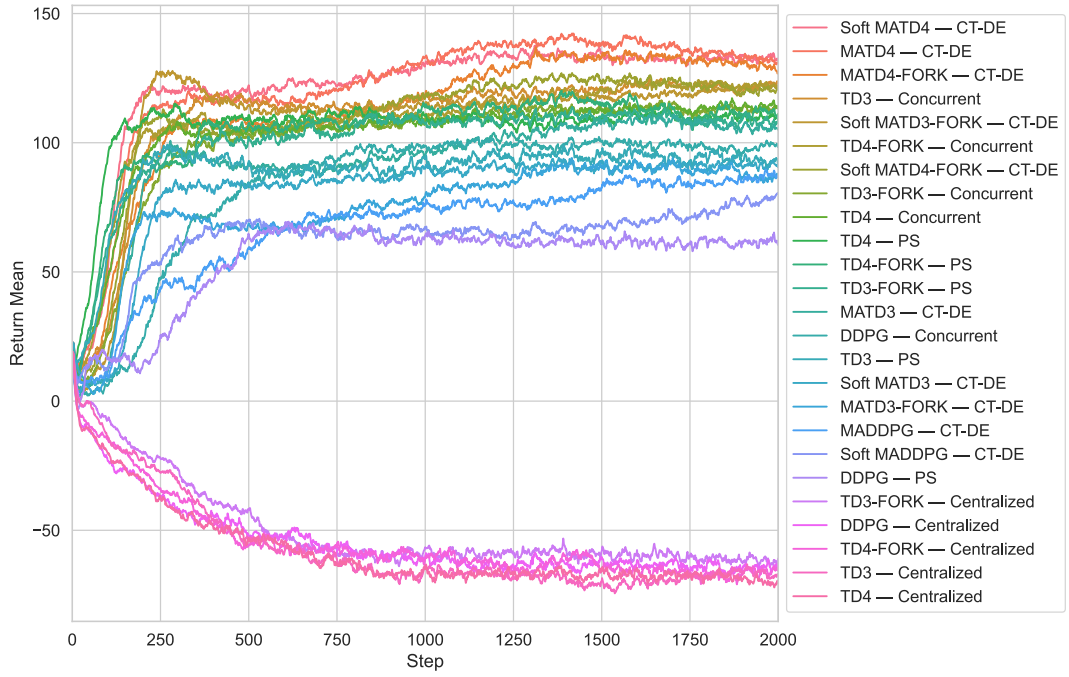


Figure 6.7: Training of MPE Simple Confuse Discrete

Algorithm — Scheme	**Step 1000**	**Step 1500**	**Step 2000**
Soft MATD4 — CT-DE	130.73 ± 64.02	134.33 ± 67.35	**131.67 ± 60.95**
MATD4 — CT-DE	**133.16 ± 62.79**	**137.66 ± 60.18**	130.77 ± 62.34
MATD4-FORK — CT-DE	118.88 ± 30.88	133.05 ± 35.41	126.85 ± 31.52
TD3 — Concurrent	117.66 ± 70.83	121.21 ± 68.82	122.93 ± 73.97
Soft MATD3-FORK — CT-DE	114.68 ± 76.54	117.71 ± 75.47	121.95 ± 73.92
TD4-FORK — Concurrent	110.27 ± 25.70	120.18 ± 32.67	121.65 ± 33.89
Soft MATD4-FORK — CT-DE	117.52 ± 36.84	122.71 ± 40.63	120.39 ± 33.28
TD3-FORK — Concurrent	109.80 ± 28.23	112.24 ± 31.18	113.66 ± 31.04
TD4 — Concurrent	107.70 ± 25.13	112.05 ± 30.07	113.13 ± 32.46
TD4 — PS	110.53 ± 93.17	108.77 ± 95.13	111.26 ± 90.47
TD4-FORK — PS	113.83 ± 136.90	114.20 ± 129.01	108.52 ± 115.81
TD3-FORK — PS	107.67 ± 93.65	109.83 ± 93.05	107.64 ± 96.69
MATD3 — CT-DE	95.42 ± 91.09	107.58 ± 103.40	105.22 ± 104.55
DDPG — Concurrent	98.94 ± 27.85	100.38 ± 26.36	98.39 ± 25.54
TD3 — PS	94.36 ± 148.68	95.72 ± 151.30	92.01 ± 153.11
Soft MATD3 — CT-DE	89.92 ± 105.15	91.97 ± 109.25	91.08 ± 108.21
MATD3-FORK — CT-DE	79.58 ± 108.00	90.22 ± 110.49	87.85 ± 109.62
MADDPG — CT-DE	76.21 ± 79.41	82.52 ± 86.87	87.00 ± 92.56
Soft MADDPG — CT-DE	67.05 ± 78.25	67.61 ± 75.00	80.64 ± 65.77
DDPG — PS	64.00 ± 155.87	60.61 ± 157.34	60.46 ± 159.04
TD3-FORK — Centralized	-56.55 ± 34.65	-58.02 ± 33.39	-60.90 ± 40.31
DDPG — Centralized	-59.94 ± 37.19	-60.30 ± 36.79	-64.07 ± 38.17
TD4-FORK — Centralized	-60.94 ± 33.82	-65.66 ± 32.61	-65.76 ± 32.89
TD3 — Centralized	-63.76 ± 30.93	-69.51 ± 28.80	-67.00 ± 32.73
TD4 — Centralized	-65.01 ± 34.23	-65.31 ± 34.02	-70.42 ± 35.32

Table 6.6: Training of MPE Simple Confuse Discrete

6.4.3 Effect of Continuous/Discrete Action Space

We have also been interested in how well the algorithms based on DDPG, which are by default applicable only to the continuous action space, stand in environment with discrete action space. The standard workaround is to equip it by the Straight-Through Gumbel-Softmax as we have already described in Section 2.6.

The comparison of Figures 6.4 and 6.5, respectively Tables 6.3 and 6.4, has shown that the training with Gumbel-Softmax is possible but significantly slower. At first sight, this can be unintuitive but we need to keep in mind that by sending gradient information "straight-through" we lose potential valuable information about the optimal optimization direction.

Also note that in our implementation we have not reduced the exponential overhead by the utilizing the factorized version of the centralized controller as described in Subsection 3.1.1. This causes the poor performance results of the centralized scheme in environments with discrete actions spaces. We are aware of this limitation but the unpromising properties of this scheme have led us to miss it out.

6.4.4 Effect of Modifications

In the previous chapters we have described several modifications for the discussed algorithms. Unfortunately, none of those modifications has overperformed the original algorithm in all measurements. On the other hand, the collected data has shown, that using the minimal value of both critics even during the actor update (the DDPG++ approach) works mostly better. As this modification is relatively simple to do and there is no additional slowdown of the training, we highly recommend to use this modification.

The *soft version* has shown a similar inconclusive improvement as the previous discussed modification. But in this case we must be careful about the training time overhead which does increase linearly with the number of agents. This was not problem in our experiments, in which we have kept the number of agents low, but can be significant in other scenarios. Therefore in general, we cannot utter any universal judgment.

6.4.5 Effect of FORK

The most interesting part has shown to be the effect of FORK to the algorithms performance. We have identified that it is essential to provide to the transition and reward approximator as many information as possible. By this observation, we explain the inferior performance of the PS and concurrent schemes compared to the centralized and CT-DE. This is only highlighted in the case of *competitive environments*, in which the changes in adversary's policies make the training even more difficult.

For the *fully cooperative environments*, the FORK has shown its potential in stabilizing the initial part of the training. Unfortunately, this has not been confirmed for the *competitive environments*. As before, we blame the "increased non-stationarity" of the environments from the agents point of view, which is caused by changes of adversary's policies.

Finally, we must not forget to mention the unobtrusive training deceleration caused by the additional approximators. This can a be significant hindrance in its potential application.

6.5 Acknowledgement

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Conclusion

This work brings a comprehensive overview of the knowledge from the Multi-Agent RL discipline. We emphasize the often neglected formalism which, as we believe, is essential to every solid scientific research. Moreover, we bring a unified notation which cherry picks the best concepts from the already proposed notations. We strongly believe that this work will serve as the brief yet not oversimplified introduction to this problematic.

We utilize this formalism and notation in a description of the most influential algorithms for Single-Agent RL, as well as Multi-Agent RL. Again, our narration starts with the simplest algorithms which are gradually complexified. This process was at most important as it enabled us to discover the miscellaneous variations of these algorithms (namely TD4-FORK, soft-MATD3, MATD4 and soft-MATD4). As far as we know, these variations have not been discussed in any scientific publications so far.

We also propose a novel model architecture called MATD3 Forward-looking Actor (MATD3-FORK) as a combination of two successful algorithms, namely MATD3 and TD3-FORK. It is built on the Centralized Training with Decentralized Execution approach, which is by the community identified as the most promising scheme for solving RL environments with multiple agents. Similarly to the previous algorithms, also in this place we propose several possible modifications.

In order to compare these algorithms justly, we have decided to reimplement them, rather than use a third party implementation. Only by this decision we can be sure of the equivalent conditions. The well documented CLI is bundled with it so other practitioners can easily reproduce and/or extend our experiments. As the testing environments we have chosen the Multi-Particle Environments from `PettingZoo` library. We have significantly extended this library by the support of continuous actions. Also, we have identified and fixed several issues in already implemented scenarios. Finally, we have extended this library with new environments which, as we believe, serve best in their performance distinction.

We have run numerous experiments, which would not have been possible without the fantastic service of MetaCentrum administrators. However, the inconsiderable overhead forced us to create several scripts as side project. We share these in a public repository in hope that it will be helpful also to the other researchers. Experiments unambiguously show that the CT-DE is the correct way in further investigation as it defeats all other proposed schemes. We have also confirmed the necessity of the suppression of the maximization bias even in multi-agent environments.

Our proposed model shows the comparable results with other SOTA algorithms when applied to the multi-agent environments. We have identified its potential in stabilizing the early phases of the training. It is especially useful for environments with discrete action spaces. On the other hand, when applied to the competitive environment the FORK causes training instabilities as the transition model of the environment depends on hidden actions of adversary agents. Moreover, the training deceleration caused by the additional approximators is significant. This can be serious downside in applying our model to the environments

with more agents.

Finally, in this work we have completely omitted to mention the Soft Actor-Critic (SAC) [Haarnoja et al., 2018], which gives comparable results to TD3. In any future work, we would like to see a utilization of this algorithm to the multi-agent environments and see comparison with TD3 approach, which we have investigated here. There is also an interesting combination of SAC together with Transformer architecture [Vaswani et al., 2017], which results in Multi-Agent Actor-Attention-Critic (MAAC) [Iqbal and Sha, 2019]. Another possibility of further a research direction can be a combination of MATD3 with attention mechanism.

Bibliography

[Achiam, 2018] Achiam, J. (2018). Spinning up in deep reinforcement learning. *GitHub repository*. `https://spinningup.openai.com/en/latest/`, visited on 2021-03-19.

[Ackermann et al., 2019] Ackermann, J., Gabler, V., Osa, T., and Sugiyama, M. (2019). Reducing overestimation bias in multi-agent domains using double centralized critics. *ArXiv e-prints*.

[Albrecht and Ramamoorthy, 2013] Albrecht, S. V. and Ramamoorthy, S. (2013). A game-theoretic model and best-response learning method for ad hoc coordination in multiagent systems. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems*, AAMAS '13, page 1155–1156, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

[Bellemare et al., 2013] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

[Bellman, 1957] Bellman, R. (1957). A Markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684.

[Bernstein et al., 2002] Bernstein, D. S., Givan, R., Immerman, N., and Zilberstein, S. (2002). The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research*, 27(4):819–840.

[Boutilier, 1996] Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. In *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge*, TARK '96, pages 195–210, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI gym. *ArXiv e-prints*.

[Campbell et al., 2002] Campbell, M., Hoane, A. J., and Hsu, F.-h. (2002). Deep Blue. *Artificial Intelligence*, 134(1):57–83.

[Clemente et al., 2017] Clemente, A. V., Castejón, H. N., and Chandra, A. (2017). Efficient parallel methods for deep reinforcement learning. *ArXiv e-prints*.

[Devlin et al., 2019] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

[Dhariwal et al., 2017] Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. (2017). OpenAI baselines. *GitHub repository*. `https://github.com/openai/baselines`, visited on 2021-03-19.

[Fakoor et al., 2020] Fakoor, R., Chaudhari, P., and Smola, A. J. (2020). DDPG++: Striving for simplicity in continuous-control off-policy reinforcement learning. *ArXiv e-prints*.

[Foerster et al., 2017] Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P. H. S., Kohli, P., and Whiteson, S. (2017). Stabilising experience replay for deep multi-agent reinforcement learning. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1146–1155. PMLR.

[Foerster, 2018] Foerster, J. N. (2018). *Deep multi-agent reinforcement learning*. PhD thesis, University of Oxford.

[Fujimoto et al., 2018] Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. *ArXiv e-prints*.

[Girshick, 2015] Girshick, R. (2015). Fast R-CNN. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448.

[Gupta et al., 2017] Gupta, J. K., Egorov, M., and Kochenderfer, M. (2017). Cooperative multi-agent control using deep reinforcement learning. In Sukthankar, G. and Rodriguez-Aguilar, J. A., editors, *Autonomous Agents and Multiagent Systems*, pages 66–83, Cham. Springer International Publishing.

[Haarnoja et al., 2018] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR.

[Hansen et al., 2004] Hansen, E. A., Bernstein, D. S., and Zilberstein, S. (2004). Dynamic programming for partially observable stochastic games. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pages 709–715. AAAI Press.

[Hasselt et al., 2016] Hasselt, H. v., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

[Hessel et al., 2018] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

[Iqbal and Sha, 2019] Iqbal, S. and Sha, F. (2019). Actor-attention-critic for multi-agent reinforcement learning. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2961–2970. PMLR.

[Jang et al., 2016] Jang, E., Gu, S., and Poole, B. (2016). Categorical reparameterization with Gumbel-Softmax. *ArXiv e-prints*.

[Koul, 2019] Koul, A. (2019). ma-gym: Collection of multi-agent environments based on OpenAI gym. *GitHub repository*. `https://github.com/koulanurag/ma-gym`, visited on 2021-03-19.

[Krizhevsky et al., 2017] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90.

[Li, 2020] Li, M. (2020). Machin. *GitHub repository*. `https://github.com/iffiX/machin`, visited on 2021-03-19.

[Liang et al., 2017] Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., and Stoica, I. (2017). RLlib: Abstractions for distributed reinforcement learning. *ArXiv e-prints*.

[Lillicrap et al., 2016] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. In *ICLR (Poster)*.

[Lowe et al., 2017] Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., and Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6382–6393, Red Hook, NY, USA. Curran Associates Inc.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA. PMLR.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[OroojlooyJadid and Hajinezhad, 2019] OroojlooyJadid, A. and Hajinezhad, D. (2019). A review of cooperative multi-agent deep reinforcement learning. *ArXiv e-prints*.

[Pineau et al., 2006] Pineau, J., Gordon, G., and Thrun, S. (2006). Anytime point-based approximations for large pomdps. *J. Artif. Int. Res.*, 27(1):335–380.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

[Schrittwieser et al., 2020] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. (2020). Mastering Atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609.

[Schulman et al., 2016] Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[Shapley, 1953] Shapley, L. S. (1953). Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

[Silver et al., 2018] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

[Silver et al., 2014] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In Xing, E. P. and Jebara, T., editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China. PMLR.

[Sunehag et al., 2017] Sunehag, P., Lever, G., Gruslys, A., Czarnecki, W. M., Zambaldi, V., Jaderberg, M., Lanctot, M., Sonnerat, N., Leibo, J. Z., Tuyls, K., and Graepel, T. (2017). Value-decomposition networks for cooperative multi-agent learning. *ArXiv e-prints*.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

[Tan and Le, 2019] Tan, M. and Le, Q. V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. *ArXiv e-prints*.

[Tan et al., 2020] Tan, M., Pang, R., and Le, Q. V. (2020). EfficientDet: Scalable and efficient object detection. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10778–10787.

[Terry et al., 2020a] Terry, J. K., Black, B., Jayakumar, M., Hari, A., Sullivan, R., Santos, L., Dieffendahl, C., Williams, N. L., Lokesh, Y., Horsch, C., and Ravi, P. (2020a). Pettingzoo: Gym for multi-agent reinforcement learning. *ArXiv e-prints*.

[Terry et al., 2020b] Terry, J. K., Grammel, N., Black, B., Hari, A., Horsch, C., and Santos, L. (2020b). Agent environment cycle games. *ArXiv e-prints*.

[Tesauro, 1995] Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *J. Int. Comput. Games Assoc.*, 18:88.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

[Wei et al., 2018] Wei, E., Wicke, D., Freelan, D., and Luke, S. (2018). Multiagent soft Q-Learning. *ArXiv e-prints*.

[Wei and Ying, 2020] Wei, H. and Ying, L. (2020). Fork: A forward-looking actor for model-free reinforcement learning. *ArXiv e-prints*.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256.

[Williams and Baird, 1993] Williams, R. J. and Baird, L. C. (1993). Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems. *Northeastern University*.

List of Figures

List of Algorithms

List of Abbreviations

A2C Advantage AC.
A3C Asynchronous Advantage AC.
AC Actor-Critic.
AEC Agent-Environment Cycle.
AECG Agent-Environment Cycle Game.
AI Artificial Intelligence.
API Application Programming Interface.

CLI Command-Line Interface.
CNN Convolutional Neural Network.
CPU Central Processing Unit.
CT-DE Centralized Training with Decentralized Execution.

DANN Deep Artificial Neural Network.
DDPG Deep Deterministic Policy Gradient.
DDPG++ Deep Deterministic Policy Gradient Plus Plus.
dec-POMDP Decentralized Partially Observable Markov Decision Process.
DL Deep Learning.
DPGT Deterministic Policy Gradient Theorem.
DQN Deep Q Network.

FORK Forward-looking Actor.

GAE Generalized Advantage Estimation.
GPU Graphics Processing Unit.

MAAC Multi-Agent Actor-Attention-Critic.
MADDPG Multi-Agent Deep Deterministic Policy Gradient.
MARL Multi-Agent RL.
MATD3 Multi-Agent Twin Delayed DDPG.
MATD3-FORK MATD3 Forward-looking Actor.
MATD4 Multi-Agent TD4.
MATD4-FORK MATD4 Forward-looking Actor.
MC Monte Carlo.
MCTS Monte Carlo Tree Search.
MDP Markov Decision Process.
MMDP Multi-Agent Markov Decision Process.
MPE Multi-Particle Environments.
MSE Mean Squared Error.

NLP Natural Language Processing.
NN Neural Network.
NoOp No Operation.

PAAC Parallel Advantage AC.
PGT Policy Gradient Theorem.
POMDP Partially Observable Markov Decision Process.
POSG Partially Observable Stochastic Game.

PS Parameter Sharing.

RL Reinforcement Learning.
RNN Recurrent Neural Network.

SAC Soft Actor-Critic.
SARL Single-Agent RL.
SG Stochastic Game.
soft-MADDPG Soft MADDPG.
soft-MATD3 Soft MATD3.
soft-MATD3-FORK Soft MATD3-FORK.
soft-MATD4 Soft MATD4.
soft-MATD4-FORK Soft MATD4-FORK.
SOTA state-of-the-art.

TD Temporal Difference.
TD3 Twin Delayed DDPG.
TD3-FORK TD3 Forward-looking Actor.
TD4 Twin Delayed DDPG++.
TD4-FORK TD4 Forward-looking Actor.
TPU Tensor Processing Unit.

Digital Attachments

The latest versions of our source codes can be found in the following repositories:

- `https://git.uhlik.me/mff/master-thesis`
- `https://git.uhlik.me/mff/marl`
- `https://git.uhlik.me/mff/PettingZoo`

Note that our modification for `PettingZoo` is stored in the `develop` branch.

```
src
├── master-thesis.tar.bz2 ..... source code of this thesis in LuaLaTeX
│   ├── thesis.tex ..... main file in LuaLaTeX
│   ├── bibliography.bib ..... bibliography file in bibTex format
│   ├── chapters ..... chapters of this thesis in LuaLaTeX
│   ├── figures ..... figures in PNG, SVG and PDF format
│   ├── tables ..... automatically generated tables in LuaLaTeX
│   └── utils ..... various utilities in LuaLaTeX
├── marl.tar.bz2 ..... source code of our implementation
│   ├── bin ..... various automation BASH scripts
│   ├── config ..... examples of configuration file for MetaCentrum
│   ├── marl ..... our implementation with RL algorithms
│   ├── notebook ..... Jupyter Notebooks for automatic plot generation
│   ├── record ..... videos and animations with successful runs
│   └── test ..... tests for our implementation
└── PettingZoo.tar.bz2 ..... source code of our PettingZoo modification
MT_Uhlik_Jan_2021.pdf ..... this thesis in PDF format
```

A. Appendix

A.1 Framework Examples

```python
import gym

env = gym.make("CartPole-v1")
obs, max_cycles = env.reset(), 1000
for _ in range(max_cycles):
  env.render()
  act = policy(obs)
  obs, rew, done, info = env.step(action)

  if done:
    obs = env.reset()
```

Listing A.1: `gym` code example

```python
import gym

env = gym.make('ma_gym:Switch2-v0')
done_n = [False for _ in range(env.n_agents)]
obs_n, max_cycles = env.reset(), 1000
for _ in range(max_cycles):
    env.render()
    act_n = [policy(obs_n[agent], agent)
             for agent in env.n_agents]
    obs_n, reward_n, done_n, info = env.step(act_n)

    if all(done):
      obs_n = env.reset()
```

Listing A.2: `ma-gym` code example

```
from pettingzoo.mpe import simple_spread_v2

env = simple_spread_v2.env()
env.reset()
max_cycles = 1000
for agent in env.agent_iter(max_cycles):
    env.render()
    observation, reward, done, info = env.last()
    action = policy(observation, agent)
    env.step(action)
```

Listing A.3: PettingZoo AEC API code example

```
from pettingzoo.mpe import simple_spread_v2

parallel_env = simple_spread_v2.parallel_env()
observations = parallel_env.reset()
max_cycles = 1000
for step in range(max_cycles):
    parallel_env.render()
    actions = {agent: policy(observations[agent], agent)
               for agent in parallel_env.agents}
    observations, rewards, dones, infos = parallel_env.step(actions)
```

Listing A.4: PettingZoo parallel API code example

A.2 Best Models Comparison

For further investigation, we have taken the models which perform the best among all seeds and run 100 test episodes. In order to equitably compare the performances we have fixed the environment seed to 42 so every model is tested on the very same environment instances. The results are shown in comparative box plots. We have highlighted mean by a green triangle, whereas the red horizontal line shows the maximal median performance.

A.2.1 Simple Target

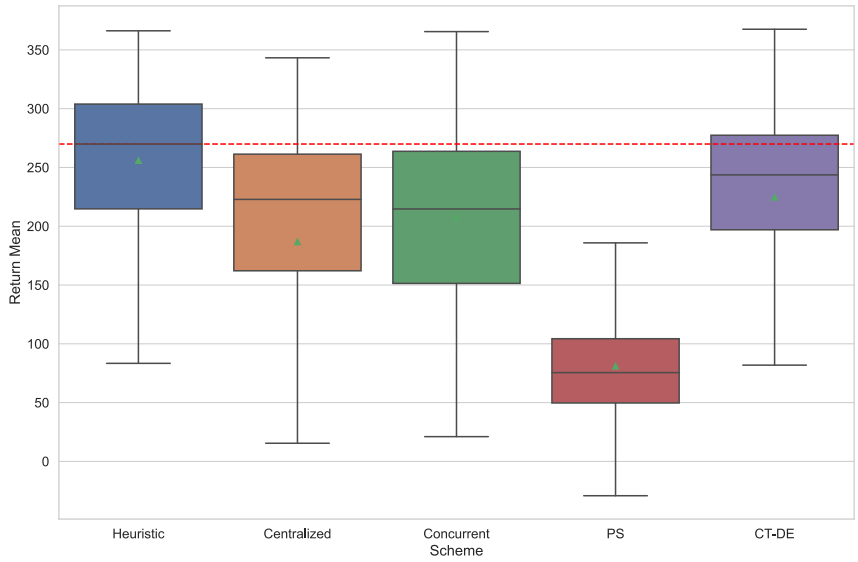


Figure A.1: Schemes comparison for MPE Simple Target Continuous

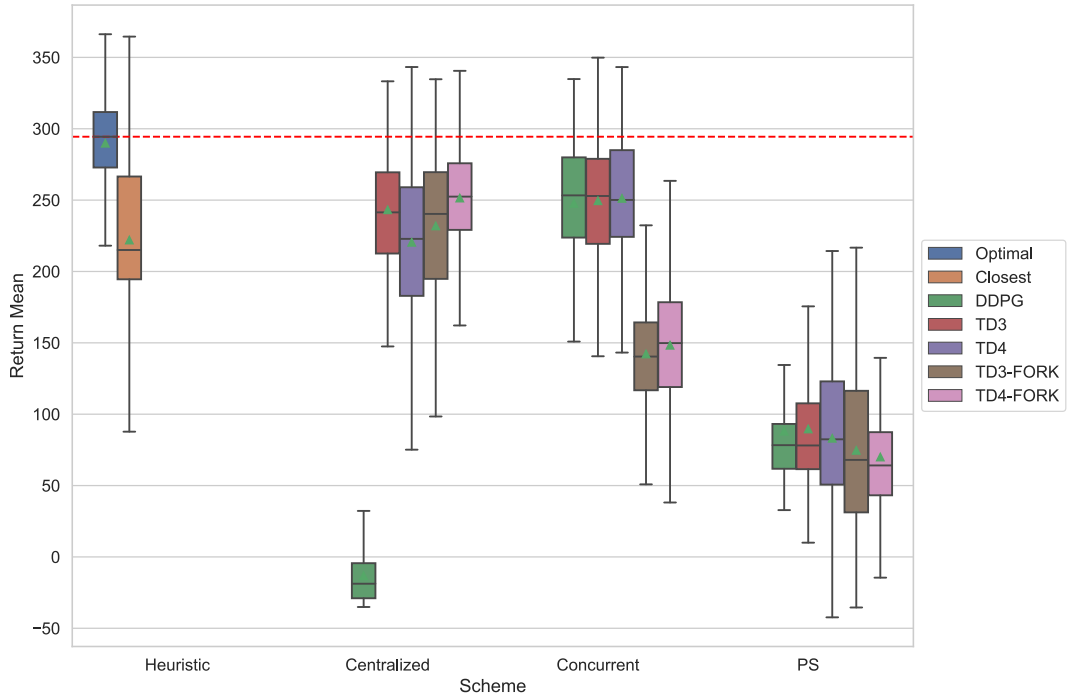


Figure A.2: SARL comparison for MPE Simple Target Continuous

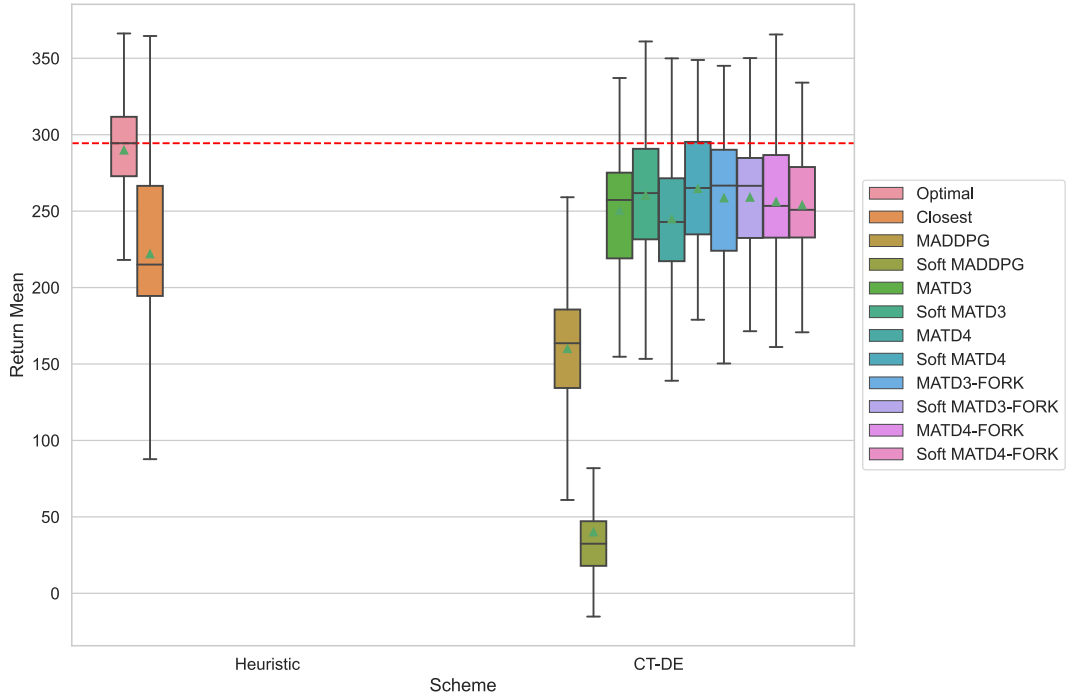


Figure A.3: CT-DE comparison for MPE Simple Target Continuous

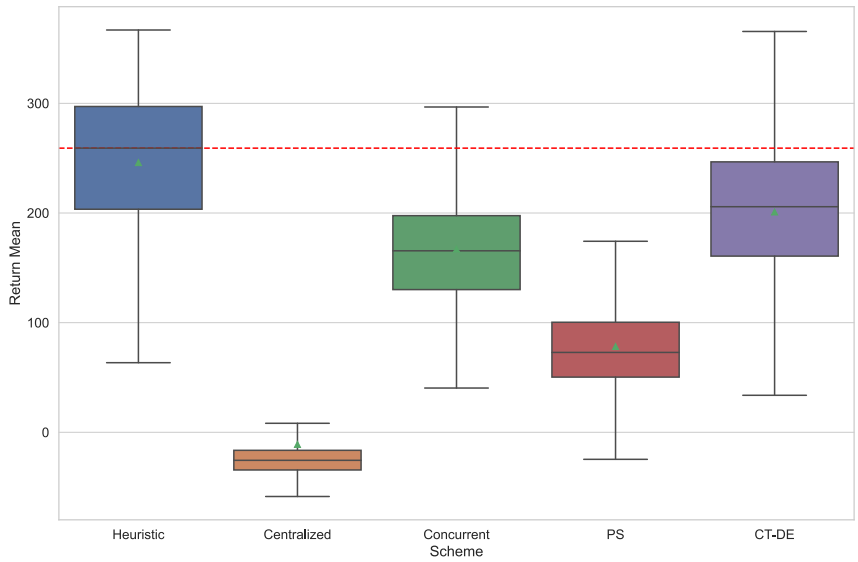


Figure A.4: Schemes comparison for MPE Simple Target Discrete

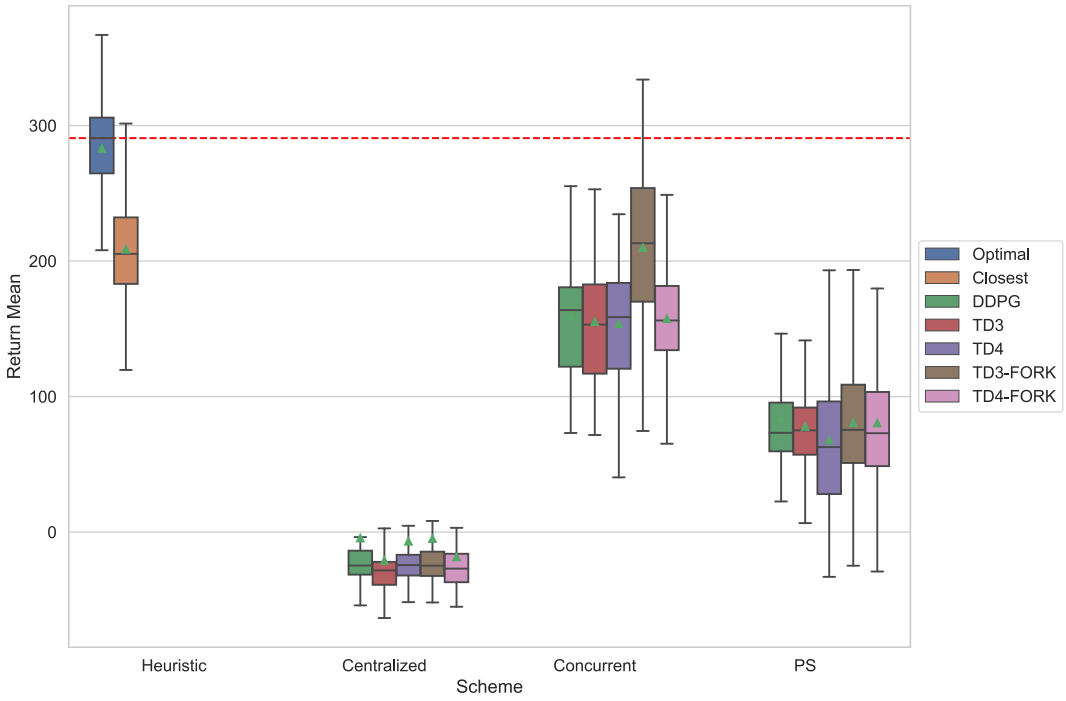


Figure A.5: SARL comparison for MPE Simple Target Discrete

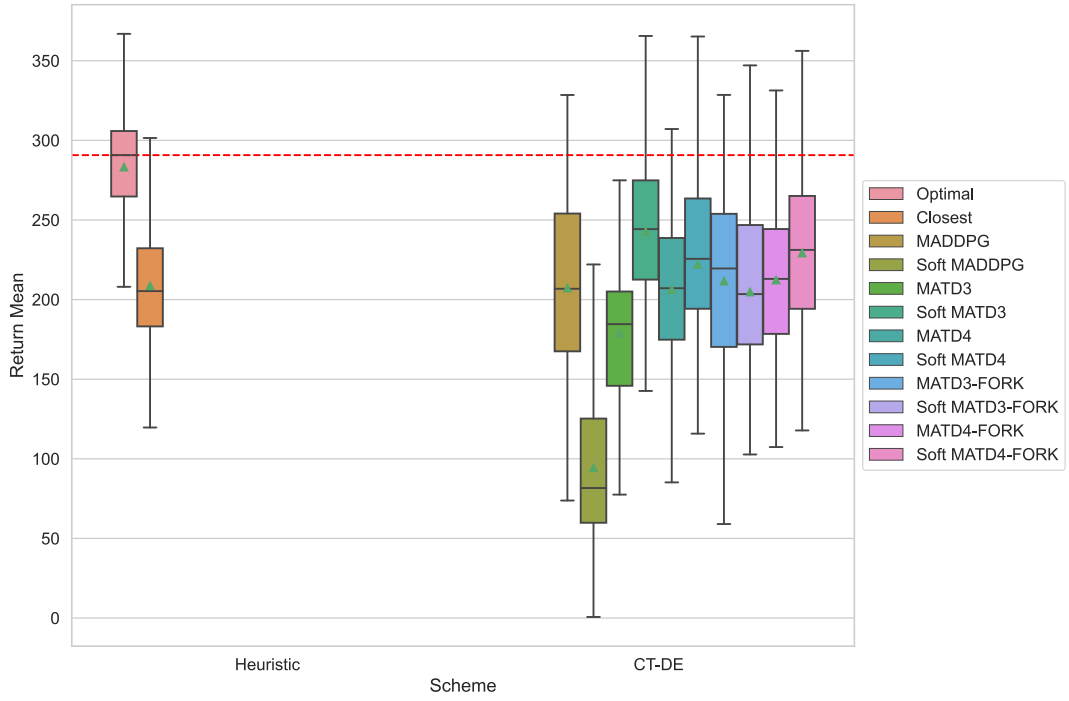


Figure A.6: CT-DE comparison for MPE Simple Target Discrete

A.2.2 Simple Collect

In order to culminate and highlight the performance differences we have taken the train models from *Simple Target* and evaluated their performances on *Simple Collect* environment. The results can be seen in Figures A.7, A.8 and A.9. Note that we have omitted the *Centralized DDPG* and *Concurrent TD4-FORK* as their performance is unsatisfactory and causes the unreadability of other results.

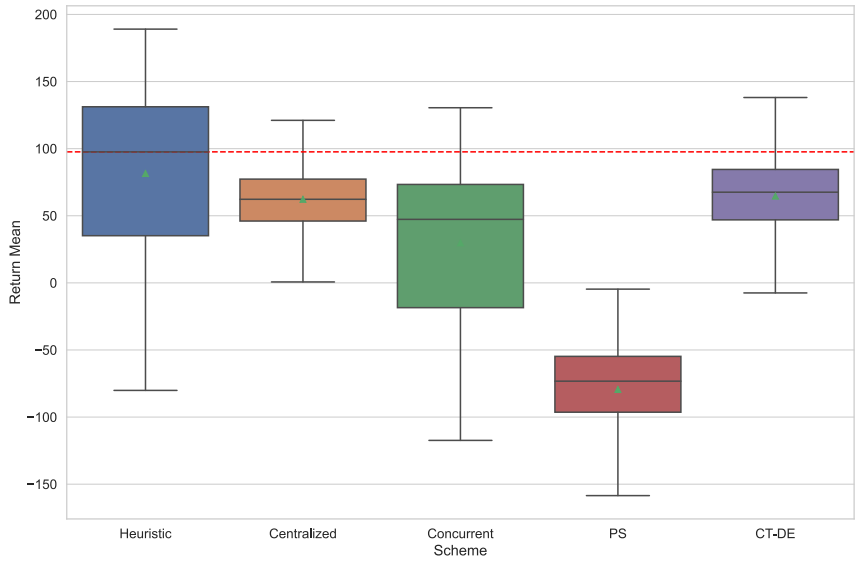


Figure A.7: Schemes comparison for MPE Simple Collect Continuous

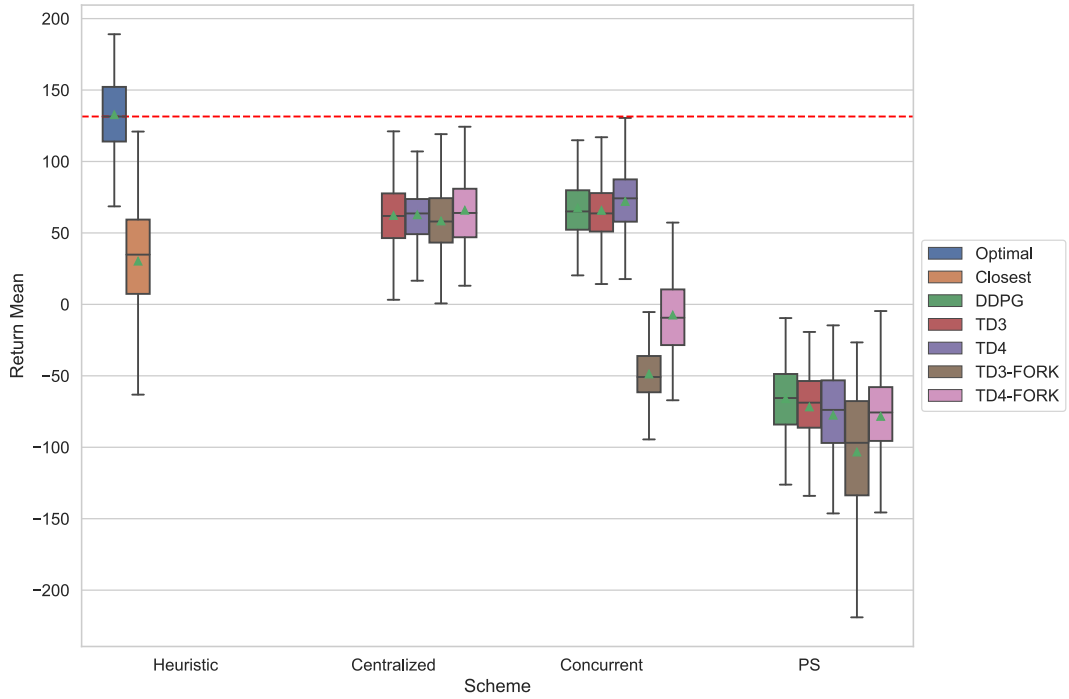


Figure A.8: SARL comparison for MPE Simple Collect Continuous

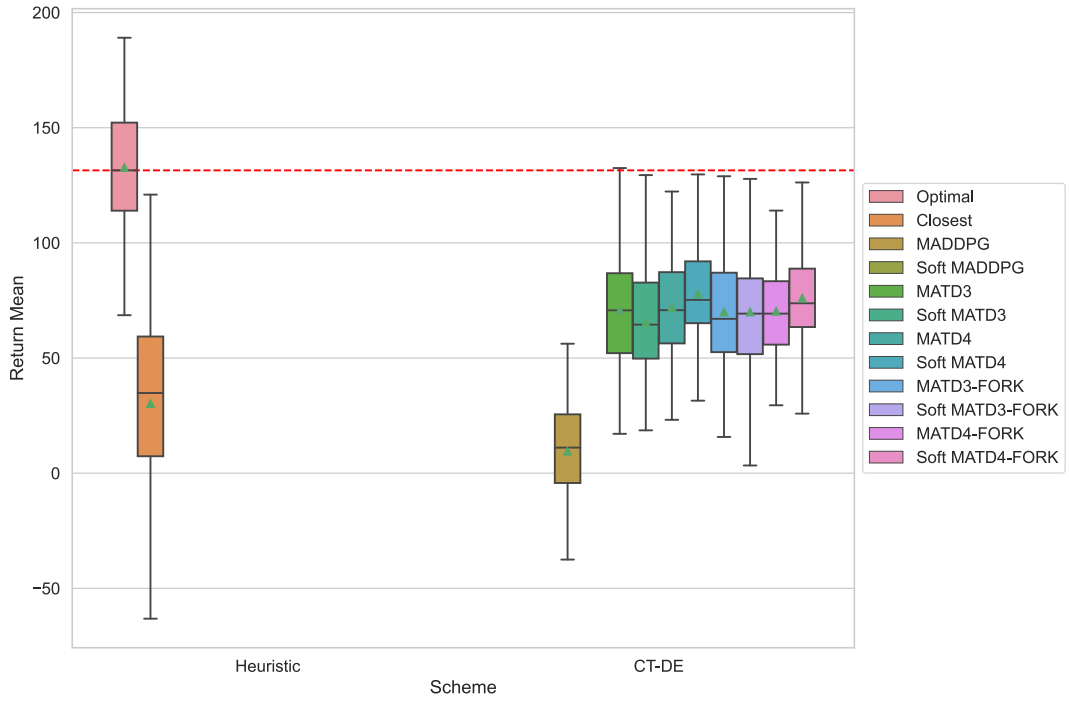


Figure A.9: CT-DE comparison for MPE Simple Collect Continuous

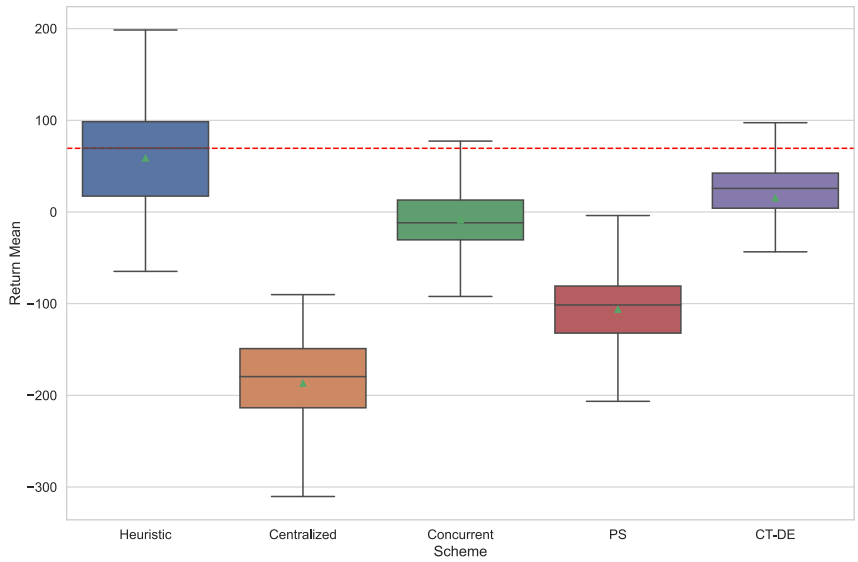


Figure A.10: Schemes comparison for MPE Simple Collect Discrete

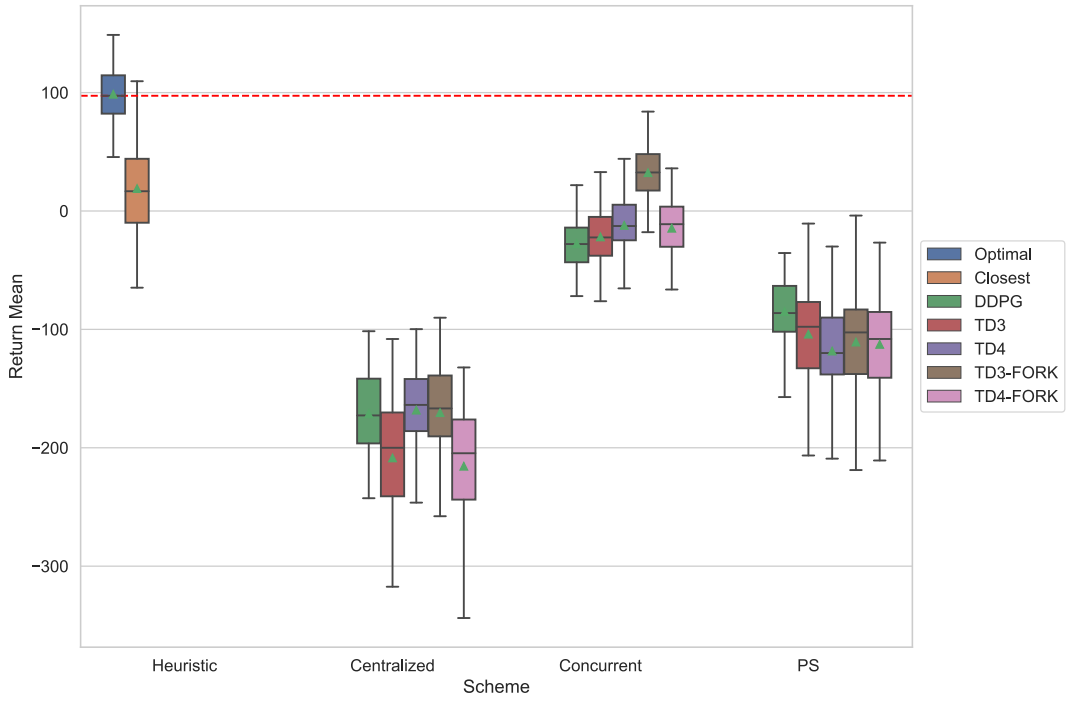


Figure A.11: SARL comparison for MPE Simple Collect Discrete

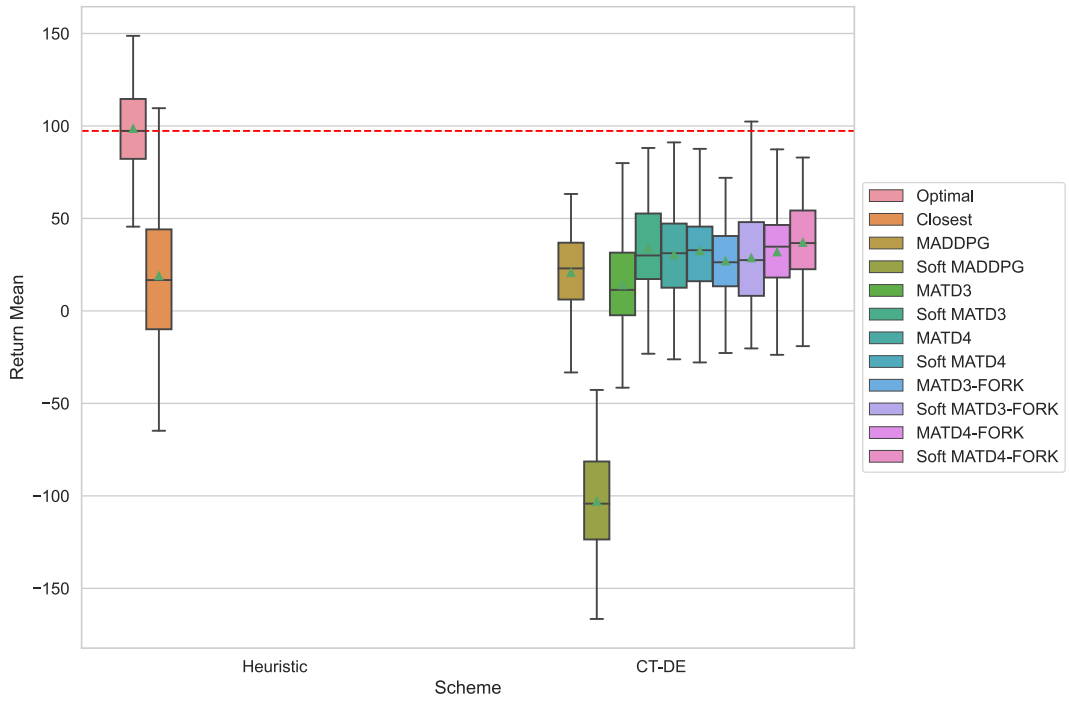


Figure A.12: CT-DE comparison for MPE Simple Collect Discrete

A.2.3 Simple Confuse

Algorithm — Scheme	Mid Target	Mid Agent	Optimal	Average
Agent Optimal — Heuristic	181.71	**187.00**	**144.24**	**170.98**
Soft MATD3-FORK — CT-DE	178.65	177.44	112.83	156.31
DDPG — PS	186.81	160.79	116.42	154.67
MATD3 — CT-DE	176.19	178.29	104.65	153.04
TD3 — PS	174.38	175.51	109.02	152.97
MATD4-FORK — CT-DE	174.04	178.28	102.14	151.49
MATD4 — CT-DE	178.31	165.92	105.64	149.96
Soft MATD4 — CT-DE	175.03	164.10	110.12	149.75
Agent Target — Heuristic	**193.68**	135.03	117.98	148.90
TD3 — Concurrent	185.52	148.51	111.88	148.64
Soft MATD3 — CT-DE	170.41	173.16	101.51	148.36
TD4 — PS	189.42	136.72	115.92	147.35
Soft MATD4-FORK — CT-DE	171.16	164.44	98.63	144.74
MATD3-FORK — CT-DE	170.29	162.38	97.87	143.51
TD4 — Concurrent	183.93	132.11	110.15	142.06
DDPG — Concurrent	172.93	141.25	104.35	139.51
TD4 — Centralized	179.15	131.99	103.89	138.34
Soft MADDPG — CT-DE	170.00	131.86	100.25	134.04
DDPG — Centralized	170.69	124.40	98.66	131.25
TD3 — Centralized	165.05	130.23	96.83	130.70
TD4-FORK — Centralized	167.56	120.82	98.34	128.91
TD4-FORK — Concurrent	158.62	118.63	94.61	123.95
TD3-FORK — Centralized	156.73	121.13	89.77	122.54
Agent Closest — Heuristic	133.47	123.27	99.30	118.68
MADDPG — CT-DE	147.37	116.77	85.85	116.66
TD3-FORK — PS	155.57	113.58	80.70	116.62
TD4-FORK — PS	158.68	105.14	83.93	115.92
TD3-FORK — Concurrent	133.51	103.64	76.33	104.50

Table A.1: Different adversary heuristics for MPE Simple Confuse Continuous

Algorithm — Scheme	Mid Target	Mid Agent	Optimal	Average
Agent Optimal — Heuristic	173.15	178.50	**133.33**	**161.66**
TD4-FORK — PS	173.64	**183.89**	109.21	155.58
MATD3-FORK — CT-DE	172.53	174.32	110.67	152.51
Soft MATD4-FORK — CT-DE	173.00	172.22	111.67	152.30
MATD4-FORK — CT-DE	167.28	179.88	108.95	152.04
TD3-FORK — Concurrent	173.27	163.06	106.58	147.64
MATD3 — CT-DE	171.63	162.75	107.75	147.37
TD4-FORK — Concurrent	170.77	162.13	105.66	146.19
TD4 — PS	162.87	169.13	105.81	145.94
Soft MATD3 — CT-DE	171.02	154.14	108.85	144.67
TD3-FORK — PS	166.57	165.24	101.05	144.29
TD4 — Concurrent	166.97	156.10	108.84	143.97
Agent Target — Heuristic	**182.83**	132.05	114.61	143.16
Soft MADDPG — CT-DE	178.59	127.75	110.51	138.95
DDPG — PS	176.22	127.57	108.20	137.33
TD3 — PS	169.51	137.18	104.66	137.12
Soft MATD3-FORK — CT-DE	166.84	137.70	102.37	135.63
TD3 — Concurrent	169.75	132.09	103.27	135.04
MADDPG — CT-DE	154.21	151.95	95.89	134.02
Soft MATD4 — CT-DE	138.94	172.20	79.91	130.35
MATD4 — CT-DE	152.68	139.94	90.11	127.58
DDPG — Concurrent	147.48	145.16	89.36	127.33
Agent Closest — Heuristic	131.27	121.96	95.30	116.18
DDPG — Centralized	-12.01	14.10	-39.74	-12.55
TD3-FORK — Centralized	-17.97	11.79	-46.37	-17.51
TD4 — Centralized	-20.19	12.65	-49.18	-18.91
TD3 — Centralized	-30.73	4.58	-57.14	-27.76
TD4-FORK — Centralized	-28.21	-0.86	-59.32	-29.46

Table A.2: Different adversary heuristics for MPE Simple Confuse Discrete

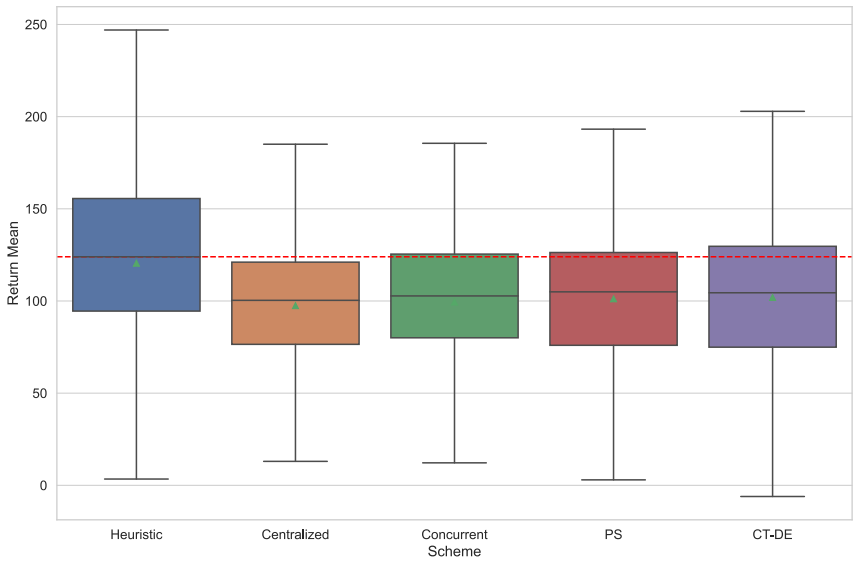


Figure A.13: Schemes comparison for MPE Simple Confuse Continuous

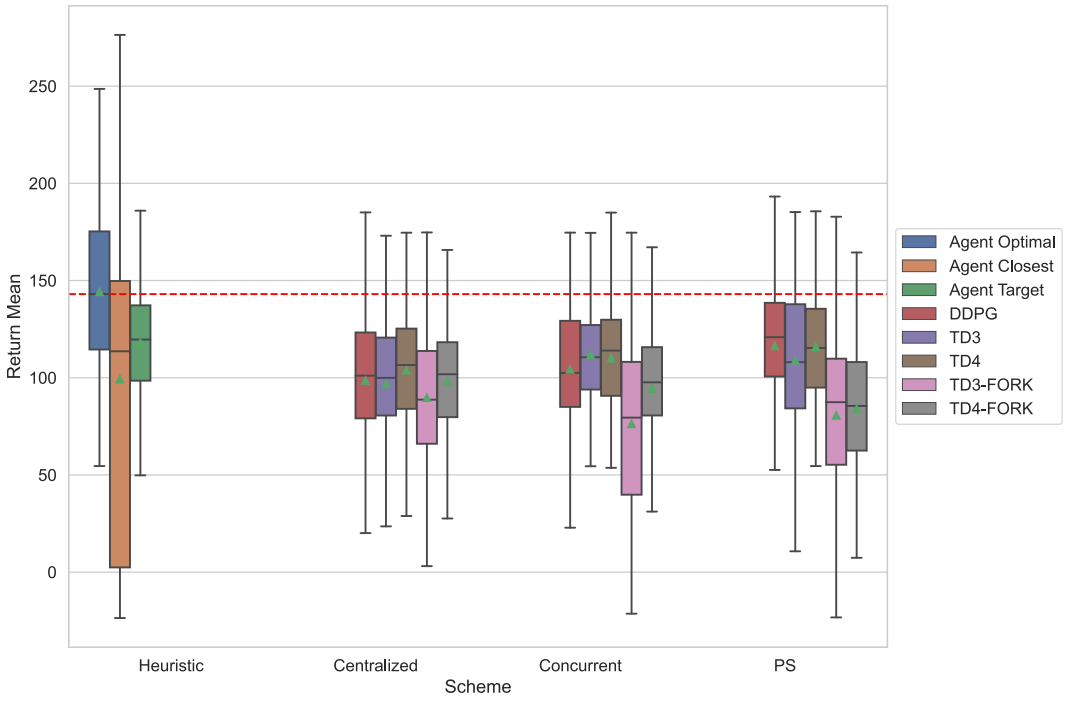


Figure A.14: SARL comparison for MPE Simple Confuse Continuous

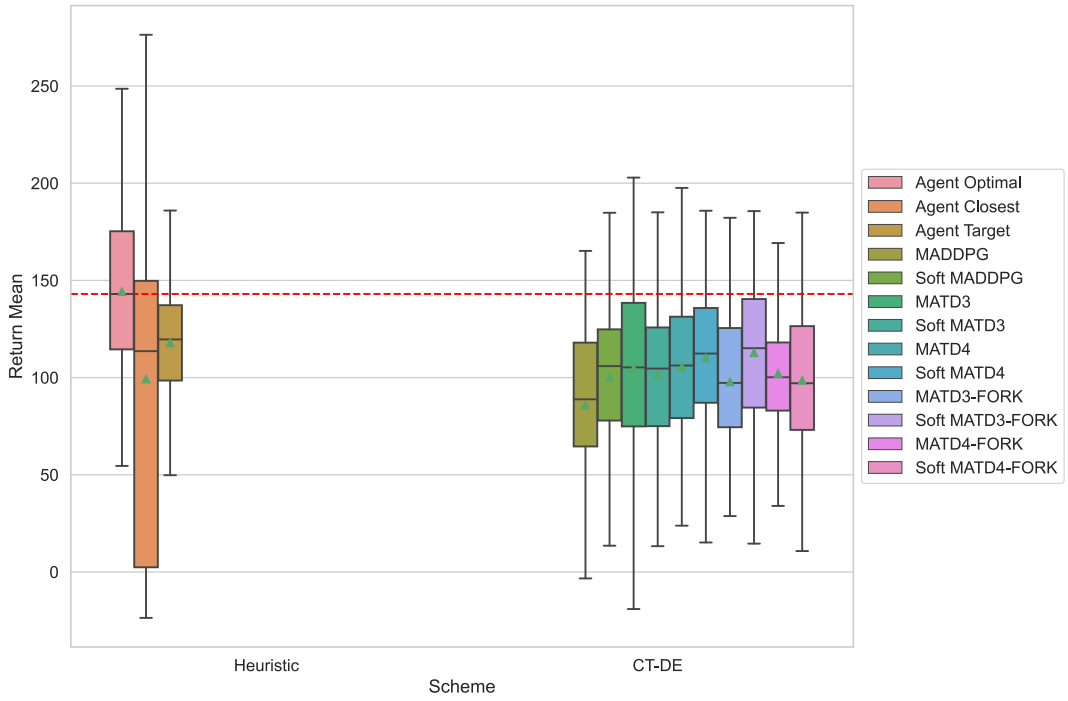


Figure A.15: CT-DE comparison for MPE Simple Confuse Continuous

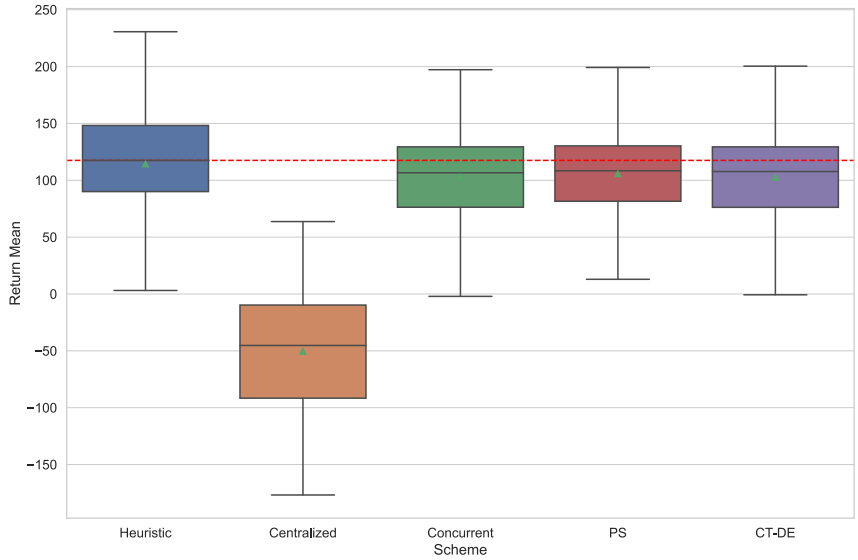


Figure A.16: Schemes comparison for MPE Simple Confuse Discrete

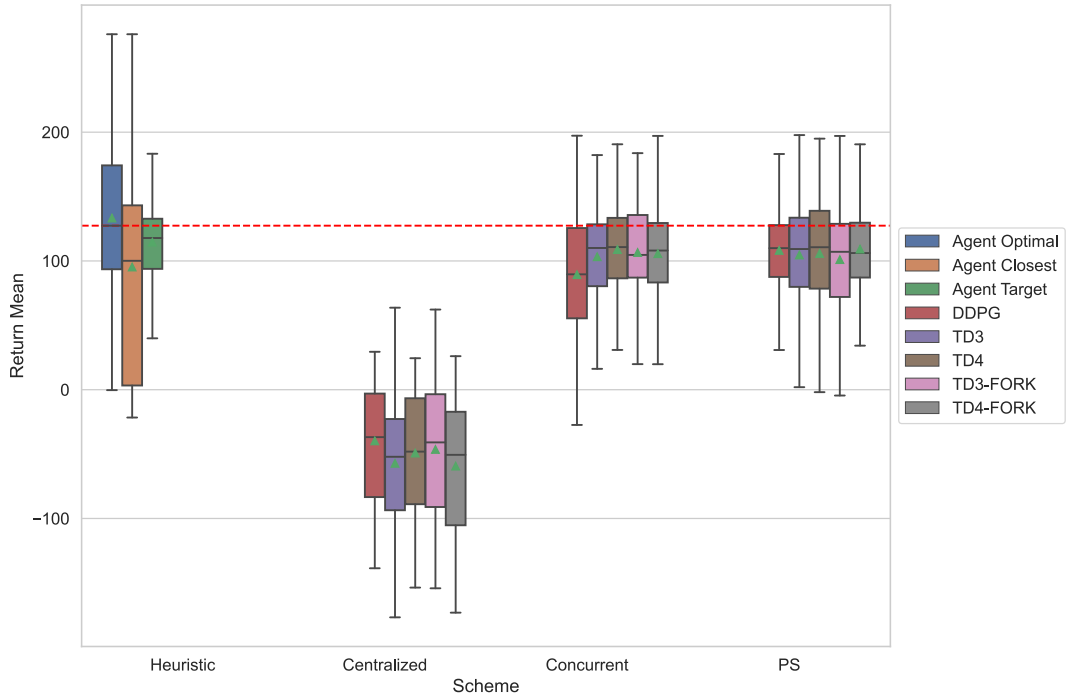


Figure A.17: SARL comparison for MPE Simple Confuse Discrete

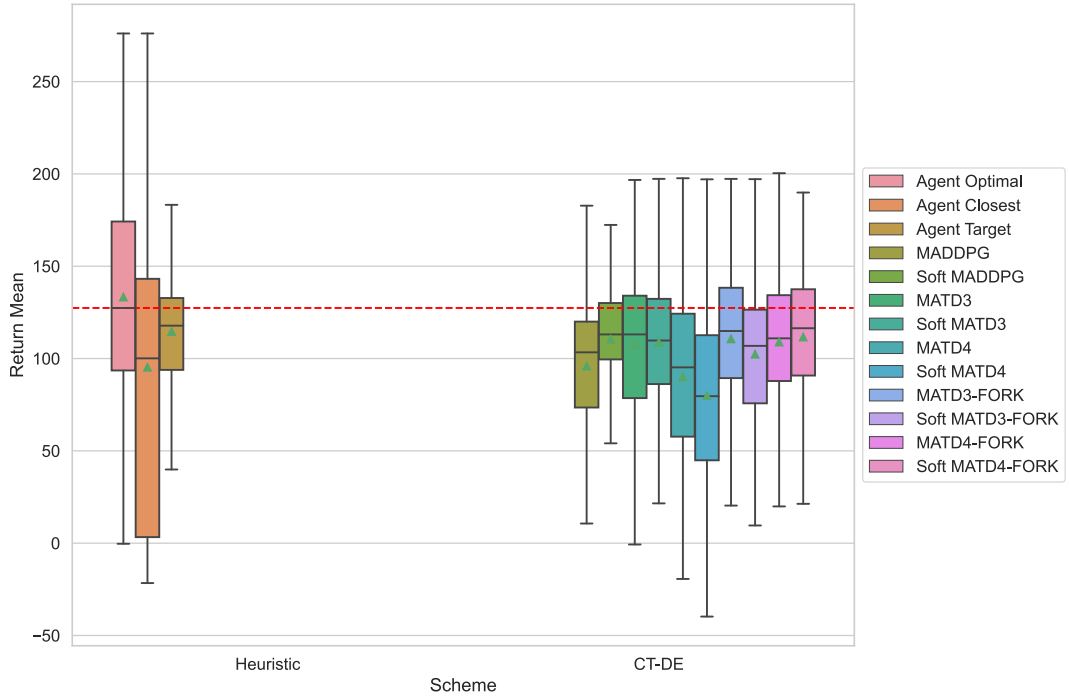


Figure A.18: CT-DE comparison for MPE Simple Confuse Discrete