MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Improving Multi-Agent Reinforcement Learning Through Agent Representation"

verfasst von / submitted by

Justus Rass, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2025 / Vienna, 2025

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 910
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Computational Science
Betreut von / Supervisor:	Assoz. Prof. Dipl.-Ing. Dr.techn. Sebastian Tschiatschek, BSc

Acknowledgments

I want to thank Sebastian Tschiatschek for supervising this project, as well as my friends and family for always being supportive during the creation of this thesis.

Abstract

In multi-agent reinforcement learning, non-stationarity poses one of the biggest challenges in learning optimal policies for agents, especially when they cannot communicate with each other. This is due to agents acting upon their own observations and thus inducing changes in the environment that other agents cannot predict, hindering convergence towards the best solution during the learning process of individually trained agents. While utilizing centralized approaches alleviates this issue for some cases, it is not always applicable. In this thesis, representations of agents, created from from interaction data, and their impact on learning are investigated. These representations aim to capture the behavior of an agent inside a low-dimensional embedding and utilizing those embeddings as privileged information during training.In experiments we show improvements in benchmark multi-agent environments using cooperative, competitive and mixed settings with two or more agents in each environment. Our approach is evaluated on decentralized training and decentralized execution, as well as centralized training and decentralized execution multi-agent learning concepts using deep learning algorithms.

Kurzfassung

Im Multi-Agenten-Deep-Reinforcement-Learning stellen nicht-stationäre Umgebungen eine der größten Herausforderungen dar, wenn es darum geht, optimale Strategien für Agenten zu erlernen, insbesondere wenn diese nicht miteinander kommunizieren können. Dies liegt daran, dass Agenten auf Grundlage ihrer eigenen Beobachtungen handeln und dadurch Veränderungen in der Umgebung hervorrufen, die andere Agenten nicht vorhersehen können, was den Entscheidungsprozess individuell trainierter Agenten destabilisiert. Während zentralisierte Ansätze dieses Problem in einigen Fällen abmildern können, sind sie nicht immer anwendbar. In dieser Arbeit werden Repräsentationen von Agenten untersucht, die aus Interaktionsdaten erstellt werden, sowie deren Einfluss auf den Lernprozess. Diese Repräsentationen zielen darauf ab, das Verhalten eines Agenten in einer niedrigdimensionalen Einbettung darzustellen und diese Einbettungen als privilegierte Information während des Trainings zu nutzen. In den Experimenten zeigen wir Verbesserungen in Benchmark-Umgebungen für Multi-Agenten-Szenarien, die kooperative, kompetitive und gemischte Aufgaben mit zwei oder mehr Agenten in jeder Umgebung umfassen. Unser Ansatz wird in Konzepten des dezentralen Trainings mit dezentraler Ausführung sowie des zentralisierten Trainings mit dezentraler Ausführung getestet, unter Verwendung von Deep-Learning-Algorithmen.

Contents

Summary of Notation

$s,\ s'$	current state, next state
$a,\ a'$	current action, next action
r	reward
t	discrete time step
γ	discount factor
$\mathcal{S}$	set of all states
$\mathcal{A}$	set of all actions
$\mathcal{R}$	set of all rewards
$s_t,\ a_t,\ r_t$	state, action, reward at time step t
$\pi,\ \pi^*$	decision making policy, optimal decision making policy
$\pi(s)$	action taken in state s under deterministic policy π
$\pi(a\|s)$	probability of taking action a in state s under stochastic policy π
$\pi_t(a)$	probability of selecting action a at time step t
$\log \pi_\theta(a\|s)$	log-probability of selecting action a in state s according to policy π_θ
G_t	cumulative return at time step t
$P(s_{t+1}\|s_t, a_t)$	transition probability to get next state s_{t+1} from state s and action a
$v_\pi, (s),\ v_*(s)$	expected return of state s under policy π/optimal policy π_*
$q_\pi, (s,a),\ q_*(s,a)$	expected return of taking action a in state s under policy π/optimal policy π_*
$v,\ v_t$	estimates of state-value function v_π or v^*
$q,\ q_t$	estimates of state-value function q_π or q^*
δ_t	temporal-difference error at time step t
$\alpha,\ \lambda$	learning rates
$J(\theta)$	performance metric for policy π_θ in Policy Gradient learning
$\theta,\ \phi$	network parameters for policy networks and value function networks
μ_θ	deterministic policy with parameters θ in deep learning methods
$o^{(i)},\ a^{(i)},\ r^{(i)}$	current observation, action, reward of agent i
$\mathcal{O}^{(i)},\ A^{(i)},\ R^{(i)}$	set of all observations, actions, rewards of agent i
$\pi^{(i)},\ \Pi$	policy of agent i, set of policies of all agents
$h_t^{(i)}$	observation history of agent i at time step t
f_θ	representation function of an agent with parameters θ
$\mathcal{E}$	observation-action pairs of an episode of an agent
$e_+,\ e_*,\ e_-$	positive, reference, negative episode for embedding
$p_e,\ r_e,\ n_e$	positive, reference, negative embedding

List of Figures

1 Introduction

With the sharp increase in computing power over recent years, solving complex problems like protein folding [JEP$^+$20], solving complex mathematical tasks [Alp24], weather predictions [LSW$^+$22], etc., deep learning models have been getting a lot of attention and seen significant progress. Be it using large language models (LLMs) like OpenAI's ChatGPT [Ope23] to interact with, generative models such as MidJourney [Mid23] to generate images from text, or reinforcement learning used in autonomous vehicles [SRR23]. This thesis focuses on reinforcement learning, with a focus on agent interaction in multi-agent environments. Multi-agent reinforcement learning is widely used for optimizing tasks that require decisions and actions from a lot of individual entities, like warehouse optimization using robots [KST$^+$24], market simulation for stock trading [LO02], video game AI for non-player characters [LFK21].

Reinforcement learning in settings where one agent interacts with its environment are well understood, since the agent can observe the whole environment and progresses according to the agent's actions on it with time. In multi-agent scenarios however, a problem arises that the environment that agent A currently observes and acts upon may not end up as predicted by agent A, because another agent B also acts according to its policy which is unknown to agent A. This can lead to issues in coordination between the agents and generalizing the problem setting itself.

This problem of agents only observing parts of their environment in a multi-agent setting, without knowing what other agents are doing, can be counteracted to a certain degree by approximating the behavior of other agents from the perspective of one agent during training [RDSF18], agents having access to the experiences of their collaborators and/or competitors using shared memory [FFA$^+$18], or using embedded information to gain advantageous knowledge [GAG$^+$18], which will be the topic of investigation of this thesis.

The approach by Grover et al. [GAG$^+$18] proposes a method to encapsulate the behavior of an agent by training neural networks, one to encode the observations and actions of an agent, the other to imitate the policy of another agent while appending the embeddings to its observations. This is shown for cooperative and competitive environments using two agents, which this thesis aims to expand upon.

The objective of this thesis is, for one, to try to bridge the information gap between completely independent agents, where each agent is following its own policy, acting on the environment [Tan93]. Further the proposed approach is evaluated with more complex environments, expanding on the results found by Grover et al. [GAG+18] by showing that it can encapsulate the behavior of multiple agents at once. For this, representations of an agent A are learned via a neural network to embed the observation and action of the agent at each time step. These embeddings are then provided to another agent B as additional information to help agent B optimize better against agent A. Experiments show embeddings can lead to better generalization and improved learning in cooperative and competitive environments, since the embedding network is trained on a diverse set of other agents following different policies, therefore improving performance.

The proposed approach is advantageous compared to others, in that it uses actual data from one agent to condition another agent instead of inferring the intentions and actions. Previous approaches include reusing older policies of the same agents in self play [Tes95], while other methods try to model other agents in the system by including additional parameters that try to encapsulate the goal of the other agent [RDSF18].

Using such embeddings does not fully resolve the information gap between individual agents, like shared memory with a centralized optimization network would, but it has the advantage of using less computational resources and such as can be more scalable. This method of using embeddings to help with generalization and information sharing between agents is being investigated in a variety of cooperative and competitive benchmark environments, using individually trained agents, as well as agents that are trained using a joint policy. By expanding on the original paper [GAG+18], which only explored one cooperative and competitive scenario involving two agents each, this work will show that this concept can be scaled to mixed cooperative-competitive scenarios, involving more than two agents.

Experiments will show that in especially competitive settings of decentralized training, the use of embeddings often leads to better performance, compared to learning with no embeddings, while cooperative settings show varying results, which is due to being more sensitive to parameter tuning.

The remainder of this thesis is structured as follows: A short breakdown of the problem setting involving embedding based expansion of agent observability in multi-agent reinforcement learning, followed by background information of single agent reinforcement learning and the challenges of multi-agent reinforcement learning. A detailed overview about the methodology used for the proposed method, as well as the implementation of algorithms, experimental setup, result evaluation and discussion of the results.

2 Problem Setting

The goal of this thesis is to evaluate the usage of agent embeddings have on the performance of multi-agent learning. With the main question being: Are representations of an AI agent helpful in the learning process useful for another agent in multi-agent reinforcement learning. These representations need to encapsulate the behavior and characteristics of an agent, while also being interpretable for another agent. Previous research has shown that in small scale scenarios, such agent representations can lead to improved performance during training. Here, this question gets expanded upon, by utilizing such embeddings in more complex tasks encapsulating the behavior of multiple agents at once.

In multi-agent reinforcement learning, agents interact with their environment and each other and learn improve their behavior by receiving reward signals through those interactions. The decision-making process for this is commonly formalized as Partially Observable Markov Decision Process (POMDP) defined by a tuple $(\mathcal{S}, (\mathcal{A}^{(i)})_{i=1}^{n}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ describes a set of states, $\mathcal{A}^{(i)}$ is a set of actions which the i-th agent can take, $\mathcal{P}$ describes the transition probability between states of the environment upon taking an action, $\mathcal{R}$ is a reward function, and γ is a discount factor. Environments can be collaborative, competitive or both, therefore the goal of collaborative agents usually receive the same reward and try to maximize the cumulative reward, while in competitive settings each agent tries to maximize its own reward. [ZYB19] The expected cumulative reward for a multi-agent setting is

$$R_t = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| (\pi^i)_{i=1}^{n}\right]$$

where $\pi^{(i)} : \mathcal{S} \times \mathcal{A}^{(i)} \times \mathcal{E} \rightarrow [0, 1]$ is the i-th agent's policy which assigns every possible action $a \in \mathcal{A}^{(i)}$ is to be taken given any possible state $s \in \mathcal{S}$. The term $(\mathcal{E})_{i=1}^{n}$ is an additional term, which represents embedded information about another agent's observation-action pairs with dimension n. All three components calculate which action a should be taken, given any state s and R_t assigning the reward obtained at time t.

Agents act on the environment given a state $s_t^{(i)}$, which only includes information that agent i is able to obtain while not observing information beyond it. This introduces uncertainty due to the agents receiving unexpected states at the following time step $s_{t+1}^{(i)}$. This uncertainty stems from actions of other agents k impacting the environment in ways that are not expected by an agent $i \neq k$ and therefore confusing it. These unexpected changes stem from the actions of all agents in an environment, affecting following observations. Existing approaches try to mitigate this information gap by utilizing centralized oversight or communication during training, which yields improved results but is not always applicable, be it due to scalability issues or in use cases where a trained agent has to act on limited observations and no communication. An approach to mitigate the problem of agents receiving limited observations of the environment is to incorporate agent representations, which are learned embeddings that encapsulate the behavior and approaches of other agents using interaction data. The goal is to find a way to create such embeddings $\mathcal{E}$, that are capable to encapsulate the behavior of one or more agents at once, while also being applicable to one or more agents of similar nature. An example for this would be encapsulating the patterns of three hunters and having two prey agents use these embeddings to escape more easily. Such scenarios are therefore not strictly competitive or cooperative, which adds another layer of complexity that needs to be overcome.

The approach used in this thesis is an extension of *Learning Policy Representations in Multiagent Systems* by Grover et. al [GAG+18], which applies the usage of agent representations during training and execution in centralized and decentralized multi-agent deep reinforcement learning algorithms. The framework consists of developing a module that learns an embedding network to encode the higher dimensional observation-action pairs of an agent at each time step into a lower dimensional representation, which can be used as privileged information by other agents. Further, learning agent policies utilizing these representations by augmenting cooperative and competitive environments to generate these embeddings at each time step, by using wrapper functions incorporating the trained embedding networks. Lastly, evaluating different approaches of training agent representations in combination with centralized and decentralized algorithms, comparing their performance against baseline results without the usage of representations.

The algorithms used focus on widely used offline model-free approaches, which do not rely on a pre-existing model, gathering all experience through trail and error, and optimize towards a action-value function $Q(s, a)$:

$$Q(s,a) = \mathbb{E}_{s',R}\left[R(s,a) + \gamma \mathbb{E}_{a'\ \pi^{(i)}}[Q(s',a')]\right]$$

where s' represents the next state and a' the next action according to policy $\pi^{(i)}$. Challenges arise in learning robust representations which capture the essential behavior of an agent, as well as trying to alleviate the issue of an agent not knowing what another agent is doing, and developing a test suite to evaluate overall and per agent performance in a plethora of scenarios.

3 Reinforcement Learning

3.1 Introduction to Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning where an agent learns how to interact inside an environment in an optimal fashion. This is achieved via trial and error by interaction with the environment, trying to maximize a numerical reward. Reinforcement learning operates in an unsupervised fashion, which means it acts more autonomous without the need of labeled datasets or validation/test data to tune against. An RL agent learns by trying out various actions inside the environment and receiving either a reward or punishment as feedback to guide further decisions on which actions to take. Over multiple iterations of observing and acting inside the environment, the agent learns which action a is more likely to lead to a reward R for a given state s, while simultaneously avoiding actions that have a negative feedback as predicted outcome. Maximizing the immediate reward is too short sighted to find the optimal course of action over extended time frames, so the agent also has to learn how to plan ahead. Therefore Reinforcement learning is combining three characteristics of being a closed-loop system, not receiving direct instructions before or during training, and the consequences of actions taken accumulating over time. [SB18]

The task an agent has to execute is typically modelled as a Markov Decision Process (MDP), for which a set of states $\mathcal{S}$, actions $\mathcal{A}$, a value function $\mathcal{Q}(\mathcal{S}, \mathcal{A})$, and transition P determine how the environment evolves. The Markov Decision Process assumes that the so called Markov property is satisfied during the decision-making process. This means that future states of the environment are only dependent on the current state and the action chosen by the agent, and independent of any previous states or environment history. [Put94]

Reinforcement learning differentiates from other forms of machine learning in that it lacks the supervision of supervised learning, or finding structures in data in the case of unsupervised learning. Unlike supervised learning, where a model is trained on labeled data of known input-output pairs, there is no correct answer for every training step in RL. Agents must discover which actions to chose for a given environment state by trial and error, so called exploration. Since there is no ground truth the agent could optimize towards, it has to figure out which actions give a reward and which actions give a punishment. To avoid locking itself into one single strategy to navigate the environment, a balance between exploration of unknown actions and exploitation of successful strategies has to be maintained. Trying unknown actions through exploration, even though working strategies have already been found, is fundamental principle to training Reinforcement Learning algorithms. [KLM96]

Although Reinforcement learning does not use labeled data for learning, it can't be categorized as unsupervised learning either. Instead of trying to find structure, it tries to maximize a reward signal. It instead focuses on sequential decision-making, where each action influences not only the immediate reward, but also future rewards. This temporal property leads to a sort of butterfly effect, where an action taken now influences every reward down the line with sometimes significant long-term consequences. This leads to an RL agent having to optimize for long-term cumulative feedback instead of short-term rewards. [SB18]

A few prominent examples are RL agents playing Atari arcade games on human- to superhuman-levels by introducing deep learning into reinforcement learning by combining Q-learning with deep neural networks to approximate an optimal action-value function from raw pixel input. The derived polices were able to surpass human performance in a number of arcade games, and also marks a pivotal point in reinforcement learning with the introduction of deep learning. [MKS+15] Another pivotal achievement for reinforcement learning was achieved with DeepMind's AlphaGo system, which was the first AI system to defeat the then reigning world champion in the board game Go, which is one of the most complex 2-player board games. It combined reinforcement learning with supervised learning by learning from human experts, while also introducing searching for the best action with deep neural networks. [SHM+16] For games involving more than one or two players, OpenAI Five is an important landmark in reinforcement learning, where five individual agents learned to play Valve's DotA 2, which is one of the most complex team based competitive multiplayer video games. By using self-play with a multi-agent learning approach, OpenAI Five managed to beat professional teams in a limited setting, where a smaller portion of playable characters was available. [BBC+19]

3.2 Markov Decision Process

The Markov Decision Process (MDP) is at the core of reinforcement learning and provides the mathematical framework for modeling decision-making processes in an environment, where the outcome is at least partially dependent on the agent. An agent interacts with the environment in a sequence of discrete time steps $t = 0, 1, 2, 3, ..., n$. During each time step t, the agent observes the current state of the environment $s_t \in \mathcal{S}$ with $\mathcal{S}$ being the set of possible states. Based on the observed state, the agent then selects and action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ are the available actions for the current state s_t. At the next time step $t + 1$ the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and the environment transitions into a new state s_{t+1} based on the action a_t as depicted in Fig. 3.1. During each of these transitions, the agent creates a mapping from states to the probabilities for selecting the best possible action. This mapping is called a policy π_t, with $\pi_t(a_t|s_t)$ being the probability of action a_t given state s_t. [SB18]

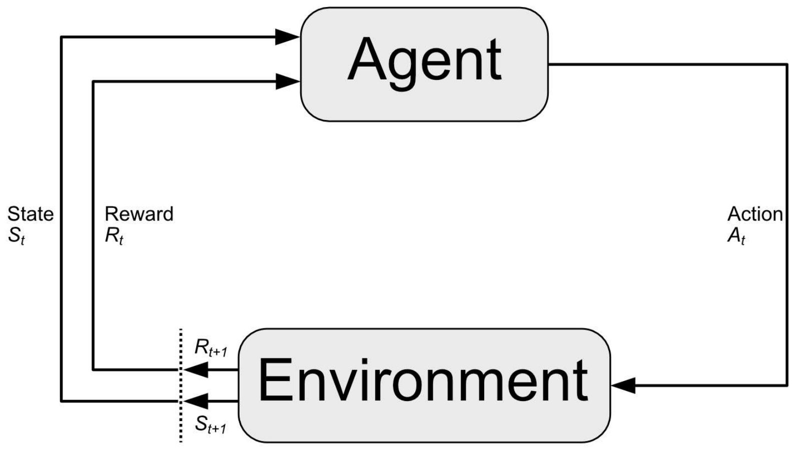


Figure 3.1: Agent-environment interaction diagram for a Markov Decision Process. Recreated from [SB18] P. 38

The Markov Decision Process is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of all possible actions, $\mathcal{P}$ is the transition probability into a state s' according to the current state and action $P(s'|s,a)$. The reward function $\mathcal{R}$ determines the immediate reward the agent receives after the environment transitions into a new state according to $R_{t+1}(s,a)$. The goal is to maximize the the accumulated reward across all time steps, with the sum of all rewards being the simplest approach:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{3.1}$$

with T being the final time step and G_t describing the cumulative reward. The last time step in the decision-making sequence is called the terminal state, and is the break point of so called episodes. The discount factor $0 \leq \gamma \leq 1$ determines how much weight is given to future rewards as compared to immediate rewards, by prompting the agent to either explore uncharted actions given the current state, or sticking to a course of action it already knows to give good rewards. [SB18]

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-1} R_T = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.2}$$

The Markov property is the underlying principle for the Markov Decision Process. It assumes that the future state of an environment only depends on the current state and action. This means that an agent only has to concern itself with the current state of the environment it is currently in without keeping track of past events. The Markov property is expressed as

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0, a_0) = P(s_{t+1}|s_t, a_t) \tag{3.3}$$

with s_t and a_t being the current state and action at time t. It describes the probability distribution over actions given a state, while disregarding any past interactions. [Put94] In environments adhering to the Markov property, it is possible to predict all future states and rewards from the knowledge of the current state at any time. The dynamics of the Markov Decision Process are now quantified by the transition probabilities in Eq. (3.3), which define how the environment evolves based on the actions of the agent. The objective of an agent adhering to the MDP is to maximize the cumulative reward over all time steps according to Eq. (3.2). [SB18]

As mentioned earlier, for an MDP to find an optimal overall solution it is important to not get stuck in a course of action that promises high immediate rewards, but rather maximize the reward over the whole episode. Exploring unknown actions, that might lead to better or worse rewards, by choosing random actions instead of the known ones is a widespread method to achieve this. Maintaining the balance between exploration and exploitation is crucial, since too much exploration can lead to suboptimal performance by skipping good known results, while too much exploitation can lead to missing out on better performing actions. One method to balance this is the so called $\epsilon - greedy$ method, where the agent sometimes selects a random action according to *if* $p < \epsilon$ *then* choose a random action, where $p = random$. [KLM96] Another method of handling exploration-exploitation in an MDP is using upper confidence bounds, where the agent selects its actions based on the estimated value and uncertainty. It samples the expected reward with an added confidence interval, which leads to more systematic exploration in the beginning and more exploitation towards the end of training. [ACF02]

3.3 Value Functions

The state-value function $v_\pi(s)$ in RL is the main concept to provide insights for being in a specific state s under policy π, while also considering long term effects. It quantifies the expected cumulative reward when an agent starts from a state $s \in \mathcal{S}$ while following the policy $\pi(a|s)$ using actions $a \in \mathcal{A}(s)$. The state-value function $v_\pi(s)$ is defined as

$$v_\pi(s) = \mathbb{E}_\pi[s_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s\right] \tag{3.4}$$

with $\mathbb{E}_\pi[\cdot]$ representing the expected value when following policy $\pi(a|s)$, $\gamma \in [0, 1]$ being the discount factor, which controls the impact of future rewards, and R_{t+1} representing the reward after transitioning from state s_t into s_{t+1}.

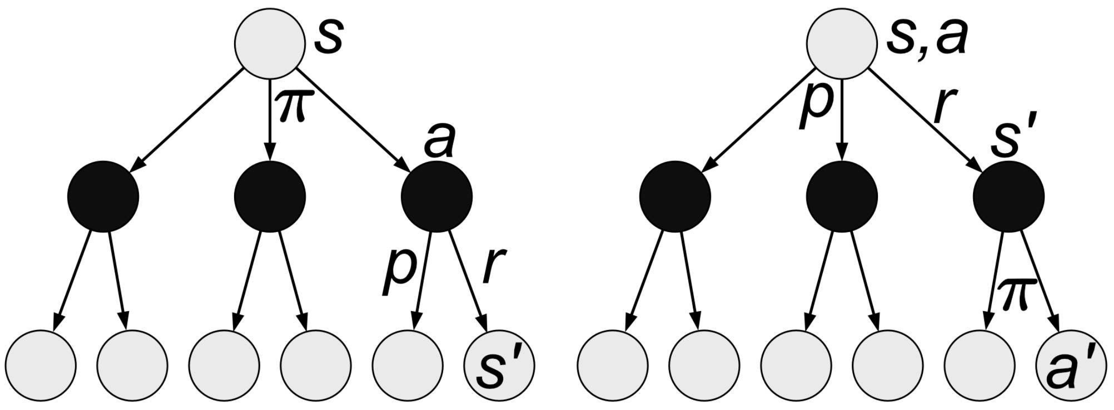


Figure 3.2: Backup diagrams for state-value function $v_\pi(s)$ (left) and action-value function $q_\pi(s,a)$ (right). Recreated from [SB18] P. 47-48

The state-value function $v_\pi(s)$ only considers the cumulative rewards based on state transitions under policy $\pi(s)$. Since RL is also reliant on the actions taken from any given state, the state-value function can be expanded into a action-value function $q_\pi(s,a)$ following the policy $\pi(s,a)$ which can be defined as:

$$q_\pi(s,a) = \mathbb{E}_\pi[s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s, a_t = a\right] \tag{3.5}$$

The state-value function, and in extension the action-value function, have a very important property of being recursive, according to the Bellman equation [Bel66]

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) \left[r + \gamma v_\pi(s')\right] \end{aligned} \tag{3.6}$$

This equation highlights the relationship between the values of a successor state and the originating state by following policy $\pi(s,a)$. In summary, the state-value function accounts both the immediate reward as well as future expected rewards, weighted by a discount factor. A flow diagram for the state-value function and action-value function is seen in Fig. 3.2.

The discount factor γ is crucial for determining how much expected rewards from future states influence the current state-value function. If $\gamma = 0$, the agent only considers the expected reward of the next step, disregarding anything beyond that. With γ approaching 1, future rewards become more prevalent in the decision-making process, and the agent gains a better degree of foresight. Thus the agent optimizes more for long term rewards. During training, finding the optimal balance between foresight and immediate gains is very important for finding an optimal state-value $v_*(s)$ or action-value $q_*(s, a)$ function. Such optimal value functions are the outcome of an agent acting according to an optimal policy π_* which is defined as being equal or better than any policy $\pi \geq \pi'$. This means that a policy $\pi \geq \pi'$ exists if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$, then an optimal policy π_* exists which is at least equal or better than all other policies. These optimal policies share the same value functions defined as

$$v_*(s) = \max_\pi v_\pi(s) \tag{3.7}$$

for all $s \in \mathcal{S}$, as well as

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{3.8}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Given a state-action pair (s, a), and following an optimal policy, it is possible to rewrite q_* in terms of v_*

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid s_t = s, a_t = a] \tag{3.9}$$

Since v_* is the value function for an optimal policy, it follows the conditions given by the Bellman equation (3.6). Given an optimal value function, the Bellman equation can be rewritten without including the policy term, since an optimal policy will always choose actions leading to optimal rewards at any point. This Bellman optimality equation is defined as

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_p i_*(s, a) \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid s_t = s, a_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{a \in \mathcal{A}(s)} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \end{aligned} \tag{3.10}$$

for the optimal state-value function v_*, with the optimal action-value function q_* being defined as

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \middle| s_t = s, a_t = a\right] \\ &= \sum_{s', r} p(s', r|s, a) \left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned} \tag{3.11}$$

which forms the basis for Q-learning. Finding these optimal value functions is used extensively in RL algorithms, particularly in value iteration and policy iteration. Value iteration updates the optimal state-value function after each episode until it converges according to Eq. (3.10). Policy iteration on the other hand, alternates between calculating the state-value function $v_\pi(s)$ for a given policy π and improving the policy by choosing the action that maximizes the expected return based on the current state-value function. [SB18] [Bel66]

3.4 Temporal Difference Learning

Temporal Difference (TD) learning is a fundamental approach in a lot of modern RL algorithms. It learns by estimating a state-value function v_π for a given policy π according to to Eq. (3.4). TD methods use a generalized version of policy iteration and experience to solve the prediction problem for choosing an action for a given state. Unlike other methods, TD based algorithms don't update the value estimations at the end of an episode, but rather after each time step. This turns this class of algorithms into online and incremental algorithms, meaning that agents learn from data they gather themselves at every time step during an episode, even if the episode is incomplete. Since the agent learns from its own gathered data during each step and is not based on an already existing model, TD learning as well as other algorithms derived from it are called model-free. [SB18]

The simplest form of Temporal Difference learning is the so called TD(0) algorithm, which updates after every time step. It does this by estimating the state-value function at $v(s_{t+1})$, and comparing it to the actual reward given at r_{t+1}. This leads to a function

$$v(s_t) \leftarrow v(s_t) + \alpha[r_{t+1} + \gamma v(s_{t+1}) - v(s_t)] \tag{3.12}$$

with α being a learning rate hyper-parameter and γ being the discount factor. In algorithmic form, TD(0) can be expressed as Alg. 1 [Sut88]

Algorithm 1 TD(0) Algorithm

```
 1: Initialize $v(s)$ arbitrarily for all states $s$ (e.g., $v(s) = 0$)
 2: Initialize step size parameter $\alpha \in (0, 1)$
 3: Initialize discount factor $\gamma \in (0, 1)$
 4: Observe initial state $s_0$
 5: for each episode do
 6:     while $s_t$ is not terminal do
 7:         Take action $a_t$ based on policy $\pi$ (e.g., $\epsilon$-greedy)
 8:         Observe reward $r_{t+1}$ and next state $s_{t+1}$
 9:         Update value function: $v(s_t) \leftarrow v(s_t) + \alpha[r_{t+1} + \gamma v(s_{t+1}) - v(s_t)]$
10:         Set $s_t \leftarrow s_{t+1}$
11:     end while
12: end for
```

To expand on only considering the most recent state, one can use eligibility traces $E(s)$ which contain n past states and actions. By updating with multiple past states, the TD(0) algorithm becomes the TD(λ) algorithm. An eligibility trace can be defined as

$$E(s_t) \leftarrow \gamma\lambda E(s_{t-1}) + 1 \tag{3.13}$$

with γ being the discount factor, $\lambda \in [0, 1]$ describing a parameter that controls how much influence previous states have, and the trace being initialized with a value of 1 at the current state, which then decays over time. The TD(λ) algorithm updates its state-value function $v(s)$ using the following update rule at each time step:

$$v(s_t) \leftarrow v(s_t) + \alpha \delta_t E(s_t) \tag{3.14}$$

with $\delta_t = [r_{t+1} + \gamma v(s_{t+1}) - v(s_t)]$ from Eq. (3.12). The TD(0) algorithm gets expanded by including said traces into the following Alg. 2:

Algorithm 2 TD(λ) Algorithm

```
1: Initialize $v(s)$ arbitrarily for all states $s$
2: Initialize step size parameter $\alpha \in (0, 1)$
3: Initialize discount factor $\gamma \in (0, 1)$
4: Initialize trace decay parameter $\lambda \in [0, 1]$
5: Initialize eligibility trace $e(s) = 0$ for all states $s$
6: Observe initial state $s_0$
7: for each episode do
8:     Initialize $e(s) = 0$ for all states $s$
9:     while $s_t$ is not terminal do
10:        Take action $a_t$ based on policy $\pi$
11:        Observe reward $R_{t+1}$ and next state $s_{t+1}$
12:        Compute the TD error: $\delta_t = r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$
13:        Update eligibility trace for the current state: $e(s_t) \leftarrow e(s_t) + 1$
14:        for each state $s$ do
15:            Update the value function: $v(s) \leftarrow v(s) + \alpha \delta_t e(s)$
16:            Decay the eligibility trace: $e(s) \leftarrow \gamma \lambda e(s)$
17:        end for
18:        Set $s_t \leftarrow s_{t+1}$
19:    end while
20: end for
```

Temporal Difference algorithms, form the basis of two foundational RL algorithms, namely SARSA [Rum94] which will not further discussed in this thesis, and Q-learning [WD92] which is the basis for the algorithms used in the later experiments and will be discussed in the following chapter.

3.5 Q-Learning

Q-learning is a fundamental and widely used model-free RL algorithm. Model-free algorithms have the ability to learn an optimal policy without knowing the transition probabilities or reward functions of an environment. Additionally it is also an off-policy algorithm, which means that it can learn an optimal policy whether it is following the current policy or exploring other actions. This leads to Q-learning being able to estimate the optimal policy while exploring via ϵ-greedy methods. The core of Q-learning is the action-value function $Q(s,a)$[1] which is the expected cumulative reward for following the optimal policy. Formally the Q-function is defined by the action-value function introduced in Eq. (3.5), and

$$Q^*(s,a) = \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} Q^*(s',a') \middle| s,a \right] \tag{3.15}$$

referring to the optimal Q-function according to the Bellman optimality, where s' is the next state, r_{t+1} referring to the reward received after going from s to s', and $max_{a'}Q^*(s',a')$ describing the maximum expected future reward when choosing the best action a' when in the next state s'. This reflects the property of the Q-function taking into account the immediate reward r_{t+1} as well as the discounted reward of the best action in the next state. [WD92]

This leads to the objective of Q-learning being to iteratively update Q-values for each state-action pair based on interactions with the environment. The basic Q-learning algorithm, also named one-step Q-learning defines the objective function as

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s,a) \middle| S_t = s, a_t = a \right] \tag{3.16}$$

with $\alpha \in (0,1]$ being a hyper parameter for the learning rate, which controls how much influence the new information has on $Q(s,a)$. The algorithm for Q-learning is outlined in Alg. 3. Q-learning typically employs 2 policies, one so called behavior policy, which the agent follows to explore the environment, and the target policy which always follows the greedy approach of using the best action for a given state. The behavior policy is usually a mix of randomly selected actions, as well as actions that follow the best actions according to their Q-values. This allows the agent to explore according to an exploration scheme, usually ϵ-greedy, while also converging towards the optimal policy π^*. [SB18]

The key strength of Q-learning is that is guaranteed to converge to optimal Q-values $Q^*(s,a)$, as long as all state-action pairs can be explored infinitely often. In conjunction with an appropriate decaying learning rate α, the learned policy will eventually become the optimal policy π^*. [WD92]

[1]To keep in line with standard notation the action-value function $q(s,a)$ will be referred to as $Q(s,a)$ from now on.

Algorithm 3 Q-learning Algorithm

```
1: Initialize $Q(s,a)$, $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$ arbitrarily
2: Initialize learning rate $\alpha \in (0,1]$
3: Initialize discount factor $\gamma \in [0,1)$
4: for each episode do
5:     Observe initial state $s_0$
6:     for each step in episode do
7:         Choose an action $a$ according to policy $\pi$ (eg. $\epsilon$-greedy)
8:         Execute action $a$ and observe next state $s'$ and reward $r_{t+1}$
9:         Update $Q(s,a)$ according to update rule
               $Q(s,a) \leftarrow Q(s,a) + \alpha \left[ r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s,a) \right]$
10:    end for
11: end for
```

3.6 Policy Gradient Learning

Policy Gradient (PG) learning is a family of RL algorithms that directly optimize the policy by computing the gradient of an objective function with respect to the policy parameters. Unlike Q-learning, for example, it does not estimate the value of states and actions to derive a policy, but rather adjusts the policy parameters directly using gradient ascent. [Wil92] Policies in PG algorithms are usually parameterized by a set of weights θ and defined as $\pi_\theta(a|s)$, which describes the probability distribution over actions for given states, which are either deterministic or stochastic. PG methods often use stochastic policies, which is useful for environments where exploration is crucial. For this, the optimization objective is to maximize the cumulative reward over time $J(\theta)$ like

$$J(\theta) = \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta} \left[r_t(s,a) \right] \tag{3.17}$$

where $\rho^\pi(s)$ is the discounted state distribution under policy π_θ. Gradient ascent is being used to optimize the policy π_θ by applying the policy gradient theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)] \tag{3.18}$$

with $Q(s,a)$ being the action-value function for policy π_θ and $\nabla_\theta \log \pi_\theta(a|s)$ describing the gradient of the log-probability of the action taken. The aforementioned policy gradient theorem provides a way to estimate the gradient of the expected return without needing derivatives of the environment's dynamics. [SMSM99] One of the most prominent stochastic on-policy PG algorithms is Proximal Policy Optimization (PPO), which places constraints on the amount the policy can update either via clipping or penalizing on each time step. [SWD$^+$17]

Deterministic Policy Gradient (DPG) methods extend Policy Gradient techniques to deterministic policies, where actions are directly determined by the state and not sampled from a probability distribution. Such deterministic policies are represented by $\mu_\theta(s) : S \to A$, which maps an action a directly from a state s with parameters θ controlling the policy instead of drawing an action from a probability distribution $\pi(a|s) : S \to P(A)$. This makes DPG methods well suited for continuous action spaces, since directly choosing an action is more efficient than sampling from a distribution. The objective for Deterministic Policy Gradient is similar to the stochastic policy gradient Eq. (3.17) and defined as

$$J(\theta) = \mathbb{E}_{s \sim \rho^\mu}[r(s, \mu_\theta(s))] \tag{3.19}$$

with the deterministic policy gradient theorem being

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s \sim \rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}] \tag{3.20}$$

where $Q^\mu(s,a)$ is the action-value function, and $\rho^\mu(s)$ is the discounted state distribution under policy μ_θ. DPG is generally implemented for off-policy algorithms and uses a replay buffer to store past experiences. These experiences are then used in batches to update the policy. These algorithms usually use the actor-critic method, where the actor represents the deterministic policy $\mu_\theta(s)$ that outputs the action a for a given state a, and the critic represents the action-value function $Q^\mu(s,a)$, which evaluates the quality of the actions taken. Usually, the actor is trained using the deterministic policy, while the critic is trained via temporal difference learning. [SLH+14]

3.7 Actor-Critic Algorithms

A widely used family of algorithms in reinforcement learning are actor-critic methods. They combine the two concepts of policy-based and value-based approaches inside one agent which can leverage the strengths of both. The agent a utilizes two part strategy where the actor is responsible for learning and executing the policy, while the critic evaluates the actions chosen by the actor by estimating the value function, and provides feedback to improve the actor. [KT99]

The actor is a policy function $\pi_\theta(a|s)$, with θ representing the parameters of the policy. The actor chooses an action a given a state s based on its policy and tries to maximize the expected cumulative reward by adjusting the policy parameters θ towards choosing higher rewarded actions. The critic, on the other hand, estimates the state-value function $V(s)$ or action-value $Q(s,a)$ of the current policy, by evaluating the actions taken by the actor and providing feedback by calculating the so called temporal difference error, which measures the difference in expected and actual reward for each action. The TD error for a value function $V(s)$ is defined as

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V(s_t) \tag{3.21}$$

where V_t is the state-value function generated by the critic or

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{3.22}$$

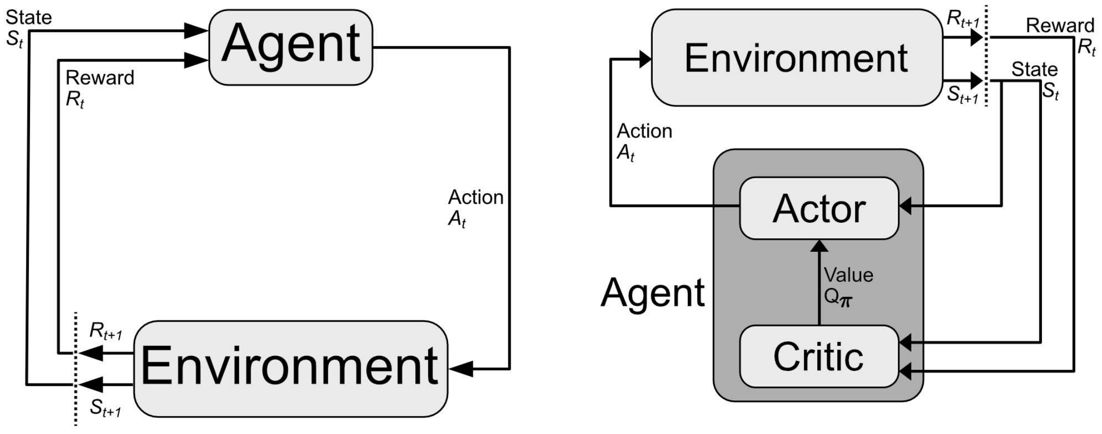


Figure 3.3: Flow-diagram showing the difference for agent-environment interaction for regular agents in an MDP (left) and actor-critic methods (right). Recreated from [SB18] P. 38 (left), [Sze10] P. 63 (right)

with Q_t being the action-value function of the critic. A positive TD-error δ_t suggests that the selected action a_t is resulting in higher rewards, whereas a negative TD-error implies that the selected action shouldn't be chosen in the future. Actor-critic methods learn by minimizing the TD-error for the critic by using TD-learning, while the actor updates its policy using policy gradient methods and feedback from the critic which actions provide good rewards. [SB18] A flow chart highlighting the difference between actor-critic methods compared to using the simple Markov Decision Process is highlighted in Fig. 3.3

The main advantage of using actor-critic methods lies in eliminating the main drawbacks of purely policy-based and value-based methods. Value-based methods such as Q-learning can struggle with environments that have continuous observation and action spaces. This is due to the algorithm having difficulties estimating the value function, which can be alleviated by function approximations like neural networks that lend themselves to actor-critic methods. Policy-based algorithms can suffer from high variance within the gradient estimates, which is a limiting factor in learning an optimal policy in algorithms like *REINFORCE* by Williams [Wil92] This variance in gradient estimation is reduced by having the critic providing feedback and thus stabilizing the learning process.

The general flow of actor-critic methods is initializing both actor and critic, as well as learning rates, then on every time step sampling an action from the actor, observing the given reward and next state, updating the critic using the TD-error, and finally updating the actor based on the feedback of the critic. This is illustrated in Alg. 4. There are 4 types of actor-critic algorithms in use that utilize this architecture. *Advantage*

Algorithm 4 General Actor-Critic Algorithm

```
 1: Initialize actor $\pi_\theta(a|s)$, critic $V_w(s)$ with parameters $\theta$ and $w$
 2: Initialize learning rates $\alpha_\theta$ and $\alpha_w$ for actor and critic
 3: Initialize discount factor $\gamma$
 4: Initialize state $s_0$
 5: for each episode do
 6:     Reset environment and observe initial state $s_0$
 7:     for each step in episode do
 8:         Sample action $a_t$ from policy $\pi_\theta(a_t|s_t)$
 9:         Execute action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
10:         Compute TD error $\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V(s_t)$
11:         Update critic parameters $w \leftarrow w + \alpha_w \delta_t \nabla_w V_w(s_t)$
12:         Compute advantage estimate $A(s_t, a_t) = \delta_t$
13:         Update actor parameters $\theta \leftarrow \theta + \alpha_\theta A(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t, s_t)$
14:         Set $s_t \leftarrow s_{t+1}$
15:     end for
16: end for
```

Actor-Critic (A2C) by Mnih et al. [MBM+16], is a parallelized algorithm which uses $A(s,a) = Q(s,a) - V(s)$ as an advantage function, which measures how much better an action a is compared to the average action in state s. Other actor-critic methods approximate this advantage function by using the TD-error instead $A(s,a) \sim \delta_t$. *Soft Actor-Critic (SAC)* by Haarnoja et al. [HZAL18] incorporates an entropy term into the objective, which encourages more exploration that favors stochastic policies that maximize the expected reward and entropy. *Proximal Policy Optimization (PPO)* by Schulman et al. [SWD+17] uses a clipped objective function to limit how much the policy can be updated for each iteration to prevent changes that might be too drastic. Lastly, *Deep Deterministic Policy Gradient (DDPG)* by Lillicrap et al. [LHP+16] uses a deterministic policy for the actor, and a Q-function for the critic. In contrast to the other actor-critic methods, it also utilizes techniques like experience replay buffers and target networks, which are commonly used in deep reinforcement learning.

3.8 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a powerful extension to regular reinforcement learning, which includes the strength of deep learning for approximating functions to solve complex decision-making problems. Policies, value functions, or models of the environment are approximated by neural networks, instead of the tabular representations or simple function approximations. With RL, at its core, trying to learn how agents should apply actions in an environment to maximize the cumulative reward, deep learning excels at extracting features from high-dimensional input data such as images, sensors, etc. Combining the two methods lends itself to learn policies for complex systems such as playing games, autonomous driving, or robotics. Similar to more simple RL methods, a state s_t is given as input for the agent, the policy $\pi(a|s)$ of which is now represented as a neural network, and an action a_t is chosen, which leads to the state s_{t+1} at each time step t. Further, the expected cumulative reward $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$, stays the same basic objective. Same as in the case of the policy, the state-value $V(s)$ or action-value function $Q(s, a)$ are now also represented as a neural network, which approximates the functions instead of keeping results in a tabular form. [SB18]

The first algorithm to introduce deep learning into RL was *Deep Q-Network (DQN)*, introduced by Mnih et al. [MKS+15], where the Q-function in environments with high-dimensional space is approximated by a deep neural network. The high-dimensional input for the first application of this method was pixel input of Atari 2600 video games. The major innovations brought forward with this paper are the inclusion of an Experience Replay Buffer, which stores state transitions in the form of (s_t, a_t, r_t, s_{t+1}) and samples random mini-batches in order to avoid correlation between updates. The other contribution is the introduction of Target Networks, which are a separate deep neural network that computes Q-values and is updated in set intervals to provide a more stable target for updates. Policy Gradient and Actor-Critic methods, such as *Deep Deterministic Policy Gradient (DDPG)* by Lillicrap et al. [LHP+16], lend themselves to be utilized with deep RL methods, since neural networks are a lot more efficient in representing complex, non-linear policies, which are the core of gradient based learning. [SWD+17]

3.8.1 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is an off-policy method of tackling high-dimensional and continuous action spaces, which pose a challenge in reinforcement learning to handle effectively. The method essentially combines DQN [MKS+15] and DPG [SLH+14] with actor-critic methods. The actor uses a deterministic policy $\mu_\phi(s)$ which is parameterized by a deep neural network and directly outputs actions from input states. The critic approximates the action-value function $Q_\theta(s, a)$ which is also parameterized by a deep neural network. Additionally, it utilizes target networks for both the actor and critic, which are updated more slowly by copying the weights from the main networks using a soft update approach. This approach limits the rate of change of the weights, leading to more stability with:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \text{ with } \tau \ll 1 \tag{3.23}$$

The soft update rule displayed in Eq. (3.23) have the parameters of the target network θ' slowly track the parameters of the learned network θ. DDPG also utilizes an Experience Replay Buffer with mini-batch selection [WT13] where past experiences are used to learn additionally to the most recent one.

To update policies, the gradient of the deterministic policy is used which is similar to the gradient ascent method used in stochastic policy gradient methods (Eq. (3.18)) and is defined as follows:

$$\nabla_\phi J(\mu_\phi) = \mathbb{E}_{s\sim S, a\sim\mathcal{D}}[\nabla_a Q_\theta(s, a)\nabla_\phi \mu_\phi(s)] \tag{3.24}$$

with $Q_\theta(s, a)$ being the action-value function, $\mathcal{D}$ representing the experience replay buffer, and μ_ϕ is the deterministic policy. To update the value function, a two step approach is used to compute the loss

$$L = \frac{1}{N}\sum_i (y_i - Q_\theta(s_i, a_i))^2,\ y_i = r_i + \gamma Q'_\theta(s_{i+1}, \mu'_\phi(s_{i+1})) \tag{3.25}$$

with Q'_θ representing the target action-value function and μ'_ϕ being the target policy.

For exploration, instead of the widely used ϵ-greedy approach, a Ornstein-Uhlenbeck noise [UO30] is used, generating a sampling noise $\mathcal{N}$ which gets added to an exploration policy $\mu' = \mu + \mathcal{N}$. The DDPG Algorithm is outlined in Alg. 5

Algorithm 5 DDPG Algorithm

```
1:  Initialize actor $\mu_\phi(s)$, critic $Q_\theta(s,a)$ with random weights $\phi$ and $\theta$
2:  Initialize target networks $\mu'_\phi$ and $Q'_\theta$ with weights $\phi' \leftarrow \phi$ and $\theta' \leftarrow \theta$
3:  Initialize Replay Buffer $\mathcal{D}$
4:  Initialize discount factor $\gamma$
5:  Initialize state $s_0$
6:  for each episode do
7:      Initialize random process $\mathcal{N}$ for exploration noise
8:      Reset environment and observe initial state $s_0$
9:      for each step in episode do
10:         Sample action $a_t$ according to $\mu_\phi(s_t) + \mathcal{N}_t$
11:         Execute action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
12:         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
13:         Sample random mini-batch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $\mathcal{D}$
14:         Set $y_i = r_i + \gamma Q'_\theta(s_{i+1}, \mu'_\phi(s_{i+1}))$
15:         Update critic by minimizing loss $L = \frac{1}{N}\sum_i (y_i - Q_\theta(s_i, a_i))^2$
16:         Update actor policy using the sampled policy gradient

            $$\nabla_\phi J \approx \frac{1}{N}\sum_i \nabla_a Q_\theta(s,a)|_{s=s_i, a=\mu(s_i)} \nabla_\phi \mu_\phi(s)|_{s_i}$$

            Update the target networks:

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

17:     end for
18: end for
```

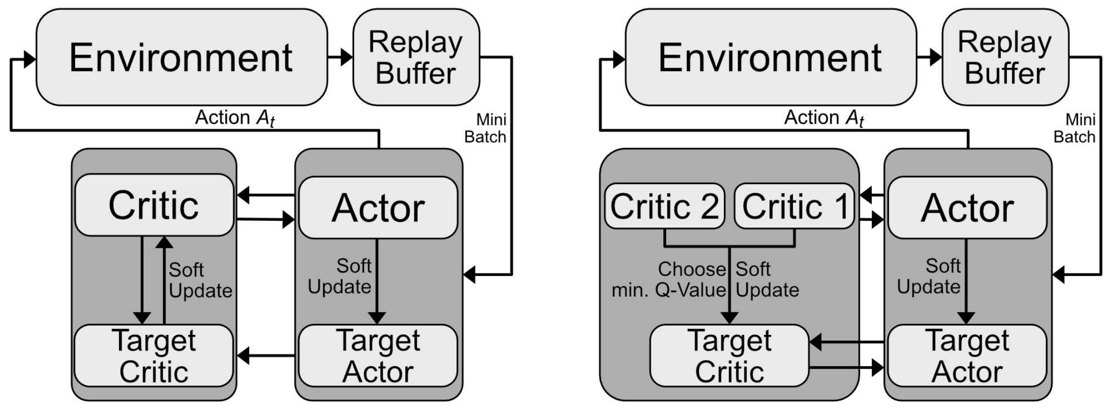


Figure 3.4: Algorithm flow-diagram for deep learning actor-critic methods DDPG (left) and TD3 (right). Recreated from [ZXQ22] P. 6, [JW22]

3.8.2 Twin Delayed Deep Deterministic Policy Gradient

The *Twin Delayed Deep Deterministic Policy Gradient (TD3)* algorithm was introduced as an evolution by Fujimoto et al. [FvHM18], and employs three key innovations that stabilize deep reinforcement learning actor-critic methods, which improves performance. An inherent problem in actor-critic methods, which persists in deep neural networks as well, is an overestimation bias and instability during training that is caused by high variance and function approximation errors in neural networks. Instead of using one critic, TD3 uses two Q-function networks as critics, based on Double Q-Learning [vHGS16]. The key differences between DDPG and TD3 are outlined in their respective algorithm flow charts in Fig. 3.4. Both critics approximated the action-value function $Q_\theta(s,a)$ and for each update step, TD3 uses the minimum value between both Q-networks for the critic update like

$$y = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', a') \tag{3.26}$$

which leads the algorithm to avoid overestimation. TD3 also delays updating the actor compared to the critics in a 1:2 ratio. This allows the Q-values to converge more before the policy is updated and therefore stabilizing the learning process. Lastly, the target policy is smoothed by adding a small random noise to the action selected by the target policy. This reduces over-fitting of the value function towards temporary peaks by using

$$a' = \mu_{\theta'}(s') + \epsilon \tag{3.27}$$

where $\epsilon \sim \text{clip}(\mathcal{N}(0,\sigma), -c, c)$ is a clipped noise that gets added to the action, and c is a small constant. Exploration is the same as in DDPG by using exploration noise rather than ϵ-greedy approaches. [FvHM18] The TD3 algorithm is outlined in Alg. 6.

Algorithm 6 TD3 Algorithm

```
1: Initialize actor $\pi_\phi$ and critics $Q_{\theta_1}$ and $Q_{\theta_2}$ with random parameters $\phi, \theta_1, \theta_2$
2: Initialize target networks $\pi'_\phi$, $Q'_{\theta_1}$ $Q'_{\theta_2}$ with weights $\phi' \leftarrow \phi, \theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2$
3: Initialize Replay Buffer $\mathcal{D}$
4: Initialize discount factor $\gamma$
5: Initialize state $s_0$
6: for each episode do
7:     Initialize random process $\mathcal{N}$ for exploration noise
8:     Reset environment and observe initial state $s_0$
9:     for each step in episode do
10:        Sample action $a_t$ according to $\pi_\phi(s) + \mathcal{N}(0, \sigma)$
11:        Execute action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
12:        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
13:        Sample random mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{D}$
14:        $a' \leftarrow \pi_{\phi'}(s') + \epsilon,\ \epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c)$
15:        Set $y = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_\phi(s'))$
16:        Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} \frac{1}{N} \sum (y - Q_{\theta_i}(s, a))^2$
17:    end for
18:    if *episode* mod $d$ then
19:        Update actor policy $\pi_\phi$ using the sampled policy gradient
               $$\nabla_\phi J(\phi) \approx \frac{1}{N} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$$
           Update the target networks:
               $$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$
               $$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$
20:    end if
21: end for
```

4 Multi-Agent Reinforcement Learning

Multi-agent reinforcement learning (MARL) is an extension to the previous single agent RL case, where multiple agents interact with the same environment and each other. It is based on the same principles as regular RL, where each agent in the system makes an observation of a state s, acts on an action $a(s)$ decided by a policy $\pi(a|s)$ following the Markov Decision Process. Systems in MARL are usually modeled as Markov Games (or Stochastic Games), which generalize having multiple agents in the same environment. Such a Markov Game consists of a finite set of agents $I = \{1, ..., n\}$, finite set of states $S \in \mathcal{S}$, and for each agent $i \in I$ a finite set of actions $A^{(i)} \in \mathcal{A}$, reward function $R^{(i)} : S \times A^{(i)} \times S$, where $A^{(i)} \in \{1, ..., n\}$. Using this extension to the single agent case, a Markov Game can now be described by a tuple $(I, S, A^{(i)}, P, R^{(i)}, \gamma)$ where P is the transition function, which now depends on the joint actions of all agents from from state s_t to the next state s_{t+1}, and γ being the discount factor. The Markov Game is the idealized version of a MARL system, which can be generalized even more by the Partially Observable Markov Decision Process (POMDP). [HBZ04] [ACS24]

For POMDPs, additional definitions for each agent $i \in I$ have to be made. These include a finite set of observations for each agent

$$\begin{gathered}\mathcal{O}^{(i)} : A \times S \times O^{(i)} \to [0, 1] \text{ such that} \\ \forall a \in A, s \in S : \sum_{o^{(i)} \in O^{(i)}} \mathcal{O}^{(i)}(a, s, o^{(i)}) = 1\end{gathered}$$

The flow of a POMDP is similar to a regular MDP, with the main difference being that each agent only observes what it can see, which may not be every parameter of the environment or actions of other agents as outlined in Fig. 4.1. Therefore, at the beginning of a time step t, each agent $i \in I$ receives an observation $o_t^{(i)} \in O_i$, given based on the observation function $\mathcal{O}^{(i)}(o_i^t | a_{t-1}, s_t)$, with the probability being conditioned on the current state and previous joint action of the system. Each agent then chooses an action $a_t^{(i)} \in A^{(i)}$ based on its own policy $\pi^{(i)}(a_t^{(i)} | h_t^{(i)})$, with $h_t^{(i)} = (o_0^{(i)}, ..., o_t^{(i)})$ representing the observation history each agent, resulting in the joint action $a_t = (a_t^1, ..., a_t^n)$. After the environment transitions into a new state according to it's transition function $P(s_{t+1} | s_t, a_t)$ each agent receives its reward $r^{(i)} = R^{(i)}(s_t, a_t, s_{t+1})$. The property of an agent only observing its own information and not the whole state and actions of other agents leads to a set of challenges that are subject of current research. [ACS24]

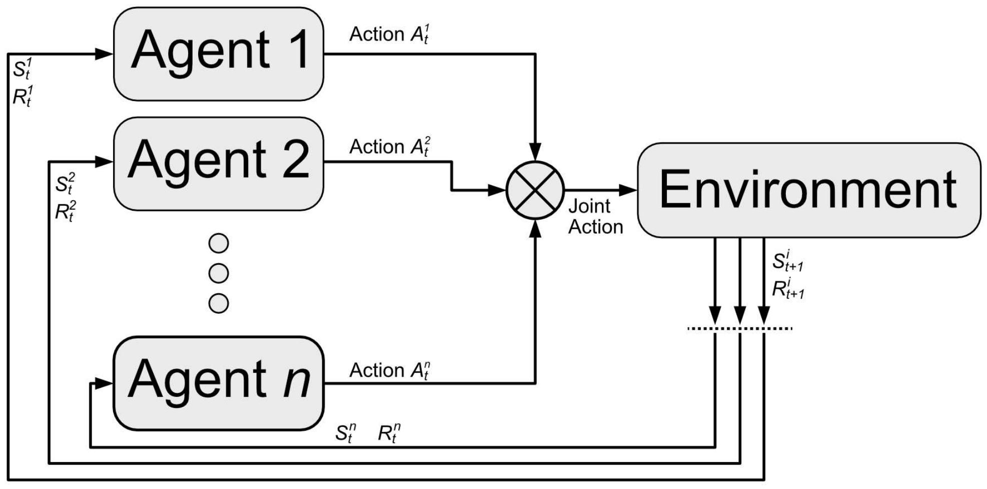


Figure 4.1: Flow-diagram for the agent-environment interaction in multi-agent reinforcement learning settings. Recreated from [ACS24] P. 6

4.1 Challenges of MARL

Non-Stationarity There are four major challenges in MARL that are not present in single-agent RL. The first challenge is non-stationarity which is caused by the learning process of agents themselves. In essence, agents in a MARL system learn the same way as in the single-agent case. This leads to agents acting on their own observations, with their actions influencing the next state of the environment, which in turn influences the next observation of each agent. Having the environment transition into a state that is influenced by more than one agent, leads to the policy of an agent to adapt to environment changes induced by agents. Given that each agent also has its own local observations $o^{(i)}$, which are usually a subset of the complete state space, and is given its own reward $r^{(i)}$, they tend to "learn in circles" around each other. Competitive MARL algorithms need to compensate for this problem of non-stationarity the most. [LMF11] [ACS24]

Equilibrium Selection The second challenge in MARL is the question of when the policies of agents in a multi-agent system are optimal. Unlike in the single-agent case, where a policy π^* can be considered optimal when it receives the maximum expected reward for every state, policies of agents in multi-agent systems depend on the other agents' policies. Introducing a measurement of optimality in MARL is part of equilibrium selection. There are a plethora of equilibrium solutions to measure optimality in a multi-agent system, but each equilibrium could lead to different returns on a per agent basis. Therefore an algorithm needs to select a proper equilibrium measure, such as Best Response [SL09] where each agent tries to maximize its own reward or the Nash Equilibrium which states that each policy $\pi^{(i)}$ is a best response to a joint policy π [Nas50], to arrive at learning robust policies. [HS88] [ACS24]

Multi-Agent Credit Assignment For an agent to know how much it contributed is the challenge of multi-agent credit assignment, which is especially prominent in scenarios with a common reward, like in the environments used for experiments later on in this thesis. In a decentralized setting where every agent is its own closed system of algorithm and replay buffer, using the Agent-Environment-Cycle, introduced in the *PettingZoo* framework [TBG+21], can assign the individual rewards $r_t^{(i)}$ for each agent $i \in I$, since the environment updates after each agent acts on the environment inside each time step t. In a centralized setting, using a joint action-value function $Q_t(s_t, a_t^1, ..., a_t^n)$ can correctly attribute the impact of each action $a_t^{(i)}$ done by an agent $i \in I$. [TA07] [ACS24]

Scalability Finally, the ability to efficiently scale to many agents in MARL systems is a vital challenge, since with a growing number of agents, the dimension of joint actions can grow exponentially with $|A| = |A^1| \cdot ... \cdot |A^n|$. While the growth of dimensionality with an increasing number of agents is an inevitable consequence on its own, it also affects all other challenges as well. Introducing more agents increases the degree of non-stationarity of the system, while multi-agent credit assignment is also more complicated. [ACS24]

4.2 Multi-Agent Deep Reinforcement Learning

Just as in single-agent RL, multi-agent deep reinforcement learning is the current state of the art way of learning policies. Training agents in multi-agent settings can be as straight forward as applying single agent methods on each agent for independent learning, include modeling of other agents, or information sharing with centralized methods. Learning in MARL can thus be divided into three main categories which are based on their modes of training and execution.

Centralized Training and Centralized Execution (CTCE) In centralized training and execution, agents utilize shared information or other mechanisms that are centrally shared with all agents in the system. This centrally shared information may contain the local observation and action history of each agent, agent models, value functions, or the policy of other agents. Methods using this approach during training and execution reduces the multi-agent problem to a single-agent problem, by effectively training a joint policy using the joint-observation history over a joint-action space. With every agent having access to all observations, the challenge of non-stationarity is practically removed, however this approach may not be feasible for certain types of systems. Since central policies have to learn with a joint-action space, the dimension of such a space grows exponentially with the number of agents. Issues with joint reward distribution across all agents can arise since the rewards for the system have to be transformed into a single reward during training. Lastly, CTCE can be problematic for agents that are either virtually or physically distributed which can lead to communication issues with a central policy. [ACS24]

Decentralized Training and Decentralized Execution (DTDE) This approach can also be understood as independent learning, where training and execution of each agent policy is done fully decentralized with no information sharing. This mode of training is the natural method for situations where agents do not have information about other participants, such as trading in financial markets [LO02]. In DTDE, each agent only has access to its own observations and does not model the existence of other agents as they are viewed as non-stationary parts of the environment. This means that agents are trained using single-agent RL methods, which has the advantage of being very scalable since it avoids having joint-observation and joint-action spaces. While also being the natural choice for systems where agents are distributed entities, it has the downside of agents not being able to leverage any information between each other which can lead to significant issues with non-stationarity. There are a number of methods to alleviate the challenge of non-stationarity, one of them is modeling other agents during training by encoding agent behavior into embeddings used during training. [ACS24] [FCA+18]

Centralized Training and Decentralized Execution (CTDE) Centralized training and decentralized execution is a hybrid approach combining both previous methods. During training, agents utilize centrally shared information of other agents to update, while each agent's policy itself only needs its own local information. This leads to the agents being able to be deployed without needing access to information of other agents during execution. CTDE methods are quite common in deep MARL, especially for actor-critic methods where a centralized critic is used to provide feedback during training to the individual agents based on information from all agents. This is achieved by the critic being able to better approximate the value function using the joint-observation and joint-action history, while the actors only have access to their local observation and action history. Since the value function isn't needed during execution, the centralized critic can be discarded after deployment. [ACS24] [LWT+17]

This thesis is utilizing both DTDE and CTDE approaches for experiments involving encoded representations of agents in multi-agent systems. For DTDE, the earlier mentioned DDPG (Alg. 5) as well as its evolution TD3 (Alg. 6) are applied, while for CTDE their centralized multi-agent variants MADDPG (Alg. 7) and MATD3 (Alg. 8) are in use.

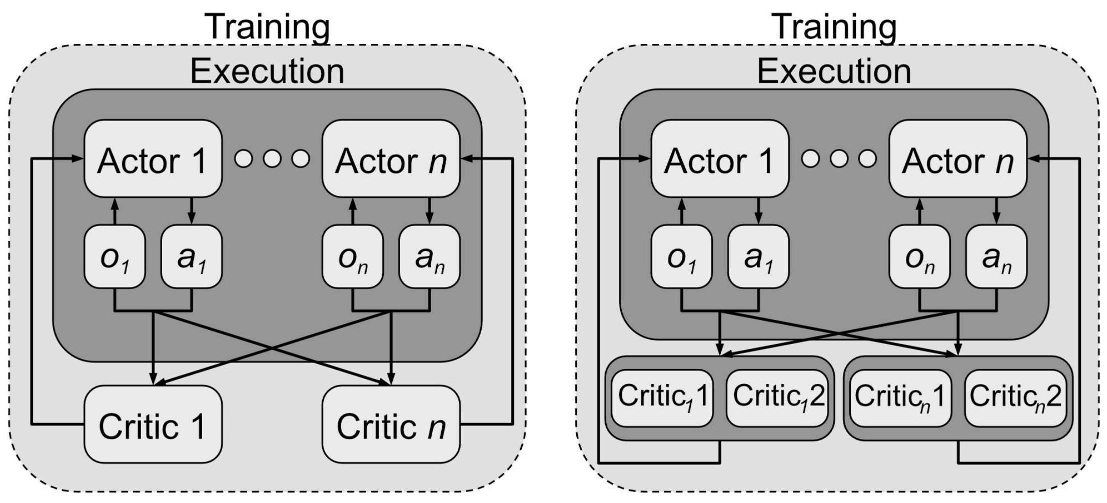


Figure 4.2: Algorithm flow-diagram for the CTDE methods of MADDPG (left) and MATD3 (right). Recreated from [FCA+18] P. 4

4.2.1 Multi-Agent Deep Deterministic Policy Gradient

Multi-Agent Deep Deterministic Policy Gradient (MADDPG) by Lowe et al. [LWT+17] is a direct multi-agent evolution of the DDPG algorithm, which is utilizing the CTDE paradigm of learning. Compared to other CTDE methods, like *Counterfactual Multi-Agent Policy Gradient* by Foerster et al. [FFA+18] which use one centralized critic, MADDPG has one centralized critic for each actor. This approach leads for the algorithm to be highly versatile in cooperative, competitive and mixed settings, since every policy is conditioned with its own value function.

Actors are trained using deterministic policies, like in the single agent case, with the addition of having multiple of those policies at once, which leads to a new notation of

$$\nabla_{\theta^{(i)}} J(\mu^{(i)}) = \mathbb{E}_{\mathbf{x},a\sim\mathcal{D}}\left[\nabla^i_\theta \mu^{(i)}(a^{(i)}|o^{(i)})\nabla_{a^{(i)}} Q^{(i)}_\mu(\mathbf{x},a^1,...,a^n)|_{a^i=\mu^{(i)}_\theta(o^{(i)})}\right] \tag{4.1}$$

with $\mathbf{x} = (o^1, ..., o^n)$ representing the observations of all agents, $Q^{(i)}_\theta$ is the action-value function for the centralized critic of agent i, and $\mathcal{D}$ being the Experience Replay Buffer containing current state, next state, action, and reward of all agents $(\mathbf{x_t}, \mathbf{x_{t+1}}, a^1, ..., a^n, r^1, ..., r^n)$. The update for the value function can be extended to

$$L(\theta^{(i)}) = \frac{1}{S}\sum_j \left(y_j - Q^{(i)}_\mu(\mathbf{x}_j, a^1_j, ..., a^n_j)\right)^2,\ y_j = r^{(i)}_j + \gamma Q^{(i)}_{\mu'}(\mathbf{x}'_j, a'_1, ..., a'_n)|_{a'_k=\mu'_k(o^k_j)} \tag{4.2}$$

where $\mu' = \{\mu_{\theta'_1}, .., \mu_{\theta'_n}\}$ is the set of target policies with the delayed parameters $\theta'^{(i)}$. The algorithm itself is outlined in Alg. 7.

Algorithm 7 MADDPG Algorithm

```
1: Initialize actor and critic networks
2: Initialize Replay Buffer $\mathcal{D}$
3: Initialize discount factor $\gamma$
4: Initialize state $\mathbf{x}_0$
5: for each episode do
6:    Initialize random process $\mathcal{N}$ for exploration noise
7:    Reset environment and observe initial state $\mathbf{x}_0$
8:    for each step in episode do
9:       For each agent $i$, sample action $a_t^{(i)}$ according to $\mu_\theta^{(i)}(o^{(i)}) + \mathcal{N}(0, \sigma)$
10:      Execute actions $a_t^{(i)} = (a_t^1, ..., a_t^n)$ and observe reward $r_t^{(i)}$ and next state $\mathbf{x}_{t+1}^{(i)}$
11:      Store transition $(\mathbf{x}_t, a_t^{(i)}, r_t^{(i)}, \mathbf{x}_{t+1})$ in $\mathcal{D}$
12:      Set $\mathbf{x}_t \leftarrow \mathbf{x}_{t+1}$
13:      for each agent $i = 1$ to $n$ do
14:         Sample random mini-batch of $S$ transitions $(\mathbf{x}_j, a_j, r_j, \mathbf{x}'_j)$ from $\mathcal{D}$
15:         Set $y_j = r_j^{(i)} + \gamma Q_{\mu'}^{(i)}(\mathbf{x}'_j, a'_1, ..., a'_n)|_{a'_k=\mu'_k(o_j^k)}$
16:         Update critics by minimizing $\mathcal{L}(\theta^{(i)}) = \frac{1}{S}\sum_j \left(y_j - Q_\mu^{(i)}(\mathbf{x}_j, a_j^1, ..., a_j^n)\right)^2$
17:         Update actor using the sampled policy gradient:
               $$\nabla_{\theta^{(i)}} J \approx \frac{1}{S}\sum_j \left[\nabla_{\theta^{(i)}} \mu^{(i)}(o_j^{(i)}) \nabla_{a^{(i)}} Q_\mu^{(i)}(\mathbf{x}_j, a_j^1, ..., a_j^n)|_{a^{(i)}=\mu^{(i)}(o_j^{(i)})}\right]$$
18:      end for
19:      Update target network for each agent $i$: $\theta'^{(i)} \leftarrow \tau\theta^{(i)} + (1-\tau)\theta'^{(i)}$
20:   end for
21: end for
```

4.2.2 Multi-Agent Twin Delayed Deep Deterministic Policy Gradient

Much like MADDPG for DDPG, *Multi-Agent Twin Delayed Deep Deterministic Policy Gradient (MATD3)* by Ackermann et al. [AGOS19] is the multi-agent evolution of the TD3 algorithm. It uses the same key concepts as the single-agent algorithm (Alg. 6) with the CTDE concepts of MADDPG. This results in an actor-dual-critic setup for every agent in the algorithm. As with TD3, the centralized critics for each agent use two Q-networks to approximate the value function, with the lower Q-value being used for updates. Additionally, the concept of delaying the policy updates for each actor by a factor d, prevents the actor from using unstable Q-values and therefore improving learning stability. While the policy updates are the same as for MADDPG (Eq. (4.1)), the update for the critic is as follows:

$$y_j = r_j^{(i)} + \gamma \min_{l=1,2} Q_{\mu'}^{(i),l}(\mathbf{x}'_j, a'_1, ..., a'_n)|_{a'_k=\mu'_k(o_j^k)+\epsilon} \tag{4.3}$$

with $\epsilon = \text{clip}(\mathcal{N}(0,\sigma), -c, c)$ being a clipped Gaussian noise that's added to the actions of all agents. Much like TD3 improving on DDPG, their respective multi-agent algorithms behave the same way with MATD3 producing consistently better results than MADDPG since learning is more stable using a dual-critic approach with delayed policy updates. The algorithm is as shown in Alg. 8.

Algorithm 8 MATD3 Algorithm

```
1: Initialize actor and critic networks
2: Initialize Replay Buffer $\mathcal{D}$
3: Initialize discount factor $\gamma$
4: Initialize state $\mathbf{x}_0$
5: for each episode do
6:   Initialize random process $\mathcal{N}$ for exploration noise
7:   Reset environment and observe initial state $\mathbf{x}_0$
8:   for each step in episode do
9:     For each agent $i$, sample action $a_t^{(i)}$ according to $\mu_\theta^{(i)}(o^{(i)}) + \mathcal{N}(0, \sigma)$
10:    Execute actions $a_t^{(i)} = (a_t^1, ..., a_t^n)$ and observe reward $r_t^{(i)}$ and next state $\mathbf{x}_{t+1}^{(i)}$
11:    Store transition $(\mathbf{x}_t, a_t^{(i)}, r_t^{(i)}, \mathbf{x}'_{t+1})$ in $\mathcal{D}$
12:    Set $\mathbf{x}_t \leftarrow \mathbf{x}'_{t+1}$
13:    for each agent $i = 1$ to $n$ do
14:      Sample random mini-batch of $S$ transitions $(\mathbf{x}_j, a_j, r_j, \mathbf{x}'_j)$ from $\mathcal{D}$
15:      Set $y_j = r_j^{(i)} + \gamma \min_{l=1,2} Q_{\mu'}^{(i),l}(\mathbf{x}'_j, a'_1, ..., a'_n)|_{a'_k=\mu'_k(o_j^k)+\epsilon}$
16:      Update both critics $l = 1, 2$ by minimizing
             $$L(\theta^{(i),l}) = \frac{1}{S}\sum_j \left(y_j - Q_\mu^{(i),l}(\mathbf{x}_j, a_j^1, ..., a_j^n)\right)^2$$
17:      if episode mod $d = 0$ then
18:        Update actor using the sampled policy gradient:
             $$\nabla_{\theta^{(i)}} J \approx \frac{1}{S}\sum_j \left[\nabla_{\theta^{(i)}}\mu^{(i)}(o_j^{(i)})\nabla_{a^{(i)}} Q_\mu^{(i)}(\mathbf{x}_j, a_j^1, ..., a_j^n)|_{a^{(i)}=\mu^{(i)}(o_j^{(i)})}\right]$$
19:        Update target network for each agent $i$: $\theta'^{(i)} \leftarrow \tau\theta^{(i)} + (1-\tau)\theta'^{(i)}$
20:      end if
21:    end for
22:  end for
23: end for
```

4.3 Learning with Embeddings

The core idea to help with the problem of non-stationarity in MARL is to use embeddings of agents in multi-agent systems, to address challenges of generalization and lack of information sharing between agents in decentralized learning systems. To this end a framework is introduced where the observation-action pairs of an agent get embedded into a space which is lower-dimensional than the original observation-action space. These embeddings are trained to capture the essential aspects and behavior of an agent's policy, and allows agents to be represented in an abstract form. Providing agents with these abstractions, they can than imply the other agents' intentions and strategies, which can lead to better adaptation to the other agent. To create such embeddings, an encoder in the form of a neural network is used, which maps an agent's behavior to its corresponding embedding. This encoder is trained using state-action pairs from the agent during roll out, which then get mapped to the the lower dimensional space. These lower dimensional embeddings then contain a reflection of the base agent's behavior. While training this encoder network, an imitation policy is also trained to reconstruct the behavior of the original policy, while also including embeddings which get appended to its observations. [GAG+18]

The general approach for learning the embeddings used in the thesis is based on the paper *Learning Policy Representations in Multiagent Systems* by Grover et al. [GAG+18] by using imitation learning to train a generative representation of an agents policy, while simultaneously training an embedding network based on the triplet loss for agent identification. The representation function utilizes episodes from an agent in a policy $\pi^{(i)} \in \Pi$, where Π are representable policies. To involve additional embedding parameters, this policy is optimized for parameters θ of function $f_\theta : \mathcal{E} \to \mathbb{R}^d$ where $\mathcal{E}$ is the space of observation-action pairs in an episode for the corresponding policy, and d is the dimension of the embedding. Since this approach is unsupervised, the agent policies are not dependent on their reward, but rather just on the interactions between agents. To learn a representation of agent i, a policy using $E_i = \cup_j E_{ij}^{(i)}$ where $E_{ij}^{(i)}$ represents the interaction between agent i and agent j during an episode. Such episodes are described by their sequence of observations and actions.

To learn an imitation policy $\pi_\theta^{(i)} : \mathcal{O} \times \mathcal{A} \to [0,1]$ for agent i using its observation-action data $e \in E_i$, the cross-entropy objective is used.

$$L_{Imitation}(\theta) = \mathbb{E}_{e \in E_i}\left[\sum_{\langle o,a\rangle \sim e}\left[\log \pi_\theta^{(i)}(a|o)\right]\right] \tag{4.4}$$

To learn a single conditional policy, a representation function $f_\theta : \mathcal{E} \to \mathbb{R}^d$ is used to condition the policy network $\pi_{\phi,\theta} : \mathcal{O} \times \mathcal{A} \times \mathcal{E} \to [0,1]$ where θ and ϕ are parameters of neural networks mapping the observation and embedding to the agent's actions. Those neural networks with parameters θ and ϕ are learned jointly by maximizing the cross-entropy objective incorporating two distinct episodes of agent i. By introducing two distinct episodes $e_1 \in E_i$ and $e_2 \in E_i \setminus e_1$ into the cross-entropy in Eq. (4.4), the objective can be expanded to include embeddings.

$$L_{Imitation} = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_{\substack{e_1 \in E_i \\ e_2 \in E_i \setminus e_1}} \left[\sum_{\langle o,a \rangle \sim e_1} \log \pi_{\phi,\theta}(a|o, f_\theta(e_2)) \right] \tag{4.5}$$

The observation-action pairs of e_2 are used to learn an embedding $f_\theta(e_2)$ which is appended to the observation of e_1 to condition the imitation policy network which is trained using the observations and actions of e_1.

The embedding function f_θ is trained on a neural network which takes observation-action pairs of three distinct episodes and maximizes an objective function based on the triplet loss [SJ03]. Two episodes, one positive episode $e_+ \sim E_i$ and one reference episode $e_* \sim E_i$ different to e_+ of agent i are selected, as well as one negative episode $e_- \sim E_j$ of agent j. The embeddings of all three episodes are used to optimize for triplet loss

$$L_{Identification} := d_\theta(e_+, e_-, e_*) = (1 + \exp\{||r_e - n_e||_2 - ||r_e - p_e||_2\})^{-2} \tag{4.6}$$

with $p_e = f_\theta(e_+)$, $n_e = f_\theta(e_-)$, $r_e = f_\theta(e_*)$. This loss function pushes positive embeddings closer to reference embeddings and away from negative embeddings. Combining objectives Eq. (4.5) and Eq. (4.6) results in the final objective to jointly train an imitation policy and embedding network:

$$L(\theta, \phi) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_{\substack{e_+ \sim E_i \\ e_* \sim E_i \setminus e_+}} \left[\sum_{\langle o,a \rangle \sim e_+} \log \pi_{\phi,\theta}(a|o, f(e_*)) \right] - \lambda \sum_{j \neq i} \mathbb{E}_{e_- \sim E_j} [d_\theta(e_+, e_-, e_*)] \tag{4.7}$$

where $\lambda > 0$ is a hyperparameter used to tune the impact of the triplet loss on the objective function. In practice, conditional policy network $\pi_{\theta,\phi}$ and corresponding embedding function f_θ are parameterized using neural networks using the Adam optimizer [KB15].

5 Related Works

While deep learning revolutionized the development of agents that can autonomously act in complex environments, scalability, non-stationarity and multi-agent credit assignment are still problems that need to be solved. [PCRA19] While purely centralized training and centralized execution methods can lead to problems during deployment, especially for environments where agents ultimately have to act on their own, centralized training and decentralized execution has been a very promising field of research in recent years.

Centralized Training and Decentralized Execution By only having a centralized critic during training, the agents perceive the environment as stationary while the policies are conditioned and can act independently during execution, since the critic is not needed anymore. One of the first methods showcasing this was *COMA* by Foerster et al. [FFA+18], where a single central critic evaluated the policies of multiple agents for micromanaging tasks in StarCraft. Other methods in this category are *MADDPG* by Lowe et al. [LWT+17], *MATD3* by Ackermann et al. [AGOS19] and *Multi-Agent PPO (MAPPO)* by Yu et al. [YVV+22] which utilize a centralized critic for each policy that has access to the global data from all agents.

Self Play When it comes to decentralized learning techniques, one of the first approaches to handle non-stationarity was to utilize self-play. This method was pioneered by Tesauro [Tes95] which managed to beat the Backgammon champion at the time. A neural network uses each agents' observations as input, and the agents play against their current policy or one from a previous checkpoint. Self-play has been extended into much more complex domains such as the board game Go, where the *AlphaGo* method by Silver et al. [SHM+16] which famously beat the world champion in 2017.

Agent Approximation Another approach to augment observations of an agent is to model other agents. One method by Raileanu et al. [RDSF18] which uses an actor-critic approach and expands the input from just being the current state to also include the goal of one agent and another agent $f(o_{f/o}, z_s, \overline{z}_o)$ with z_s being the goal of the "self" agent and $\overline{z}_o$ is the goal of the other agent. Methods like *Learning with Opponent Learning Awareness (LOLA)* by Foerster et al. [FCA+18] include an additional learning rule that accounts for the impact of one agent's policy on the expected policy update of the other agent in 2-player tit-for-tat games.

Modeling Agents via Embeddings Using representations of agents is an approach where the characteristics, general strategies and behavior of an agent is encoded into a neural network to be leveraged by other agents in order to boost generalization when encountering new opponents. One such approach is *Machine Theory of Mind* by Rabinowitz et al. [RPS+18], where a Theory of Mind network tries to infer another agent's beliefs, goals and preferences from limited observations. Generating agent embeddings in a single agent setting has been researched by Chang et al. [CKCL19] where the neural networks of agents are compressed into a low-dimensional embedding by vectorizing their weights and learning a generative model from it. *Learning Meta Representations for Agents in Multi-Agent Reinforcement Learning* by Zhang et al. [ZSHS23] proposes a framework to generalize agents across varying population sizes by capturing meta representations. These representations aim to learn common strategic knowledge and strategic relationships between different types of agents eg. predator and prey agents. *Attention-Guided Contrastive Role Representations for Multi-agent Reinforcement Learning* by Zican et al. [HZL+24] aims to to capture the behavior of agents into compact role representations to enhance coordination and heterogeneity between agents. For this, they propose a method to learn distinct roles by maximizing mutual information between agents while also including a mechanism to focus on these representations during training.

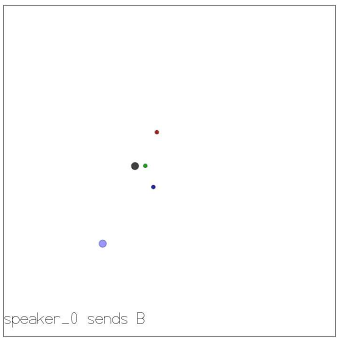
Ad Hoc Teamwork Recent research on the topic of enhancing teamwork between agents is the use of ad hoc methods, where agents dynamically detect changes in their teammates behavior and try to adapt new strategies. The key part of ad hoc teamwork is that agents can dynamically adapt to each other without prior coordination or communication. One such method proposed by Ravula et al. [RAS19], uses a so called Change Point Detection (CPD) network to monitor the observations in the environment utilizing a Convolutional Neural Network (CNN). The advantage of using a CNN is that it can use a sequence of observations of the whole environment and detect abrupt change more efficiently. Upon detecting sudden changes in the agents behavior, its teammate can then adapt by using learned representations. A more comprehensive survey about the key challenges of ad hoc learning where one agent is trained using this approach is presented by Mirsky et al. [MCR+22]. Since this method usually assumes control of one agent during learning, applying it to MARL settings can be difficult when it comes to the problem of scalability or poor performance of an ad hoc trained agent when encountering unseen teammates.

6 Methodology

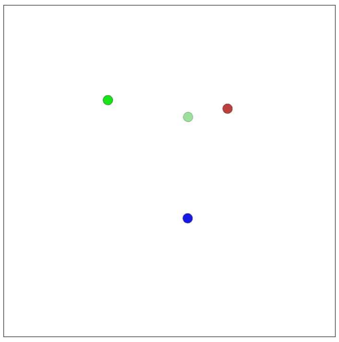
To tackle the goal of implementing agent representations during training, embeddings are utilized. These representations encode observation-action pairs of an agent from their original dimensions into a lower dimensional embedding, which is appended to the observation of another agent. This approach is inspired by Grover et al.'s [GAG+18] work, which is highlighted in Chapter 4.3. This method utilizes embedded agent representations, as well as jointly trained imitation policies to train agents utilizing privileged information about the behavior of their counterparts for a wide variety of strategies. This can lead to better performance for the agent with this additional information and also to better overall performance in cooperative settings, or to better single agent results in asymmetric competitive environments. The goal is to expand on this method by training agent representations of one or more agents at once, in a multitude of environments. While the original paper showed results on a one versus one competitive and a two agent cooperative tasks, the methodology presented here is augmenting this approach by scaling up the amount of agents in an environment, capturing the combined behavior of multiple agents at once and applying those embeddings to one or more agents of the same nature. This is applied, not only in strictly competitive and cooperative environments, but also in mixed settings using multiple embedding networks where a subset of agents tries to cooperate with each other while competing against another subset of agents in the same environment.

6.1 Environments

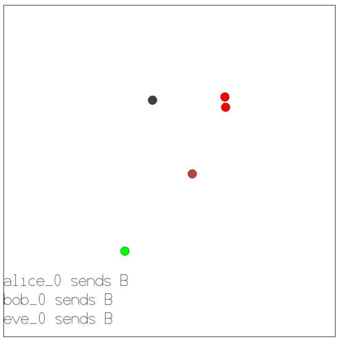
In reinforcement learning, agents interact with an environment based on the observations they make at each time step. When an agent makes an action upon the environment, the environment evolves accordingly and provides a reward, which is usually a scalar value, back to the agent based on that action. Agents can observe the state of an environment as a state vector that is directly provided by the environment, or through images that are usually processed using convolutional neural networks. In multi-agent reinforcement learning, multiple agents interact in a shared environment and with each other, while only being able to observe parts of it. Multi-agent environments can usually be classified to be either cooperative, competitive or mixed where at least two teams of agents compete against each other, while cooperating with their respective team mates.



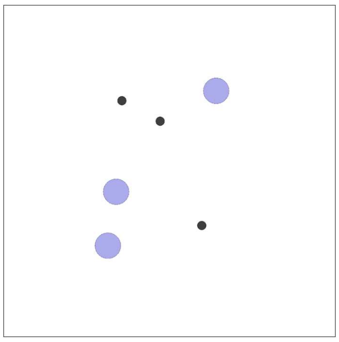
(a) Simple Speaker Listener



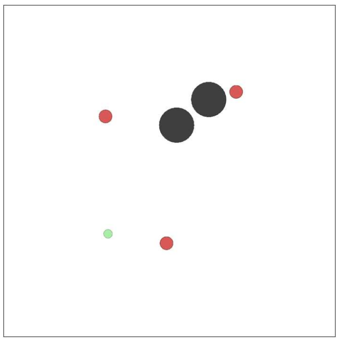
(b) Simple Push



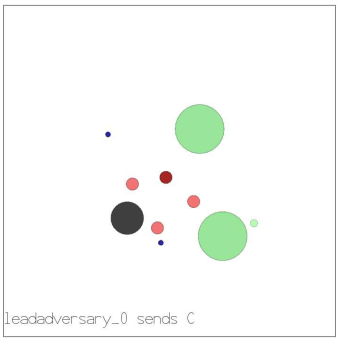
(c) Simple Crypto



(d) Simple Spread



(e) Simple Tag



(f) Simple World Comm

Figure 6.1: Multi-Agent Particle Environments used for experiments.

The environments used for the experiments in this thesis are widely used benchmarks consisting of cooperative, competitive and mixed settings with varying number of agents, known as the *OpenAI Multi-Agent Particle Environment* first introduced by Lowe et al. [LWT$^+$17]. The environments in this suite of benchmarks are based on two dimensional grid world scenarios that incorporate cooperative communication tasks, competitive predator-prey settings, as well as team based combinations of both. Agents in these environments can see each other, in most cases move around, push each other, and interact with fixed landmarks. For evaluation the following environments are chosen:

- Cooperative environments
  - Simple Speaker Listener (Fig. 6.1a): A two agent environment, where a static *speaker* agent observes the color of one of three fixed target landmarks, as well as giving commands to the *listener* agent. The listener then tries to get as close as possible to the target landmark observed by the *speaker*.
  - Simple Spread(Fig. 6.1d): A three agent environment, where three agents try to get as close as possible to three fixed landmarks, while taking care not to hit each other. Therefore agents need to learn to coordinate between each other which agent covers which landmark while avoiding collisions.
- Competitive environments
  - Simple Push(Fig. 6.1b): This environment features two agents, one *good* agent which tries to get as close as possible to a fixed landmark, while the *bad* agent tries to push the former away.
  - Simple Tag(Fig. 6.1e): A four agent environment, which features three hunter agents and one prey agent, as well as two fixed obstacles. The the prey agent tries to escape the three, slightly slower moving hunter agents, while not straying too far away from the point of origin.
- Mixed environments
  - Simple Crypto(Fig. 6.1c): Here, two agents *Alice* and *Bob* try to send an encrypted message between each other, while a bad agent *Eve* tries to reconstruct and read the message before the two good agents manage to do the same.
  - Simple World(Fig. 6.1f): The most complex environment which features multiple hunter agents, which are coordinated by an additional overseer, as well as multiple prey agents that try to collect *food* without getting caught by the hunters.

A more detailed description of all environments, along the observation and action spaces of each agent, can be found in the respective sections in Ch. 8.

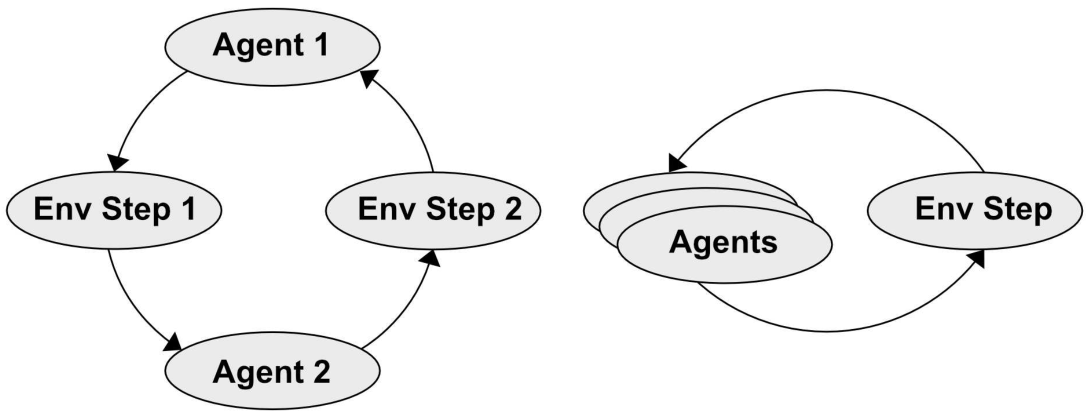


Figure 6.2: PettingZoo environment interactions. Agent-Environment-Cycle (AEC) left, parallel environment interaction right. Recreated from [TBG+21] P. 7

The Multi-Agent Particle Environments (MPE) [LWT+17] are implemented in the *PettingZoo* library [TBG+21], which is a popular framework for multi-agent reinforcement learning research. PettingZoo has two modes of environment execution illustrated in Fig. 6.2:

1. The Agent Environment Cycle (AEC), which is a turn based approach where agents take actions sequentially and therefore interact with the environment one after another. After an agent acts on the environment, the environment gets updated and provides new observations for the next agent based on the action of the previous agent. The time step of an episode gets advanced once all agents have acted on the environment once.

2. Parallel interaction, where all agents act simultaneously. Contrary to AEC this leads to every agent having the same observation and acting upon it, without taking into consideration the actions of other agents at the current time step.

6.2 Learning Framework

For this thesis, two different approaches to train agents in a multi-agent setting are considered. Decentralized learning with decentralized execution (DTDE), as well as centralized learning with decentralized execution (CTDE). Both methods are based on offline deep learning actor-critic algorithms. These algorithms learn by utilizing previous experiences collected during training. They employ multiple neural networks, one for the actor and one or more for the critic per agent, depending on the algorithm. The actor network is the policy that selects an action based on the state the agent observes, while the critic network represents the action-value function that gives feedback and guides the actor during training by limiting updates to smaller values. [KT99]. The key difference between DTDE and CTDE is that decentralized training employs one actor-critic policy with separate replay buffer, thus only having access to the local information for each agent, while centralized training has a shared replay buffer and global critics to guide all agents using all observations and actions.

For decentralized training, *Deep Deterministic Policy Gradient* (DDPG) [LHP$^+$16] (Ch. 3.8.1) and *Twin Delayed Deep Deterministic Policy Gradient* (TD3) [FvHM18] (Ch. 3.8.2) are utilized to train agents. The difference between the two algorithms is that TD3 is an evolution of DDPG which uses two critic networks to determine better guidance for the actor network, while also delaying updates to the policy network to smooth out the learning process. The Agent Environment Cycle (Fig. 6.2 left) is used since every agent has its own individual policy and replay buffer. Here each agent observes the environment and acts according to its policy in succession for each time step in an episode. Since every agent is its own *closed system* there is no communication or sharing of experience between each other, which is outlined in Alg. 10.

In the case centralized training, the dedicated multi-agent variants for DDPG and TD3 are utilized. *Multi Agent Deep Deterministic Policy Gradient* (MADDPG) [LWT$^+$17] (Ch. 4.2.1) and *Multi Agent Twin Delayed Deep Deterministic Policy Gradient* (MATD3) [AGOS19] (Ch. 4.2.2) are based on the same principle as their single agent counterparts but utilize global critics for each actor and shared memory. These global critics guide the individual actors based on all observations and actions taken by all agents during training. Since all agents share the same replay buffer and critics, parallel environment interaction (Fig. 6.2 right) is chosen to train policies. Here all agents choose actions based on the same observation, without the environment updating for each agent like in the case for DTDE. The implementation for this training method is outlined in Alg. 12.

6.3 Embeddings

Embeddings representing the behavior of agents are trained to convey what an agent sees and how it wants to act in an encoded fashion. They are in a lower dimensional space than the original observation-action pairs. The dimension of embeddings is chosen as roughly half of the dimension of the corresponding observation-pairs, to strike a balance between retaining information of the embedded agent and the additional computational load of encoding observation-action pairs at each time step. All input and embedding dimension being listed in the environments table Tab 6.1. To train these embeddings the approach outlined in Grover et al. [GAG$^+$18] (Ch. 4.3) is implemented. An encoding network, alongside an imitation policy network, is trained on full episodes of an agent which are recorded during evaluation runs from a set of baseline policies. For a chosen agent i two episodes e_+ and e_* are selected with $e_+ \neq e_*$ which are referred to as positive episode e_+ and reference episode e_*. The reference episode serves as basis to train the imitation network, while the embedding network also utilizes both e_+, e_*, and a negative episode e_- from an agent $j \neq i$ of a different policy. This conditions the imitation and embedding networks to be closer tied to the agent of which the positive and reference episodes are selected. To train these embeddings, a positive and reference episode e_+ and e_* with $e_+ \neq e_*$ are selected from an agent i, and the imitation loss in Eq. (4.5) is calculated. Further, a negative episode e_- from every agent $j \neq i$ is selected to calculate the triplet loss in Eq. (4.6), and then jointly optimize the imitation policy network and embedding network using the combined loss function Eq. (4.7). The Algorithm for training the imitation and embedding networks is shown in Alg. 9:

Algorithm 9 Policy Embedding Function f_θ

```
1:  Initialize $\theta$ and $\phi$
2:  **for** $i = 1$ to $n$ **do**
3:      Sample a positive episode $p_e = e_+ \sim E_i$
4:      Sample a reference episode $r_e = e_* \sim E_i \setminus e_+$
5:      Compute Imitation Loss $-\sum_{\langle o,a\rangle \sim e_+} \log \pi_{\phi,\theta}(a|o, e_*)$
6:      **for** $j = 1$ to $n$ **do**
7:          **if** $j \neq i$ **then**
8:              Sample a negative episode:$n_e = e_- \sim E_j$
9:              Compute Identification Loss $d_\theta(e_+, e_-, e_*)$
10:             Set Loss $\leftarrow$ Imitation Loss $+ \lambda$ Identification Loss
11:             Update $\theta$ and $\phi$ to minimize $Loss$
12:         **end if**
13:     **end for**
14: **end for**
15: Return $\theta$ and $\phi$
```

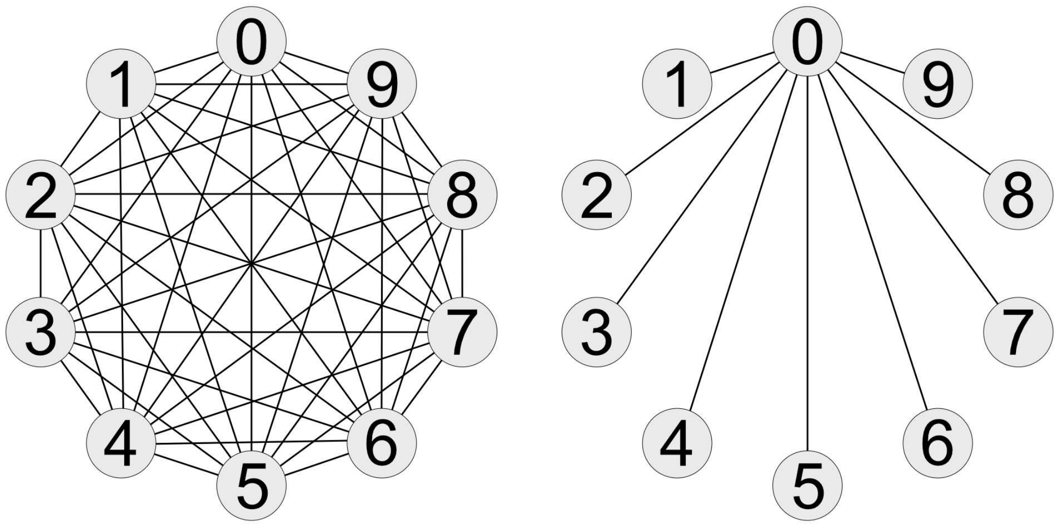


Figure 6.3: Agent selection for negative for generalized embeddings (left) and fixed base policy embeddings (right). Numbered nodes represent the base agent of policy i, with their connections representing negative episodes of agent $j \neq i$.

6.3.1 Training Embeddings

Embeddings trained using Alg. 9 are separated into two cases. The first case trying to encapsulate the behavior of many agents into one *generalized embedding* where episodes e_+ and e_* are sampled from agent i and e_- from an agent $j \neq i$ of a different policy. The agents i from which e_+ and e_* are sampled iterate until all policies have provided one pair of positive and reference episodes which are optimized against negative episodes e_- from agents $j \neq i$ of the other policies as illustrated in Fig. 6.3 (left).

The other method of training agent embeddings is to keep the agent i from which e_+ and e_* are sampled static, and optimizing the imitation and embedding networks through multiple iterations against negative episodes e_- from agents $j \neq i$. This attempts to specifically encapsulate the behavior of agent i into a more direct *imitation embedding*, compared to the other approach which tries to encapsulate the behavior of many agents at once, which is depicted in Fig. 6.3 (right).

The resulting embedding networks are then implemented into the environment via wrapper functions, which take the observation-action pairs of the relevant agents, encodes them using the embedding network and appends the resulting embeddings to the observation of the recipient agent. These wrapper functions are outlined in Alg. 11 for DTDE and Alg. 13 for CTDE.

Environment	Base Agents	Target Agents	Input Dim	Embedding Dim	Type
Speaker	listener_0	speaker_0	16	5	Coop
Push	agent_0	adversary_0	24	10	Comp
$\text{Crypto}_{\text{coop}}$	alice_0	bob_0	12	5	Coop
$\text{Crypto}_{\text{comp}}$	alice_0; bob_0	eve_0	24	10	Comp
$\text{Crypto}_{\text{mixed}}$	alice_0; bob_0	eve_0; bob_0	[24, 12]	[10, 5]	Mixed
Spread	$\text{agent}_{0,1,2}$	$\text{agent}_{0,1,2}$	69	35	Coop
Tag	$\text{adversary}_{0,1,2}$	agent_0	63	30	Comp
$\text{World}_{\text{coop}}$	$\text{adversary}_{0,1,2}$	leadaversary_0	117	60	Coop
$\text{World}_{\text{comp}}$	leadaversary_0	$\text{agent}_{0,1}$	43	20	Comp
$\text{World}_{\text{mixed}}$	$\text{adv}_{0,1,2}$; leadadv_0	leadadv_0; $\text{agent}_{0,1}$	[117, 43]	[60, 20]	Mixed

Table 6.1: Table listing parameters of the base and target agents used to train embedding and imitation networks.

6.3.2 Agent Selection

Choosing which agents are to be embedded varies depending on the environment. The behavior of the agent with the highest dimensional observation-action space is chosen to be encoded, because it has the most information to pass along. In cases where there are multiple agents having the same observation-action space, considerations have to be made which one to generate embeddings from, or to embed multiple agents at once. For the environments used in this thesis, agents that are equal in their observation-action space are combined into one embedding, with all observations and actions of all agents concatenated into one input vector. The choice to combine the observations and actions into one big input instead of averaging or encoding them is made to avoid the possibility of diluting the observation-action pair of each agent, resulting in inaccurate input data for the embedding network.

A detailed listing of base agents, which are encoded, target agents, from which the imitation policy is trained, input dimensions, embedding dimensions and type of environment is listed in Table 6.1:

6.4 Evaluation

To evaluate how the usage of agent representations impacts the learning process, the overall performance, as well as, single agent performance are compared to average results of 20 baseline policies. Further comparisons are made with random embeddings and all-zero embeddings, to determine if appending random or empty information to observations yields similar or worse performance compared to utilizing embeddings. For generalized embeddings, a new set of 20 policies applying the same embedding networks are trained and their average performance is compared to the baseline results.

For imitation embeddings using a fixed agent as its base, 15 of the 20 baseline policies are randomly selected to train representations from, while the other 5 are held out to compete against. Using the imitation embedding networks, 15 new policies are trained. During training of these 15 new agents, the imitation policy networks, which are jointly trained with their respective embedding networks, are used as a starting point for the actor networks of agents receiving embeddings. The 5 best performing agents utilizing imitation embeddings, are then evaluated against the other 5 agents of the hold out set of baseline policies to verify if they can still produce competitive results with agents they have never seen before.

7 Implementation and Setup

To train the policies according to the algorithms used in the experiments, *AgileRL v0.1.21* [UAP] is used with two different training loops for DTDE and CTDE. OpenAI's *Multi Particle Environments* (MPE) [LWT+17] are implemented using the *PettingZoo v1.24.1* [TBG+21] library. To implement the embedding and imitation policy networks, custom neural networks are created using *PyTorch v2.0.1* [PGM+19]. The following sections highlight the experimental setup for centralized and decentralized training, as well as implementation of the respective training loops, wrapper functions for the environments.

7.1 Algorithms

7.1.1 Decentralized Training - Decentralized Execution

Decentralized Training Loop

The two algorithms used for are *Deep Deterministic Gradient Policy (DDPG)* [LHP+16] and *Twin Delayed Deep Deterministic Gradient (TD3)* [FvHM18]. For all environments a fully connected MLP neural network is used with two hidden layers of size [64, 64]. The Adam optimizer is used where learning rates for the actors of each agent are set to 0.001, while the optimizer learning rate for their respective critics is set to 0.01, with a learning interval of every 5 episodes. The experience replay buffers are set to $1 \cdot 10^6$ with a mini-batch size of 64 samples per learning interval. Policies are trained for 10000 episodes, with 25 environment time steps per episode. The training loop for the DTDE approach is shown in Alg. 10. After initializing the environment, agents and replay buffers, the environment can be wrapped using Alg. 11 and agents can be replaced with imitation policies that are trained together with their corresponding embedding networks. Following that, at the start of each episode, the environment gets reset and the initial states for all agents are observed. Then, for each time step in an episode, agents are iterated where each agent i observes its current state and reward, samples an action according to its policy, executes that action and observes the next state before storing it into its replay buffer $\mathcal{D}^{(i)}$. An episode ends if the time steps reach the pre-defined threshold or the environment reports back a termination flag. At the end of each episode, all observations, actions and rewards are stored on a per agent basis, with the fully trained agents being saved after iterating through all episodes. The same algorithm is used for evaluation, omitting the the initializing and storing transitions into the replay buffers $\mathcal{D}^{(i)}$ and learning steps, since those are not needed for roll out.

Algorithm 10 DTDE Training Loop with Environment, Embeddings, and Imitation Policy

```
 1: Initialize Hyperparameters
 2: Initialize AEC Environment
 3: Initialize DDPG/TD3 actor and critic networks $\pi_\theta^{(i)}$
 4: Initialize Replay Buffers $\mathcal{D}^{(i)}$
 5:
 6: if Embeddings are used then
 7:     Load embedding network
 8:     Wrap environment
 9: end if
10:
11: if Imitation policy is used then
12:     Replace target agent with imitation policy
13: end if
14:
15: for each episode do
16:     Reset environment
17:     Observe initial states $\mathbf{x}_0^{(i)}$
18:     for each step in episode do
19:         for each agent i do
20:             Observe state $\mathbf{x_t^{(i)}}$ and reward $r_t^{(i)}$
21:             Sample action $a_t^{(i)}$ according to $\pi_\theta^{(i)}$
22:             Execute action $a_t^{(i)}$
23:             Observe next state $\mathbf{x}_{t+1}^{(i)}$
24:             Store transition $\left(\mathbf{x}_t^{(i)}, a_t^{(i)}, r_t^{(i)}, \mathbf{x}_{t+1}^{(i)}\right)$ in $\mathcal{D}^{(i)}$
25:             Set $\mathbf{x}_t^{(i)} \leftarrow \mathbf{x}_{t+1}^{(i)}$
26:             if current episode mod learn frequency $= 0$ then
27:                 Sample random mini-batch of transitions in $\mathcal{D}^{(i)}$
28:                 Update $\pi_\theta^{(i)}$
29:             end if
30:         end for
31:     end for
32:     Store all transitions for each agent
33: end for
34:
35: return trained $\pi_\theta^{(i)}$ and interaction episodes
```

AEC Embedding Wrapper

Since the environments don't support the inclusion of embedding networks by default, wrapper functions are used to take the observation-action pair of the base agents at each time step, calculating the embeddings according to the given embedding network, and lastly appending the embeddings to the observation of the target agent. The WRAPPER class, as outlined in Alg. 11 augments the RESET, STEP, and LAST functions of the corresponding PettingZoo environment to store the observations and actions of the relevant agents at every time step, with the EMBED function concatenating the observation-action pair of the agent that is to be embedded, calculating the embedding and then appending the embedding to the target agent's observation.

Alg. 11 shows the general structure of the environment wrapper used for decentralized training. It augments the RESET, STEP, and LAST functions used in a regular PettingZoo environment by saving the latest observations and actions of the base agents at each time step. Once the target agent gets its turn to act, the EMBED function gets called to calculate the embeddings of the base agent according to the given embedding network. The embeddings are then concatenated to the observation of the target agent and returned into the training loop.

Algorithm 11 AEC Wrapper with Embedding Network and Agent Interaction

```
 1: class WRAPPER
 2:     Attributes:
 3:         Environment
 4:         Embedding Network
 5:         Base Agent
 6:         Target Agent
 7:
 8: function RESET(Observations, Actions)
 9:     Store initial Observations and Actions of all Agents
10:     Concatenate Observation-Action pair of Base Agent
11:     Embed Observation-Action of Base Agent
12:     Append Embedding to Observation of Target Agent
13:     return Observations
14: end function
15:
16: function EMBED(Observations, Actions)
17:     Concatenate current Observation-Action pair of Base Agent
18:     Generate Embedding
19:     Append Embedding to current Observation of Target Agent
20:     return Target Agent Observation
21: end function
22:
23: function STEP(Action, Agent ID)
24:     Take step in Environment according to Action
25:     if Agent ID is Base Agent then
26:         Store current Observation of Base Agent
27:     end if
28: end function
29:
30: function LAST(Agent ID)
31:     if Agent ID is Target Agent then
32:         Generate and append Embedding for Observation of Target Agent
33:     end if
34:     return Observation
35: end function
```

7.1.2 Centralized Training - Decentralized Execution

Centralized Training Loop

The algorithms used for centralized training - decentralized execution are the multi-agent variants of DDPG [LHP+16] and TD3 [FvHM18], *Multi-Agent Deep Deterministic Gradient Policy (MADDPG)* [LWT+17] and *Multi-Agent Twin Delayed Deep Deterministic Policy Gradient (MATD3)* [AGOS19] respectively. As with decentralized training - decentralized execution, the training parameters for all environments are fully connected MLP neural networks with a hidden layer size of [64, 64]. The Adam optimizer is used with learning rates of 0.001 for the actors and 0.01 for the central critics. For learning, the experience replay buffer is set to hold $1 \cdot 10^6$ samples with a batch of 64 samples on learning interval of 5 episodes. The training loop for CTDE is implemented as laid out in Alg. 12.

The training loop for centralized training is more compact, since the need to iterate through each agent and updating the environment multiple times at every time step is omitted. All actor networks are kept in the collection of policies with global critics that use all observations and actions of all actors as input, therefore optimizing every actor using the information of all actors. This lends itself to parallel execution, which is why a parallel environment is chosen for these particular algorithms. This leads to agents acting simultaneously instead of the environment updating multiple times per time step for each agent's action, like it is the case for decentralized training using the AEC approach. Alg. 12 functions nearly the same as Alg. 10 for decentralized training, with the key difference of all agents observing their environment state and executing an action based on their individual observations all at once. Another difference is that one combined replay buffer $\mathcal{D}$ is used to store the interaction data of all agents, in contrast to having a dedicated replay buffer $\mathcal{D}^{()}$ for each agent like in the decentralized case. At the end of each episode, all observations, actions and rewards for each agent are stored in a dictionary, with fully trained policies being saved after iterating through all episodes. Evaluation is done the same way, while leaving out initializing the replay buffer $\mathcal{D}$ and storing interaction data into it, as well as omitting the learning step

Parallel Embedding Wrapper

The WRAPPER class implemented in Alg. 13 used to wrap environments for centralized learning augments the RESET and STEP functions in PettingZoo environments using parallel execution. In both of these functions, the observations and actions of all relevant agents are stored at each time step, with the EMBED function concatenating the observation-action pairs of the agents that are to be embedded, calculating their embedding, using the provided embedding network, and appending them to the observation of the target agent.

Algorithm 12 CTDE Training Loop with Embedding and Imitation Policy

```
 1: Initialize Hyperparameters
 2: Initialize Parallel Environment
 3: Initialize MADDPG/MATD3 actor and critic networks $\pi_\theta^{(i)}$
 4: Initialize centralized Replay Buffer $\mathcal{D}$
 5:
 6: if Embeddings are used then
 7:     Load embedding network
 8:     Wrap environment
 9: end if
10:
11: if Imitation policy is used then
12:     Replace target agent with imitation policy
13: end if
14:
15: for each episode do
16:     Reset Environment
17:     Observe initial states $\mathbf{x}_0^{(i)}$
18:     for each step in episode do
19:         Sample actions $a_t^{(i)}$ according to $\pi_\theta^{(i)}$
20:         Execute action $a_t^{(i)}$
21:         Observe next state $\mathbf{x}_{t+1}^{(i)}$
22:         Store transition $\left(\mathbf{x}_t^{(i)}, a_t^{(i)}, r_t^{(i)}, \mathbf{x}_{t+1}^{(i)}\right)$ in $\mathcal{D}$
23:         Set $\mathbf{x}_t^{(i)} \leftarrow \mathbf{x}_{t+1}^{(i)}$
24:         if $\textit{current episode} \bmod \textit{learn frequency} = 0$ then
25:             Sample random mini-batch of transitions in $\mathcal{D}$
26:             Update $\pi_\theta^{(i)}$
27:         end if
28:     end for
29:     Store all transitions for each agent
30: end for
31:
32: return Trained $\pi_\theta^{(i)}$ and interaction episodes
```

Algorithm 13 Parallel Wrapper with Embedding Network and Agent Interaction

```
class WRAPPER
    Attributes:
        Environment
        Embedding Network
        Base Agent
        Target Agent

function RESET(Observations, Actions)
    Store initial Observations and Actions of all Agents
    Concatenate Observation-Action pair of Base Agent
    Embed Observation-Action of Base Agent
    Append Embedding to Observation of Target Agent
    return Observations
end function

function EMBED(Observations, Actions)
    Concatenate current Observation-Action pair of Base Agent
    Generate Embedding
    Append Embedding to current Observation of Target Agent
    return Target Agent Observation
end function

function STEP(Actions)
    Take step in Environment according to Actions
    Record all current Observations of all Agents
    Embed Base Agent Observation-Action pairs
    Replace Target Agent Observation with Observation+Embedding
    return Observations
end function
```

7.2 Embeddings

Per environment and algorithm, 20 agents are trained to be used as baseline agents. From these agents, embedding networks and imitation policies are trained using Alg. 9. Episodes involving agent i are chosen randomly from a set of 1000 roll out episodes of the corresponding, fully trained policy while choosing different episodes for the positive embedding e_+ and reference embedding e_*. The embeddings are generated by forwarding all observation-action pairs of an episode through the embedding network f_θ, and then averaging each entry over all time steps. To calculate the *imitation loss*, line 5 of Alg. 9 is used, where the observations at each time step get concatenated with the embedding $r_e = f_\theta(e_*)$. The *identification loss* is calculated via the *triplet loss* outlined in Equation 4.6 using all three embeddings $p_e = f_\theta(e_+)$, $r_e = f_\theta(e_*)$, $n_e = f_\theta(e_-)$. This leads to any given positive episodes e_+, reference episodes e_* and their according *identification loss* being optimized against n negative episodes e_- of agents $j \neq i$ from other policies. Both the imitation policy and embedding networks get updated using the combined loss, with a hyperparameter λ being used to tune the impact of the *identification loss*.

7.3 Experimental Setup

For each algorithm and environment, 20 policies are trained for 10000 episodes to use as baseline for further testing and generating imitation policies and embedding networks from. To gather all the data needed for the embedding algorithm, every *observation*, *action*, and *reward* of every agent at every time step is recorded during evaluation runs consisting of 1000 episodes. To train the embedding and imitation policy networks, two approaches are used. The first is to train a *generalized embedding* involving one positive episode e_+ and reference episode e_* of agent i against one negative episode e_- of all other 19 agents $j \neq i$. Every agent is optimizing against an episode of all other agents. For this approach, only the embedding network is used and the imitation policy network is being disregarded to investigate if a shallow optimization across many agents yields an uplift in results. The second approach is to fix agent i and sample 20 sets of positive and reference episodes and optimize them against a negative episode of 14 other agents $j \neq i$. This leads to 15 embedding and imitation policies optimized towards a specific agent.

To investigate if the shallow embedding network can enhance performance, 20 fresh policies are trained with the generalized embedding network being used in the environment wrapper. Apart from that, the training and evaluation setup is the same as for the baseline policies. For the agent specific imitation policies and embeddings, the environment gets wrapped with the relevant embedding network, as well as the target agents being replaced with the imitation policies. Training for these 15 imitation policies uses the same setup as the baseline policies. For evaluation, the 5 best performing agents using imitation policies are selected and loaded into a set of 5 hold out policies. The idea behind this is that an agent trained using imitation policies and specific embeddings should generalize better when interacting with other agents it has never seen before, since the same base agent has been optimized against many other agents of different policies. The following list summarizes the parameters for the experimental and evaluation setup:

- **Training Agents**
  - Hidden Layer Size: [64, 64]
  - Network Architecture: MLP
  - Activation Function: Sigmoid
  - Experience Replay Buffer: $1 \cdot 10^6$
  - Batch Size: 64
  - Actor/Critic Learning Rate: 0.001/0.01
  - Learn Step: 5 episodes
  - Training/Evaluation Episodes: 10000/1000
  - Steps per Episode: 25
- **Training Embeddings**
  - Hidden Layer Size: [100, 100]
  - Network Architecture: MLP
  - Activation Function: Sigmoid
  - λ/Adam Learning Rate: 0.01/0.001
  - Agent i Episodes: 20
  - Agent j Episodes:
    - Generalized Embeddings: 19
    - Imitation Embeddings: 14

8 Experiments

To evaluate training results, the episodic rewards for all sets of policies: 20 baseline, 20 with generalized embeddings, 15 with imitation embeddings, 5 with random embeddings and 5 with all zero embeddings are averaged and plotted using exponential moving average smoothing, with a smoothing factor of 0.95. Training curves are plotted for each environment and algorithm, one displaying training performance with generalized embeddings compared to baseline and all random/zero embeddings, the other comparing imitation embeddings to baseline and all random/zero embeddings. To evaluate roll out policies, the results of 1000 evaluation episodes are averaged on a per agent and overall basis, with the 5 best performing policies using imitation embeddings being tested against previously unseen agents.

8.1 Simple Speaker Listener

The *Simple Speaker Listener* environment (Fig. 6.1a) includes two agents, a *speaker* and a *listener*, as well as three colored landmarks. The *speaker* agent knows the correct landmark but cannot move. The *listener* does not know which landmark is the correct one, but is being able to move. The objective of this environment is for the *speaker* to communicate the location of the correct landmark to the *listener*, strengthening the communication and cooperation between the two. Both agents get rewarded globally based on the distance of the *listener* to the target landmark. [LWT+17] [TBG+21]

The agents in this environment have the following observation-action spaces: $speaker \leftarrow (3, 3)$, $listener \leftarrow (11, 5)$ with the *speaker* communicating the position of the correct landmark and the *listener* trying to move towards it. For the embeddings, the listener agent is used as the basis for embeddings, since it has the larger observation-action space with $dim = 16$. The embedding dimension is chosen to be $dim = 5$ in accordance with the already conducted experiments in the original paper by Grover et al. [GAG+18].

Using embeddings generated as described in Alg. 9 and incorporating them into the training and evaluation process with Alg. 11 for DTDE and Alg. 13 for CTDE, yields no particular improvement in the case of decentralized training, while showing slight improvements in the early learning process when using the MADDPG algorithm as seen in Fig. 8.3 for generalized embeddings. Using imitation embeddings alongside imitation policies, results in slightly worse performance during training for both CTDE algorithms (Fig. [8.3 8.4]), while DTDE methods show similar learning curves compared to their baseline results with no embeddings (Fig. [8.1 8.2]).

When evaluating roll out episodes, the 20 policies trained using generalized embeddings display performance which is largely in line with using either random or all zero embeddings (Tab.[8.3 8.4]), suggesting that these embeddings don't provide any useful information for this task. This trend can be observed for all further results, as using one sample each with 20 agents as basis for one embedding seems to be too shallow to capture meaningful information. When evaluating roll out episodes using imitation embeddings against agents from the set of hold out policies, the results dip dramatically, which suggests that the imitation policy and embedding based agent did not manage to generalize the task but rather only works with its original training partner (Tab. [8.1 8.2]). This behavior is recurring with other communication tasks in further experiments.

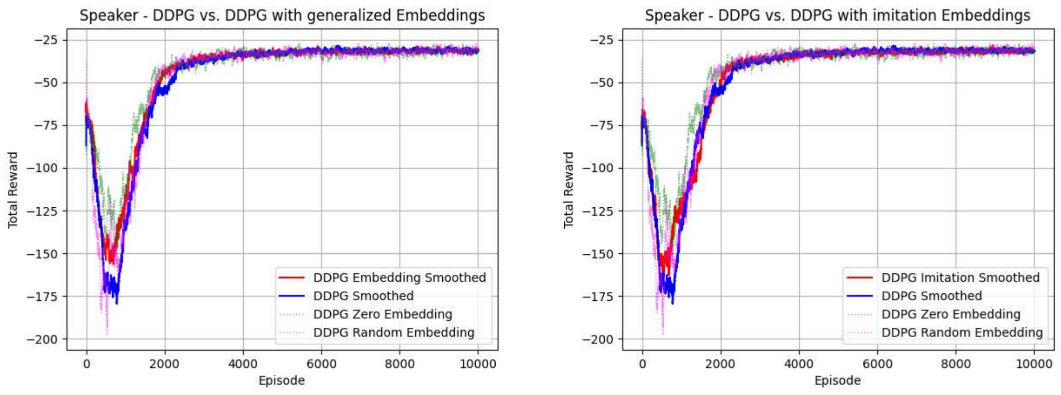


Figure 8.1: Training curves of the simple speaker listener environment with DDPG using generalized embeddings (left) and imitation embeddings (right).

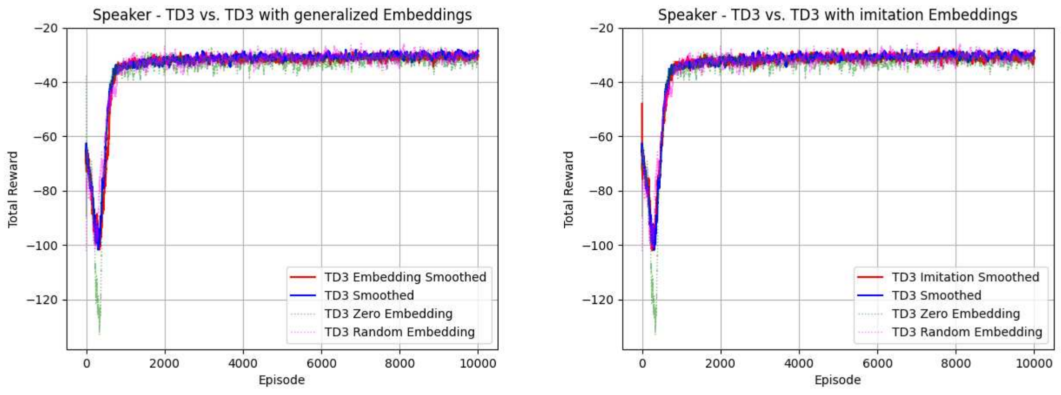


Figure 8.2: Training curves of the simple speaker listener environment with TD3 using generalized embeddings (left) and imitation embeddings (right).

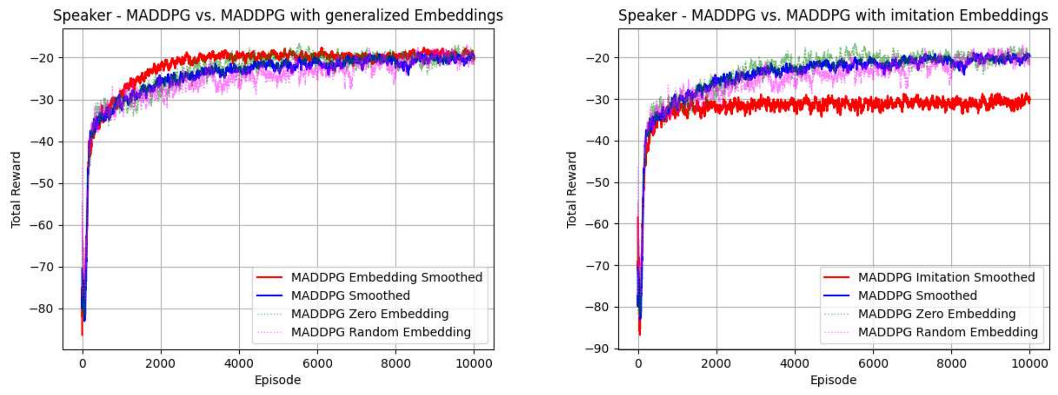


Figure 8.3: Training curves of the simple speaker listener environment with MADDPG using generalized embeddings (left) and imitation embeddings (right).

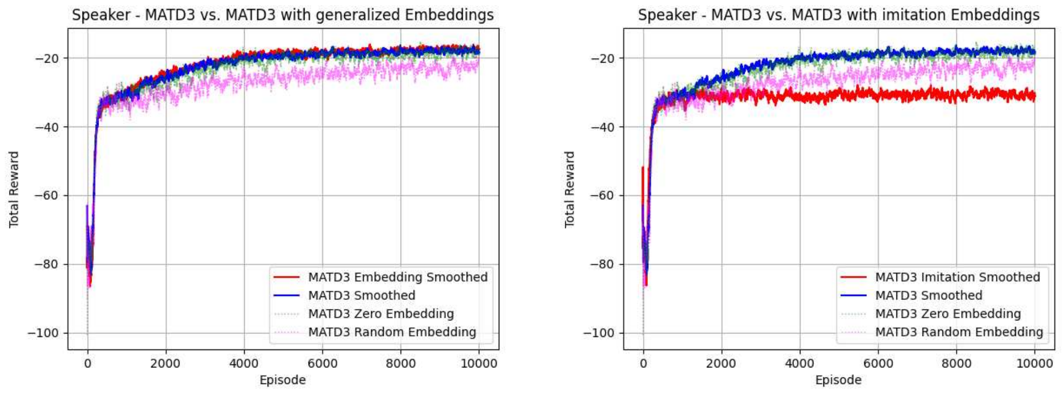


Figure 8.4: Training curves of the simple speaker listener environment with MATD3 using generalized embeddings (left) and imitation embeddings (right).

Embedding Type	speaker_0	listener_0	Total Reward
DDPG	**-15.27**	**-15.27**	**-30.54**
$\text{DDPG}_{\text{general emb}}$	-51.26	-51.26	-102.52
$\text{DDPG}_{\text{imitation emb}}$	-294.0	-294.0	-588.0
$\text{DDPG}_{\text{zero emb}}$	-54.31	-54.31	-108.62
$\text{DDPG}_{\text{random emb}}$	-54.57	-54.57	-109.14

Table 8.1: Table listing the evaluation results for the simple speaker listener environment, trained on the DDPG algorithm.

Embedding Type	speaker_0	listener_0	Total Reward
TD3	**-14.72**	**-14.72**	**-29.44**
$\text{TD3}_{\text{general emb}}$	-51.85	-51.85	-103.7
$\text{TD3}_{\text{imitation emb}}$	-301.39	-301.39	-602.78
$\text{TD3}_{\text{zero emb}}$	-52.84	-52.84	-105.68
$\text{TD3}_{\text{random emb}}$	-49.17	-49.17	-98.34

Table 8.2: Table listing the evaluation results for the simple speaker listener environment, trained on the TD3 algorithm.

Embedding Type	speaker_0	listener_0	Total Reward
MADDPG	**-11.65**	**-11.65**	**-23.3**
$\text{MADDPG}_{\text{general emb}}$	-57.87	-57.87	-115.74
$\text{MADDPG}_{\text{imitation emb}}$	-342.24	-342.24	-684.48
$\text{MADDPG}_{\text{zero emb}}$	-60.93	-60.93	-121.86
$\text{MADDPG}_{\text{random emb}}$	-48.97	-48.97	-97.94

Table 8.3: Table listing the evaluation results for the simple speaker listener environment, trained on the MADDPG algorithm.

Embedding Type	speaker_0	listener_0	Total Reward
MATD3	**-8.37**	**-8.37**	**-16.74**
$\text{MATD3}_{\text{general emb}}$	-54.25	-54.25	-108.5
$\text{MATD3}_{\text{imitation emb}}$	-344.88	-344.88	-689.76
$\text{MATD3}_{\text{zero emb}}$	-59.96	-59.96	-119.92
$\text{MATD3}_{\text{random emb}}$	-48.66	-48.66	-97.32

Table 8.4: Table listing the evaluation results for the simple push environment, trained on the MATD3 algorithm.

8.2 Simple Push

The *Simple Push* environment (Fig. 6.1b) involves a *good agent*, an *adversary* and a landmark. The *good agent* tries to come as close to the landmark as possible, while the *adversary* tries to block the *good agent* and keep it as far away as possible. The *good agent* is rewarded based on its distance to the landmark, while the *adversary* gets rewarded by staying close to the landmark and also keeping the *good agent* away. In this competitive scenario, the *adversary* has to learn a strategy to keep the *good agent* away, while keeping close to the landmark as well. [LWT+17] [TBG+21]

The observation-action space dimensions for the *Simple Push* environment are *good agent* $\leftarrow (19, 5)$ and $adversary \leftarrow (8, 5)$. With the *good agent* having a total dimensionality of $dim = 24$, it is used as base agent with an embedding dimension of $dim = 10$. Generalized embeddings show no real improvements during training for DTDE (Fig. [8.5 8.6]), but they display better results for overall performance during evaluation of roll out episodes (Tab. [8.5 8.6]). The opposite is the case for CTDE, where the training curves show better performance (Fig. [8.7 8.8]), while the overall average reward after 1000 evaluation episodes performs worse (Tab. [8.7 8.8]). In both cases, the *good agent* loses a lot of performance, while the *adversary* gains significant performance for DTDE and slightly better results for CTDE.

When training using imitation policies and imitation embeddings, improvements are seen in the training curves for both DTDE algorithms. TD3 and DDPG start out with decent performance and maintain it through the whole training process, while the baseline agents lose performance over time (Fig. [8.5 8.6]). This is also reflected during evaluation, where generalized embeddings improve compared to the baseline results, and imitation embeddings display the best results with much stronger performance for the *adversary* (Tab [8.5 8.6]). The opposite is the case when looking at CTDE algorithms with imitation embeddings. While still performing better than generalized embeddings, during training the training curves behave similar to DTDE, as they start out at a certain reward range and don't improve over time. (Fig. [8.7 8.8]). Looking at the per agent results for CTDE (Tab. [8.7 8.8]), the performance of both agents is much more extreme with the *adversary* showing very good results and the *good agent* struggling a lot. This leads to overall worse global performance for the environment compared to using no embeddings at all, since the two agents are a lot more balanced in performance in the baseline results.

The use of embeddings for this environment leads to an overall uplift in performance for DTDE methods, with the per agent performance being advantageous to the *adversary* which can make use of them against the *good agent.* While overall performance in CTDE methods is worse compared to the baseline results when using embeddings, the single agent performance also heavily favors the *adversary* which makes good use of them.

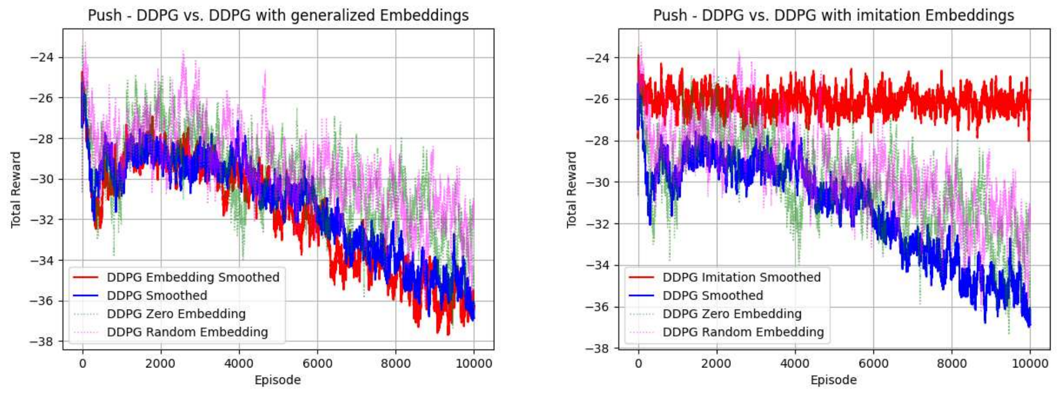


Figure 8.5: Training curves of the simple push environment with DDPG using generalized embeddings (left) and imitation embeddings (right).

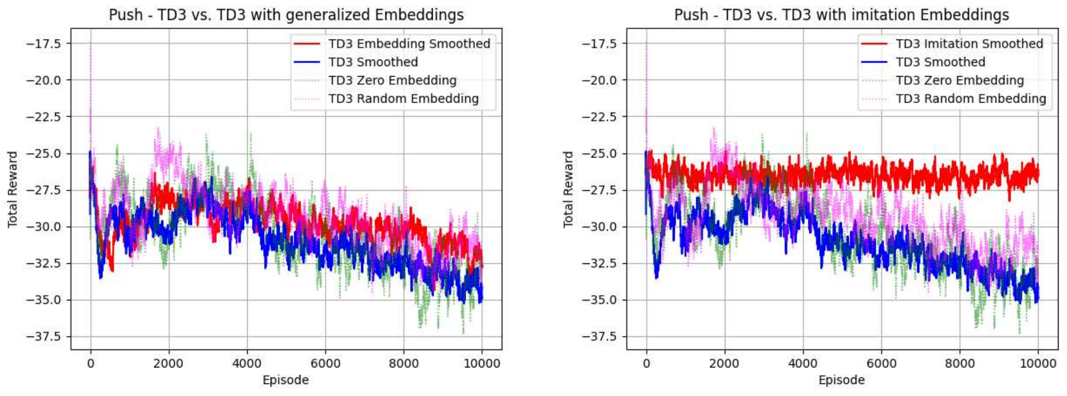


Figure 8.6: Training curves of the simple push environment with TD3 using generalized embeddings (left) and imitation embeddings (right).

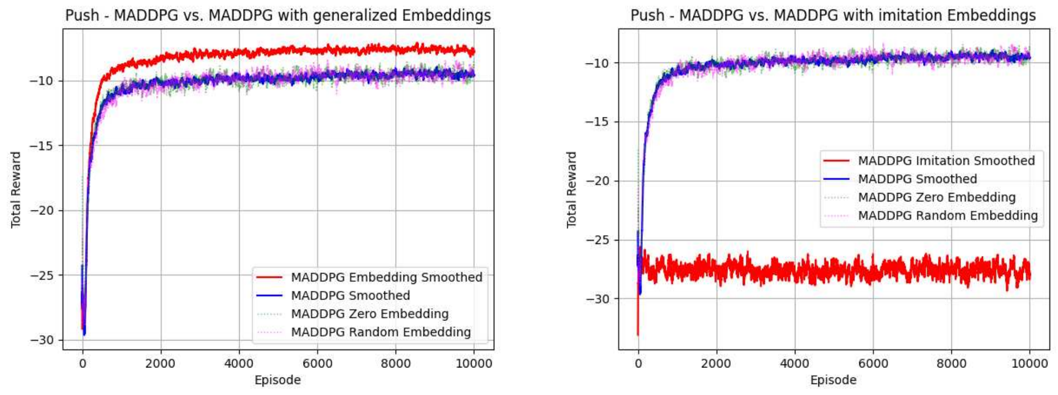


Figure 8.7: Training curves of the simple push environment with MADDPG using generalized embeddings (left) and imitation embeddings (right).

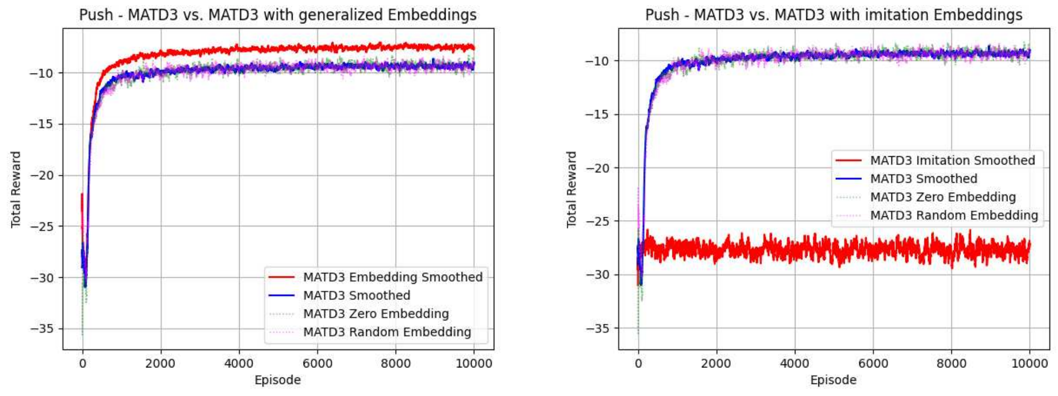


Figure 8.8: Training curves of the simple push environment with MATD3 using generalized embeddings (left) and imitation embeddings (right).

Embedding Type	adversary_0	agent_0	Total Reward
DDPG	-30.79	**-7.56**	-38.35
$\text{DDPG}_{\text{general emb}}$	2.48	-32.32	-29.84
$\text{DDPG}_{\text{imitation emb}}$	**52.95**	-78.45	**-25.5**
$\text{DDPG}_{\text{zero emb}}$	0.46	-30.53	-30.07
$\text{DDPG}_{\text{random emb}}$	3.44	-33.84	-30.4

Table 8.5: Table listing the evaluation results for the simple push environment, trained on the DDPG algorithm.

Embedding Type	adversary_0	agent_0	Total Reward
TD3	-26.58	**-8.76**	-35.34
$\text{TD3}_{\text{general emb}}$	-0.71	-30.76	-31.47
$\text{TD3}_{\text{imitation emb}}$	**50.47**	-75.92	**-25.45**
$\text{TD3}_{\text{zero emb}}$	-0.25	-30.85	-31.1
$\text{TD3}_{\text{random emb}}$	1.21	-30.67	-29.46

Table 8.6: Table listing the evaluation results for the simple push environment, trained on the TD3 algorithm.

Embedding Type	adversary_0	agent_0	Total Reward
MADDPG	-2.09	**-6.77**	**-8.86**
$\text{MADDPG}_{\text{general emb}}$	1.03	-32.5	-31.47
$\text{MADDPG}_{\text{imitation emb}}$	**58.81**	-85.46	-26.65
$\text{MADDPG}_{\text{zero emb}}$	-0.83	-33.03	-33.86
$\text{MADDPG}_{\text{random emb}}$	0.1	-32.14	-32.04

Table 8.7: Table listing the evaluation results for the simple push environment, trained on the MADDPG algorithm.

Embedding Type	adversary_0	agent_0	Total Reward
MATD3	-2.07	**-6.64**	**-8.71**
$\text{MATD3}_{\text{general emb}}$	-0.5	-31.44	-31.94
$\text{MATD3}_{\text{imitation emb}}$	**59.1**	-85.55	-26.45
$\text{MATD3}_{\text{zero emb}}$	-1.41	-33.73	-35.14
$\text{MATD3}_{\text{random emb}}$	3.91	-34.99	-31.08

Table 8.8: Table listing the evaluation results for the simple push environment, trained on the MATD3 algorithm.

8.3 Simple Crypto

In *Simple Crypto* (Fig. 6.1c) two agents try to communicate an encrypted message while an adversarial agent tries to intercept the message. Agents *Alice* tries to send an encrypted private message to *Bob* via a public channel. If *Bob* manages to decode the message, both *Alice* and *Bob* get rewarded, but get the reward revoked if the adversarial agent *Eve* also manages to reconstruct the message during an episode. *Eve* gets punished if it can't reconstruct the message and is rewarded nothing if it can. This environment combines cooperative communication between *Alice* and *Bob* and competitive aspects with *Eve* trying to intercept the message. [LWT+17] [TBG+21] For this mixed cooperative-competitive environment, multiple experiments are conducted, using cooperative embeddings to help *Alice* and *Bob*, competitive embeddings to help of *Eve* versus *Alice* and *Bob*, and a mixed approach where both embeddings are utilized.

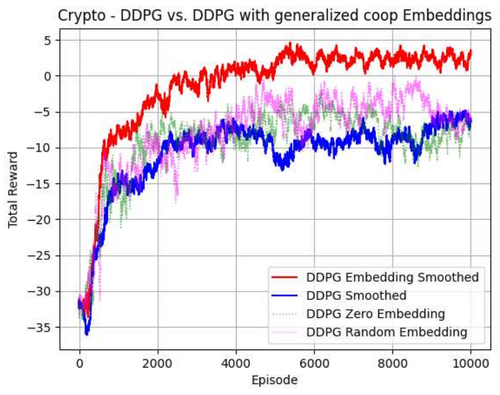
The dimensions for the observation-action spaces of these agents are $Eve \leftarrow (4, 4)$, $Alice \leftarrow (8, 4)$ and $Bob \leftarrow (8, 4)$. Thus for embeddings emphasizing cooperation, *Alice* is used as a basis with dimension $dim = 12$ and embedding dimension $dim = 5$, since *Alice* is sending the message and the embeddings aim to convey *Alice's* intent to *Bob*. For embeddings utilized in a competitive setting, both *Alice* and *Bob* are jointly embedded with *Eve* receiving as much information about its adversaries as possible. Therefore a summed base dimension of $dim = 24$ is embedded into dimension $dim = 10$.

8.3.1 Cooperative Embeddings

For cooperative embeddings, only the embedding network involving *Alice* as the base agent and *Bob* as the target is used. When using generalized embeddings, an uplift in training performance can be observed across all algorithms, while using imitation embeddings with imitation policies leads reduced performance during training when compared to the baseline policies. (Fig. [8.9 8.10 8.11 8.12]).

Similar to the results observed in the *Simple Speaker Listener* environment, when implementing embeddings for a communication based task, performance of agents using embeddings is reduced during evaluation. While the loss of performance is not as dramatic, the results when using generalized embeddings and imitation embeddings are largely in line with using random/all zero embeddings (Tab. [8.9 8.10 8.11 8.12]). While *Eve* experiences especially harsh losses when using generalized embeddings, all agents are a lot more balanced in terms of their per agent rewards when implementing imitation embeddings.

Contrary to the uplift in training performance using generalized embeddings, they seem to not generalize the task of this environment very well, while the implementation of imitation embeddings seems to not work at all when an agent utilizing these embeddings gets paired up with another agent it has never interacted with before.



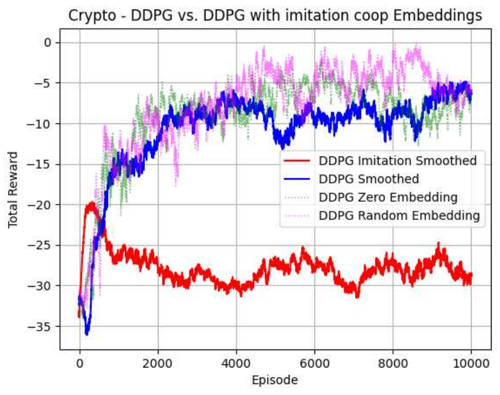
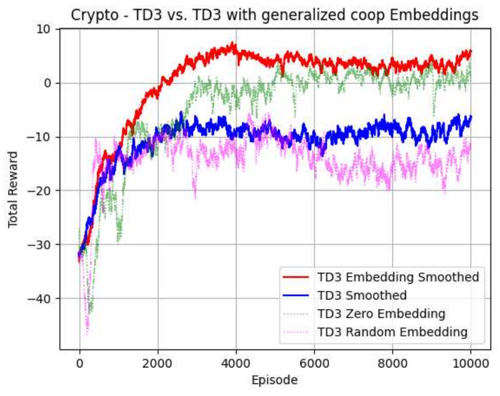


Figure 8.9: Training curves of the simple crypto environment with DDPG using cooperative generalized embeddings (left) and imitation embeddings (right).



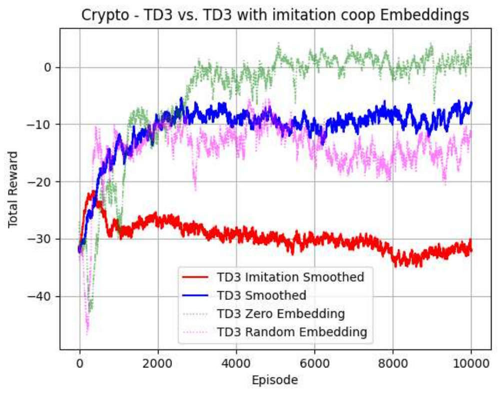


Figure 8.10: Training curves of the simple crypto environment with TD3 using cooperative generalized embeddings (left) and imitation embeddings (right).

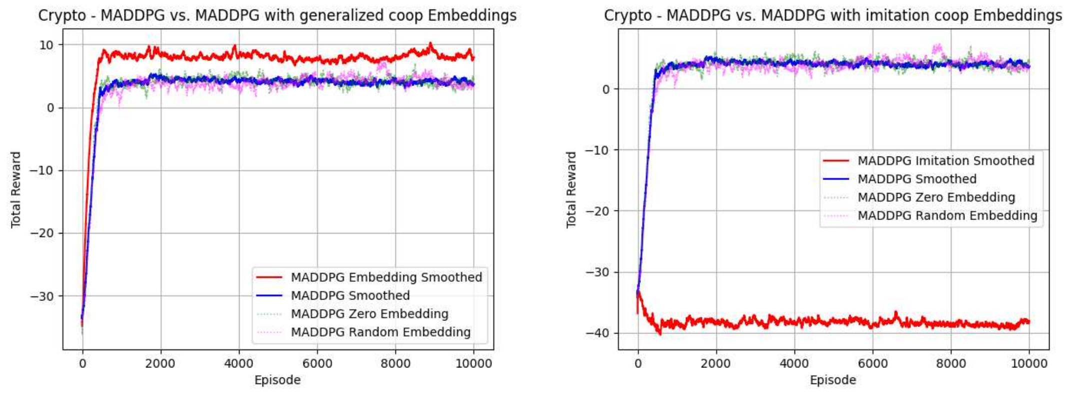


Figure 8.11: Training curves of the simple crypto environment with MADDPG using cooperative generalized embeddings (left) and imitation embeddings (right).

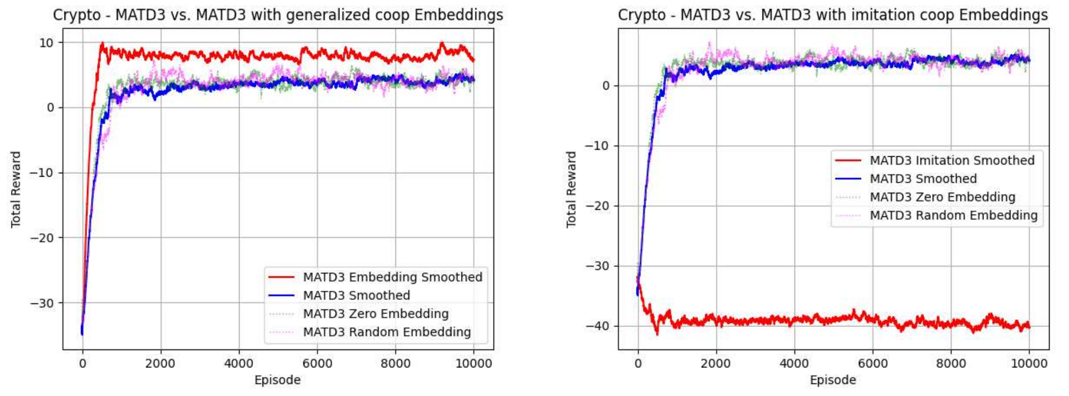


Figure 8.12: Training curves of the simple crypto environment with MATD3 using cooperative generalized embeddings (left) and imitation embeddings (right).

Embedding Type	eve_0	bob_0	alice_0	Total Reward
DDPG	**-13.05**	**5.75**	**5.75**	-1.55
$\text{DDPG}_{\text{general emb}}$	-30.4	-1.62	-1.62	-33.64
$\text{DDPG}_{\text{imitation emb}}$	-13.16	-10.5	-10.5	-34.16
$\text{DDPG}_{\text{zero emb}}$	-26.57	-2.46	-2.46	-31.49
$\text{DDPG}_{\text{random emb}}$	-27.03	-2.98	-2.98	-32.99

Table 8.9: Table listing the evaluation results for the simple crypto environment, trained on the DDPG algorithm using cooperative embeddings.

Embedding Type	eve_0	bob_0	alice_0	Total Reward
TD3	-14.75	**6.99**	**6.99**	**-0.77**
$\text{TD3}_{\text{general emb}}$	-28.77	-0.21	-0.21	-29.19
$\text{TD3}_{\text{imitation emb}}$	**-11.95**	-11.6	-11.6	-35.15
$\text{TD3}_{\text{zero emb}}$	-30.88	2.53	2.53	-25.82
$\text{TD3}_{\text{random emb}}$	-28.01	-1.73	-1.73	-31.47

Table 8.10: Table listing the evaluation results for the simple crypto environment, trained on the TD3 algorithm using cooperative embeddings.

Embedding Type	eve_0	bob_0	alice_0	Total Reward
MADDPG	**-13.38**	**11.72**	**11.72**	**10.06**
$\text{MADDPG}_{\text{general emb}}$	-27.9	-3.27	-3.27	-34.44
$\text{MADDPG}_{\text{imitation emb}}$	-13.9	-11.45	-11.45	-36.8
$\text{MADDPG}_{\text{zero emb}}$	-27.9	1.26	1.26	-25.38
$\text{MADDPG}_{\text{random emb}}$	-26.9	-2.32	-2.32	-31.54

Table 8.11: Table listing the evaluation results for the simple crypto environment using cooperative embeddings, trained on the MADDPG algorithm.

Embedding Type	eve_0	bob_0	alice_0	Total Reward
MATD3	-14.19	**12.57**	**12.57**	**10.95**
$\text{MATD3}_{\text{general emb}}$	-30.81	-1.07	-1.07	-32.95
$\text{MATD3}_{\text{imitation emb}}$	**-13.82**	-9.41	-9.41	-32.64
$\text{MATD3}_{\text{zero emb}}$	-34.82	1.49	1.49	-31.84
$\text{MATD3}_{\text{random emb}}$	-27.26	2.27	2.27	-22.72

Table 8.12: Table listing the evaluation results for the simple crypto environment, trained on the MATD3 algorithm using cooperative embeddings.

8.3.2 Competitive Embeddings

Using the combined observation-action pairs of *Alice* and *Bob* as basis for embeddings for *Eve* as the embedding recipient, shows a significant uplift in training performance when using imitation embeddings and imitation policies as basis for DTDE and CTDE (Fig. [8.13 8.14 8.15 8.16]). Implementing generalized embeddings, training performance falls short of showing improvements in all algorithms compared to baseline results, with significantly worse training performance for CTDE (Fig. [8.15 8.16]).

This behavior is explained when looking at the per agent performance, where the evaluation results for *Eve* sharply declines for all algorithms when using embeddings of any kind compared to baseline results (Tab [8.13 8.14 8.15 8.16]). Similar to using cooperative embeddings, giving access to embeddings for a communication based task, reduces the performance of agents receiving them. The impact of this is especially visible for evaluation results using imitation embeddings, where *Eve* is over-conditioned to the *Alice* and *Bob* agents it trained with, such that *Eve* is incapable of intercepting the majority of messages an unseen pair of *Alice* and *Bob* are sending between each other. Counter-intuitively, this leads to the best overall performance, because *Alice* and *Bob* receive positive rewards while *Eve* is consistently receiving negative rewards.

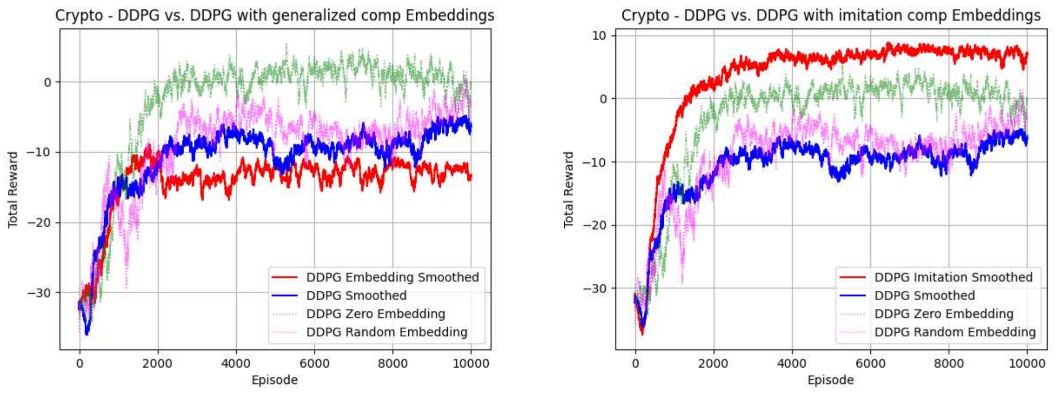


Figure 8.13: Training curves of the simple crypto environment with DDPG using competitive generalized embeddings (left) and imitation embeddings (right).

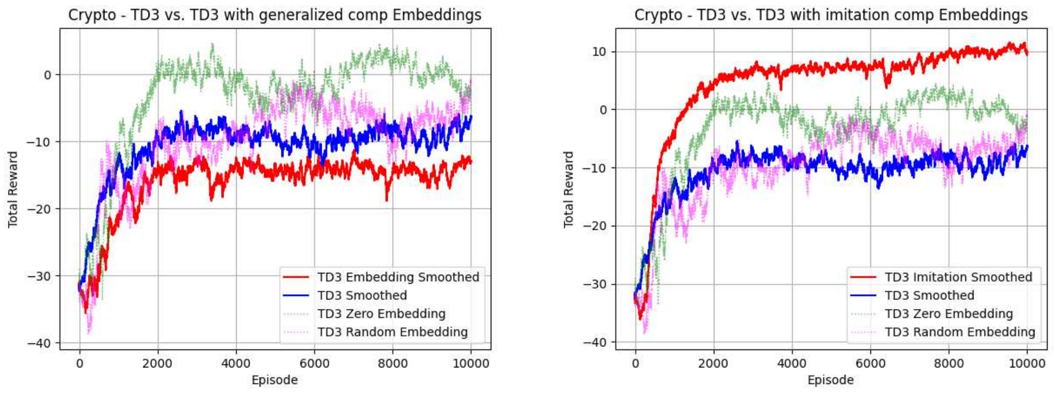


Figure 8.14: Training curves of the simple crypto environment with TD3 using competitive generalized embeddings (left) and imitation embeddings (right).

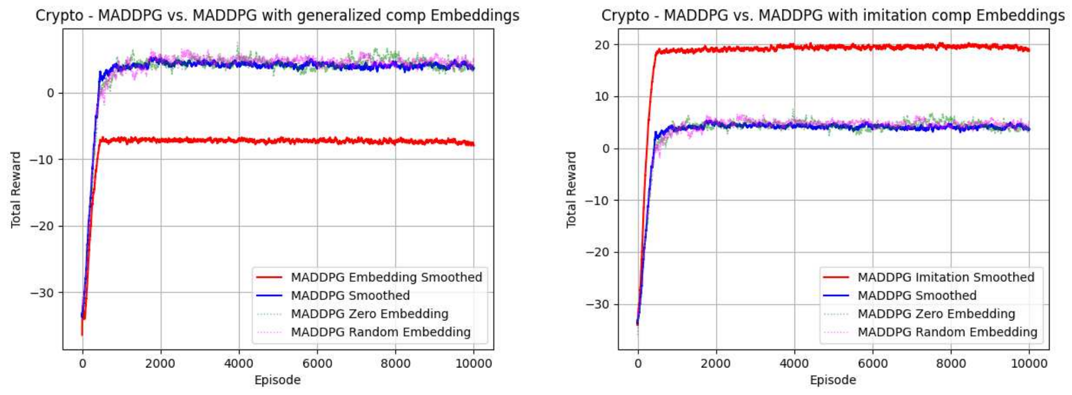


Figure 8.15: Training curves of the simple crypto environment with MADDPG using competitive generalized embeddings (left) and imitation embeddings (right).

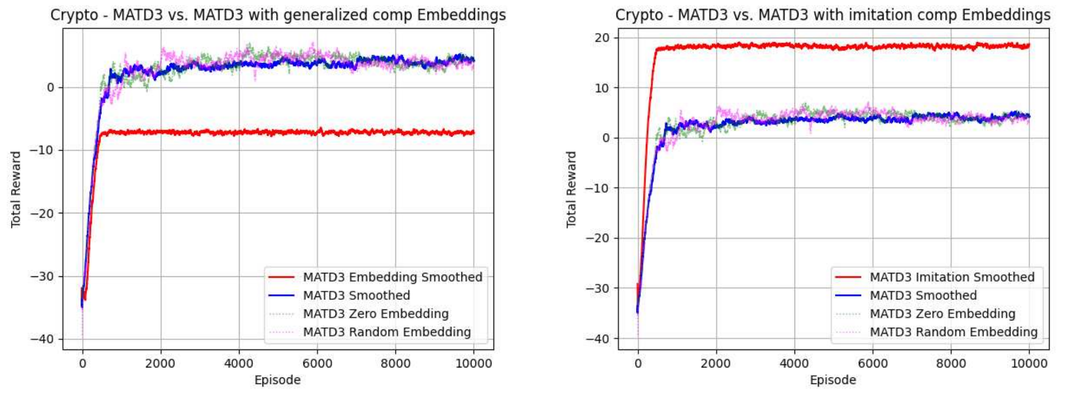


Figure 8.16: Training curves of the simple crypto environment with MATD3 using competitive generalized embeddings (left) and imitation embeddings (right).

Embedding Type	eve_0	bob_0	$alice_0$	Total Reward
DDPG	**-13.05**	5.75	5.75	-1.55
$\text{DDPG}_{\text{general emb}}$	-27.38	-1.97	-1.97	-31.32
$\text{DDPG}_{\text{imitation emb}}$	-28.01	**24.0**	**24.0**	**19.99**
$\text{DDPG}_{\text{zero emb}}$	-33.56	3.2	3.2	-27.16
$\text{DDPG}_{\text{random emb}}$	-31.58	-3.27	-3.27	-38.12

Table 8.13: Table listing the evaluation results for the simple crypto environment, trained on the DDPG algorithm using competitive embeddings.

Embedding Type	eve_0	bob_0	$alice_0$	Total Reward
TD3	**-14.75**	6.99	6.99	-0.77
$\text{TD3}_{\text{general emb}}$	-27.92	-2.5	-2.5	-32.92
$\text{TD3}_{\text{imitation emb}}$	-28.3	**18.43**	**18.43**	**8.56**
$\text{TD3}_{\text{zero emb}}$	-26.93	-1.34	-1.34	-29.61
$\text{TD3}_{\text{random emb}}$	-30.81	0.83	0.83	-29.15

Table 8.14: Table listing the evaluation results for the simple crypto environment, trained on the TD3 algorithm using competitive embeddings.

Embedding Type	eve_0	bob_0	$alice_0$	Total Reward
MADDPG	**-13.38**	11.72	11.72	10.06
$\text{MADDPG}_{\text{general emb}}$	-28.49	-3.21	-3.21	-34.91
$\text{MADDPG}_{\text{imitation emb}}$	-30.46	**28.85**	**28.85**	**27.24**
$\text{MADDPG}_{\text{zero emb}}$	-28.58	-0.28	-0.28	-29.14
$\text{MADDPG}_{\text{random emb}}$	-30.61	2.03	2.03	-26.55

Table 8.15: Table listing the evaluation results for the simple crypto environment, trained on the MADDPG algorithm using competitive embeddings.

Embedding Type	eve_0	bob_0	$alice_0$	Total Reward
MATD3	**-14.19**	12.57	12.57	10.95
$\text{MATD3}_{\text{general emb}}$	-27.91	0.1	0.1	-27.71
$\text{MATD3}_{\text{imitation emb}}$	-31.47	**29.82**	**29.82**	**28.17**
$\text{MATD3}_{\text{zero emb}}$	-24.49	-1.19	-1.19	-26.87
$\text{MATD3}_{\text{random emb}}$	-30.27	-1.16	-1.16	-32.59

Table 8.16: Table listing the evaluation results for the simple crypto environment, trained on the MATD3 algorithm using competitive embeddings.

8.3.3 Mixed Embeddings

When using both cooperative and competitive embeddings simultaneously, it can be observed that for DTDE, training generalized embeddings initially shows better performance before converging towards the training curves of the baseline policies. Using imitation embeddings with DTDE, displays a loss of performance during training induced by cooperative embeddings outweighing the uplift in overall performance gained from implementing competitive embeddings (Fig. [8.17 8.18]).

For CTDE algorithms, generalized embeddings show consistently worse performance during training with the losses from implementing the competitive embedding network outweighing the gains of the cooperative embeddings. The same observation is made when training with imitation embeddings, where the losses from using cooperative embeddings outweigh the gains from using competitive embeddings (Fig. [8.19 8.20]).

Introducing two embedding networks into a mixed environment yields consistently worse evaluation performance for *Eve*, while *Alice* and *Bob* see a less significant drop when using both cooperative and competitive generalized embedding networks. While the evaluation results using imitation embeddings for *Alice* and *Bob* are on par with the baseline results for DTDE algorithms (Tab. [8.17 8.18]), they still experience a drop in performance for CTDE algorithms, albeit not as much as with generalized embeddings, with *Eve* experiencing a consistent loss in performance (Tab. [8.19 8.20]).

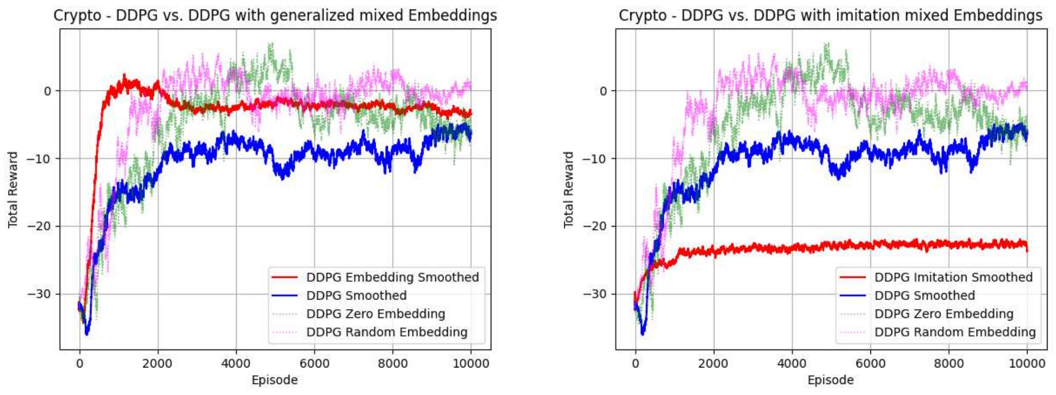


Figure 8.17: Training curves of the simple crypto environment with DDPG using mixed generalized embeddings (left) and imitation embeddings (right).

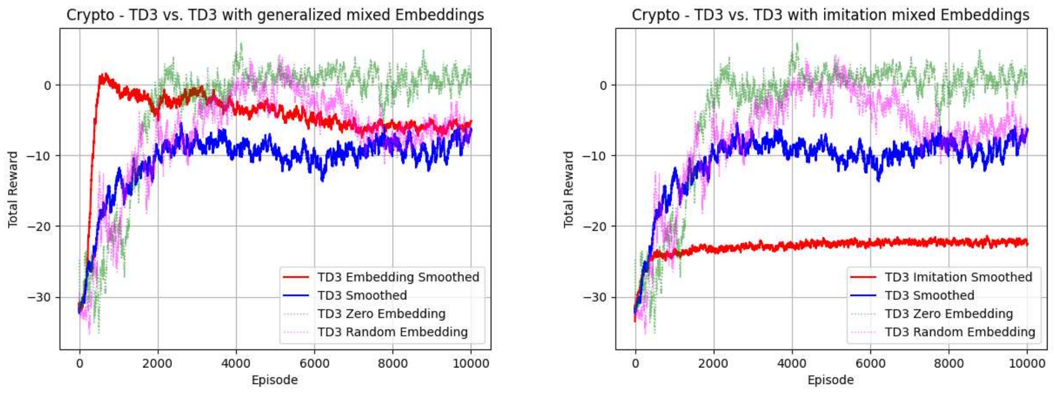


Figure 8.18: Training curves of the simple crypto environment with TD3 using mixed generalized embeddings (left) and imitation embeddings (right).

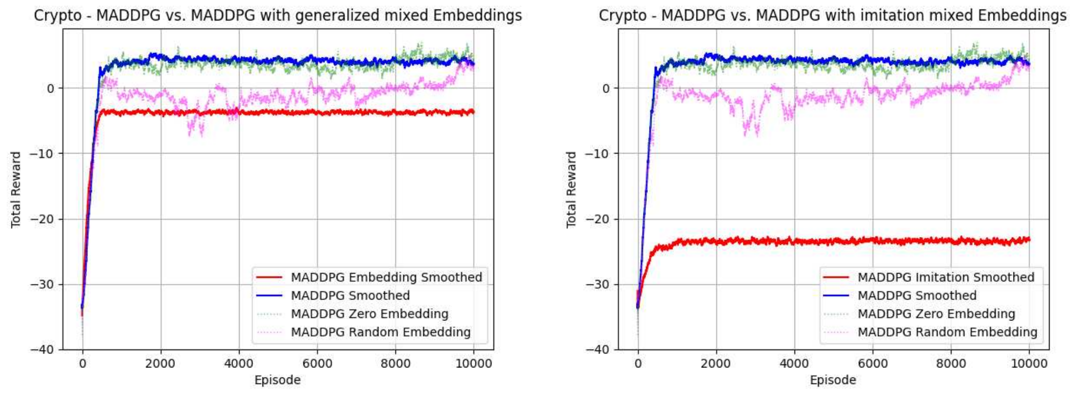


Figure 8.19: Training curves of the simple crypto environment with MADDPG using mixed generalized embeddings (left) and imitation embeddings (right).

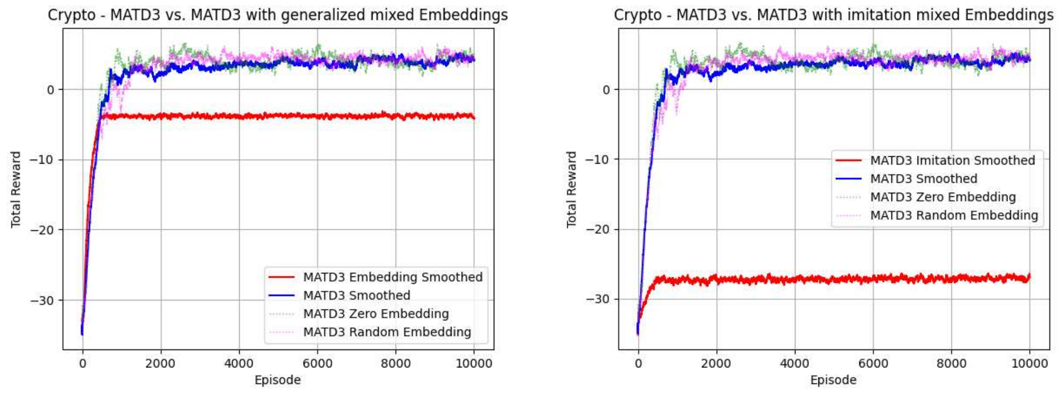


Figure 8.20: Training curves of the simple crypto environment with MATD3 using mixed generalized embeddings (left) and imitation embeddings (right).

Embedding Type	eve_0	bob_0	$alice_0$	Total Reward
DDPG	**-13.05**	5.75	5.75	**-1.55**
$DDPG_{\text{general emb}}$	-29.69	1.2	1.2	-27.29
$DDPG_{\text{imitation emb}}$	-28.24	5.85	5.85	-16.54
$DDPG_{\text{zero emb}}$	-30.79	**9.68**	**9.68**	-11.43
$DDPG_{\text{random emb}}$	-27.53	-1.35	-1.35	-30.23

Table 8.17: Table listing the evaluation results for the simple crypto environment, trained on the DDPG algorithm using mixed embeddings.

Embedding Type	eve_0	bob_0	$alice_0$	Total Reward
TD3	**-14.75**	**6.99**	**6.99**	**-0.77**
$TD3_{\text{general emb}}$	-26.08	-3.34	-3.34	-32.76
$TD3_{\text{imitation emb}}$	-29.19	6.85	6.85	-15.49
$TD3_{\text{zero emb}}$	-28.34	-0.62	-0.62	-29.58
$TD3_{\text{random emb}}$	-31.48	1.98	1.98	-27.52

Table 8.18: Table listing the evaluation results for the simple crypto environment, trained on the TD3 algorithm using mixed embeddings.

Embedding Type	eve_0	bob_0	$alice_0$	Total Reward
MADDPG	**-13.38**	**11.72**	**11.72**	**10.06**
$MADDPG_{\text{general emb}}$	-28.83	-2.19	-2.19	-33.21
$MADDPG_{\text{imitation emb}}$	-28.64	6.08	6.08	-16.48
$MADDPG_{\text{zero emb}}$	-34.93	6.51	6.51	-21.91
$MADDPG_{\text{random emb}}$	-32.52	3.83	3.83	-24.86

Table 8.19: Table listing the evaluation results for the simple crypto environment, trained on the MADDPG algorithm using mixed embeddings.

Embedding Type	eve_0	bob_0	$alice_0$	Total Reward
MATD3	**-14.19**	**12.57**	**12.57**	**10.95**
$MATD3_{\text{general emb}}$	-29.77	-1.63	-1.63	-33.03
$MATD3_{\text{imitation emb}}$	-25.87	2.3	2.3	-21.27
$MATD3_{\text{zero emb}}$	-25.26	0.01	0.01	-25.24
$MATD3_{\text{random emb}}$	-30.44	2.98	2.98	-24.48

Table 8.20: Table listing the evaluation results for the simple spread environment, trained on the MATD3 algorithm using mixed embeddings.

8.4 Simple Spread

Simple Spread (Fig. 6.1d) is a cooperative environment where three or more agents try to cover distinct landmarks, while avoiding collisions between each other. This environment is built on teamwork and coordination of movement between the agents. The agents are globally rewarded based on how far the closest agent of each landmark is located, while getting punished for every collision.[LWT+17] [TBG+21]

For the experiments, 3 agents with 3 landmarks are used, which all have observation-action spaces of dimension $agent_i \leftarrow (18, 5)$. Each agent observes the position of itself, the landmarks, other agents, and communication. To train the embeddings, all three agents are used as base agents with their observations and actions concatenated into a $dim = 69$ input vector and $dim = 35$ output embedding. The resulting imitation policies are then used as base for all three agents when training with imitation embeddings.

In the case of generalized embeddings, the learning curves for both DTDE and CTDE learning closely follow the baseline learning curves. However, training results when using imitation embeddings stays static during the whole duration of training (Fig. [8.21 8.22 8.23 8.24]). This suggests that training three imitation policy based agents with embeddings that include all agents does not lead to any tangible improvements for any agent during the learning process.

During evaluation, DTDE trained agents show slightly worse performance when using generalized embeddings, and nearly the same performance as the baseline policies when imitation embeddings are implemented (Tab. [8.21 8.22]). Evaluating CTDE based agents, shows this behavior in a more tangible way, where the baseline algorithms show better performance than using embeddings of any kind in this cooperative task (Tab. [8.23 8.24]). It is of note that across all algorithms, the evaluation results when using embeddings on all agents largely stays the same.

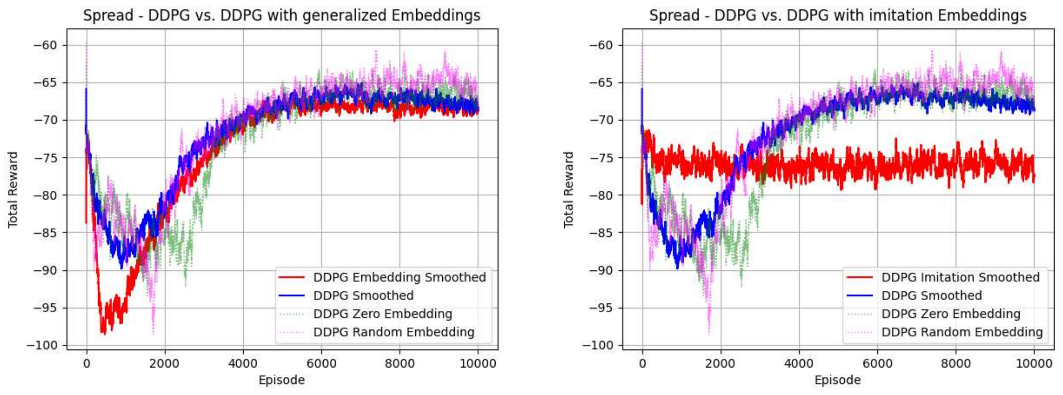


Figure 8.21: Training curves of the simple spread environment with DDPG using generalized embeddings (left) and imitation embeddings (right).

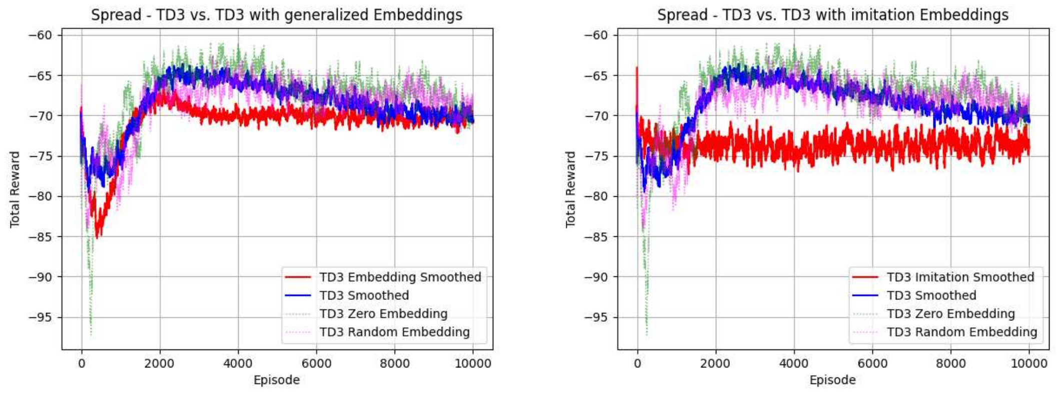
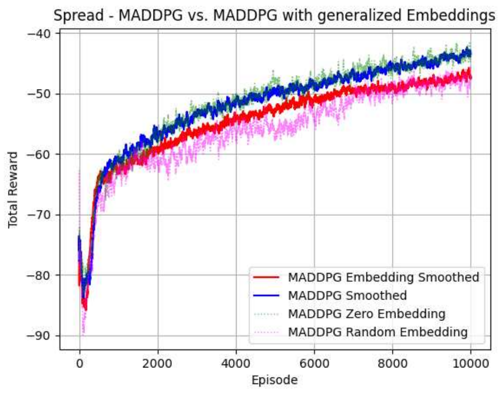


Figure 8.22: Training curves of the simple spread environment with TD3 using generalized embeddings (left) and imitation embeddings (right).



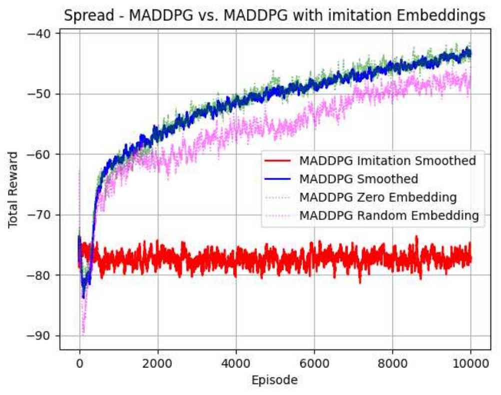
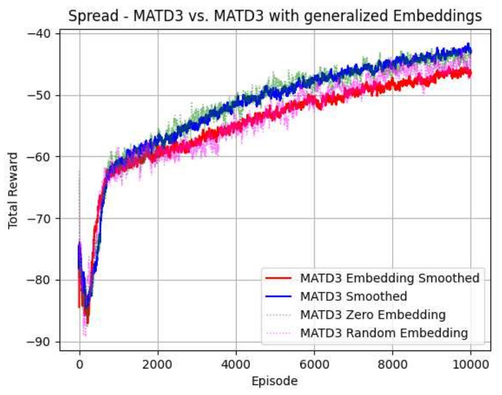


Figure 8.23: Training curves of the simple spread environment with MADDPG using generalized embeddings (left) and imitation embeddings (right).



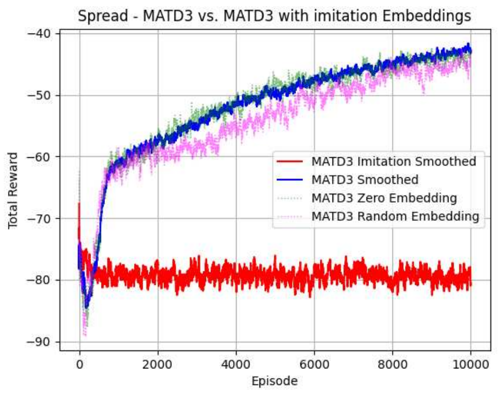


Figure 8.24: Training curves of the simple spread environment with MATD3 using generalized embeddings (left) and imitation embeddings (right).

Embedding Type	agent_0	agent_1	agent_2	Total Reward
DDPG	**-22.83**	**-22.86**	**-22.87**	**-68.56**
$\text{DDPG}_{\text{general emb}}$	-28.35	-28.35	-28.35	-85.05
$\text{DDPG}_{\text{imitation emb}}$	-24.69	-24.69	-24.69	-74.07
$\text{DDPG}_{\text{zero emb}}$	-27.53	-27.51	-27.51	-82.55
$\text{DDPG}_{\text{random emb}}$	-27.02	-27.04	-27.03	-81.09

Table 8.21: Table listing the evaluation results for the simple spread environment, trained on the DDPG algorithm.

Embedding Type	agent_0	agent_1	agent_2	Total Reward
TD3	**-23.43**	**-23.42**	**-23.47**	**-70.32**
$\text{TD3}_{\text{general emb}}$	-27.7	-27.71	-27.71	-83.12
$\text{TD3}_{\text{imitation emb}}$	-23.6	-23.61	-23.62	-70.83
$\text{TD3}_{\text{zero emb}}$	-28.15	-28.16	-28.17	-84.48
$\text{TD3}_{\text{random emb}}$	-29.78	-29.8	-29.8	-89.38

Table 8.22: Table listing the evaluation results for the simple spread environment, trained on the TD3 algorithm.

Embedding Type	agent_0	agent_1	agent_2	Total Reward
MADDPG	**-14.21**	**-14.21**	**-14.21**	**-42.63**
$\text{MADDPG}_{\text{general emb}}$	-29.33	-29.32	-29.34	-87.99
$\text{MADDPG}_{\text{imitation emb}}$	-24.51	-24.49	-24.49	-73.49
$\text{MADDPG}_{\text{zero emb}}$	-29.48	-29.45	-29.5	-88.43
$\text{MADDPG}_{\text{random emb}}$	-27.76	-27.74	-27.73	-83.23

Table 8.23: Table listing the evaluation results for the simple spread environment, trained on the MADDPG algorithm.

Embedding Type	agent_0	agent_1	agent_2	Total Reward
MATD3	**-13.79**	**-13.78**	**-13.82**	**-41.39**
$\text{MATD3}_{\text{general emb}}$	-29.75	-29.76	-29.74	-89.25
$\text{MATD3}_{\text{imitation emb}}$	-25.17	-25.15	-25.17	-75.49
$\text{MATD3}_{\text{zero emb}}$	-28.63	-28.61	-28.64	-85.88
$\text{MATD3}_{\text{random emb}}$	-29.32	-29.32	-29.31	-87.95

Table 8.24: Table listing the evaluation results for the simple spread environment, trained on the MATD3 algorithm.

8.5 Simple Tag

Simple Tag (Fig. 6.1e) is a competitive environment, which consists of three or more *adversary* agents hunting one ore more *prey* agent, while also having two or more impassable landmarks as obstacles. For the experiments conducted, three *adversaries*, one *prey* and two landmarks are used. The *adversary* agents move slightly slower than the *prey* agent and need to learn to coordinate in order to corner the *prey* and collide with it. Meanwhile the *prey* agent tries to avoid getting tagged, but can't just run away in a straight line, because it will be punished by a bounding function the further it runs away from the center of the environment. *Adversaries* are rewarded for each collision, while the *prey* agent is punished for each collision. [LWT+17]. [TBG+21]

For *Simple Tag*, the dimensions of the observation-action pairs for are $adversary_i \leftarrow (16, 5)$ and for $prey \leftarrow (14, 5)$ resulting in a total dimension of $dim = 82$ for three *adversaries* and one *prey*. All agents observe their own position, velocity, landmark position as well as other agents position and velocities. For experiments, the three *adversaries* are used as base agents with their observation-action pairs concatenated into one input vector of $dim = 63$ and embedding output dimension of $dim = 30$.

DTDE algorithms show a common pattern during training when using generalized embeddings, which closely follows the performance of baseline training curves, while imitation embeddings show static behavior (Fig. [8.25 8.26]). CTDE methods show rapid improvements during training when using imitation embeddings compared to the baseline policies, while the utilization of generalized embeddings closely follows the training curves of the initial set of policies (Fig. [8.27 8.28]). This suggests that in centralized approaches, the *adversaries* manage to consistently outperform the *prey* agent during training. This is something that does not hold when pitching the imitation embedding based *prey* agent against *adversaries* it has never seen before.

While evaluation results suggest that using no embeddings yields the best overall results, that is misleading for this environment. Since the *prey* agent does not receive positive rewards at all, and only gets punished by either straying too far from the center of the environment or colliding with *adversary* agents, per agent results suggest highly improved performance for the *prey* agent in CTDE algorithms. This is shown by the three *adversary* agents receiving very little rewards when hunting a *prey* agent that is using imitation embeddings, while the *prey* agent itself is punished less by not getting captured (Tab. [8.27 8.28]). Evaluation results for DTDE show slightly worse performance for all agents, when imitation embeddings are employed (Tab. [8.25 8.26]), while using generalized embeddings yields worse results across all algorithms.

These results suggest that using imitation embeddings of adversaries in a many versus one environment can yield very advantageous performance for the agent that's facing multiple opponents which have been jointly trained with global critics.

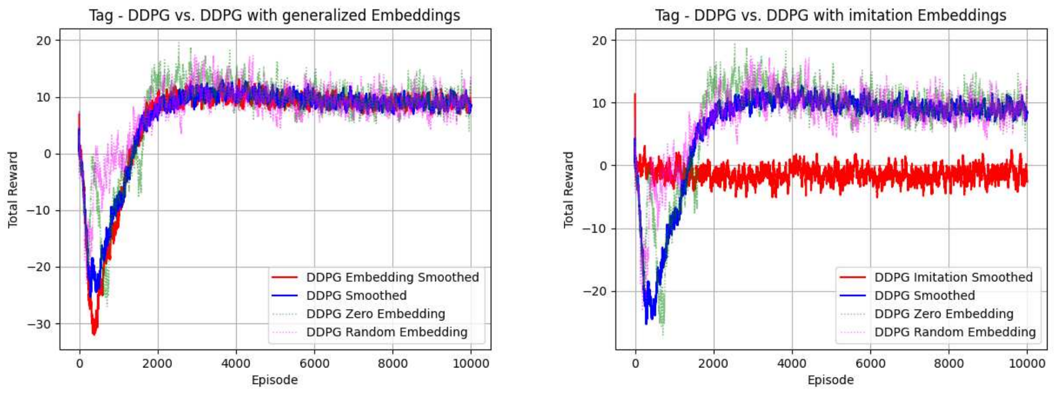


Figure 8.25: Training curves of the simple tag environment with DDPG using generalized embeddings (left) and imitation embeddings (right).

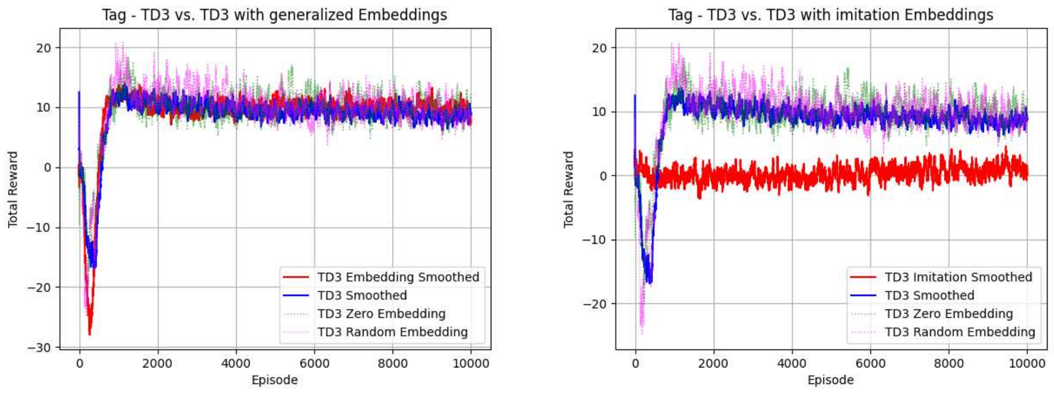


Figure 8.26: Training curves of the simple tag environment with TD3 using generalized embeddings (left) and imitation embeddings (right).

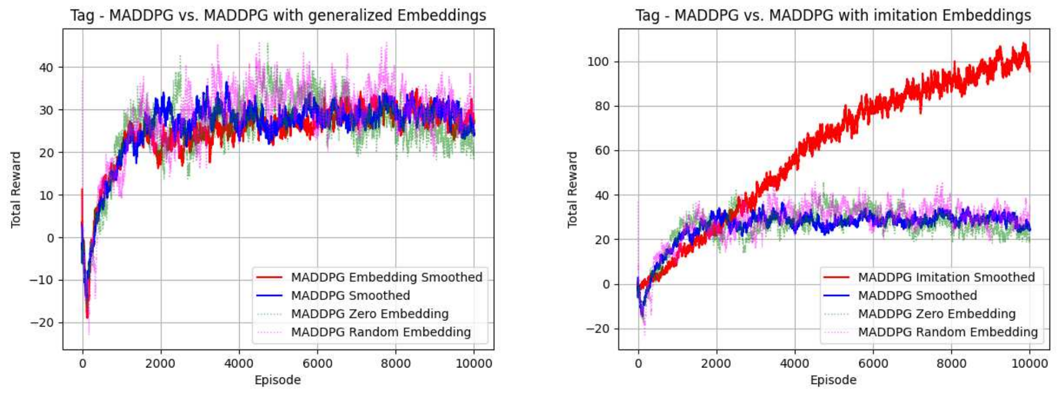


Figure 8.27: Training curves of the simple tag environment with MADDPG using generalized embeddings (left) and imitation embeddings (right).

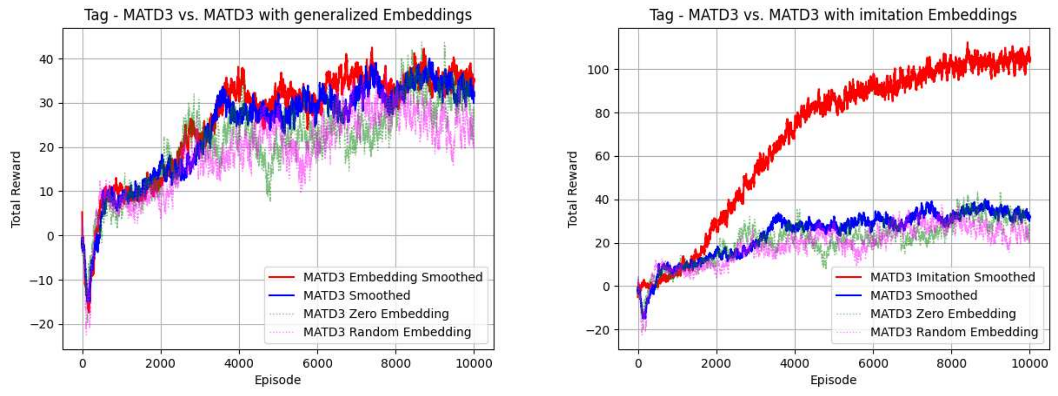


Figure 8.28: Training curves of the simple tag environment with MATD3 using generalized embeddings (left) and imitation embeddings (right).

Embedding Type	adv_0	adv_1	adv_2	$agent_0$	Total Reward
DDPG	**4.7**	**4.7**	**4.7**	**-5.36**	**8.74**
$DDPG_{\text{general emb}}$	3.91	3.91	3.91	-24.55	-12.82
$DDPG_{\text{imitation emb}}$	3.79	3.79	3.79	-10.29	1.08
$DDPG_{\text{zero emb}}$	3.99	3.99	3.99	-35.85	-23.88
$DDPG_{\text{random emb}}$	4.15	4.15	4.15	-32.24	-19.79

Table 8.25: Table listing the evaluation results for the simple tag environment, trained on the DDPG algorithm.

Embedding Type	adv_0	adv_1	adv_2	$agent_0$	Total Reward
TD3	**4.67**	**4.67**	**4.67**	**-5.66**	**8.35**
$TD3_{\text{general emb}}$	4.08	4.08	4.08	-22.66	-10.42
$TD3_{\text{imitation emb}}$	3.85	3.85	3.85	-10.32	1.23
$TD3_{\text{zero emb}}$	3.54	3.54	3.54	-27.04	-16.42
$TD3_{\text{random emb}}$	3.66	3.66	3.66	-24.37	-13.39

Table 8.26: Table listing the evaluation results for the simple tag environment, trained on the TD3 algorithm.

Embedding Type	adv_0	adv_1	adv_2	$agent_0$	Total Reward
MADDPG	**17.12**	**17.12**	**17.12**	-25.59	**25.77**
$MADDPG_{\text{general emb}}$	3.96	3.96	3.96	-29.77	-17.89
$MADDPG_{\text{imitation emb}}$	0.49	0.49	0.49	**-5.73**	-4.26
$MADDPG_{\text{zero emb}}$	3.79	3.79	3.79	-31.24	-19.87
$MADDPG_{\text{random emb}}$	3.44	3.44	3.44	-29.26	-18.94

Table 8.27: Table listing the evaluation results for the simple tag environment, trained on the MADDPG algorithm.

Embedding Type	adv_0	adv_1	adv_2	$agent_0$	Total Reward
MATD3	**20.48**	**20.48**	**20.48**	-26.19	**35.25**
$MATD3_{\text{general emb}}$	4.01	4.01	4.01	-28.44	-16.41
$MATD3_{\text{imitation emb}}$	1.04	1.04	1.04	**-7.73**	-4.61
$MATD3_{\text{zero emb}}$	4.39	4.39	4.39	-32.58	-19.41
$MATD3_{\text{random emb}}$	3.93	3.93	3.93	-30.0	-18.21

Table 8.28: Table listing the evaluation results for the simple tag environment, trained on the MATD3 algorithm.

8.6 Simple World Comm

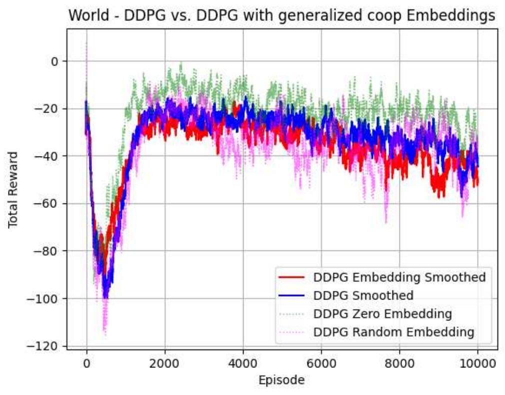
The *Simple World Comm* environment (Fig. 6.1f) is an extension of *Simple Tag*, which adds more complexity to it. On top of having three or more *adversary* agents, there is also a *leader adversary* agent that coordinates the *adversaries*, while not actively participating in the chase itself. The two or more *prey* agents have safe zones in which they can hide their positional data from the *adversaries*, while trying to evade them and staying as close as possible to *food* landmarks. Additionally, the *prey* agents are bound by the same bounding function as in *Simple Tag*. *Adversary* agents get rewarded for every collision with a *prey* agent and punished based on their minimum distance to a *prey* agent. The *prey* agents on the other hand get rewarded for collecting food landmarks, and punished for colliding with *adversaries*, or being too far away from the center of the environment and *food* landmarks. [LWT+17] [TBG+21]

Simple World Comm is the most complex environment of the Multi-Agent Particle Environments with 6 agents of following dimensions for observation-action pairs:
$leader\ adversary \leftarrow (34, 9)$, $adversary_i \leftarrow (34, 5)$, $prey_i \leftarrow (28, 5)$. Using 3 *adversaries* and 2 *prey* agents leads to a total observation-action space of dimension $dim = 226$. All agents observe their own position, velocity, landmark location, if they are in the safe zones, as well as the location and velocity of other agents. The *leader adversary* can also communicate suggested actions to the other *adversaries*. Due to simultaneous cooperation and competition in this environment, a similar approach to *Simple Crypto* is used, where embeddings promoting cooperation, competition and a applying both at once is considered. For cooperation, the three *adversaries* are used as base agents with the *leader adversary* being the target, to create better instructions for the *adversaries*. An input with dimension $dim = 117$ consisting of observation-action pairs of the *adversaries* is concatenated, with an output embedding dimension of $dim = 60$. In the case of competitive embeddings, the *leader adversary* is used as a base agent with the *prey* agents as targets to convey the coordination intent of the *leader adversary*, thus embedding the observation-action pairs of the *leader adversary* from dimension $dim = 43$ into output dimension $dim = 20$. For the mixed setting, both embedding networks are used simultaneously.

8.6.1 Cooperative Embeddings

Using cooperative embeddings has no real impact on training curves, whether generalized embeddings or imitation embeddings are used for DTDE (Fig. [8.29 8.30]), while CTDE methods display slightly worse training performance, with MATD3 showing slightly better early training results when using generalized embeddings (Fig. [8.318.32]). For evaluation results, using generalized embeddings does not lead to any improvements, with even zero embeddings outperforming all but one methods (Tab. [8.29 8.30 8.31]).

When evaluating cooperative embeddings with imitation embeddings, the same behavior as seen in *Simple Speaker Listener* can be observed, with the anomalous agents being the *prey* agents that did not receive any embeddings. While the *prey* agents show deteriorating



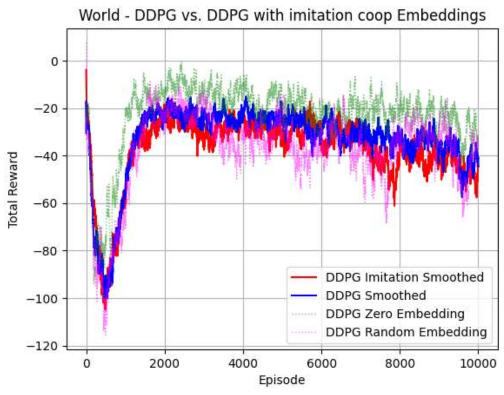
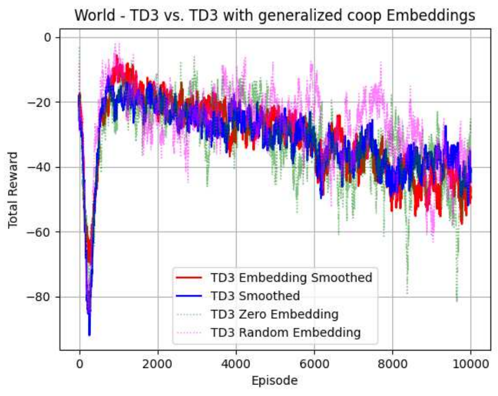


Figure 8.29: Training curves of the simple world comm environment with DDPG using cooperative generalized embeddings (left) and imitation embeddings (right).



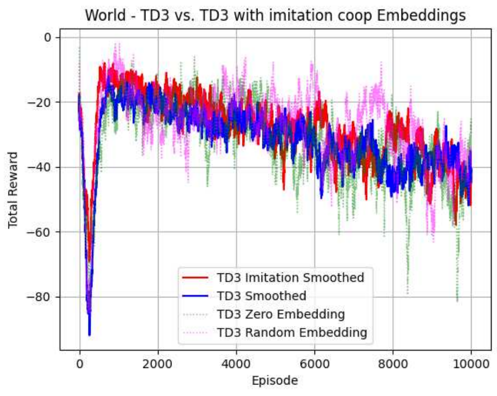


Figure 8.30: Training curves of the simple world comm environment with TD3 using generalized cooperative embeddings (left) and imitation embeddings (right).

performance across all algorithms, DTDE methods seem to have one agent perform much worse than the other, while for CTDE, both agents exhibit a similarly steep decline.

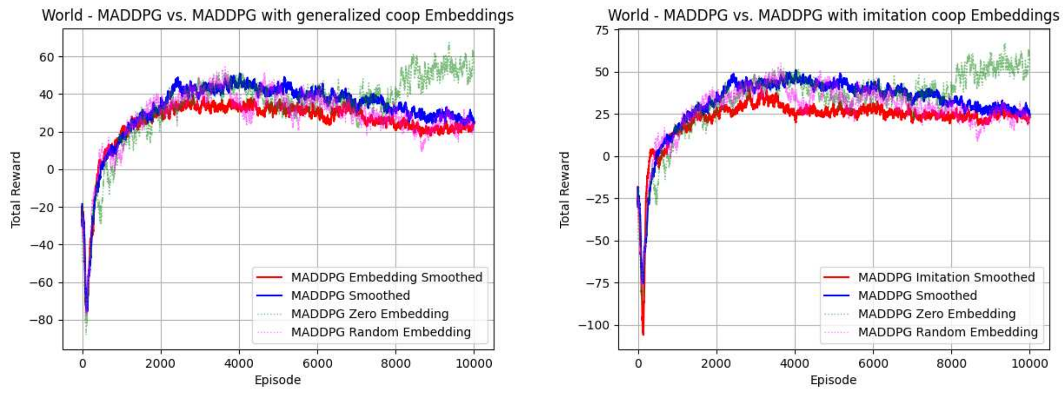


Figure 8.31: Training curves of the simple world comm environment with MADDPG using cooperative generalized embeddings (left) and imitation embeddings (right).

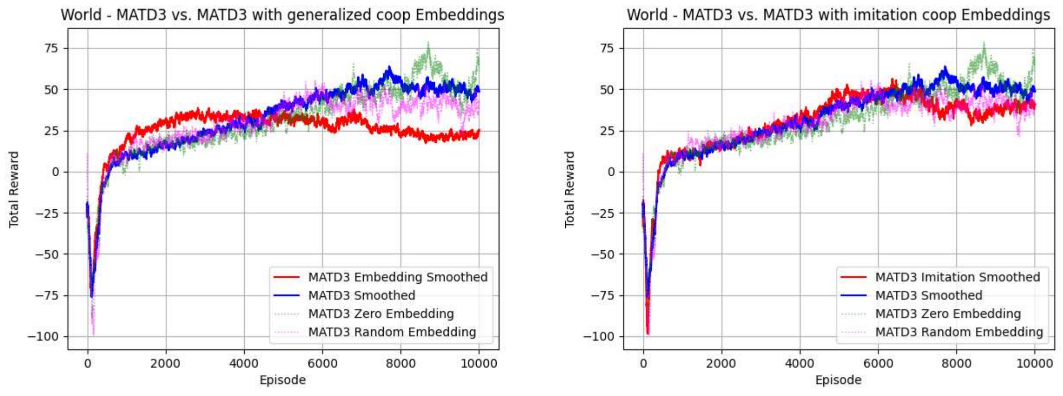


Figure 8.32: Training curves of the simple world comm environment with MATD3 using cooperative generalized embeddings (left) and imitation embeddings (right).

Embedding Type	lead_0	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
DDPG	**4.61**	**4.64**	**4.6**	**4.59**	-60.88	**-4.95**	**-47.39**
$\text{DDPG}_{\text{general emb}}$	2.56	2.5	2.51	2.54	**-46.47**	-44.75	-81.11
$\text{DDPG}_{\text{imitation emb}}$	-0.26	-0.89	-0.95	-0.85	-157.53	-323.93	-484.41
$\text{DDPG}_{\text{zero emb}}$	2.92	2.96	3.0	3.08	-35.87	-51.52	-75.43
$\text{DDPG}_{\text{random emb}}$	2.25	2.25	2.16	2.43	-54.09	-57.82	-102.82

Table 8.29: Table listing the evaluation results for the simple world com environment, trained on the DDPG algorithm using cooperative embeddings.

Embedding Type	lead_0	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
TD3	**4.53**	**4.63**	**4.66**	**4.56**	-60.6	**-4.92**	**-47.14**
$\text{TD3}_{\text{general emb}}$	2.42	2.4	2.44	2.44	**-55.61**	-47.01	-92.92
$\text{TD3}_{\text{imitation emb}}$	-0.09	-0.58	-0.48	-0.49	-126.83	-329.89	-458.36
$\text{TD3}_{\text{zero emb}}$	2.59	2.59	2.52	2.76	-35.82	-40.25	-65.61
$\text{TD3}_{\text{random emb}}$	2.64	2.7	2.73	2.68	-47.22	-38.97	-75.44

Table 8.30: Table listing the evaluation results for the simple world com environment, trained on the TD3 algorithm using cooperative embeddings.

Embedding Type	lead_0	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
MADDPG	**12.0**	**11.96**	**11.95**	**11.98**	**-9.3**	**-10.13**	**28.46**
$\text{MADDPG}_{\text{general emb}}$	2.46	2.44	2.44	2.44	-54.18	-72.13	-116.53
$\text{MADDPG}_{\text{imitation emb}}$	-1.3	-2.03	-1.88	-1.97	-302.62	-326.33	-636.13
$\text{MADDPG}_{\text{zero emb}}$	2.8	2.89	2.66	2.86	-38.63	-61.39	-88.81
$\text{MADDPG}_{\text{random emb}}$	2.32	2.51	2.17	2.36	-57.8	-68.05	-116.49

Table 8.31: Table listing the evaluation results for the simple world com environment, trained on the MADDPG algorithm using cooperative embeddings.

Embedding Type	lead_0	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
MATD3	**18.88**	**18.76**	**18.81**	**18.81**	**-12.12**	**-12.68**	**50.46**
$\text{MATD3}_{\text{general emb}}$	2.66	2.67	2.67	2.73	-47.18	-49.32	-85.77
$\text{MATD3}_{\text{imitation emb}}$	-1.92	-2.38	-2.45	-2.51	-315.09	-334.76	-659.11
$\text{MATD3}_{\text{zero emb}}$	2.39	2.45	2.41	2.49	-51.74	-71.68	-113.68
$\text{MATD3}_{\text{random emb}}$	2.67	2.52	2.53	2.55	-35.28	-49.41	-74.42

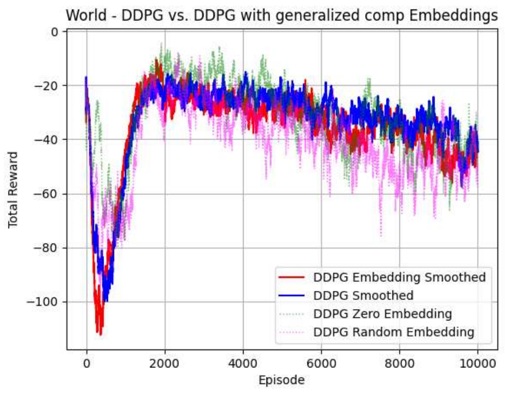
Table 8.32: Table listing the evaluation results for the simple world com environment, trained on the MATD3 algorithm using cooperative embeddings.

8.6.2 Competitive Embeddings

For the competitive embeddings, the learning curves for DTDE remain roughly the same when using generalized embeddings, while imitation embeddings exhibit static behavior during training (Fig. [8.33 8.34]). In the case of CTDE methods, generalized embeddings also perform similar to the baseline policies during training, while imitation embeddings show worse episodic rewards over the training period but exhibit slow improvements (Fig. [8.35 8.36).

Similar to *Simple Tag*, the goal for this experiment is not to maximize the total reward, but to help the *prey* agents avoid their *adversaries.* For imitation embeddings, this leads to increased *prey* performance for both DTDE methods (Tab. [8.33 8.34]), which also result in the best overall performance. This is achieved by both *prey* agents acting similarly, instead of one performing significantly better than the other, while also be able to evade the *adversaries.*

For CTDE algorithms, the performance of the *prey* agents is slightly reduced compared to baseline policies, but performance of their *adversaries* is also significantly reduced, which implies that they are more adapt in not getting caught by keeping away from the *adversaries*, while sticking close to the center or inside the safe zone. Utilizing generalized embeddings leads to worse results across all algorithms and is on par with using random or all zero embeddings.



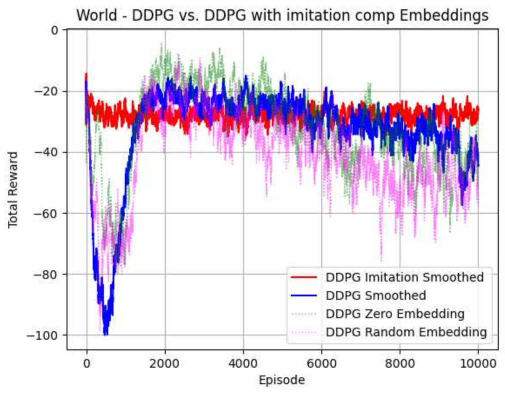
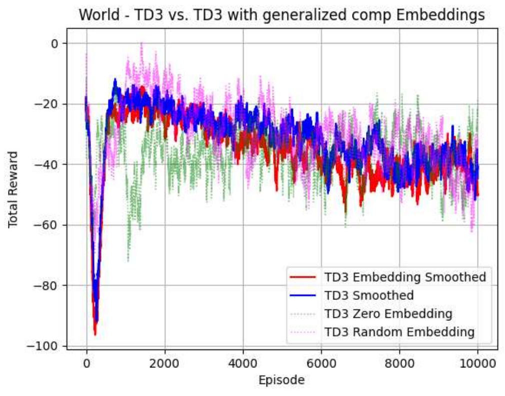


Figure 8.33: Training curves of the simple world comm environment with DDPG using competitive generalized embeddings (left) and imitation embeddings (right).



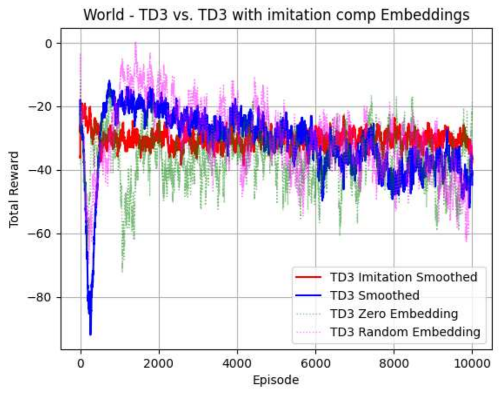


Figure 8.34: Training curves of the simple world comm environment with TD3 using generalized competitive embeddings (left) and imitation embeddings (right).

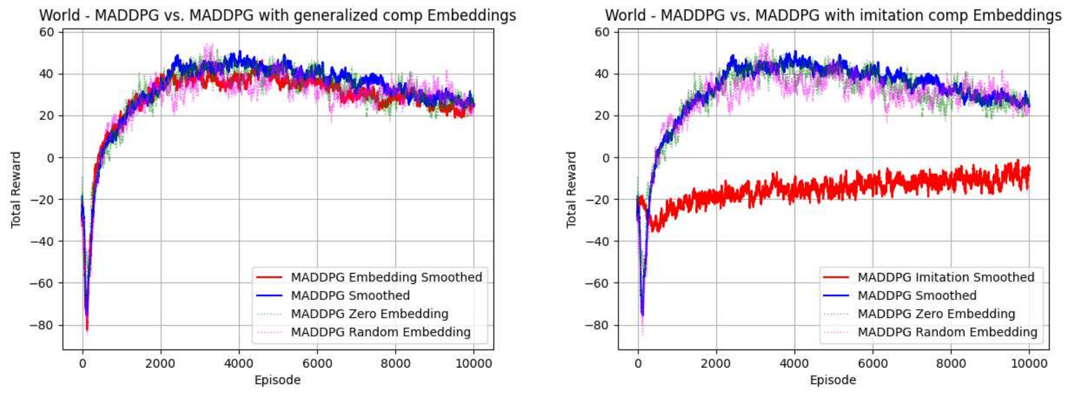


Figure 8.35: Training curves of the simple world comm environment with MADDPG using competitive generalized embeddings (left) and imitation embeddings (right).

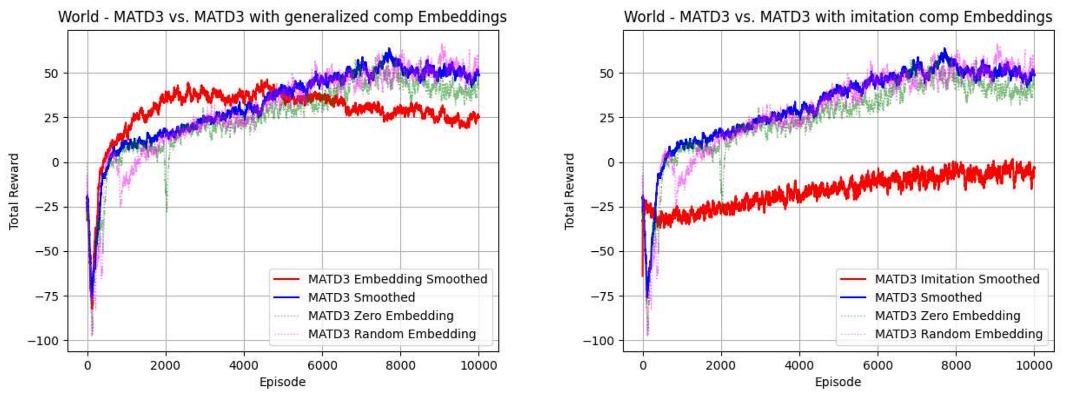


Figure 8.36: Training curves of the simple world comm environment with MATD3 using competitive generalized embeddings (left) and imitation embeddings (right).

Embedding Type	lead_0	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
DDPG	**4.61**	**4.64**	**4.6**	**4.59**	-60.88	**-4.95**	-47.39
$\text{DDPG}_{\text{general emb}}$	2.47	2.43	2.46	2.52	-52.9	-52.66	-95.68
$\text{DDPG}_{\text{imitation emb}}$	2.1	2.14	2.08	2.16	**-12.92**	-12.92	**-17.36**
$\text{DDPG}_{\text{zero emb}}$	2.98	3.01	2.97	2.89	-57.7	-45.77	-91.62
$\text{DDPG}_{\text{random emb}}$	2.49	2.56	2.34	2.45	-45.06	-40.6	-75.82

Table 8.33: Table listing the evaluation results for the simple world com environment, trained on the DDPG algorithm using competitive embeddings.

Embedding Type	lead_0	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
TD3	**4.53**	**4.63**	**4.66**	**4.56**	-60.6	**-4.92**	-47.14
$\text{TD3}_{\text{general emb}}$	2.55	2.5	2.55	2.58	-51.31	-49.53	-90.66
$\text{TD3}_{\text{imitation emb}}$	2.47	2.29	2.5	2.48	**-13.52**	-12.99	**-16.77**
$\text{TD3}_{\text{zero emb}}$	2.74	2.61	2.83	2.65	-47.31	-42.42	-78.9
$\text{TD3}_{\text{random emb}}$	2.52	2.49	2.61	2.56	-50.56	-47.32	-87.7

Table 8.34: Table listing the evaluation results for the simple world com environment, trained on the TD3 algorithm using competitive embeddings.

Embedding Type	lead_0	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
MADDPG	**12.0**	**11.96**	**11.95**	**11.98**	**-9.3**	**-10.13**	**28.46**
$\text{MADDPG}_{\text{general emb}}$	2.44	2.49	2.47	2.47	-60.96	-48.58	-99.67
$\text{MADDPG}_{\text{imitation emb}}$	-0.18	-0.93	-0.89	-0.93	-12.39	-13.05	-28.37
$\text{MADDPG}_{\text{zero emb}}$	2.17	2.06	2.17	2.13	-52.69	-43.75	-87.91
$\text{MADDPG}_{\text{random emb}}$	2.88	2.84	2.96	2.88	-55.31	-42.6	-86.35

Table 8.35: Table listing the evaluation results for the simple world com environment, trained on the MADDPG algorithm using competitive embeddings.

Embedding Type	lead_0	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
MATD3	**18.88**	**18.76**	**18.81**	**18.81**	**-12.12**	**-12.68**	**50.46**
$\text{MATD3}_{\text{general emb}}$	2.56	2.47	2.54	2.6	-64.75	-56.94	-111.52
$\text{MATD3}_{\text{imitation emb}}$	-0.39	-1.12	-1.21	-1.27	-18.59	-18.44	-41.02
$\text{MATD3}_{\text{zero emb}}$	2.51	2.57	2.73	2.63	-35.92	-68.24	-93.72
$\text{MATD3}_{\text{random emb}}$	2.33	2.32	2.29	2.38	-36.6	-54.85	-82.13

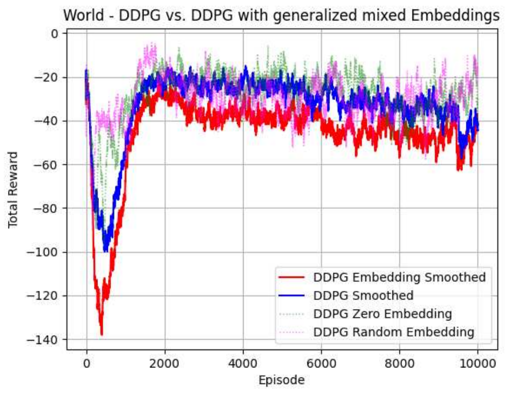
Table 8.36: Table listing the evaluation results for the simple world com environment, trained on the MATD3 algorithm using competitive embeddings.

8.6.3 Mixed Embeddings

When using both cooperative and competitive embeddings at the same time, all agents experience a loss in performance when using generalized embeddings. Using both cooperative and competitive imitation embeddings does not cause abnormal results like solely implementing cooperative embeddings. Rather, the behavior displayed when using both cooperative and competitive imitation embeddings, aligns more with the result of only using the competitive embeddings. Implementing both generalized embedding networks does not show different results compared to only using either cooperative or competitive generalized embeddings.

This can be interpreted as the competitive embeddings overshadowing the impact of the cooperative embeddings. The best overall performance for DTDE models is achieved when using imitation embeddings and imitation policies as baselines for the three target agents *leader adversary*, *agent*$_1$, *agent*$_2$ (Tab. [8.37 8.38. CTDE methods behave in a similar manner compared to only using competitive embeddings, where *prey* agents perform slightly worse but manage to evade the *adversaries* much better while not straying too far from the environment origin, therefore lowering their per agent and overall reward significantly (Tab. [8.39 8.40]).

In summary, when using both cooperative and competitive imitation embeddings simultaneously, the influence of competitive embeddings overshadows the impact of cooperative embeddings and leads to better overall *prey* performance in DTDE, and worse significantly *adversary* performance in CTDE.



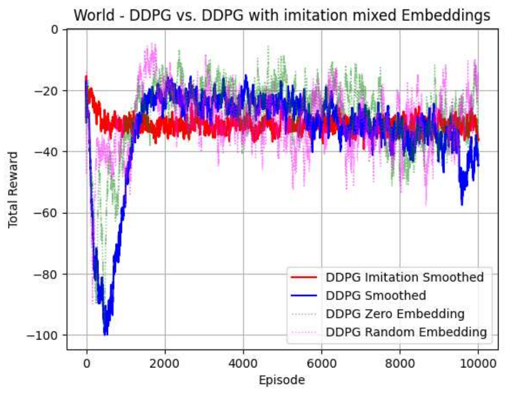
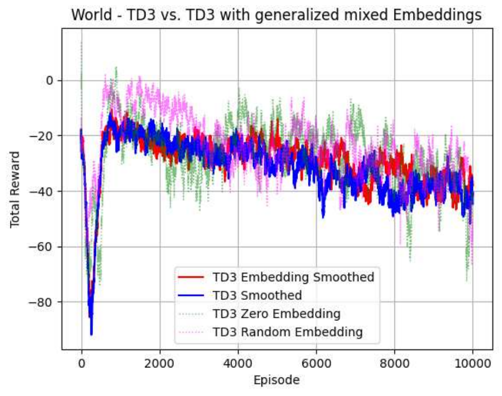


Figure 8.37: Training curves of the simple world comm environment with DDPG using mixed generalized embeddings (left) and imitation embeddings (right).



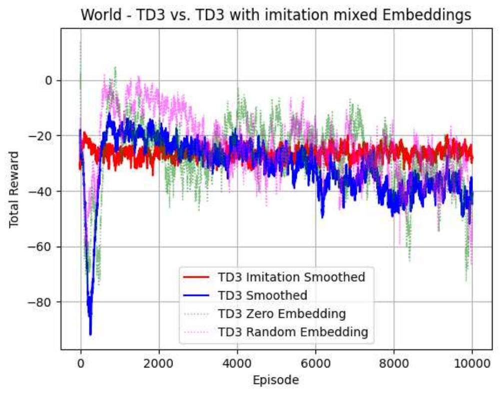


Figure 8.38: Training curves of the simple world comm environment with TD3 using mixed generalized embeddings (left) and imitation embeddings (right).

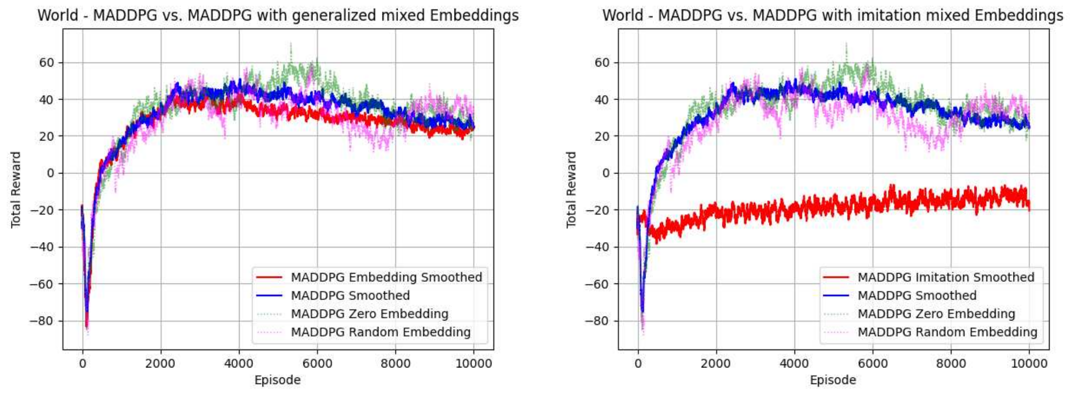


Figure 8.39: Training curves of the simple world comm environment with MADDPG using mixed generalized embeddings (left) and imitation embeddings (right).

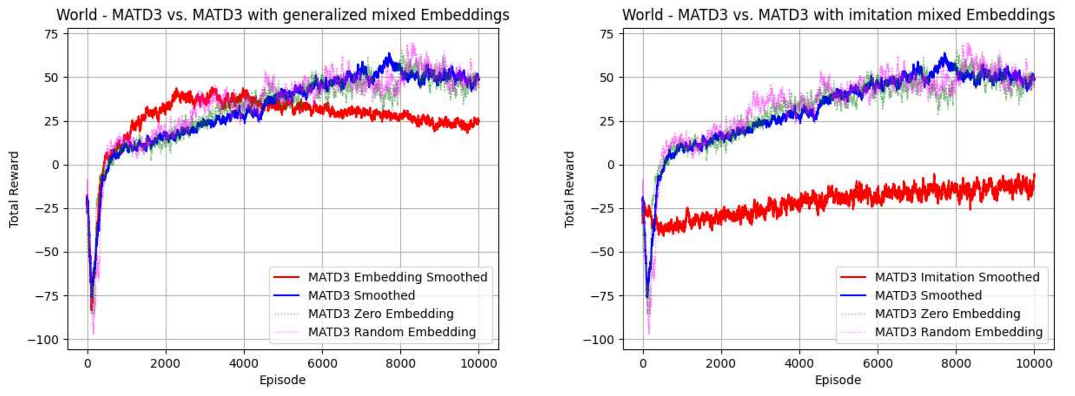


Figure 8.40: Training curves of the simple world comm environment with MATD3 using mixed generalized embeddings (left) and imitation embeddings (right).

Embedding Type	$lead_0$	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
DDPG	**4.61**	**4.64**	**4.6**	**4.59**	-60.88	**-4.95**	-47.39
$DDPG_{general\ emb}$	2.49	2.44	2.5	2.53	-58.65	-44.93	-93.62
$DDPG_{imitation\ emb}$	2.5	1.73	1.65	1.68	**-15.25**	-15.09	**-22.78**
$DDPG_{zero\ emb}$	2.37	2.5	2.44	2.28	-50.54	-43.58	-84.53
$DDPG_{random\ emb}$	2.76	2.74	2.79	2.69	-53.8	-46.93	-89.75

Table 8.37: Table listing the evaluation results for the simple world com environment, trained on the DDPG algorithm using mixed embeddings.

Embedding Type	$lead_0$	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
TD3	**4.53**	**4.63**	**4.66**	**4.56**	-60.6	**-4.92**	-47.14
$TD3_{general\ emb}$	2.56	2.58	2.55	2.59	-47.41	-55.7	-92.83
$TD3_{imitation\ emb}$	2.72	1.96	2.14	2.15	**-13.37**	-13.22	**-17.62**
$TD3_{zero\ emb}$	2.22	2.17	2.12	2.19	-65.63	-46.76	-103.69
$TD3_{random\ emb}$	2.47	2.56	2.52	2.5	-31.71	-53.89	-75.55

Table 8.38: Table listing the evaluation results for the simple world com environment, trained on the TD3 algorithm using mixed embeddings.

Embedding Type	$lead_0$	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
MADDPG	**12.0**	**11.96**	**11.95**	**11.98**	**-9.3**	**-10.13**	**28.46**
$MADDPG_{general\ emb}$	2.53	2.57	2.53	2.51	-59.5	-63.06	-112.42
$MADDPG_{imitation\ emb}$	0.8	-1.14	-1.07	-1.03	-11.94	-11.55	-25.93
$MADDPG_{zero\ emb}$	2.93	2.78	2.69	2.89	-52.47	-52.58	-93.76
$MADDPG_{random\ emb}$	2.06	2.14	2.12	1.99	-42.78	-66.97	-101.44

Table 8.39: Table listing the evaluation results for the simple world com environment, trained on the MADDPG algorithm using mixed embeddings.

Embedding Type	$lead_0$	adv_0	adv_1	adv_2	ag_0	ag_1	Total Reward
MATD3	**18.88**	**18.76**	**18.81**	**18.81**	**-12.12**	**-12.68**	**50.46**
$MATD3_{general\ emb}$	2.49	2.42	2.45	2.45	-57.41	-55.98	-103.58
$MATD3_{imitation\ emb}$	0.6	-1.19	-1.21	-1.38	-13.43	-14.19	-30.8
$MATD3_{zero\ emb}$	2.58	2.44	2.49	2.39	-53.55	-42.53	-86.18
$MATD3_{random\ emb}$	2.83	2.71	2.95	2.86	-58.43	-48.32	-95.4

Table 8.40: Table listing the evaluation results for the simple world com environment, trained on the MATD3 algorithm using mixed embeddings.

9 Conclusion and Future Works

Including agent representations by using embeddings of agents as privileged information for other agents can lead to meaningful improvements for that agent, especially in competitive settings. For cooperative settings, employing embeddings didn't lead to the expected results of overall better performance, which lends itself as a topic for further investigation. Competitive settings are a lot more receptive to the inclusion of agent representations, where they display elevated target agent performance in all but one scenarios. Imitation embeddings lead to overall best performance for DTDE models in *Simple Push*, as well as the competitive and mixed implementation of *Simple World Comm* and best target agent performance in *Simple Tag*. While CTDE methods suffered in overall performance due to implementing competitive embeddings, the uplift in performance of the *good* or *prey* agents of those environments results in getting caught less often by the *adversary* agents. Using generalized embeddings doesn't yield any improvements across all experiments, which could be the result of trying to squeeze the information of too many policies into one singular representation. Determining an approach that does manage to properly capture the behavior of many different policies could also be a topic for further research.

Using imitation embeddings can therefore lead to improved per agent performance in both CTDE and DTDE approaches, but is most useful when training in a decentralized manner. Since CTDE methods utilize one or more central critic per actor during training, which conditions the actor networks based on the combined information of all agents, thus raising overall performance is more difficult to achieve. Increasing individual performance of agents is more easily achievable at the cost of the performance of other agents in competitive settings, and overall rewards of a competitive environment. This is due to centralized methods already having mostly solved the biggest hurdle of multi-agent reinforcement learning, which is the non-stationary nature of an environment during training from an agent's point of view. While the individual actors don't have full access to all information at any point in time, the central critics condition them as if they did.

In the case of DTDE algorithms, where every agent is its own closed off actor-critic algorithm with no shared experience, using embeddings of one or more agents can be a sufficient means of propagating agent information to increase performance. While in cooperative settings the overall results were consistently worse than not using embeddings at all, improvements with further tuning of the embedding network and other hyper parameters could provide better results.

While delivering promising results, especially for decentralized learning in competitive settings, the extra effort of training baseline policies, training embedding and imitation policy networks from those baselines, and then re-training new agents that can utilize said embeddings adds a lot more complexity and computational effort, which can be a drawback. While centralized learning suffers from its own problems, like rapidly growing dimensionality for the critic networks, advancements in modern computer hardware can compensate for that while training competitive policies with less effort. Although this is only applicable for scenarios which allow for centralized training.

Further research can be done by studying the impact of neural network parameters and embedding sizes when encoding multiple agents into one representation. Since the amount of episodes during training was on the lower side compared to training an environment for $1 \cdot 10^6$ episodes, due to the volume of policies that had to be trained, investigations into longer training periods can be done to evaluate their impact on performance, especially for cooperative settings. Another topic for consideration is to find a way to only apply the embeddings during training while not having to rely on them during execution as well, which is where most of the discrepancies between training and evaluation performance stemmed from.

Bibliography

[ACF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[ACS24] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches.* MIT Press, 2024.

[AGOS19] Johannes Ackermann, Volker Gabler, Takayuki Osa, and Masashi Sugiyama. Reducing overestimation bias in multi-agent domains using double centralized critics. *CoRR*, abs/1910.01465, 2019.

[Alp24] AlphaProof and AlphaGeometry teams. Ai achieves silver-medal standard solving international mathematical olympiad problems. *DeepMind Blog*, July 2024.

[BBC$^+$19] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[Bel66] Richard Bellman. Dynamic programming. *science*, 153(3731):34–37, 1966.

[CKCL19] Oscar Chang, Robert Kwiatkowski, Siyuan Chen, and Hod Lipson. Agent embeddings: A latent representation for pole-balancing networks. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019*, pages 656–664. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[FCA$^+$18] Jakob N. Foerster, Richard Y. Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 122–130. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018.

[FFA$^+$18] Jakob N. Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 2974–2982. AAAI Press, 2018.

[FvHM18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1582–1591. PMLR, 2018.

[GAG$^+$18] Aditya Grover, Maruan Al-Shedivat, Jayesh K. Gupta, Yuri Burda, and Harrison Edwards. Learning policy representations in multiagent systems. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1797–1806. PMLR, 2018.

[HBZ04] Eric A. Hansen, Daniel S. Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 709–715. AAAI Press / The MIT Press, 2004.

[HS88] John C Harsanyi and Reinhard Selten. A general theory of equilibrium selection in games. *MIT Press Books*, 1, 1988.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[HZL$^+$24] Zican Hu, Zongzhang Zhang, Huaxiong Li, Chunlin Chen, Hongyu Ding, and Zhi Wang. Attention-guided contrastive role representations for multi-agent reinforcement learning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[JEP$^+$20] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Kathryn Tunyasuvunakool, Olaf Ronneberger, Russ Bates, Augustin Žídek, Alex Bridgland, et al. Alphafold 2. *Fourteenth Critical Assessment of Techniques for Protein Structure Prediction*, page 13, 2020.

[JW22] Caiyu Jiang and Jianhua Wang. A portfolio model with risk control policy based on deep reinforcement learning. *Mathematics*, 11:19, 12 2022.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res.*, 4:237–285, 1996.

[KST$^+$24] Aleksandar Krnjaic, Raul D. Steleac, Jonathan D. Thomas, Georgios Papoudakis, Lukas Schäfer, Andrew Wing-Keung To, Kuan-Ho Lao, Murat Cubuktepe, Matthew Haley, Peter Börsting, and Stefano V. Albrecht. Scalable multi-agent reinforcement learning for warehouse logistics with robotic and human co-workers. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2024, Abu Dhabi, United Arab Emirates, October 14-18, 2024*, pages 677–684. IEEE, 2024.

[KT99] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 1008–1014. The MIT Press, 1999.

[LFK21] Cheng Li, Levi Fussell, and Taku Komura. Multi-agent reinforcement learning for character control. *Vis. Comput.*, 37(12):3115–3123, 2021.

[LHP$^+$16] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[LMF11] Guillaume J. Laurent, Laëtitia Matignon, and Nadine Le Fort-Piat. The world of independent learners is not markovian. *Int. J. Knowl. Based Intell. Eng. Syst.*, 15(1):55–64, 2011.

[LO02] Jae Won Lee and Jangmin O. A multi-agent q-learning framework for optimizing stock trading systems. In *Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*, volume 2453 of *Lecture Notes in Computer Science*, pages 153–162. Springer, 2002.

[LSW$^+$22] Rémi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Alexander Pritzel, Suman V. Ravuri, Timo Ewalds, Ferran Alet, Zach Eaton-Rosen, Weihua Hu, Alexander Merose, Stephan Hoyer, George Holland, Jacklynn Stott, Oriol Vinyals, Shakir Mohamed, and Peter W. Battaglia. Graphcast: Learning skillful medium-range global weather forecasting. *CoRR*, abs/2212.12794, 2022.

[LWT+17] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6379–6390, 2017.

[MBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1928–1937. JMLR.org, 2016.

[MCR+22] Reuth Mirsky, Ignacio Carlucho, Arrasy Rahman, Elliot Fosong, William Macke, Mohan Sridharan, Peter Stone, and Stefano V. Albrecht. A survey of ad hoc teamwork research. In *Multi-Agent Systems - 19th European Conference, EUMAS 2022, Düsseldorf, Germany, September 14-16, 2022, Proceedings*, volume 13442 of *Lecture Notes in Computer Science*, pages 275–293. Springer, 2022.

[Mid23] MidJourney. Midjourney: Ai-based image generation tool, 2023. Accessed: 2024-09-13.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533, 2015.

[Nas50] John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950.

[Ope23] OpenAI. Chatgpt: A language model for dialogue applications, 2023. Accessed: 2024-09-13.

[PCRA19] Georgios Papoudakis, Filippos Christianos, Arrasy Rahman, and Stefano V. Albrecht. Dealing with non-stationarity in multi-agent deep reinforcement learning. *CoRR*, abs/1906.04737, 2019.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley Series in Probability and Statistics. Wiley, 1994.

[RAS19] Manish Ravula, Shani Alkoby, and Peter Stone. Ad hoc teamwork with behavior switching agents. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 550–556. ijcai.org, 2019.

[RDSF18] Roberta Raileanu, Emily Denton, Arthur Szlam, and Rob Fergus. Modeling others using oneself in multi-agent reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 4254–4263. PMLR, 2018.

[RPS$^+$18] Neil C. Rabinowitz, Frank Perbet, H. Francis Song, Chiyuan Zhang, S. M. Ali Eslami, and Matthew M. Botvinick. Machine theory of mind. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 4215–4224. PMLR, 2018.

[Rum94] G. A. Rummery. On-line q-learning using connectionist systems. *CTIT technical reports series*, January 1994.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[SHM$^+$16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

[SJ03] Matthew Schultz and Thorsten Joachims. Learning a distance metric from relative comparisons. In *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pages 41–48. MIT Press, 2003.

[SL09] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations.* Cambridge University Press, 2009.

[SLH$^+$14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31th International Conference on Machine Learning, ICML*

2014, Beijing, China, 21-26 June 2014, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 387–395. JMLR.org, 2014.

[SMSM99] Richard S. Sutton, David A. McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 1057–1063. The MIT Press, 1999.

[SRR23] Maximilian Schier, Christoph Reinders, and Bodo Rosenhahn. Deep reinforcement learning for autonomous driving using high-level heterogeneous graph representations. In *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, pages 7147–7153. IEEE, 2023.

[Sut88] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3:9–44, 1988.

[SWD$^+$17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[Sze10] Csaba Szepesvári. *Algorithms for Reinforcement Learning.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

[TA07] Kagan Tumer and Adrian K. Agogino. Distributed agent-based air traffic flow management. In Edmund H. Durfee, Makoto Yokoo, Michael N. Huhns, and Onn Shehory, editors, *6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007), Honolulu, Hawaii, USA, May 14-18, 2007*, page 255. IFAAMAS, 2007.

[Tan93] Ming Tan. Multi-agent reinforcement learning: Independent versus cooperative agents. In Paul E. Utgoff, editor, *Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993*, pages 330–337. Morgan Kaufmann, 1993.

[TBG$^+$21] Justin K. Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S. Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, Niall L. Williams, Yashas Lokesh, and Praveen Ravi. Pettingzoo: Gym for multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 15032–15043, 2021.

[Tes95] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[UAP] Nicholas Ustaran-Anderegg and Michael Pratt. AgileRL.

[UO30] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[vHGS16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 2094–2100. AAAI Press, 2016.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Mach. Learn.*, 8:279–292, 1992.

[Wil92] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, 1992.

[WT13] Paweł Wawrzyński and Ajay Kumar Tanwani. Autonomous reinforcement learning with experience replay. *Neural Networks*, 41:156–167, 2013. Special Issue on Autonomous Learning.

[YVV$^+$22] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre M. Bayen, and Yi Wu. The surprising effectiveness of PPO in cooperative multi-agent games. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[ZSHS23] Shenao Zhang, Li Shen, Lei Han, and Li Shen. Learning meta representations for agents in multi-agent reinforcement learning. In *Conference on Lifelong Learning Agents, 22-25 August 2023, McGill University, Montréal, Québec, Canada*, volume 232 of *Proceedings of Machine Learning Research*, pages 292–317. PMLR, 2023.

[ZXQ22] Haifei Zhang, Jian Xu, and Jianlin Qiu. [retracted] an automatic driving control method based on deep deterministic policy gradient. *Wireless Communications and Mobile Computing*, 2022(1):7739440, 2022.

[ZYB19] Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *CoRR*, abs/1911.10635, 2019.