Dynamic Multi-Agent Reward Sharing

Dynamisches Teilen von Belohnungen in Multi-Agent Reinforcement Learning
Master thesis by Tristan Tisch
Date of submission: September 30, 2024

1. Review: Aryaman Reddi
2. Review: Prof. Carlo D'Eramo
Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Tristan Tisch, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

English translation for information purposes only:

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Tristan Tisch, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt without any outside support and using only the quoted literature and other sources. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I have clearly marked and separately listed in the text the literature used literally or in terms of content and all other sources I used for the preparation of this academic work. This also applies to sources or aids from the Internet.

This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Datum / Date: 30.09.2024

Unterschrift/Signature: 

Abstract

The framework of Reinforcement Learning has led to astonishing results in games that were previously dominated by humans, such as Atari arcade games or the complex board game Go, and also in real-world applications such as robotics. While the focus of RL research has been predominantly on single-agent environments, the recent development of a growing number of environments involving multiple intelligent systems has yielded the need for methods that can efficiently solve complex tasks in multi-agent systems, especially systems that require cooperation like network routing or traffic flow control.

Many multi-agent reinforcement learning algorithms have the downside that they require the agents to learning to communicate with each other, or base their method on joint action learning both of which incurs a computational overhead. Analogously to humans encouraging cooperation in other humans by incentivizing them with some reward, we investigate methods that use reward-sharing to promote actions that are beneficial to reaching the shared goal. To this end, we present an extension to an existing reward-sharing algorithm and a novel approach and evaluate both of them on diverse environments.

Zusammenfassung

Reinforcement Learning-Algorithmen haben erstaunliche Ergebnisse in Aufgaben erziehlen können, die zuvor von Menschen beherrscht wurden, so zum Beispiel in Spielen der Atari-Konsole oder dem komlexen Brettspiel Go, und ebenso in realen Anwendungsfällen wie der Robotik. Während der Schwerpunkt der RL-Forschung vorwiegend auf Einzelagentenumgebungen lag, hat die jüngste Entwicklung einer wachsenden Zahl von Umgebungen mit mehreren intelligenten Systemen zu einem Bedarf an Methoden geführt, die effizient komplexe Aufgaben lösen können, die Kooperation benötigen, wie zum Beispiel Netzwerkrouting oder Verkehrsflusskontrolle.

Viele Multi-Agent Reinforcement-Learning-Algorithmen haben den Nachteil, dass die Agenten lernen müssen, miteinander zu kommunizieren, oder basieren auf dem lernen gemeinsamer Aktionen, was beides mit erhöhter Komplexität verbinden ist. Analog zu Menschen, die andere Menschen zur Kooperation bewegen, indem sie einen Anreiz durch Belohnungen schaffen, untersuchen wir Methoden, die das Teilen von Belohnungen benutzen, um Aktionen zu fördern, die für das Erreichen eines gemeinsamen Ziels zuträglich sind. Zu diesem Zweck stellen wir eine Erweiterung eines bestehenden Algorithmus sowie einen neuen Ansatz vor und evaluieren beide in verschiedenen Umgebungen.

Contents

1. Introduction

Cooperation between intelligent systems plays a more and more important role as the world becomes increasingly connected and tasks can be solved by smart agents. From robots picking stock for orders in a warehouse to self-driving cars on the roads, a growing number of problems are no longer constrained to a single agent acting detached from the outside world but usually involve multiple agents interacting with and influencing each other. Additionally, in many environments where multiple agents play a role, they are faced with a common goal that they need to work towards together (e.g. minimizing the wait time of orders before they can be packed and shipped, or optimizing traffic flow while ensuring the safety of all vehicles), where it typically does not suffice to act selfishly.

Because of this, and the fact that reinforcement learning offers an extensible and well-studied framework for modeling such environments, *Multi-Agent Reinforcement Learning* (MARL), and especially cooperation in MARL is a growing field of research. While there are multiple approaches to training agents in a MARL setting to cooperate with each other, such as introducing communication channels [1] [2] [3], learning joint actions [4] [5] or supplying the agents with additional information [6] [7], in this thesis, we focus on investigating methods that share rewards between agents. This has the benefit that we can directly influence the objective that the agents are already optimizing and can use this signal as an implicit form of communication without having to learn a separate communication protocol.

Our two main contributions are to modify an existing reward-sharing method (LToS [8]) to improve its performance and make it more robust, by letting agents learn a representation of other agents' policies that they condition their action selection on (we call this extension *Policy-Encoded Reward Exchange*, or PERC), and our own novel method (*Bribes for Incentivizing Behavior*, or BRIBE) which uses a more principled way to modify the reward signal of the agents by approximating the expected advantage that another agent's action has on our reward compared to other actions they might take and rewarding actions which are advantageous.

In Chapter 2, we will first introduce the reinforcement learning and multi-agent reinforcement learning frameworks, as well as key concepts and algorithms needed for our methods. We then give an overview of related work in this field in Chapter 3 and explain LToS, the method that we extend, in detail. Chapter 4 then presents our two methods, explaining the motivation behind them and their architecture. Chapter 5 introduces different cooperative MARL environments and presents the results of our algorithms in these environments, which we discuss in Chapter 6, where we also give some pointers to future work in this field.

2. Background

In this chapter, we introduce the key concepts needed to understand this thesis. We start by examining the formal definition of a reinforcement learning problem in Section 2.1 and then extend it to the multi-agent setting in Section 2.2, where we also introduce some key challenges. Lastly, we discuss some common deep reinforcement learning algorithms which will be fundamental building blocks for MARL algorithms later on.

2.1. Reinforcement Learning

Reinforcement learning concerns itself with a class of problems in which an agent interacts with an environment by selecting an action from a predefined action space in each step, based on the state that the environment is currently in. The goal of the agent is to maximize the reward that it is given after each interaction with the environment. Because the agent does not know the underlying logic that hands out these rewards, it has to learn in which states to take which actions by repeatedly trying different strategies and adjusting to the rewards it receives.

2.1.1. Markov Decision Process

We will now more formally define the setting and goal of reinforcement learning algorithms. Environments can be modeled as a Markov Decision Processes (cf. [9]):

Definition 2.1.1 (Markov Decision Process)**.** A Markov Decision Process (MDP) is a tuple (S, A, P, R, I) where

- S is a set of states that the environment can be in,
- A is a set of actions that the agent can take,

- $P : S \times A \times S \to [0, 1]$ is the transition probability function that determines the probability $P(s, a, s')$ of transitioning into state s' when being in state s and taking action a,
- $R : S \times A \to \mathbb{R}$ is the reward function that returns a real-valued reward for each state-action pair, and
- $I : S \to [0, 1]$ is the initial state distribution, where $I(s)$ determines the probability that s is the initial state.

It obeys the Markov Property, stating that the evolution of the stochastic process is not dependent on its history because the transition probability function only depends on the current state and the current action taken, but not on previous states that the agent has been in.

An interaction between an agent and an environment can be depicted as follows: The environment starts in state s_0 drawn from the initial state distribution I. The agent then decides on an action $a_0 \in A$ to take, usually by drawing it from its learned policy $\pi : S \times A \to [0, 1]$. The environment then transitions into the next state s_1 by using its transition function P. The agent receives a reward r_0 that was determined by the reward function R and continues the process by choosing a new action in the new state according to its policy. This process is also illustrated in Fig. 2.1.

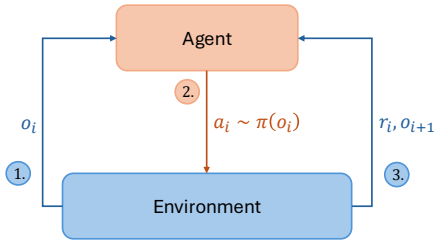


Figure 2.1.: Agent-Environment interaction in a MDP

2.1.2. Objective

This interaction between the agent and the environment is repeated until the environment reaches a terminal state. If there is no natural terminal state in the domain of the environment, one usually lets the interaction terminate after a fixed number of steps T. The interaction up until that point, which can be described by the list of states, actions, and rewards $s_0, a_0, r_0, s_1, a_1, r_1, ..., s_T, a_t, r_T$ is called an *episode*.

We can measure the quality of an episode τ by its *return*, which defines a measure of quality over an episode and is usually a weighted sum of the rewards:

$$G_0(\tau) = \sum_{t=0}^{T} \gamma^t r_t \tag{2.1}$$

Here, $\gamma \in [0, 1]$ is the discount factor and is usually close to 1. It determines how much influence rewards that lie in the future should have on the maximization objective. In the case of $\gamma = 1$, the return is called *undiscounted*. However, in most problems, it is natural to assume that we care more about immediate rewards than rewards in the future and it is also mathematically convenient to do so. Generally, we define $G_i(\tau)$ to be the return of the subsequence $\tau_i = s_i, a_i, r_i, ..., s_T, a_T, r_T$. Also note that in cases where we don't terminate an episode after a fixed number of steps, all definitions still work by replacing T with ∞. The goal of the agent usually is to learn a policy π that maximizes the expected return over all episodes

$$\max \mathbb{E}_\pi[G_0(\tau)] = \max \mathbb{E}_\pi[\sum_{t=0}^{T} \gamma^t r_t]. \tag{2.2}$$

The key idea behind a lot of reinforcement learning algorithms is to learn how "good" it is to be in certain states and then try to choose actions that lead to the best possible state in the next step. More formally, the "goodness" of a state is called its *value* $V^\pi(s)$ and is equal to the expected return that the agent gains when it starts from state s and follows policy π from then on:

$$V^\pi(s_i) = \mathbb{E}_\pi[\sum_{t=i}^{T} \gamma^t r_t | s_t = s_i]. \tag{2.3}$$

Notice that the value function is always defined with respect to a policy π that the agent follows, but if that policy is clear from the context, we sometimes omit it and write $V(s)$. Because it is sometimes useful to work with the value of state-action pairs instead of just states, we can define the "quality" of a state-action pair with respect to a policy by the *Q-function* $Q^\pi(s, a)$ which is equal to the expected return that the agent gets when it starts from state s by taking action a and then follows the policy π afterwards:

$$Q^\pi(s_i, a_i) = \mathbb{E}[\sum_{t=i}^{T} \gamma^t r_t | s_t = s_i, a_t = a_i]. \tag{2.4}$$

Because of the recursive property that the return follows ($G_t = \gamma^t + \gamma^{t+1} G_{t+1}$), both the value- and Q-function can also be defined recursively:

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(s),\ r, s' \sim P(s,a)}[r + \gamma V^{\pi}(s')]. \tag{2.5}$$

More importantly, we know that an optimal policy π^*, i.e. a policy that yields more or equally as much expected return in all states as any other policy, has to follow the *Bellman optimality equation*

$$V^{\pi^*}(s) := v_*(s) = \max_{a \in A} \mathbb{E}_{r, s' \sim P(s,a)}[r + \gamma v_*(s)]. \tag{2.6}$$

This equation is important because it can easily be turned into an update rule

$$V(s) \leftarrow max_a \mathbb{E}_{r, s' \sim P(s,a)}[r + \gamma V(s')], \tag{2.7}$$

where the expectation on the right-hand side can be approximated by one or multiple samples in practice. This update rule is at the core of many value-based reinforcement learning algorithms and also plays a key role in Deep Q-Learning, which we will introduce in Section 2.3.1.

An important extension to the MDP defined in this section are environments in which the agent cannot observe the entire current state. In these *partially observable MDPs* (POMDP), the agent observes an observation of the current state $o(s_i) = o_i \in O$, which can be any representation of the state that can be expressed as a function of it, for example, a feature vector or a local region centered around the current position of the agent. The definitions introduced in this section, e.g. the definition of the agent's policy, are still valid if one replaces the state s with the corresponding observation $o(s)$.

2.2. Multi-Agent Reinforcement Learning

The definitions and formalisms we have defined so far have been developed under the assumption that only a single agent interacts with the environment. However, a variety of problems involve multiple decision-making entities, such as players in a game, robots in a warehouse, or network nodes in a packet routing process. Although we could technically model multiple entities as one agent and reduce the problem to a single-agent reinforcement learning problem, we will later see that this is not efficient and introduces unwanted assumptions. Therefore, we will introduce stochastic games, which we can use to model multi-agent environments (cf. [10]):

Definition 2.2.1 (Stochastic game)**.** A stochastic game for n agents is a tuple (S, A, P, R, I) where

- S is a set of states that the environment can be in,
- $A = \{A_1, ..., A_n\}$, with A_i being the set of actions agent i can be in.
- $P : S \times \mathbf{A} \times S \to [0, 1]$, where $\mathbf{A} = A_1 \times ... \times A_n$, is the transition probability function that determines the probability $P(s, \mathbf{a}, s')$ of transitioning into state s' when being in state s and the agents having taken the joint action $[a_1, ..., a_n]$,
- $R = \{R_1, ..., R_n\}$, where $R_i : S \times A \to \mathbb{R}$ is the reward function for agent i, and
- $I : S \to [0, 1]$ is the initial state distribution.

Here, we can also restrict agents to only be able to make decisions based on observations of the state, not the entire state itself. Now, each agent gets a different observation of the current state $o_i(s_i^t) = o_i^t$. Since it is common to restrict the agents' knowledge about the environment in MARL settings, we will continue our notation including this extension.

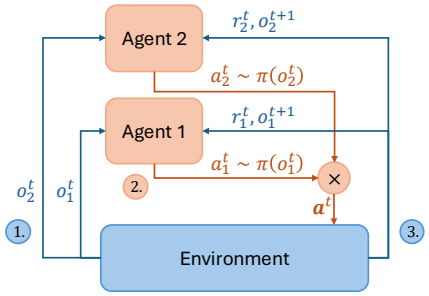


Figure 2.2.: Agent-Environment interaction in MARL

The dynamics of the game are similar to MDPs described for single-agent environments in Section 2.1, with every agent choosing an action $a_i \in A_i$ to take and the environment transitioning into a new state s_1 by drawing it from the transition probability function $P(s^1|s^0, [a_1^0, ..., a_n^0])$ which depends on the actions of all agents $\mathbf{a}$, also called the joint action. Each agent i then receives a reward r_i^0 that was determined by the reward function R_i. Fig. 2.2 shows this process. When dealing with multiple agents, we add the agent's index as the subscript to the definitions in Section 2.1.2 and denote the timestep as the superscript if necessary (e.g. $G_a^t(\tau)$ is the return of agent a starting at timestep t).

As this definition is quite broad, there are a lot of different environments and situations that can be modeled as a stochastic game [11]. One can model games in which agents act as adversaries to each other and fight against each other for reward or model agents as

belonging to one of multiple groups that have different motives. In this thesis though, we will focus on settings in which agents work towards a shared goal and need to learn to cooperate.

2.2.1. Goals & Challenges

To be able to develop adequate algorithms to solve multi-agent reinforcement problems, one first needs to define the goal of the desired algorithm (i.e. the property that the solution should fulfill). Common solution concepts are (cf. [10]):

- **Nash equilibrium:** For all agents i, given the set of fixed policies of all other agents, the policy of agent i should maximize the expected return for that agent. This comes from a game theory background. However, there can be many sets of policies that fulfill the Nash equilibrium criterion with different expected returns, and therefore agents can converge to suboptimal Nash equilibria.
- **Pareto optimality [12]:** In addition to being in a Nash equilibrium, this criterion demands that there is no other policy that yields a higher expected reward for one agent while not decreasing the expected reward for all other agents. This tries to account for the aforementioned problem with Nash equilibria.
- **Fairness [13]:** A policy is fairness-optimal if it maximizes the product of all agents' expected returns.
- **Welfare [14]**: A policy is welfare-optimal if it maximizes the sum of all agents' expected returns.

In this thesis, we will try to learn policies that maximize the agents' welfare. This is reasonable as we consider environments in which agents work towards a common goal whose success is measured by the reward the environment hands out to any agent.

As mentioned previously, it is possible to model a MARL problem as a single-agent problem. This can be done in two ways:

(1) One can introduce a super-agent who takes in the observations from all agents $\mathbf{o} = \{o_1^t, ..., o_n^t\}$, decides on a joint action $\mathbf{a} = \{a_1^t, ..., a_n^t\} \in \mathbf{A}$ and then receives some

function of the agents' rewards (i.e. the sum of them)[1]. From the outside, i.e. from the perspective of the environment, we are now only dealing with one agent and can apply all algorithms and formalisms we know from single-agent RL. The main drawback is that this super-agent's action space is now exponentially bigger than that of any single agent of the original problem (assuming that the action spaces of all agents are equally large). Another downside is that we cannot let the agents execute their policies separately from each other, even after training is finished, because they have to be controlled by the super-agent. This is a limitation that is problematic in situations where agents are not able to communicate with each other or the outside world after they are deployed.

(2) Another way to reduce the MARL problem to a single-agent problem is to train each agent independently from the others (cf. [15]). This does not have the drawbacks of exploding action space or limited application to certain environments as the centralized-control approach. Breaking down a multi-agent environment in multiple single-agent environments does however break the Markov property. From the perspective of a single agent i the next state should only depend on the current state, the agent's action, and the transition probability function of the environment, which should not change over time (i.e. $P(s'|s,a)$ should always be the same probability distribution during the training process). In reality, the environment transition probability function depends on all agents' actions, and therefore on their policies, which change over time as they are being learned. From a single agent's perspective, $P(s'|s,a)$ thus seems to change change over time. This problem is called *non-stationarity* and is the reason why most of the convergence guarantees from single-agent RL do not translate to independent MARL and what makes it challenging to develop universal algorithms in this field.

When developing custom MARL algorithms that don't reduce to a single-agent problem in one of the ways described above, one can categorize different approaches in training and execution based on what information they are allowed to have access to (cf. [10]):

- **Centralized Training, Centralized Execution**: The algorithm uses privileged information during training time such as other agents' policies or observations, and also relies on this information during execution. This can help make training more stable and increase performance, however might not be applicable in some environments as sharing information between agents after deployment might not always be possible.

[1] It might also be challenging to find a suitable function for some environments, e.g. when the sum of all agents rewards is always 0.

- **Decentralized Training, Decentralized Execution**: Agents don't rely on centralized or shared information at all. This is the most general approach as it works with all environments but might suffer from stationarity problems.
- **Centralized Training, Decentralized Execution**: This approach combines the advantages of both methods above. As long as agents don't rely on shared information when choosing their next action, there is no downside to using this information during training.

The algorithms we present in this thesis will mainly fall into the third category, although we will make a well-grounded exception when it comes to sharing information between the agents during execution time.

2.3. Deep Reinforcement Learning Algorithms

So far, we have only hinted at a concrete reinforcement learning algorithm by introducing the *Bellman Optimality Equation* (Eq. (2.6)) and transforming it into an update rule to iteratively learn the value of different states. In this section, we will introduce two classic deep reinforcement learning algorithms that build on this idea and leverage the power of neural networks as universal function approximators [16]. These algorithms can be used in single-agent RL and will also be fundamental building blocks in the MARL algorithms we will present in Chapter 4.

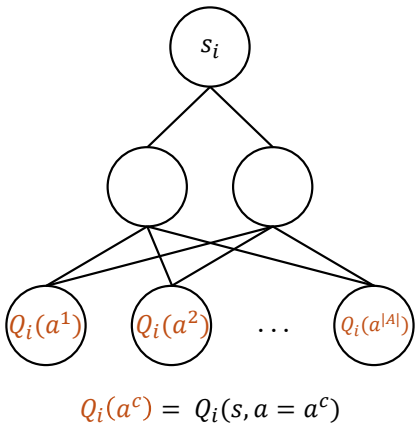


Figure 2.3.: The Q-network of agent i, taking in the current state and predicting Q-values for all different actions.

2.3.1. Deep Q-Networks (DQN)

The approach of Q-learning is for the agent to learn the quality of state-action pairs (i.e. the Q-function), so that, during execution time, it can follow the policy of choosing

Algorithm 1 Deep Q-networks (DQN)

Initialize Q-network Q with parameters θ
Initialize target network Q' with parameters $\theta' \leftarrow \theta$
Initialize replay buffer $R = \{\}$
for $t = 0, \ldots$ **do**
  Observe s_t, choose action with ϵ-greedy policy
  Store sample (s_t, a_t, r_t, s_{t+1}) in replay buffer, reset environment if s_{t+1} is terminal
  Draw minibatch $\mathcal{D} = \{(s_i, a_i, r_i, s_i')\}$ of b samples from replay buffer R
  if s_i' is terminal **then**
    Set target $y_i \leftarrow r_i$
  else
    Set target $y_i \leftarrow r_i + \gamma \max_a Q'(s_i', a)$
  end if
  Update θ by minimizing $\mathcal{L}(\theta) = \frac{1}{b} \sum_{k=1}^{b} (y^k - Q(s^k, a^k))$
  if $t \bmod$ update frequency $= 0$ **then**
    Update target network: $\theta' \leftarrow \theta$
  end if
end for

the action that has maximal Q-value for the current state. Deep Q-Networks [17] do this by training a neural network to approximate the Q-function $Q^\pi(s, a)$. The neural network gets the observation at the current timestep o_t as input and outputs a real-valued number for every action $a \in A$. An illustration of this network is shown in Fig. 2.3. Note that action space must be discrete and finite to be able to approximate Q-values in this way.

The training process consists of two steps:

(1) Collecting samples: This is done by letting the agent interact with the environment. Usually, the agent follows an ϵ-greedy policy, meaning it chooses a random action with probability ϵ and otherwise chooses the action that maximizes the current approximation of the Q-function. This policy realizes the tradeoff between exploration and exploitation and the parameter ϵ will usually decrease as the algorithm is being trained to exploit good states after the state space has been explored sufficiently.
The samples (s, a, r, s') are then stored in a replay buffer, which holds the k newest samples. The benefit of sampling batches from this buffer when training the network is that one can reuse old samples improving sample efficiency and can do batchwise updates.

(2) Training the network: We sample a batch of interactions from the replay buffer. All

the following calculations can be done batch-wise: First, the target y is calculated following the equation Eq. (2.7) that we have derived from the Bellman optimality equation (where the Q-value of the next state s' is omitted if the state is terminal). We then update the network by taking a step in the gradient direction of the loss which we calculate to be the error between the predicted Q-value and the target y, averaged over all samples in the batch.
An important thing to note is that we use a separate *target network* to compute the target, which has the same architecture as the Q-network and is not trained, but only updated with the parameters of the Q-network once every fixed number of steps. This helps combat the *moving target problem* where we would update the same network that we use for calculating our target if we were to not use two separate networks [18]. Algorithm 1 shows pseudocode for this algorithm.

Although DQN is a widely-used algorithm in RL, both standalone and as a building block for more complex algorithms, it has the drawback that it cannot be used in environments with discrete action spaces. This is because we are computing the target y using the maximum Q-value in the next state (cf. line 11), which we can only do if there is a finite amount of states.

2.3.2. Deep Deterministic Policy Gradient (DDPG)

An approach that aims to mitigate this problem is the family of policy gradient algorithms. The idea is to train a policy network μ directly, which takes in the current state and outputs a mean action in the continuous action space. A stochastic policy can then be used to handle the tradeoff between exploration and exploitation based on this mean action. This has the benefit that the policy is more flexible than an ϵ-greedy policy over a Q-network (it can approximate any probability distribution), but most importantly we can train this policy network directly. There are many variants of policy gradient algorithms, but the idea behind all of them is to update the policy network by a gradient that makes action (-sequences) with high returns more likely and ones with low returns less likely. The simplest algorithm *REINFORCE* minimizes the loss

$$\mathcal{L}(\phi) = -\frac{1}{T}\sum_{t=0}^{T}(\sum_{i=0}^{T-1}\gamma^{i-t}r_i)\log\pi(a_t|s_t) \tag{2.8}$$

which is calculated from a whole episode.

Deep Deterministic Policy Gradient (DDPG) combines the approach of learning a Q-function approximation from DQN and uses it to update the policy network by calculating a bootstrapped gradient. Pseudocode for this algorithm can be found in Algorithm 2.

Additionally to the Q-network, we now also have a network μ that takes in the current state s and outputs a mean action a. Collecting samples is done in the same way as in DQN, only that we use our policy network for choosing the actions. Here, we usually apply a noise function to realize the trade-off between exploration and exploitation. The original paper [19] uses additive Gaussian noise, but other options include time-correlated noise processes such as the Ornstein-Uhlenbeck process. We also train the Q-network in the same way as in 2.3.1, with the small adjustment that we don't use the action from the replay buffer but recompute what action we would take under our current policy (cf. line 11). This is necessary because the original policy gradient theorem assumes that the expected return is calculated under the current policy and samples from the replay buffer might already be outdated.

After having trained the Q-network, we update the parameters of the policy network by taking a step in the direction of the gradient that maximizes the Q-network. Instead of doing "hard" target updates (setting the parameters of the target networks to the current parameters of the network every fixed number of steps), DDPG updates the targets "softly". This is done by updating them in every step, but only by a weighted sum of the current weights and the target weights (with weighting parameter τ) in order to improve stability during training.

Algorithm 2 Deep Deterministic Policy Gradient (DDPG)

Initialize policy network μ with parameters ϕ, Q-network Q with parameters θ
Initialize target networks μ', Q' with parameters $\phi' \leftarrow \phi, \theta' \leftarrow \theta$
Initialize replay buffer $R = \{\}$
for $t = 0, \dots$ **do**
  Observe s_t, choose action with exploration noise $a = \mu(s) + \epsilon$
  Store sample (s_t, a_t, r_t, s_{t+1}) in replay buffer, reset environment if s_{t+1} is terminal
  Draw minibatch $\mathcal{D} = \{(s_i, a_i, r_i, s'_i)\}$ of b samples from replay buffer R
  if s'_i is terminal **then**
    Set target $y_i \leftarrow r_i$
  else
    Set target $y_i \leftarrow r_i + \gamma \max_a Q'(s'_i, \phi(s'_i))$
  end if
  Update θ by minimizing $\mathcal{L}(\theta) = \frac{1}{b} \sum_{k=1}^{b} (y^k - Q(s^k, a^k))$
  Update ϕ by maximizing $\frac{1}{b} \sum_{k=1}^{b} Q(s^k, \phi(s^k))$
  Update target networks softly: $\phi' \leftarrow \tau\phi' + (1-\tau)\phi$ and $\theta' \leftarrow \tau\theta' + (1-\tau)\theta$
end for

3. Related Work

In this chapter, we will first give an overview of different approaches to solving the cooperate MARL problem and then focus in on methods that reshape the reward signal in order to incentivize cooperation. We will then present some approaches to model the policy of other agents, which can help better predict the evolution of the environment and thus improve learning performance. We will use this technique in one of our methods as well, which we describe in more detail in section Chapter 4. Lastly, since one of our contributions is to extend the algorithm *Learning To Share* (LToS) presented in [8], we introduce this method in detail.

3.1. Cooperative MARL

There have been numerous different approaches to solving the problem of cooperative MARL. Foerster et al. [7] use an actor-critic method with decentralized actors and a centralized critic which is trying to solve the multi-agent credit assignment problem (i.e. the problem of determining which agent was responsible for how much of the attained reward) by rewarding the agent based on the advantage of the taken action over a baseline. The disadvantage here is that the method still requires a central critic whose complexity scales with the number of agents (and actions) in the environment.

Sunehag et al. [20] try to decompose the joint value function into multiple agent-specific value functions that sum to the global one. QMIX from Rashid et al. [21] improves on this idea by realizing that the constraint that the local value function must sum to the global one at all points can be relaxed to only holding in the arg max case. This makes them able to represent a larger class of value functions. Wang et al. [22] introduce QPLEX, splitting the Q-function approximation into a value network and an advantage network, which are both decomposed for each agent. Variations of this also don't decompose the value function for each agent individually, but build factors. For these kinds of value

decomposition, it can however be hard to find the combination of factors that yields the best performance, and it is especially hard to do this dynamically.

Zhang et al. [23] also use a centralized critic, which is updated by the consensus of all individual updates. Qu et al. [24] propose intention propagation between agents, updating each agent's policy based on the intentions shared by other agents. This can however converge slowly because the intentions have to be propagated through the network of agents.

Chu et al. [25] devise a communication protocol where agents exchange both their current observations and beliefs, as well as a fingerprint. This communication is however costly and provides a significant overhead in computation.

3.2. Reward Shaping

The methods we presented above mostly tackled the MARL problem by adding means of communication between agents or tried to bridge the gap between joint learning and independent learning by supplying the networks with additional information. There is also related work that takes the approach of modifying the agents' rewards in order to promote cooperation:

Peysakhovich et al. [26] have limited their research to two-player stag hunt games, examining how agents with different prosociality (receiving some convex combination of the individual rewards) play against each other. However, the prosociality values (similar to our sharing weights) are fixed and hand-tuned. Jaques et al. [27] give out additional rewards to agents who choose actions that cause a high change in the probabilities of other agents' actions, maximizing mutual information and also add a message channel where messages are rewarded in the same manner.

Mguni et al. [28] also modify the reward function, but use black-box optimization that is trying to learn the optimal way to give out additional rewards given some fixed budget of reward. These however stay the same during an entire episode and can thus not react to dynamic environments and have no guarantees, as the modification of the reward signal is unprincipled. When dealing with environments that give out only a global reward, Wang et al. [29] redistribute this reward by using the idea of the Shapley value, which measures each player's marginal contribution in all possible coalitions. Yang et al. [30] learn an incentive function to give additional rewards to other agents.

Hostallero et al. [31] change the reward function of the stochastic game iteratively by evaluating the current performance and employing a kind of temporal difference error. However, because of this iterative process, it can take time to converge, and there are situations where convergence is not guaranteed. In the method from Kölle et al. [32], agents hold shares of other agents, based on which they receive parts of their reward and they can decide to acquire more shares or reduce their number of shares in each step. Their experiment setups and initial share distribution are however still hand-crafted.

Lupu et al. [33] also investigate the idea of letting agents give out additional rewards to other agents (which they call gifting). Agents can gift rewards from a finite pool, but this is not chosen in a principled manner, and the gifted reward goes to *all* other agents equally, making it impossible to incentivize agents differently in more complex scenarios. Wang et al. [34] extend this idea and analyze it theoretically.

3.3. Agent Modeling

The idea of modeling other agent's behavior to better predict how the environment is gonna evolve has also been studied in a few papers: Lowe et al. [6] introduce Multi-Agent DDPG (MADDPG), where the critic is augmented with information about other agents' policies.

He et al. [35] indirectly model opponents by extending the Q-function with opponent features. Jin et al. [36] try to learn what action other agents will take by training a function that takes in two environment states and tries to predict the action taken to transfer the environment from one state to another. Because they are however always one step behind, as they can only know the next state of the environment after the current action has been taken, the performance varies.

Foerster et al. [37] take a different approach and refrain from actually modeling other agents based on their behavior. Instead, they additionally condition the agent's policy on a fingerprint that contains information about at which point on their learning trajectory they currently are. This can be for example the number of training iterations or the value of the exploration parameter. The goal is to mitigate the problem that other agents' policies change as they are being trained. This is however also an advantage we get for free when we model another agent's behavior (although with more computational cost).

3.4. Learning to Share (LToS)

The underlying idea of reward sharing as it is done in LToS is that agents can decide to share parts of the reward they receive with other agents on a per-step basis. This leads the agent receiving the shared reward to have an incentive to select its action in a way that is also beneficial to the agent that shares its reward, as this will directly influence its own reward. This assumes that the actions of agents have an effect on the reward of other agents, which is given in most MARL environments where cooperation is needed, especially between agents that are spatially close to each other. The fraction being shared (also called the *sharing weight*) is communicated to each agent before they choose the action they want to execute so that agents can base their choice of action on how the rewards are gonna be redistributed in this step.

The further goal of LToS is that agents do not just incentivize all other agents to cooperate with them equally and at all times, but rather that agents can choose who and how much they want to incentivize to cooperate at each step, dependent on the current state of the environment. This makes the method flexible and not environment-dependent and should make it possible for agents to learn the optimal sharing behavior without any need for hand-crafted communication.

The LToS approach formulates this setting as a bi-level optimization problem, where the *high-level* problem is to optimize the way to choose the sharing weights (which we will call the *high-level policy*), given the current policies of all agents (which we call the *low-level policy*) and the low-level problem is to optimize the way to choose an action in the environment given the current high-level policies of all agents. We will now describe how this problem is solved by in turn updating the low-level and high-level policy. A diagram of the interaction between the different policies and agents can be found in Fig. 3.1 and pseudocode for the training process is shown in Algorithm 3.

Each agent has two components, an actor-network ϕ that chooses the high-level action (i.e. the sharing weights) and a Q-network Q that tries to approximate the quality of the triple (state, high-level action, low-level action) $Q(s, w^{in}, a)$. The Q-network is used to sample the low-level action, as well as being the critic for the actor ϕ.

(1) Collecting Samples: The agents' interaction with the environment is now split into two steps. First, each agent k chooses the sharing weights w_k^{out} by following a noisy policy over the actor ϕ (we use noise generated by an Ornstein-Uhlenbeck process). The vector w_k^{out} has n entries (w_{kj}^{out} denotes the fraction of reward agent j will get from agent k)

Algorithm 3 Learning to Share (LToS)

Initialize high-level network ϕ_k with parameters θ_k and target network ϕ'_k with parameters θ'_k for every agent k
Initialize low-level network Q_k with parameters μ_k and target network Q'_k with parameters μ'_k for every agent k
Initialize replay buffer $R = \{\}$
for $t = 0, \ldots$ **do**
  for each agent k **do**
    Observe s_k^t, choose high-level action $w_k^{out,t} \leftarrow \phi_k(s_k^t)$ with noisy policy
    Exchange $w_k^{out,t}$ to get $w_k^{in,t}$
    Choose low-level action a_k^t with ϵ-greedy policy from $Q(s_k^t, w_k^{in,t}, a)$
    Store sample $(s_k^t, w_k^{out,t}, a_k^t, r_k^t, s_k^{t+1})$ in replay buffer, reset environment if s_k^{t+1} is terminal
  end for
  for each agent k **do**
    Draw minibatch $\mathcal{D} = \{(s_k^i, w_k^{out,i} a_k^i, r_k^i, s_k'^i)\}$ of b samples from replay buffer R
    Draw next high-level action $w_k'^{out} \leftarrow \phi'_k(s'_k)$ and exchange to get w'^{in}
    if s'_k is terminal **then**
      Set target $y_k \leftarrow r_k$
    else
      Set target $y_k \leftarrow r_k + \gamma \max_a Q'_k(s'_k, w_k^{in'}, a)$
    end if
    Update θ_k by minimizing $\mathcal{L}(\theta) = \frac{1}{b} \sum_{i=1}^{b} (y_k^i - Q_k(s_k^i, w_k^{in,i}, a_k^i))$
    Redraw $w_k^{out} \leftarrow \phi_k(s_k)$ and exchange to get w_k^{in}
    Compute $g_k^{in} = \nabla_{w_k^{in}} \arg\max_a Q_k(s_k, w_k^{in}, a)$ and exchange to get g_k^{out}
    Update μ_k by maximizing $\frac{1}{b} \sum_{i=1}^{b} (\nabla_{\theta_k} \phi_k(s_k^i))^T g_k^{out,i}$
    Update target networks softly
  end for
end for

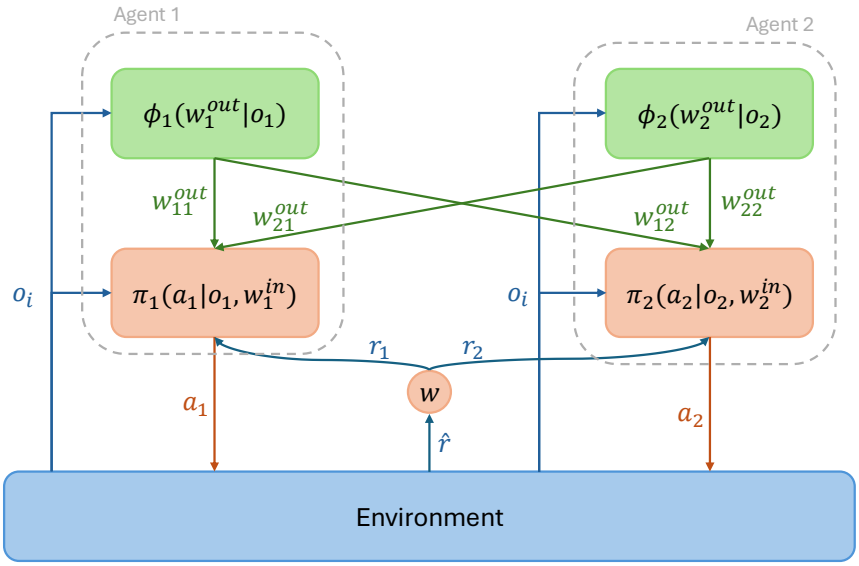


Figure 3.1.: The LToS method from [8] in an environment with two agents

which are between 0 and 1 and all entries sum to 1, as all of the reward that agent k gets should be redistributed (to other agents or to agent k itself). We make sure that this is the case by applying a softmax over the output of the last layer of the network and additionally renormalizing the sharing weights after the noise policy has been applied. The agents then exchange the sharing weights and each agent receives a vector $w_k^{in} = \begin{pmatrix} w_{1k}^{out} & \dots & w_{nk}^{out} \end{pmatrix}$ of sharing weights directed to them.

Second, the agents choose a low-level action a_k by following an ϵ-greedy policy over $Q_k(s, w_k^{in}, a)$. Note that this is only possible if the action space of the environment is discrete (and finite). It would also be possible to extend this method to continuous action spaces by employing a similar strategy to DDPG, but we only concern ourselves with discrete action spaces in this thesis. Lastly, the agents execute a step in the environment and each receive a reward, which is calculated based on the base rewards of the environment $\hat{r}_i^t$ and the sharing weights and is calculated by

$$r_i^t = \sum_{j=1}^{n} w_{ji}^{out,t} \hat{r}_j^t. \tag{3.1}$$

(2) Training: We again train on a minibatch sampled from the replay buffer where all calculations can be done batch-wise. First, the low-level Q-network is trained, similar to the process in DQN. The only difference is that in order to compute the bellman target, we need to draw the high-level actions in the next state s' (and exchange the sharing weights between the agents) because we need both the next state and the next high-level actions to determine the next maximum Q-value. After we have updated the parameters of the Q-network, we only need to update the parameters of actor ϕ by taking a step in the direction of the gradient of the Q-function with respect to its parameters θ.

To this end, we can see that the gradient with respect to the high-level parameters of agent k of the Q-value of agent j with fixed state and action is

$$\begin{aligned} G_{kj} &= \nabla_{\phi_k} \max_a Q_j(s, w_k^{in}, a) \\ &= \nabla_s \max_a Q_j(s, w_k^{in}, a) \nabla_\theta s \\ &\quad + \nabla_{w_k^{in}} \max_a Q_j(s, w_k^{in}, a) \nabla_\theta \phi_\theta(s) \\ &\quad + \nabla_a \max_a Q_j(s, w_k^{in}, a) \nabla_\theta a \\ &\approx \nabla_{w_k^{in}} \max_a Q_j(s, w_k^{in}, a) \nabla_\theta \phi_\theta(s). \end{aligned} \tag{3.2}$$

Because the sharing weights of all agents influence the Q-value, we need to sum over all gradients G_{kj} when updating the parameters of agent k:

$$\theta_k \leftarrow \theta_k + \alpha \sum_{i=1}^{n} G_{ki}. \tag{3.3}$$

Because we need gradient information from all agents in order to update a single agent's parameters, we need to exchange this information (in the pseudocode and actual implementation we exchange $g_{kj}^{out} = \nabla_{w_k^{in}} \max_a Q_j(s, w_k^{in}, a)$ instead of G_{kj} and multiply them with $\nabla_\theta \phi_\theta(s)$ afterward in line 24). Lastly, we update both the high-level and low-level target networks softly, which are being used in calculating the next Q-value in the calculation of the bellman target.

4. Methods

In this chapter, we introduce the two MARL algorithms we have developed for training agents in a stochastic game to maximize the total welfare. As previously mentioned, there are multiple different approaches to making agents learn to cooperate. We want to confine ourselves to methods that can be executed decentralized to keep the range of applicable problem settings broad. We don't allow access to the joint action of all agents or other privileged information during execution that would break the scalability of the method. However, in many MARL algorithms, a communication channel either between all agents or between agents that share some form of locality is allowed to aid communication.

Communication can be modeled as being part of the agent's action, however not influencing the environment state. Communication actions (or *messages*) can be of different forms: They can be selected from a set of possible messages, which can either have a predefined meaning assigned to them that the agents know about or agents might need to learn a common language. They can also be continuous or even noisy if one models the communication channel as unreliable. For our first method, PERC (Policy-Encoded Reward-Communication), we allow a reliable communication channel between local agents with continuous actions.

Many other methods that use a communication channel between agents require the agents to learn a common language (and to use that language properly) to develop a helpful messaging system or share hand-crafted information with each other such as properties of the current state. This has the downside of being quite dependent on the messaging system being developed by the agents or the helpfulness of environment features being communicated.

Our methods try to circumvent this problem by using rewards as a means of communication. As the agents are already trying to maximize their own rewards, it is natural to let other agents modify this signal in order to influence the behavior. Our methods are thus general enough to work with any environment and just reshape the reward function each agent tries to maximize.

We present two methods in this chapter:

PERC (Policy-Encoded Reward Sharing): Our first method, which we will present in Section 4.1 is an extension to the LToS (Learning to Share) framework proposed in [8], which lets agents dynamically share parts of their reward with other agents, which they communicate to them.

BRIBE (Bribes for Incentivizing Behavior): In Section 4.2, we will describe our second method, in which agents can specify an additional reward (which we call *bribe*) that is given to another agent if it takes the action specified by the bribing agent.

4.1. Policy-Encoded Reward-Exchange (PERC)

In our method PERC, we extend LToS in order to improve two main aspects: Firstly, because the high-level network draws the sharing weights w_k^{out} in one forward pass as a single vector, the high-level network needs to learn which agent corresponds to which index in the vector. If the environment changes (i.e. agents disappear, new agents get added or agents change indices in some other way), this relationship gets lost and has to be learned anew.

Secondly, we hypothesize that it would be beneficial if the agent had more information about another agent's policy when deciding on the sharing weights. The intuition behind this is that in order to decide what fraction of its reward an agent should share with another agent, it should determine whether and in which way cooperation with that agent is needed in the next step. Other papers that we have presented in Section 3.3 have shown that conditioning policies on models of other agents can have a significant impact on performance. To that end, it would be beneficial to condition the high-level policy on a representation of the other agent's policy (which ideally would hold information about the next action that the agent takes).

In the original LToS method, this information about other agents' policies has to be implicitly learned from experience samples. This is an extra layer of complexity and is especially a problem in environments where agents' policies change during training. In those cases, it can be beneficial to explicitly condition the agent's policy on the policies of all other agents in order to be able to use up-to-date information about the behavior of other agents.

In an ideal world, we would like to use the actual actions that all other agents are going to take in this step, but this is mostly not feasible as we don't have access to all actions in the decentralized execution paradigm after deployment. Furthermore, we would introduce a circular dependency because the action of one agent would depend on the actions of all other agents. Another approach would be to use the *policy* of all other agents. This has the downside that in deep reinforcement learning, policies are usually represented by neural networks, i.e. by a large number of parameters, typically too large to effectively condition policies on efficiently.

This leads us to the idea of learning a compact representation of the agents' policies that is small enough to be able to be used efficiently when drawing actions, but large enough to capture the policies' characteristics sufficiently. Encoder-decoder networks have been found to work well with learning compact representations of larger data and then making predictions from them. Because we want to predict an agent's next action a_i from the current state s (as this is the only information available to us), we train an encoder e_i and a decoder d_i for every agent. The encoder receives the current state and outputs an intermediate representation (or *latent state*) $e_i(s) = l_i$, where the dimensionality of l_i is sufficiently small. The decoder then takes in the latent state and outputs a vector, which we aim to approximate the action probabilities of agent i: $d_s(l_i) = \hat{a}_i = \begin{pmatrix} \hat{a}_i^1 & \dots & \hat{a}_i^{|A|} \end{pmatrix}$, where we want to train the encoder-decoder network, such that $\hat{a}_i^k$ approximates the probability with which agent i takes action k in the current step. We achieve this by training the encoder and decoder jointly each time a training step in LToS is done by minimizing the cross-entropy loss between the predicted action probability of the network and the true action that the agent has taken which we observe from the sample:

$$\min \mathcal{L}(\hat{a}_i, a_i) = \log(\frac{\exp(\hat{a}_i^{a_i})}{\sum_{k=1}^{|A|} \exp(\hat{a}_i^k)}). \tag{4.1}$$

This way, the encoder learns to encode information into l_i which is important for predicting the action probabilities of agent i and the decoder learns to use that information correctly.

When we now draw a high-level action (the sharing weights), we could give the network both the state s and the encodings for all other agents $l_{-i} = \{l_1, \dots, l_{i-1}, l_{i+1}, \dots, l_n\}$ or alternatively choose to directly condition on the predicted action probabilities $\hat{a}_{-i}$ instead of the encodings. The advantage of using the predicted action probabilities directly is that we make the information more explicit, whereas the idea behind using the encodings is that the encoder-decoder has hopefully learned to structure this information in a more usable way during the training process. We will investigate the difference between these two ideas in Chapter 5.

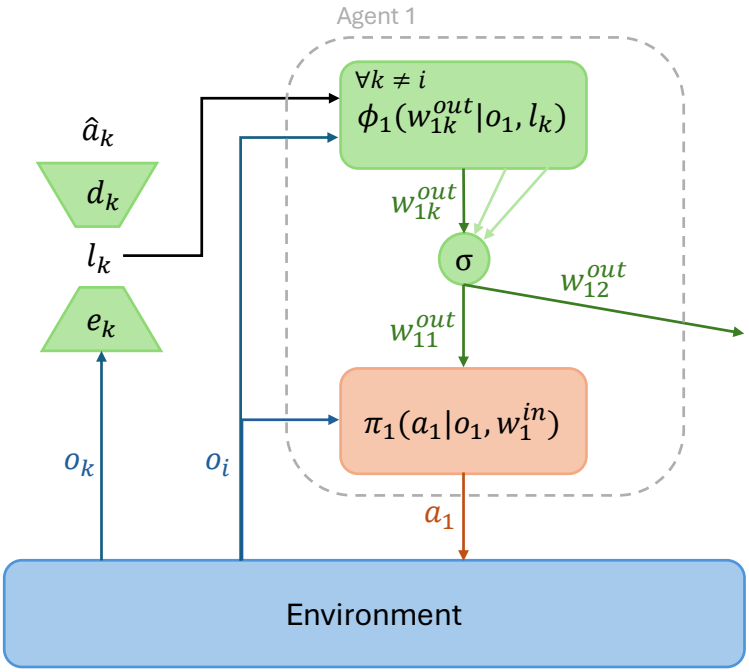


Figure 4.1.: A PERC agent's interaction with the environment, using information about other agents' policies.

However, conditioning agent i's policy on *all* other agents' policies when drawing all sharing weights in one single pass has two main drawbacks. Firstly, since we draw all sharing weights in one pass, the high-level agent has to learn which encoding it should pay attention to when computing the sharing weight for a specific agent. Conversely, when deciding on the sharing weight w_{ij}^{out}, all encodings except for l_j should not be of much value. So, we unnecessarily blow up the input space of the high-level policy and make it harder to learn the relationships between the policy encodings. Secondly, drawing the weights in one pass would carry over the problem of LToS that the architecture is not flexible towards changes in the environment due to tying agents to indices in the sharing weight vector.

To circumvent these problems, we change the high-level agent's network such that it takes in the state s and *one* policy-encoding l_j and only outputs *a single* sharing weight w_{ij}^{out}. We then determine the sharing weights by separately drawing the individual sharing weights and then applying a softmax (which we now cannot have as the last layer of the network anymore):

$$w_i^{out} = \sigma\left(\begin{pmatrix}\phi_i(s, l_1)\\ \vdots \\ \phi(s, l_n)\end{pmatrix}\right). \tag{4.2}$$

Fig. 4.1 shows a diagram of this process. The full pseudocode for PERC can be found in Appendix A.

The modifications compared to LToS are listed below:

Algorithm 4 Modifications to LToS for PERC

Initialize encoder e_k and decoder d_k with parameters ψ_k^d and ψ_k^d for every agent k
...
Observe s_k^t, draw encodings l_j and predicted action $\hat{a}_j^t$ for every agent j, choose high-level action $w_k^{out,t} \leftarrow \sigma(\phi_k(s_k^t, l_1), \ldots, \phi_k(s_k^t, l_n))$ with noisy policy
...
Update ψ_k^d and ψ_k^d jointly by minimizing $\frac{1}{b}\sum_{i=1}^{b}(log(\frac{\exp(\hat{a}_i^{a_k^i})}{\sum_{c=1}^{|A|}\exp(\hat{a}_i^c)}))$

4.2. Bribes for Incentivizing Behavior (BRIBE)

Most cooperative MARL methods that follow the idea of sharing rewards between agents or influencing the reward of other agents in some other way (including LToS & PERC) influence the actions of other agents only indirectly through a modified reward signal. This reward signal can however be unreliable at times. For example, even if agent i chooses an action that is beneficial to agent j, it might not get a high reward signal, because agent j itself chose a suboptimal action (especially during the start of training time when the exploration parameter of the policy is still high) or because either some other agent or the stochastic environment transition led to a suboptimal state by chance.

Additionally, some reward-shaping methods like LToS give agents a lot of freedom in how to share/modify rewards. Because the reward landscape from the point of view of a single agent depends so much on the modification that other agents make, its policy can show a high variance depending on what the current reward signal looks like. This added layer of dimensionality to the original reinforcement learning problem can lead to more local optima or cases where agents don't converge to a coordinated way of modifying each other's reward signal.

This is why we developed a method that creates an explicit way for agents to suggest to other agents which action they should take and give them an incentive to do so. Additionally, both the suggested action and the incentive are chosen in a principled way based on the current learned state values.

The idea behind *Bribes for Incentivizing Behavior (BRIBE)* is that each agent i, additionally to selecting an action a_i, selects an action for each other agent j that it wants this agent to take. We call this action b_{ij} the *bribe action*. Additionally, it attaches a reward to this action, which agent j will receive if $a_j = b_{ij}$. We call this additional reward the *bribe value* v_{ij}. This is the only modification compared to the standard MARL setting, i.e. the agents' policies are still only conditioned on the current observation, not on the bribe actions and bribe values of other agents. This is because we want to avoid additional complexity when learning the policies and additional overhead that this communication step would entail. We suppose that solely reshaping the reward function of the agents in this way - encouraging single actions explicitly - will suffice to learn to cooperate.

As explained above, we want to choose the bribe actions in a principled way, actually reflecting what agents currently believe to be the best action another agent can take (i.e. which maximizes their expected reward). Similarly, we want to choose the bribe value in such a way that it accurately reflects, *how* beneficial this single action is for them. We

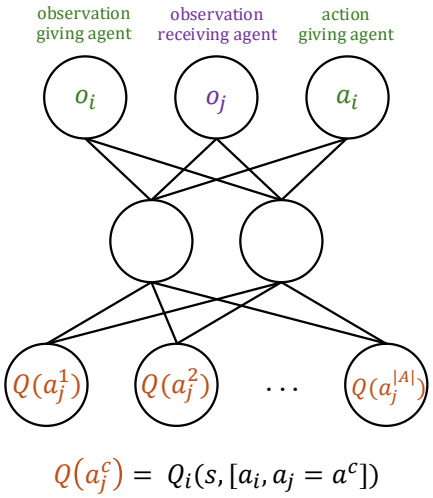


Figure 4.2.: The bribe network predicts the Q-values for agent i given which action agent j takes in this step.

achieve this by giving out additional rewards equal to how much higher the expected return given that the other agent executes the bribe action is, compared to the average expected return over all possible actions.

In order to do this, we need two Q-networks per agent Q_k and $\hat{Q}_k$. Q_k is trained to predict the Q-value of state-action pairs using the shaped rewards r, which include the additional rewards gained from bribes. This network is used to train the agents, just as in DQN. $\hat{Q}_k$ is trained to predict the Q-values of state-action pairs using the base rewards $\hat{r}$ which do not include the bribe values. It will become apparent later on why we need this additional network per agent.

The third component is another Q-network B, which we will use to predict the expected value for agent i taking action a_i in state s and agent j taking action a_j. Therefore, the inputs to this network are o_i, o_j and a_i, and each of its $|A|$ outputs a_j^c denotes the Q-value for agent i if agent j takes action a_j^c. An illustration of this network architecture is shown in Fig. 4.2.

(1) Collecting samples: When drawing the next action a_i, agents choose them as they normally would in DQN, using the Q-network with shaped rewards Q_i. Agents then choose

the bribe action for agent j b_{ij} by

$$b_{ij} = \arg\max_{a_j} B(s, a_i, a_j). \tag{4.3}$$

This corresponds to choosing the action for agent j that maximizes the expected return for agent i in the current state-action pair. Additionally, the agent calculates the bribe value v_{ij} as the reward-over-average of this action, which can be determined by first calculating the expected return-over-average

$$\bar{G}_{ij} = \arg\max_{a_j} B(s, a_i, a_j) - \sum_{a_j \in A} \frac{1}{|A|} B(s, a_i, a_j). \tag{4.4}$$

We now want to convert this expected return to an expected reward, because the bribe values we hand out should have the same magnitude as the rewards an agent usually gets. Because we don't know how the expected return is distributed over future rewards, we assume that all rewards are equal in our calculation. Additionally, we don't know how many steps are still to go until the environment terminates after T steps. On average, there are still $T/2$ steps to go, which is why we use this in our calculation. Together, we thus have

$$\begin{aligned} \bar{G}_{ij} &\approx \sum_{t=0}^{T/2} \gamma^t r \\ \Leftrightarrow r &= \bar{G} \frac{1-\gamma}{1-\gamma^{T/2}}, \end{aligned} \tag{4.5}$$

where $v_{ij} = r$ is an approximation of the reward-over-average that the agent selects as the bribe value.

The agents interact with the environment as normal, storing both the environmental rewards $\hat{r}$ and the modified rewards r, which include possible rewards from bribes if agents have chosen the corresponding bribe action in the replay buffer.

(2) Training: Having drawn a minibatch from the replay buffer, we first train the Q-networks Q_k and $\hat{Q}_k$ for every agent as in DQN. We then want to train the bribe network so that it predicts the correct Q-values. To that end, the target for $B_k(s, a_k, a_j)$ should be equal to the environmental reward that agent k received in this step $\hat{r}_k$ plus the maximum Q-value without bribes in the next state $max_a \hat{Q}'_k(s', a)$, where we use the target network. The parameters of the bribe network are then updated by minimizing the standard DQN loss using the target described above.

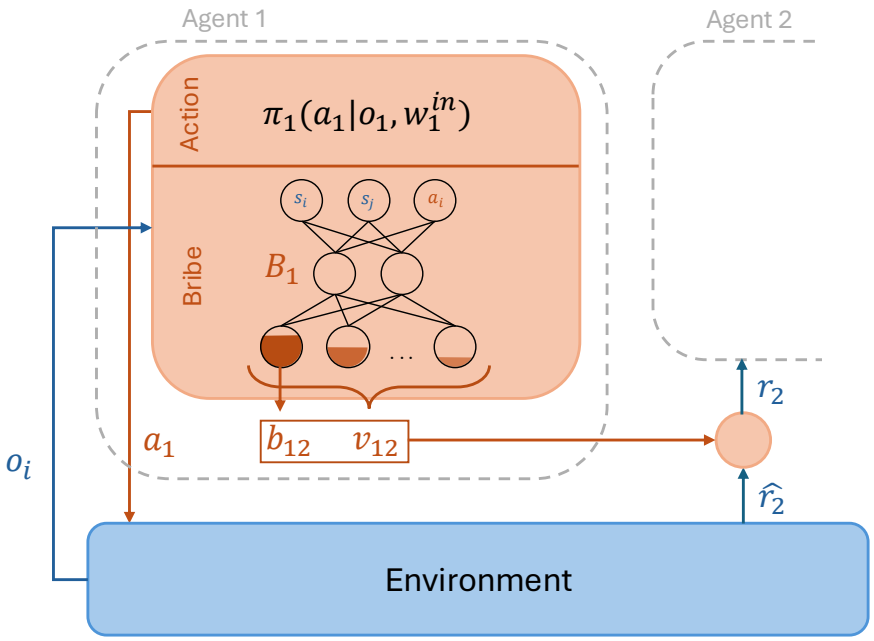


Figure 4.3.: In BRIBE, an agent i chooses an action a_i, and a bribe action b_{ij} and a bribe value v_{ij} for every other agent j, using the bribe network B_i. Agent j then additionally gets v_{ij} reward if it chose action b_{ij}.

An important note and the reason why two separate Q-networks (one for environmental rewards and one for shaped rewards) are necessary, is that we need to use environmental rewards and Q-values here. Were we to use Q-values that have been calculated based on shaped rewards, we would have created a feedback loop: Giving out bribe values increases the expected Q-values using shaped rewards (as there are additional rewards from bribes now). This leads to the bribe network also predicting higher Q-values and thus the agents giving out higher bribes. This process would continue indefinitely and the bribe values would increase with every training step. Therefore, we use solely environmental rewards to update the bribe network's parameters. Lastly, the parameters of all networks are updated softly. Algorithm 5 shows pseudocode for this process, which is also illustrated in Fig. 4.3.

Algorithm 5 Bribes for Incentivizing Behavior (BRIBE)

Initialize Q-network Q_k with parameters θ_k and target network Q'_k with parameters θ'_k for every agent k
Initialize Q-network without bribes $\hat{Q}_k$ with parameters $\hat{\theta}_k$ and target network $\hat{Q}'_k$ with parameters $\hat{\theta}'_k$ for every agent k
Initialize Q-network for bribe actions and bribe values B_k with parameters ϕ_k and target network B'_k with parameters ϕ'_k for every agent k
Initialize replay buffer $R = \{\}$
for $t = 0, \ldots$ **do**
  for each agent k **do**
    Observe s_k^t, choose action a_k^t with ϵ-greedy policy
    Chose bribe-action $b_k = (b_{k1}, \ldots, b_{kn})$ from B where $b_{kj} = \arg\max_{a_j^t} B(s_k^t, a_k^t, a_j^t)$
    Set bribe-value $v_k = (v_{k1}, \ldots, v_{kn})$ where $v_{kj} = (B(s_k^t, a_k^t, b_j^t) - \sum_{a \in A} \frac{1}{|A|} B(s_k^t, a_k^t, a)) \frac{1-\gamma}{1-\gamma^{h/2}}$
    Observe base rewards $\hat{r}_k^t$ and rewards with bribes r_k^t and store sample $(s_k^t, a_k^t, \hat{r}_k^t, r_k^t, s_k^{t+1})$ in replay buffer, reset environment if s_k^{t+1} is terminal
  end for
  for each agent k **do**
    Draw minibatch $\mathcal{D} = \{(s_k^i, a_k^i, \hat{r}_k^t, r_k^i, s_k'^i)\}$ of b samples from replay buffer R
    Compute targets $y_k, \hat{y}_k$ for $Q_k, \hat{Q}_k$ and update $\theta_k, \hat{\theta}_k$ by minimizing the corresponding loss (difference between target and prediction)
    Set target $y^B \leftarrow \hat{r}_k + \max_a \hat{Q}'_k(s'_k, a)$
    for $j \neq k$ **do**
      Update ϕ_k by minimizing $\mathcal{L}(\phi_k) = \frac{1}{b} \sum_{i=1}^{b} (y^B - B_k(s_k^i, a_k^i, a_j^i))$
    end for
    Update all target networks softly
  end for
end for

5. Experiments

In this chapter, we will describe the experiments we have conducted in order to test our methods on different environments, examine different modifications on our methods, and understand their limitations. We will first give an overview of some important implementation details and describe our training process. A more comprehensive list of hyperparameters can also be found in Appendix B. We used the framework MushroomRL [38] for our experiments. Then, we will present the results of our algorithms' performances in three different environments, each having its own challenges. We will also investigate the reward-sharing mechanisms that are being learned as well as perform some ablation studies of different parts of the algorithms. Lastly, we will discuss the results and identify limitations of our methods and how those could be tackled.

5.1. Setup & Training

5.1.1. LToS

In our implementation of LToS, we try to stick to the implementation of the original paper [8] as closely as possible. For the actor-network of the high-level policy (which takes in the current state and outputs the weights shared with all agents), we use a standard neural network with two fully connected hidden layers with 128 neurons, each with a ReLU activation function. Since the outputs of the network should sum to one, a softmax is applied after the last layer.

During training, samples from the high-level policy are drawn with a time-correlated Ornstein-Uhlenbeck noise, because it increases state-space coverage compared to a non-correlated Gaussian noise. Because the noise modifies the sharing weights randomly and they might not sum to one after manipulation, we normalize the sharing weights after

applying the policy in order to ensure that rewards are just redistributed and not lost or created.

For the low-level agent, the original LToS paper uses a graph convolutional network [39], which convolves features of the observations of neighboring agents like a convolutional neural network (CNN). We decide to use a simple DQN instead, because it is well-studied, easier to implement, and does not introduce an additional layer of complexity during training. Also, the environments we test our methods on don't have an inherent notion of adjacency, so it would take additional work to dynamically model a graph between the agents given their current positions.

Our Q-network has the same architecture as the high-level network, only that the number of output neurons correspond to the size of the action space of the current environment and there is no need to apply a softmax over the last layer as we want to predict Q-values. During training, we use an ϵ-greedy policy to draw samples, where we let ϵ decay during training.

We start training by collecting enough samples to fill the replay buffer without fitting our networks. We then train our algorithms for multiple epochs, where we simulate 2,000 environment steps (i.e. agents observe the current state, choose actions, observe reward and next state) in every epoch. Fitting the agents happens every third environment step, which we found to be a good trade-off between training often and experiencing diverse samples from the replay buffer. Because the high-level agent should be updated more slowly than the low-level agent, we adjust the learning rates such that three updates of the high-level agent correspond to one update of the low-level agent.

5.1.2. PERC

For PERC, the architecture of the high- and low-level networks are the same as for LToS, except that the high-level policy now additionally takes in the latent state of the encoder-decoder. The encoder-decoder is a simple neural network with the encoder having one hidden layer of size 128, and its output layer being the latent state. The decoder has the same architecture, only that the input layer takes in the latent state of the encoder and the output layer corresponds to the number of actions in the environment. As with all other networks, we train this encoder-decoder with Adam [40]. We train the encoder-decoder jointly with cross-entropy loss with the target being the action that the other agent actually took, with the training frequency being the same as the one of the low-level network, because it is the function we want to approximate.

During our experiments, we find that conditioning the low-level policy with the predicted action probabilities $\hat{a}_i$ (i.e. the output of the decoder) instead of the latent state l_i (i.e. the output of the encoder) also to be a viable option. We present more data on this in Section 5.3.

We also experimented with adding a regularization term to our loss function that penalizes the high-level policy for choosing sharing weights that deviate too much from sharing the reward equally to all agents, as erratic sharing weights could lead to instability during training, but we found this to only have a very marginal effect on the overall performance and the weighting of this penalty would be a parameter that would need to be fine-tuned for each environment separately.

5.1.3. BRIBE

For BRIBE, we use the same architecture for the low-level DQN agent as we use for the low-level network in LToS. The bribe agent is also a simple neural network that has two hidden layers and outputs the expected q-values for agent i given which action agent j takes. We train the bribe network whenever we train the Q-network, using the same sample. In order to do that, we must store the joint action of all agents for each sample in the replay buffer, because we need to know which action the other agent took for training the bribe network.

During our experiments, we also compare the performance of BRIBE in two settings: Either every agent uses the same bribe network which thus has to learn to differentiate which agent is the giving agent and which one is the receiving agent, based on information contained in the observations. Or every agent has its own bribe network that is only trained with that agent in the position of the agent that gives out bribes and thus might not need to generalize as much. We present more data on this in Section 5.3.

5.2. Environments

5.2.1. Trapped Agent

Trapped Agent is a social dilemma environment that we have devised. It is played on a 2x3 grid world with 3 agents, where one agent is "trapped" and cannot move (i.e. no matter which action it takes, it will always stay on the square it started on). The other two agents

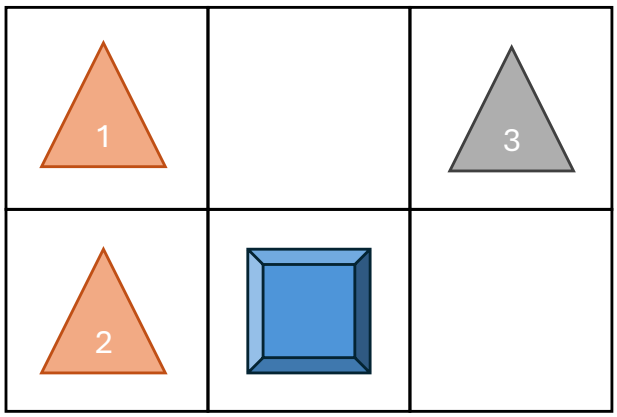


Figure 5.1.: The starting configuration of the *Trapped Agent* environment with one trapped agent (grey) and two free agents (orange). The optimal behavior to maximize the joint reward would be e.g. both agent 1 and agent 2 to move right on their first step, agent 2 to move back in the next step (and agent 2 staying in its new position), and finally agent 2 to move down onto the square with the blue button.

can move freely (up, down, left or right) in the confinement of the grid as long as the square is not occupied by another agent already. Fig. 5.1 shows the environment in its starting configuration. The trapped agent receives a constant negative reward of -1 until it is "freed". That only happens after both free agents have moved onto the square with the button one after the other.

Inherently, the free agents have no incentive to do so and even incur a small negative reward of 0.1 if they step onto the square with the button. Once the trapped agent is freed, it no longer receives a negative reward. We have set up the environment like this because independently training the agents will not lead to the trapped agent being freed, thus not achieving the highest possible joint reward. It is required that the agents (in particular the trapped agent) create some incentive (in our case in the form of shared reward or reward from bribes) to maximize the welfare. We always let the environment terminate after 10 steps so that the returns are comparable across different episodes and there is an incentive to free the trapped agent as early as possible.

5.2.2. Jungle

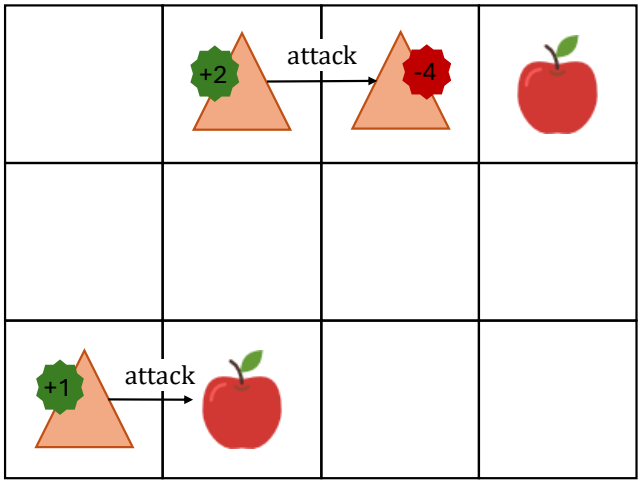


Figure 5.2.: A 3x4 *Jungle* environment with 3 agents (shown as orange triangles) and 2 fruits (apples). An agent receives more reward for attacking an agent than "eating" food, but the attacked agent receives a high negative reward, such that the cumulative reward over all participating agents in the attack scenario is lower than in the eating scenario.

Jungle is a grid-world environment that also poses a social dilemma to the agents. In an MxN grid with A agents, F stationary fruits are spawned in each episode. Agents can move (up, down, left or right) in the confinements of the grid as long as the square is not occupied by another agent yet. They can also attack a neighboring square (up, down, left or right).

When an agent attacks a square that is occupied by another agent, it receives a reward of $+2$ and the attacked agent receives a reward of -4. When an agent attacks a square that is occupied by a fruit, the agent receives a reward of $+1$. Fig. 5.2 illustrates these two scenarios. The same fruit or agent can be attacked by multiple agents at the same time. When an agent attacks an empty square, it receives a small penalty of -0.1 in order to prevent excessive attacking.

Thus, a single agent will inherently favor attacking another agent over attacking a food, since that gives it a higher reward. However, when all agents follow this strategy, which leads to a Nash equilibrium, both the reward of a single agent and the total reward of

all agents will be suboptimal because the penalty for being attacked is higher than the reward for attacking. If all agents only eat food and refrain from attacking, both the total reward and the reward for a single agent will be higher than in the scenario above. To that end, agents need to create incentives for other agents not to attack them in order to reach the optimal strategy of collaborating and only eating fruit.

5.2.3. Give Way

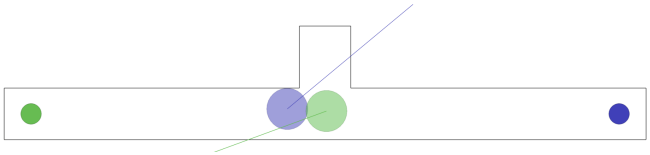


Figure 5.3.: Two agents (big circles) and their current velocity vector (thin lines from their center) in the *Give Way* environment.

The last environment that we conduct experiments in is a two-player environment taken from [41], who provide a series of vectorized multi-agent environments. Two agents start at opposite ends of a corridor and their objective is to reach their respective goals, which are positioned where the other agent starts. In the discretized version of this environment, which we use, the agents can accelerate in one of 8 directions, which then updates their (continuous) velocity or do nothing (keeping their current velocity). The agents get rewarded based on whether they reduced the distance to their goal, and an additional reward if they reached their goal in the last step. In order for the agents to pass each other and reach their corresponding goals, they have to coordinate so that one agent "gives way" to the other by moving up into the nook at the halfway point. The agent that makes way forgoes some of its potential reward because it does not decrease the distance to its goal while making space for the other agent, which is why it should be helpful to train the agents with some incentive mechanism that promotes cooperation.

5.3. Results

We train our algorithms PERC and BRIBE for 100 epochs in the *Trapped Agent* environment. To compare their performances, we also train our implementation of LToS. We also compare our methods against training all agents individually with DQN, each agent maximizing

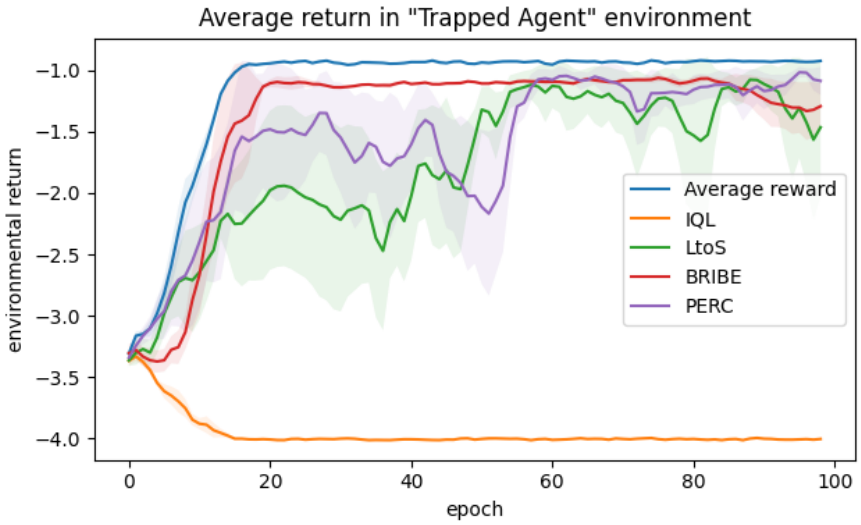


Figure 5.4.: The mean environmental return over all agents during an episode in the Trapped Agent environment. The solid lines represent the mean over five runs and the shaded area is a 90% confidence interval.

their own rewards, which we call *Independent Q-Learning* (IQL), and against training all agents individually, but with each agent receiving the average reward of all agents in the environment, which we call *Average Reward*. IQL and Average Reward can also be seen as special cases of LToS, where the sharing weights are fixed and set to $w_{ij} = \delta_{ij}$ (IQL) or $w_{ij} = 1/n$ where n is the number of agents (Average Reward).

All plots we show in this section are averaged over five runs with different random seeds and show the mean return over all agents in one episode, averaged over all runs, as well as a 90% confidence interval. Fig. 5.4 shows the performance of all algorithms. As expected, IQL does not solve the environment (i.e. the agents don't learn to cooperate and free the trapped agent), which is logical as the free agents have no incentive to accept the negative reward for stepping onto the button, because they don't get compensated for it.

We can see that all other algorithms manage to solve the environment. PERC performs slightly better than LToS, gaining more return more quickly and showing less variance

over the different runs. Furthermore, BRIBE shows an even better and more consistent performance.

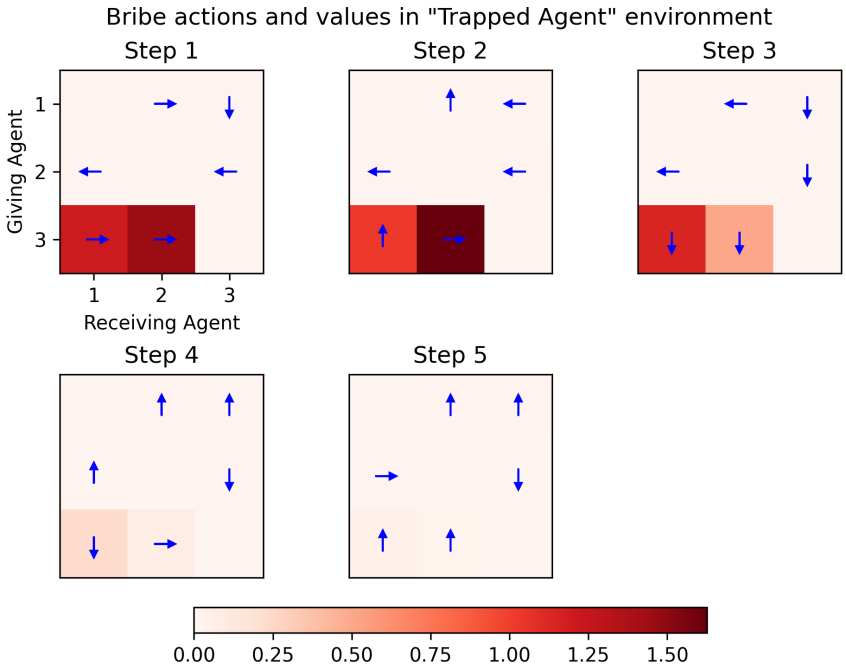
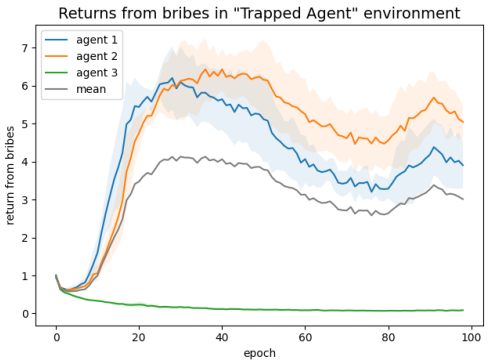


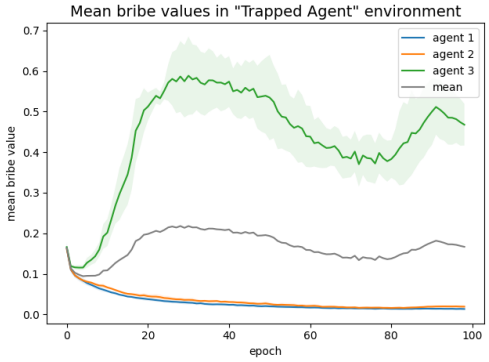
Figure 5.5.: For every environment step in *Trapped Agent*, we show a matrix of bribe actions and values that are handed out. A square (i, j) in the matrix shows the bribe action a_{ij} (from agent i to agent j) as an arrow (up, down, left or right) and the bribe value b_{ij} as the background color (darker red is higher). For example, the bottom left square in the first matrix shows that agent 3 gives agent 1 an additional reward of 1.2, if agent 1 moves right in the first step.

To investigate the learning process of BRIBE further, we can look at further statistics about how bribes are being handed out and if agents learn to accept them. Fig. 5.5 shows a matrix of which bribe actions and bribe values are being handed out by the different agents in each environment step. We can see that, as expected, only the trapped agent (index 3) gives out significant bribes. This is because its reward directly depends on the actions of the other two agents, whereas the free agents' rewards only depend on their own actions, more precisely on whether they step onto the square with the button or not. We can also observe that the actions suggested to the free agents exactly correspond to

a solution in the environment: First, both agents move to the right (agent 1 moves onto the button), then agent 1 moves to the right again, and agent 2 does not move (it hits the upper wall of the environment) and finally, agent 2 moves down onto the button. After the button has been pressed by both agents, the trapped agent is freed and we can observe that it is no longer handing out significant bribe values, as its reward now does not depend on the other agents' actions anymore.



(a) The return from bribes for all agents in the *Trapped Agent* environment throughout training.



(b) Mean bribe values being handed out in a single step for all agents in the *Trapped Agent* environment throughout training.

Figure 5.6.: Bribe values and how they are being earned by other agents during training.

Fig. 5.6 shows how much reward from bribes the agents get throughout the training and the mean bribe value the agents give out in a single step. Since agents only get the bribe reward when they choose the action that was suggested to them, we can observe whether and how fast the agents learn to conform to the actions that are being suggested to them. In Fig. 5.6a, we can see that immediately after the warmup phase (first 10 epochs), agents 1 and 2 learn to follow the bribe action and gain rewards from bribes. Conversely, Fig. 5.6b shows that agent 3 quickly learns to give out bribes. If we compare the two plots, we can see that agents 1 and 2's return from bribes closely matches the mean bribe value that agent 3 hands out, i.e. they learn to conform to the bribe actions quickly.

Lastly, we compare the performance when using a single bribe network across all agents to using agent-specific bribe networks. Fig. 5.7 shows the return for both variants. We can observe that the performance is virtually the same. We suspect this to be the case because the environment is relatively simple and because the agents' observations already

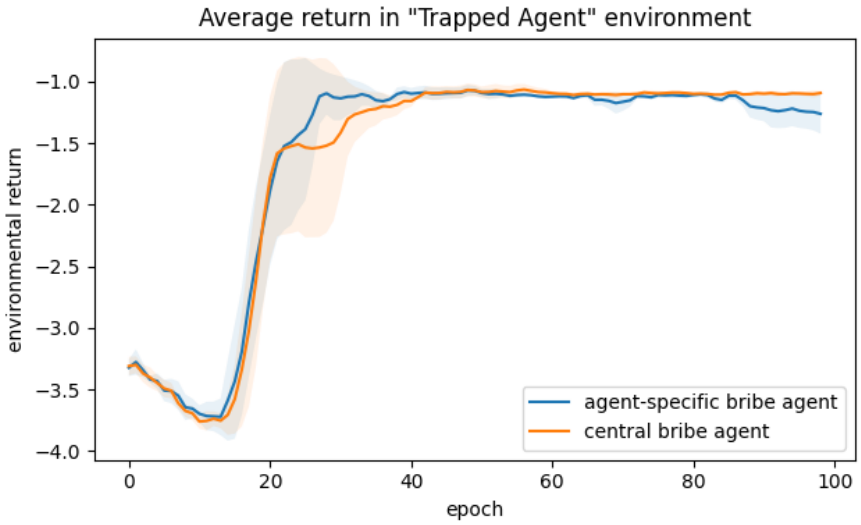


Figure 5.7.: Comparison of the environmental reward when using a shared bribe network across all agents or using one bribe network per agent.

include the index of the agent whose observation it is. Thus, the bribe network can easily determine which agent is currently in the position of giving bribe values and receiving bribe values. Additionally, in this environment, agents have a limited range of motion, and thus agent indices can relatively easily be determined from the agent's position.

We will now look into some more statistics of PERC, in particular how well the encoder-decoder is able to approximate the actions taken by other agents and what information is more helpful for the high-level policy to be conditioned on.

Fig. 5.8 shows the accuracy of the encoder-decoder over the course of training, which corresponds to the percentage of correctly predicted maximum-probability actions. We can see that the accuracy rises relatively quickly, settling around 40%. While this shows us that the encoder-decoder is able to learn a connection between the current state and the chosen action of an agent, it still seems like there is potential for improvement. A reason why the accuracy does not rise even more could be that there are multiple ways to solve this environment that are all equally good in terms of total reward. Therefore, it is

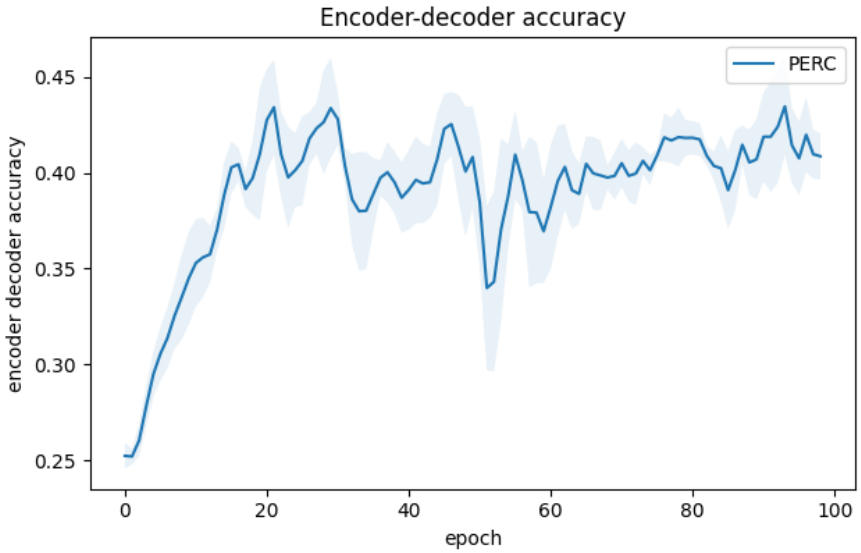


Figure 5.8.: The accuracy of the encoder-decoder in PERC (i.e. how often the action that the encoder-decoder assigned the highest probability was taken by the other agent) in the Trapped Agent environment throughout training.

hard for the encoder-decoder to predict, which of these solutions is being taken in the current episode, reducing the accuracy statistic.

Additionally, during training, we use an ϵ-greedy policy for choosing the actions, such that even though the encoder-decoder might have predicted the maximum likely action correctly, the agent chose their action randomly. Lastly, since we only look at the action that the encoder-decoder assigned the highest probability to, there could be cases where the actual action taken was also assigned a high probability by the encoder-decoder, but not the maximum.

As mentioned in Section 4.1, we initially use the latent state of the encoder-decoder l_i to pass it to the high-level policy as additional information about agent is policy. We compare this with instead passing the predicted action probabilities $\hat{a}_i$, and passing only the action with maximal predicted probability $\arg\max \hat{a}_i$. In theory, the advantage of using the latent state could be that the encoder-decoder has learned to represent useful

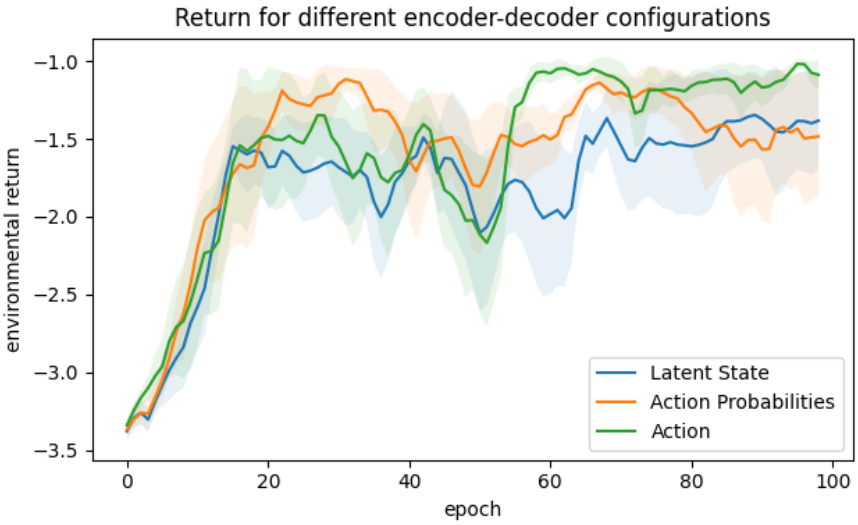


Figure 5.9.: The environmental return in the Trapped Agent environment throughout training for PERC, once conditioning the high-level policy on the hidden state of the encoder-decoder and once on the predicted action probabilities directly.

information to predict the taken action in a structured way.

However, using the predicted action probabilities directly is more explicit and since they are the key information we care about supplying the high-level policy with, we could hope that it is easier to use this explicit form of information. Fig. 5.9 shows a comparison of these variations. We can see that the difference between the modifications is only marginal, but directly passing the most probable action to the high-level policy yields a slightly better return and shows the lowest variance across different runs, so we use this modification in all further experiments.

Fig. 5.10 shows the mean return of all agents in the *Jungle* environment during training over the course of 500 epochs. We set up the environment on a 4x4 grid, two fruits at a time and three agents.

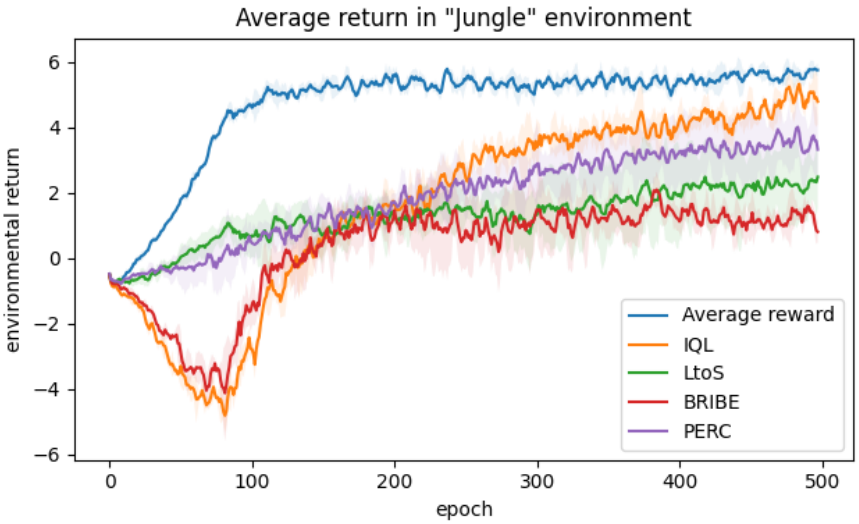


Figure 5.10.: The mean environmental return over all agents during an episode in the *Jungle* environment. The solid lines represent the mean over five runs and the shaded area is a 90% confidence interval.

Fig. 5.11 shows the same plot for two agents in the *Give Way* environment over 200 epochs.

In both environments, we can see that our algorithms PERC and BRIBE, as well as LToS gradually increase the environmental return, but perform worse than the baselines *Average Reward* and IQL. PERC performs slightly better than LToS and also exhibits a lower variance across different runs, but the difference is only marginal. BRIBE's performance settles in the same range as PERC and LToS.

In terms of the behavior of the agents in the Jungle environment, LToS, PERC, and BRIBE all learn to not attack other agents but fail to optimally eat all fruits in the fastest way possible, while the baselines are able to do that consistently. In the Give Way environment, all algorithms learn to swap positions, but the baselines give way in a more coordinated manner and thus the agents pass each other more quickly, while our algorithms take longer until one agent decides to make way for the other, receiving less environmental reward in the process.

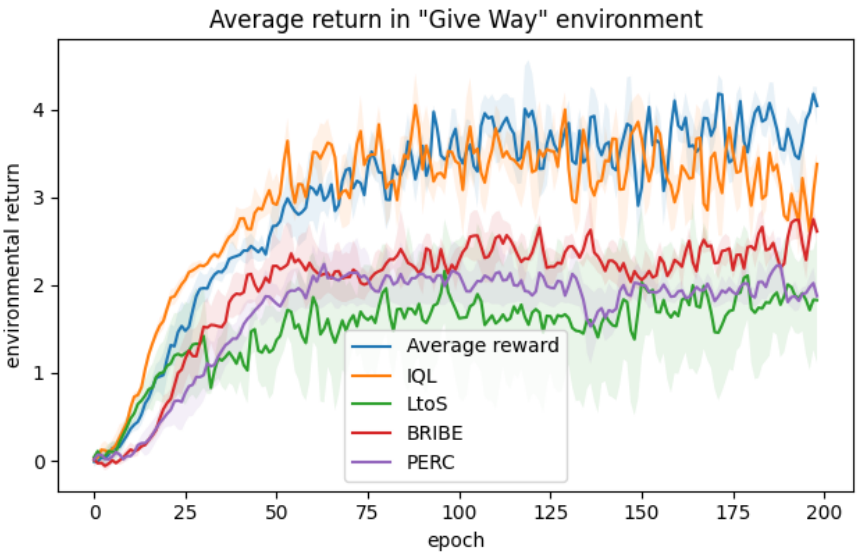


Figure 5.11.: The mean environmental return over all agents during an episode in the *Give Way* environment. The solid lines represent the mean over five runs and the shaded area is a 90% confidence interval.

6. Discussion

We have seen that our methods show some promising results in simple social dilemmas such as the Trapped Agent environment and have analyzed some of their behavior and modifications in more detail. There we could observe, that the BRIBE approach did help to promote actions to other agents, that were part of the optimal solution when optimizing for maximum welfare and that the agents quickly learned to modify their policies so that they conform to those actions and maximize the reward from bribes. However, we found that BRIBE could not translate this performance to the other environments we tested it on. This could have multiple reasons: As the observation space and the action space of the environment increase, the bribe network's approximation task becomes significantly more complex. Not only does the size of the input increase but since the possible combination of the state that agent i observes and the state that agent j observes increases quadratically with the observation space, the bribe network will in the worst case see much fewer samples for a given state-state-action pair. This could demand a deeper or more elaborate network structure than the one that we tried our experiments with.

Furthermore, environments such as Give Way, where the action space is naturally continuous, but discretized into a finite set of actions could be challenging for BRIBE. Because there are oftentimes multiple ways to execute a continuous action with multiple discrete actions, it is harder for BRIBE to know which composition of the desired continuous action it should promote. Since the algorithm only promotes one single action at each step, no matter if other actions were similarly beneficial to the bribing agent, there can be situations where this has an unwanted impact on the reward landscape. A modification that could mitigate this effect could be to not only hand out bribe values to the most beneficial action of the bribing agent, but to the top N actions, or even to all actions. The bribe values could then be calculated in the same way, if one allows for negative bribe values. Or, independently from this change, one could use a different measure of advantage for computing the bribe values which behaves differently to the linear reward-over-mean approach we used, or additionally considers other factors such as regularity in its action

selection.

Concerning PERC, we could see that our modifications to LToS yielded an improvement in environmental return. However, in more complex environments, neither LToS nor PERC were able to learn the best possible policy and were beaten by *Average Reward*. Although our baselines performed well in the environments we presented, one has to note that they are very inflexible. They perform well here because both of the environments can be solved with symmetrical cooperation and even approximated well with selfish behavior. We have already seen in the *Trapped Agent* environment that selfish behavior can often not solve cooperative environments, and similarly, there are situations where symmetric cooperation between all agents is not the best behavior.

Another reason why the performance of our methods lags behind the baselines in our experiments could be that due to the added layer of complexity in order to train the weight-sharing policy (and the encoder-decoder in PERC), the space of the optimization problem grows and the corresponding reward landscape could develop some new local optima. We have briefly looked into regularizing the weight-sharing network by adding a regularization penalty, and this approach could be built upon in order to reduce the degrees of freedom that were introduced into the optimization problem. Additionally, due to a limitation in computing resources, our neural networks used in both the high-level and low-level agent might be not powerful enough to efficiently and robustly learn an optimal policy. Although we have tried to stick to the implementation that was detailed in the original LToS paper [8] as closely as possible, there are still some differences, the biggest one being that we used a standard DQN for our low-level policy where Yi et al. used graph convolutional reinforcement learning. Although a standard DQN approach was also used in our baselines, showing good results, the ability of graph convolutional networks to directly take into account relations to neighboring agents could provide needed stability when it comes to training our more complex algorithms.

6.1. Future Work

As seen in Section 5.3, we did not manage to translate the results of our methods in smaller environments to more complex ones, but we have ideas for further improvements that might be promising. First of all, as our implementation of LToS seems to perform worse than the results reported in the original paper [8], it would be worth implementing the graph convolutional network from [39] and using it instead of a standard DQN approach.

Increasing the size of the networks, which we did not get around to because of limiting computation time is also an area for improvement potential. In order to use computation time more efficiently, one could also make use of the feature of VMAS [41] to run multiple environment instances in parallel. This batching would require adapting the reinforcement framework MushroomRL [38] to handling vectorized environments, but would enable us to obtain more samples efficiently. Also, since PERC seemed to be too unstable during training for more complex environments, enforcing more regularity when drawing the sharing weights could be an area of improvement. We have already briefly tried using a penalty term for sharing weights that deviate too much from sharing reward equally, but analogously rewarding smoothness of the high-level policy across adjacent inputs (states) could be conceivable. Another area of improvement could be experimenting with different architectures for encoding the policy of other agents, such as CNNs for grid-based states.

Concerning BRIBE, a natural extension would be to not only hand out bribe values for the best action but for the best N or even all actions, if one allows to give out negative rewards to discourage bad actions. Also, different measures of "goodness" of an action from the perspective of another agent than our approach of reward-over-average could be experimented with, such as rewarding agents for being predictable or taking into account joint actions with third agents. One could also deviate from our current approach for handing out bribes and allow agents to choose them in a black-box optimization manner.

Another area that would be interesting to investigate and which we have not gotten around to inspect due to time constraints is to conduct experiments in environments that change throughout training, or in which agents get added, removed, or changed from teammate to adversary. There has been little research in these types of environments, so evaluating our methods in these settings would be interesting. Similarly, the methods introduced in this thesis could be applied to semi-cooperative environments, for example in environments where different groups of agents compete against each other, but agents need to work together within their groups, e.g. predator-prey environments [42].

Bibliography

[1] K. Cao, A. Lazaridou, M. Lanctot, J. Z. Leibo, K. Tuyls, and S. Clark, "Emergent communication through negotiation," 2018.

[2] T. Lin, M. Huh, C. Stauffer, S.-N. Lim, and P. Isola, "Learning to ground multi-agent communication with autoencoders," 2021.

[3] D. Simoes, N. Lau, and L. Reis, "Multi agent deep learning with cooperative communication," *Journal of Artificial Intelligence and Soft Computing Research*, vol. 10, pp. 189–207, 07 2020.

[4] G. Tesauro, "Extending q-learning to general adaptive multi-agent systems," in *Advances in Neural Information Processing Systems* (S. Thrun, L. Saul, and B. Schölkopf, eds.), vol. 16, MIT Press, 2003.

[5] V. Conitzer and T. Sandholm, "Awesome: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents," 2003.

[6] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," 2020.

[7] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual multi-agent policy gradients," 2017.

[8] Y. Yi, G. Li, Y. Wang, and Z. Lu, "Learning to share in multi-agent reinforcement learning," 2022.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[10] S. V. Albrecht, F. Christianos, and L. Schäfer, *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2024.

[11] P. Hoen, K. Tuyls, L. Panait, S. Luke, and J. Poutré, "An overview of cooperative and competitive multiagent learning.," pp. 1–46, 01 2005.

[12] D. T. Luc, "Pareto optimality," *Pareto optimality, game theory and equilibria*, pp. 481–515, 2008.

[13] J. Jiang and Z. Lu, "Learning fairness in multi-agent systems," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[14] M. Zimmer, C. Glanois, U. Siddique, and P. Weng, "Learning fair policies in decentralized cooperative multi-agent reinforcement learning," in *International Conference on Machine Learning*, pp. 12967–12978, PMLR, 2021.

[15] K. M. Lee, S. G. Subramanian, and M. Crowley, "Investigation of independent reinforcement learning algorithms in multi-agent environments," 2021.

[16] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[17] V. Mnih, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[18] J. F. Hernandez-Garcia and R. S. Sutton, "Understanding multi-step deep reinforcement learning: A systematic study of the dqn target," *arXiv preprint arXiv:1901.07510*, 2019.

[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[20] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, and T. Graepel, "Value-decomposition networks for cooperative multi-agent learning," 2017.

[21] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. Foerster, and S. Whiteson, "Monotonic value function factorisation for deep multi-agent reinforcement learning," 2020.

[22] J. Wang, Z. Ren, T. Liu, Y. Yu, and C. Zhang, "Qplex: Duplex dueling multi-agent q-learning," 2021.

[23] K. Zhang, Z. Yang, H. Liu, T. Zhang, and T. Başar, "Fully decentralized multi-agent reinforcement learning with networked agents," 2018.

[24] C. Qu, S. Mannor, H. Xu, Y. Qi, L. Song, and J. Xiong, "Value propagation for decentralized networked deep multi-agent reinforcement learning," 2019.

[25] T. Chu, S. Chinchali, and S. Katti, "Multi-agent reinforcement learning for networked system control," 2020.

[26] A. Peysakhovich and A. Lerer, "Prosocial learning agents solve generalized stag hunts better than selfish ones," 2017.

[27] N. Jaques, A. Lazaridou, E. Hughes, C. Gulcehre, P. A. Ortega, D. Strouse, J. Z. Leibo, and N. de Freitas, "Social influence as intrinsic motivation for multi-agent deep reinforcement learning," 2019.

[28] D. Mguni, J. Jennings, S. V. Macua, E. Sison, S. Ceppi, and E. M. de Cote, "Coordinating the crowd: Inducing desirable equilibria in non-cooperative systems," 2019.

[29] J. Wang, Y. Zhang, T.-K. Kim, and Y. Gu, "Shapley q-value: A local reward approach to solve global reward games," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, p. 7285–7292, Apr. 2020.

[30] J. Yang, A. Li, M. Farajtabar, P. Sunehag, E. Hughes, and H. Zha, "Learning to incentivize other learning agents," 2020.

[31] D. E. Hostallero, D. Kim, S. Moon, K. Son, W. J. Kang, and Y. Yi, "Inducing cooperation through reward reshaping based on peer evaluations in deep multi-agent reinforcement learning," in *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 520–528, 2020.

[32] M. Kölle, T. Matheis, P. Altmann, and K. Schmid, "Learning to participate through trading of reward shares," 2023.

[33] A. Lupu and D. Precup, "Gifting in multi-agent reinforcement learning," in *Proceedings of the 19th International Conference on autonomous agents and multiagent systems*, pp. 789–797, 2020.

[34] W. Z. Wang, M. Beliaev, E. Bıyık, D. A. Lazar, R. Pedarsani, and D. Sadigh, "Emergent prosociality in multi-agent games through gifting," 2021.

[35] H. He, J. Boyd-Graber, K. Kwok, and H. D. I. au2, "Opponent modeling in deep reinforcement learning," 2016.

[36] Y. Jin, S. Wei, J. Yuan, X. Zhang, and C. Wang, "Stabilizing multi-agent deep reinforcement learning by implicitly estimating other agents' behaviors," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3547–3551, 2020.

[37] J. Foerster, N. Nardelli, G. Farquhar, T. Afouras, P. H. S. Torr, P. Kohli, and S. Whiteson, "Stabilising experience replay for deep multi-agent reinforcement learning," 2018.

[38] C. D'Eramo, D. Tateo, A. Bonarini, M. Restelli, and J. Peters, "Mushroomrl: Simplifying reinforcement learning research," *Journal of Machine Learning Research*, vol. 22, no. 131, pp. 1–5, 2021.

[39] J. Jiang, C. Dun, T. Huang, and Z. Lu, "Graph convolutional reinforcement learning," 2020.

[40] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[41] M. Bettini, R. Kortvelesy, J. Blumenkamp, and A. Prorok, "Vmas: A vectorized multi-agent simulator for collective robot learning," *The 16th International Symposium on Distributed Autonomous Robotic Systems*, 2022.

[42] M. Kölle, Y. Erpelding, F. Ritz, T. Phan, S. Illium, and C. Linnhoff-Popien, "Aquarium: A comprehensive framework for exploring predator-prey dynamics through multi-agent reinforcement learning algorithms," 2024.

A. Full algorithm of PERC

Algorithm 6 Policy-Encoded Reward-Exchange (PERC)

Initialize high-level network ϕ_k parameters θ_k and target network ϕ'_k with parameters θ'_k for every agent k
Initialize low-level network Q_k parameters μ_k and target network Q'_k with parameters μ'_k for every agent k
Initialize encoder e_k and decoder d_k with parameters ψ_k^d and ψ_k^d for every agent k
Initialize replay buffer $R = \{\}$
for $t = 0, \ldots$ **do**
  for each agent k **do**
    Observe s_k^t, draw encodings l_j and predicted action $\hat{a}_j^t$ for every agent j, choose high-level action $w_k^{out,t} \leftarrow \sigma(\phi_k(s_k^t, l_1), \ldots, \phi_k(s_k^t, l_n))$ with noisy policy
    Exchange $w_k^{out,t}$ to get $w_k^{in,t}$
    Choose low-level action a_k^t with ϵ-greedy policy from $Q(s_k^t, w_k^{in,t}, a)$
    Store sample $(s_k^t, w_k^{out,t}, \hat{a}_k^t, a_k^t, r_k^t, s_k^{t+1})$ in replay buffer, reset environment if s_k^{t+1} is terminal
  end for
  for each agent k **do**
    Draw minibatch $\mathcal{D} = \{(s_k^i, w_k^{out,i}, \hat{a}_k^i, a_k^i, r_k^i, s_k'^i)\}$ of b samples from replay buffer R
    Update ψ_k^d and ψ_k^d jointly by minimizing $\frac{1}{b}\sum_{i=1}^{b}(log(\frac{\exp(\hat{a}_i^{a_k^i})}{\sum_{c=1}^{|A|}\exp(\hat{a}_i^c)}))$
    Draw next high-level action $w_k'^{out} \leftarrow \phi'_k(s'_k)$ and exchange to get w'^{in}
    if s'_k is terminal **then**
      Set target $y_k \leftarrow r_k$
    else
      Set target $y_k \leftarrow r_k + \gamma \max_a Q'_k(s'_k, w_k^{in'}, a)$
    end if
    Update θ_k by minimizing $\mathcal{L}(\theta) = \frac{1}{b}\sum_{i=1}^{b}(y_k^i - Q_k(s_k^i, w_k^{in,i}, a_k^i))$
    Redraw $w_k^{out} \leftarrow \phi_k(s_k)$ and exchange to get w_k^{in}
    Compute $g_k^{in} = \nabla_{w_k^{in}} \arg\max_a Q_k(s_k, w_k^{in}, a)$ and exchange to get g_k^{out}
    Update μ_k by maximizing $\frac{1}{b}\sum_{i=1}^{b}(\nabla_{\theta_k}\phi_k(s_k^i))^T g_k^{out,i}$
    Update target networks softly
  end for
end for

B. Hyperparameters

Hyperparamater	Value
discount factor γ	0.99
batch size	32
encoder-decoder latent state size	32
capacity of replay memory	10,000
$\epsilon_{\text{start}}, \epsilon_{\text{end}}$ (reached after 2/3 num. epochs)	1, 0.1
Ornstein-Uhlenbeck σ, θ, μ	0.01, 0.15, 1e-2
optimizer	Adam
learning rate high-level agent	1e-4
learning rate low-level agent	3e-4
τ for soft update	0.01
update interval	3 steps
steps per epoch	2,000

Figure B.1.: Hyperparameters used for training our methods during the experiments detailed in Chapter 5.