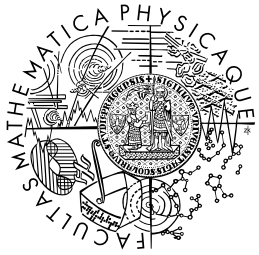


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Přemysl Bašta

Cooperation with Unknown Agents in Multi-agent Environment

Department of Theoretical Computer Science and Mathematical Logic

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

Prague 2023

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

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

In ..... date ....

Author's signature

First of all, I would like to thank my beloved wife, Adéla, who carried me through most of my master's studies and especially during the writing of this thesis, provided me with endless words of motivation and encouragement, and created an overall nurturing work environment. I would also like to thank my mother for her lifelong encouragement.

A big thank you goes to my wonderful supervisor Martin, who not only actively supported me during the time we worked on this thesis, but also inspired me to pursue nature-inspired science and multi-agent systems in the first place. I would also like to thank Milan Straka, whose amazing courses on Deep Learning and Deep Reinforcement Learning made it possible for me to dive deeper into the field through this thesis.

Finally, I would like to thank the MetaCentrum (MetaVO) organization for providing me with cluster computing resources, which made all the necessary computations possible.

Title: Cooperation with Unknown Agents in Multi-agent Environment

Author: Bc. Přemysl Bašta

Department: Department of Theoretical Computer Science and Mathematical Logic

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

Abstract: Over the past few decades, we have witnessed great successes in the field of deep and reinforcement learning. Great achievements have been made in many competitive settings, both in single-agent and multi-agent environments, where AI has managed to outperform human experts and even entire teams of human experts. However, the situation is much more difficult when cooperation is required in purely cooperative environments. We first give a brief overview of reinforcement learning theory and it's current state of the art algorithms. We then extend the theory to multi-agent systems, where several related issues are discussed. And finally, we propose novel approaches of agent training where we use a simplified multi-agent cooperative cooking game environment based on the popular game Overcooked, we attempt to train agents that are robust and capable of ad hoc agent cooperation.

Keywords: Reinforcement Learning Multi-Agent Systems Ad-hoc Cooperation

Contents

Introduction

In recent years, people have witnessed rapid progress in artificial intelligence (AI) in all kinds of fields. Beating world champions at chess or Go is no longer a problem for AI models. The same pattern can be seen in more recent popular games such as Dota (OpenAI [2019]) or Starcraft, where even great human-AI team cooperation behavior has been achieved. However, all of these examples share the common trait of being competitive. The ultimate goal of our society is to create AI that cooperates with humans, not competes with them.

Recent work has shown that cooperative AI models trained together on purely cooperative tasks tend to rely on near-optimal behavior from their partners, and fail to cooperate with partners who don't meet this condition. This is bad news for us humans, because our behavior is rarely optimal.

A great example of human-AI cooperation where humans do not always perform perfectly are self-driving cars. In a situation where an accident is imminent, humans have to react quickly without having enough time to consider all possible reactions or even analyze the entire current road situation. However, car accidents are perhaps even too extreme an example of human suboptimal behavior. Nevertheless, people often fail at the even simpler task of following standard traffic rules when they have enough time to react. We can imagine that predicting human behavior is not an easy task for a self-driving car.

In this work, we will first operate in a single-agent environment, revisiting the definition of Markov decision, the building block of reinforcement learning. Based on this theory, we will intuitively introduce popular reinforcement learning algorithms divided into two categories of Q-learning and policy optimization. We primarily focus on policy branch of reinforcement algorithms, especially policy learning algorithm proximal policy optimization, which is considered as state of the art algorithm masively deployed in many successful projects.

Subsequently, we extend the theory developed for single-agent environments to more complex scenarios where more agents are involved. We highlight problems related to multi-agent settings, where the observability of the world is often an issue compared to single-agent environments. We illustrate problems with multi-agent training, where changes in one agent affect the behavior of the environment from the point of view of other agents, introducing the problem of non-stationarity. Finally, some single-agent variants of reinforcement learning algorithms are extended to multi-agent settings, where we mention the important aspects of these extensions.

We will use a simplified cooperative cooking environment based on the popular video game Overcooked, where two partners are forced to coordinate a shared task of cooking and delivering soup to a customer. We will familiarize ourselves with several different kitchen layouts, as each layout may offer different cooperative obstacles. Here we summarize what approaches have been tested in related work with respect to ad hoc agent cooperation. We mention the problem of defining the robustness of agent cooperation and different possible definitions of robustness.

And in the last part of this work, we prepare our working environment by modifying the stable-baselines3 library designed for reinforcement learning algorithms. We try to reimplement some methods of previously related work regarding

the problem of ad hoc coordination in AI-AI settings, both for verification purposes and also for building us an evaluation tool for our experiments. And finally, we propose a diversification method for building a population of agents that are designed to try to differentiate their behavior from those they have encountered in the training population. After tuning and optimizing our experiments on a selected kitchen layout, we then evaluate our approach on some of the remaining layouts.

1. Introduction to reinforcement learning

In this introductory chapter, we slowly build up the intuition and motivation behind reinforcement learning. We start by defining the mathematical model of the Markov decision process and then proceed with other related properties and relationships. The following pages are inspired by the introduction to reinforcement learning as presented by the authors of the spinningup library (Achiam [2018]) and also by the introductory book to reinforcement learning (Sutton and Barto [2018]).

1.1 Markov decision process

Definition

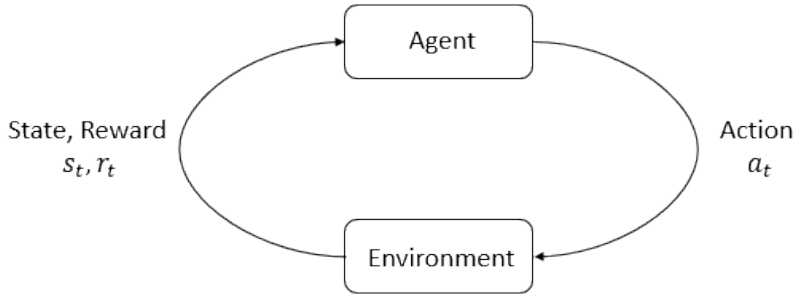
Markov Decision Process is tuple $\langle S, A, R, P, \rho_0 \rangle$, where

- S is the set of all valid states,
- A is the set of all valid actions,
- $R : S \times A \times S \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$ being reward obtained when transitioning from state s_t to s_{t+1} using action a_t.
- $P : S \times A \to \mathcal{P}(S)$ is the transition probabilty function, where $P(s_{t+1}|s_t, a_t)$ is probability of transition from state s_t to s_{t+1} after taking action a_t.
- ρ_0 is starting state distribution.

The name Markov comes from the fact that the system satisfies the Markov property, which states that the history of previous states has no effect on the next state and that only the current state is considered for state transitions.

1.2 Single-agent environment

Having defined the mathematical model of the environment, let's review the related concepts. The whole problem of reinforcement learning (RL) can be best described by the following visualization.



Environment represents a kind of world with its internal rules and properties. The agent is then an entity that exists within this world, observes **state s** of the world, decides to react with **action a** on the basis of this state, and receives **reward r** as a consequence of this action. The entire mechanism of this environment can then be broken down into these cycles of states, actions, and rewards. The goal of an agent is to interact with the environment in such a way as to maximize its cumulative reward.

Observability

State s contains all information about environment at given time. However, in some environment agent can perceive only **observation o** where some information about environment can be missing. In this case we say that environment is **partially observable** as opposed to **fully observable** environment where agent has all information available at it's observation. Nevertheless, this problem is more related to multi-agent environments discussed in the next chapter and not so much to single-agent settings.

Actions and policies

Environments can also differ in terms of what actions are possible within a given world. The set of possible actions is called **action space**, which again can be divided into two types. **Discrete** action space contains finite number of possible actions. And **continuous** action space, which allows the action to be any real-valued number or vector.

The agent's choice of action can then be described by a rule called **policy**. A common notation is that if the action selection is deterministic, we say the policy is **deterministic** and denote by

$$a_t = \mu(s_t).$$

If policy is **stochastic** it is usually noted as

$$a_t \sim \pi(\cdot|s_t).$$

Policies are the main object of interest of reinforcement learning, as this action selection mechanism of an agent is what we are trying to learn. A policy, for optimization purposes, is a function often parametrized by a neural network whose parameters are usually denoted by the symbol θ , therefore, parameterized deterministic and stochastic policies are represented by the symbols $\mu_\theta(s_t), \pi_\theta(\cdot|s_t)$ respectively.

Trajectory

The next important definition is the notion of trajectory, also known as episode or rollout. A trajectory is a sequence of states and actions in an environment.

$$\tau = (s_0, a_0, s_1, a_1, ...)$$

The initial state s_0 of an environment is sampled from **start-state distribution**, denoted as ρ_0. Subsequent states follow the transition function of the environment. These may again be deterministic

$$s_{t+1} = f(s_t, a_t)$$

or stochastic,

$$s_{t+1} \sim P(\cdot|s_t, a_t)$$

Return

We have already mentioned the agent's desire to maximize cumulative rewards. Now we combine it with trajectories and derive the formulation of **return**.

$$R(\tau) = \sum_{t=0}^{|\tau|} r_t \quad \text{(finite-horizon)}$$

$$R(\tau) = \sum_{t=0}^{|\tau|} \gamma^t r_t \quad \text{(infinite-horizon discounted return)}$$

Infinite-horizon discounting is both intuitive and mathematically convenient.

Optimal policy

In general, the goal of RL is to find such a policy that maximizes the expected return when acted upon. Suppose both the environment state transitions and the policy are stochastic. Then we can define the probability of the trajectory as

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

The expected return $J(\pi)$ can then be expressed as

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)].$$

And finally, we can finish by defining the optimal policy

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

which is also an expression describing the central RL optimization problem.

Value functions

Once we have some policy π it would be useful to define value of observed state. For that matter we define two functions.

On-Policy Value Function $V^\pi(s)$, which yields value of expected return when starting from state s and following policy π:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s]$$

Similarly we define **On-Policy Action-Value Function** $Q^\pi(s,a)$ which adds the possibility to say that in state s we take an arbitrary action a that does not necessarily have to come from policy π:

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$$

For the optimal policy we further define **optimal value function** $V_{\pi^*(s)}$ and **optimal action-value function** $Q_{\pi^*(s,a)}$:

$$V^*(s) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s],$$
$$Q^*(s,a) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$$

Bellman equations

There exist formulations called Bellman equations that provide a way how to express value function in terms of action-value function and vice versa. They are based on the idea that the value of a state is equal to the reward you get in a given state, plus the value of the state you will obtain in the next transition. This idea also provides recursive relation.

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_{a\sim\pi(s)}[Q_\pi(s,a)] \\ &= \mathop{\mathbb{E}}_{a\sim\pi(s), s'\sim P(\cdot|s,a)}[R(s,a,s') + \gamma V^\pi(s')] \end{aligned}$$

$$\begin{aligned} Q^\pi(s,a) &= \mathop{\mathbb{E}}_{s'\sim P(\cdot|s,a)}[R(s,a,s') + \gamma V_\pi(s')] \\ &= \mathop{\mathbb{E}}_{s'\sim P(\cdot|s,a)}[R(s,a,s') + \gamma \mathop{\mathbb{E}}_{a'\sim\pi(s')}[Q_\pi(s',a')]] \end{aligned}$$

The most important theorem for us is the reformulation of Bellman's equations for optimal policies:

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s'\sim P}[R(s,a,s') + \gamma V^*(s')]$$
$$Q^*(s,a) = \mathop{\mathbb{E}}_{s'\sim P}[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

As we will see in the next section. The first RL algorithm will be a straightforward application of the Bellman equation for optimal policy.

Advantage function

Now that we've spent a few sections defining functions for the absolute value of actions or state-action pairs, it's worth considering relative value as well. Often, when dealing with RL problems, it is not so important for us to know the exact value of the action-state pair, but rather whether and by how much a given action is, on average, relatively better than others. In other words, we want to know the relative advantage of a given action over others. For a given policy π, the advantage function $A^\pi(s,a)$ describes how much better it is to take action a over

randomly sampled actions following policy π, assuming that policy π is followed in all subsequent steps. Mathematically, the advantage function is defined as follows

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s).$$

The concept of an advantage function will be an integral part of policy gradient based methods, as we will see in a later section.

2. Reinforcement learning algorithms

In the previous chapter, we built the theoretical foundation of reinforcement learning. In this chapter, we translate that theory into practical algorithms. There are no precise classification boundaries between RL algorithms, as many of the techniques described are shared and crossed by all kinds of algorithms. Therefore, it is difficult to come up with an absolutely definitive taxonomy. However, for our purposes and the scope required, the following division into Q-learning and policy optimization is sufficient.

2.1 Q-Learning

Idea

We will start with a family of algorithms that focuses on learning the action value approximator $Q_\theta(s, a)$, as described in the previous chapter. For this reason, the group of such algorithms can also be referred to as Q-learning. Our primary goal in the RL problem is to find a policy that the agent can follow. In the case of Q-learning, once we have learned the approximator $Q_\theta(s, a)$, we can derive the policy by always taking the best possible action in the given state according to the learned action-value function.

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

By incorporating Bellman's optimal policy equations, we can directly train the Q-network by minimizing the loss

$$L(\theta) = (r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a))^2.$$

And by computing the loss gradient, we arrive at the update rule

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

which is the backbone of the 2.1 algorithm bearing the same name.

Instability

The algorithm has rarely been used in this pure form. It has primarily been described for tabular methods, where the action-value function is represented by a table instead of a network approximator. In its simplest form, training is unstable and suffers from a number of significant shortcomings. Most notable is the theoretical deadly triad counter example Sutton and Barto [2018], which consists of a combination of value approximation, bootstrapping, and off-policy training that can lead to instability and divergence. The value approximation condition is met because we use a Q-network to approximate the action value. Bootstrapping means that the estimate is used to compute the targets, this is also true in Q-learning for the same reason. Finally, the term off-policy stands for an approach where training data is collected using a different distribution than that of a target policy.

Algorithm 2.1: Q-learning

Input: initial action-value aproximator Q parameters θ .

repeat

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

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

Execute a in the environment.

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

if *d is true* **then**

Reset environment state.

end if

Compute targets

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

Update Q-network taking one step of gradient decent on

$$(y(r, s', d) - Q_\theta(s', a))^2$$

until *convegence*;

DQN

One of the most outstanding papers based on Q-learning was the algorithm Deep Q-learning 2.2, which demonstrated super-human results on several Atari games Mnih et al. [2015]. We give the pseudocode of the algorithm in it's original form. The notation may seem a bit different from ours, but it represents the same mechanisms that we expect. To address the problem of correlated transition sequences of data, they introduce experience replay, where previously sampled transitions are stored. During training, data is sampled from this buffer, thus smoothing the training distribution over different past behaviors. However, probably the most important idea was the usage of a target Q-network, which broke the value approximation condition of the deadly triad, thus making the algorithm more robust. The target network is a copy of the original Q-network, with its parameters frozen and updated only once in a while based on the parameters of the main network. Its sole purpose is to compute target estimates that are not directly dependent on the Q-network function.

Rainbow

Deep Q Learning was a significant contribution that led to the study of further Q Learning capabilities. The Rainbow project Hessel et al. [2017] could probably be called the pinnacle of such research. In this paper, they further investigate several isolated ideas of possible improvements and try to combine them. To name a few, they use double Q-network to address the problem of maximization

Algorithm 2.2: Deep Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for *episode = 1,M* **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for *t=1, T* **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observer reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set

$$y_j = \begin{cases} r_j & \text{if terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$

        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
        Every C steps reset $\hat{Q} = Q$
    end for
end for

bias and enhance sampling from experience buffer by considering the priority of stored individual data samples. Together with all the other improvements, they achieved state-of-the-art performance on the Atari 2600 benchmark, both in terms of data efficiency and final performance.

2.2 Policy gradient methods

2.2.1 Idea

In the previous section, we were acquainted with the first group of RL algorithms, where we derived the final policy by acting according to $argmax$ of our Q-function approximator. Since our objective is to find an optimal policy, this approach of considering Q values may seem a bit indirect. Fortunately, there is a whole other family of algorithms that deal with this very issue. As the name suggests, policy gradient algorithms focus on directly optimizing the policy $\pi_\theta(a|s)$. This is achieved by directly taking steps along the gradient of the expected return return $J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$. The optimization step then has the form

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

and the gradient $\nabla_\theta J(\pi_\theta)$ is called **policy gradient**.

Before converting this into an algorithm, we have to figure out how to compute the policy gradient numerically. Such an expression can be obtained as a result of the Policy Gradient Theorem (PGT).

TODO: Nasledujici veta a dukaz je inspirovany odtud `https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html` kde se odkazuji na puvodni literaturu, kde je skutecne PGT dokazany. Ale je tam pouze v Sumarni podobe s hodne predpoklady a celkove dost jinou reprezentaci. Nikde se tam veta nebo dukaz pro integral nevyskytuje. Dava mi intuitivne dost smysl dukaz prepsat z diskretniho stavu do spojiteho a tvarit se ze vse plati jako predtim, ale nevim, nevim, nepodarilo se mi najit nikde jinde odkaz na vysledek v teto podobe. Zajimave bylo kdyz jsem se koukal do diplomky od Honzy Uhlika, ze on to tam uvadi take v integralni podobe, zavadi si tam jeste distribuci pro pocatecni stav a do zneni vety rovnou zahrne i fakt za expected return lze nahradit za jinou nahodnou promenou nezavislou na akci - take tam ma pouze odkaz na tu samou puvodni literaturu. Tak nevim, jak se k tomu postavit. Jen tak prohlasit ze kdyz to plati pro diskretni podminky, tak ze to plati i pro spojite, mi pripada zvlastni.

2.2.2 Policy Gradient Theorem

Policy Gradient Theorem(Sutton and Barto [2018]). It holds:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{|\tau|}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)]$$

This can be proven by rewriting the formula in the following way:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) \\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) \\
&= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)R(\tau)] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{|\tau|} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)]
\end{aligned}$$

It is useful to have a look at the expression and what it represents. There are two important components of the expression: $\pi_\theta(a_t|s_t)$ and $R(\tau)$. Taking the gradient step of this objective, we are actually stating that we want to make the change in log probability $\pi_\theta(a_t|s_t)$ weighted by how good the expected return was. However, this may feel slightly counterintuitive, since the expected return takes into account all the rewards of the episode. We may want to restrict ourselves to the future consequences of a given action. Fortunately, it can be shown that $R(\tau)$ can be replaced by many other useful functions(Schulman et al. [2015b]):

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{|\tau|} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right],$$

where Ψ_t, can be one of the following:

$\sum_{t=0}^{|\tau|} r_t$: total reward of the trajectory

$\sum_{t=t'}^{|\tau|} r_t$: reward following action a_t

$\sum_{t=t'}^{|\tau|} r_t - b(s_t)$: baselined version of previous formula

$Q^\pi(s_t, a_t)$: state-action value function

$A^\pi(s_t, a_t)$: advantage function

2.2.3 Vanilla Policy Gradient

Common practice for policy gradient algorithms is to use some form of advantage function, where the baseline function is the value approximator $b(s_t) = V^\pi(s_t)$. The value approximator is typically represented by another neural network and is learned in parallel with the policy. There is a naming convention for the two types of networks. The policy network is usually called an actor since it's job is to provide a policy for the agents to act on. The value network, on the other hand, is often referred to as a critic, because it produces, in a sense, a critique of the value of the state. Incorporating the value approximator and the idea of the utility function reduces the variance in the sample estimation and leads to more stable and faster learning.

With all of this said, we present the first algorithm of this class.

Algorithm 2.3: Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for *k = 0,1,2,...* **do**
  Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Estimate policy gradient as

$$\hat{g_k} = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

  Compute policy update, either using standard gradient ascent,

$$\theta_{l+1} = \theta_k + \alpha_k \hat{g_k},$$

  or via another gradient ascent algorithm like Adam. Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2,$$

  typically via some gradient descent algorithm.
end for

Policy gradient methods differ from Q-learning in several ways.

First, the absolute value of the objective function we are optimizing cannot be interpreted in terms of performance outcome. In fact, taking a step in the gradient descent does not even guarantee an improvement in the expected return in general. On a given set of samples, a value of even $-\infty$ can be achieved. However, the expected return of a changed policy would most likely be abysmal.

And second, policy gradient algorithms are **on-policy**, meaning that only data samples collected using the most recent policy are used for training. This is in contrast to the off-policy approach of Q-learning, where training data samples collected throughout the training process are used for learning steps. For this reason, policy gradient algorithms, especially vanilla policy gradient algorithms, are often considered sample-inefficient compared to off-policy algorithms.

2.2.4 Trust Region Policy Optimization

Although the standard policy gradient step makes a small policy change within the parameter space, it turns out that it may have a significant change on the performance difference. Therefore, vanilla policy gradient algorithm must be careful not to take large steps, making it even more sample-inefficient.

The next algorithm in the policy gradient family attempts to address this problem. As its name suggests, Trust Region Policy Optimization (Schulman et al. [2015a]) tries to take steps within the trusted region in which it is con-

strained so as not to degrade performance. TRPO proposes theoretical update of parameterized policy π_θ as

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}(\theta_k, \theta)$$
$$\text{s.t.} \bar{D}_{KL}(\theta||\theta_k) \leq \delta$$

where

$$\mathcal{L}(\theta_k, \theta) = \mathop{\mathbb{E}}_{s,a\sim\pi_{\theta_k}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)\right]$$

is a surrogate advantage measuring the performance of the policy π_θ relative to the old policy π_{θ_k}.

And

$$\bar{D}_{KL}(\theta||\theta_k) = \mathop{\mathbb{E}}_{s\sim\pi_{\theta_k}} [D_{KL}(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s))]$$

is the average KL-divergence(Kullback [1959]) between policies evaluated on states visited by the old policy. However, working with theoretical updates of TRPO in this form is not an easy task. Therefore, approximations obtained by applying Taylor expansion around θ_k are being used:

$$\mathcal{L}(\theta_k, \theta) \approx g^T(\theta - \theta_k)$$
$$\bar{D}_{KL}(\theta||\theta_k) \approx \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k)$$

And the original problem can then be reformulated as an approximate optimization problem:

$$\theta_{k+1} = \arg\max_\theta g^T(\theta - \theta_k)$$
$$\text{s.t.} \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta$$

Such an approximate reformulation can be solved analytically using methods of Lagrangian duality (Rockafellar [1970]), yielding the solution:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

However, since we employed Taylor expansion, the approximation error could violate the KL divergence constraint. For this reason, TRPO incorporates the idea of backtracking line search (Armijo [1966]):

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

where $\alpha \in (0, 1)$ is the backtracking coefficient and j is the smallest non-negative integer such that the KL divergence constraint is fulfilled and the surrogate advantage is positive.

Algorithm 2.4: Trust Region Policy Optimization

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K
for *k = 0,1,2,...* **do**
    Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
    Estimate policy gradient as

$$\hat{g_k} = \frac{1}{|D_k|} \sum_{\tau \in D_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

    Use the conjugate gradient algorithm to compute

$$\hat{x_k} \approx \hat{H}_k^{-1} \hat{g}_k,$$

    where $\hat{H}_k^{-1}$ is the Hessian of the sample average KL-divergence.
    Compute the policy by backtracking line search with

$$\theta_{l+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

    where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
    Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2,$$

    typically via some gradient descent algorithm.
end for

2.2.5 Proximal Policy Optimization

Finally, we will conclude this chapter with last algorithm from family of policy gradient algorithms, which is considered in many aspects to be the state of the art algorithm. As the authors of this next algorithm state. With the leading contenders Q-learning, "vanilla" policy gradient methods, and trust region policy gradient methods, there is still room for the development of a method that is

- scalable - large models and parallel implementations
- data efficient
- robust - successful on a variety of problems without hyperparameters tuning

Q-learning is poorly understood and fails on many trivial problems. Vanilla policy gradient methods suffer from poor data efficiency and robustness. And trust region policy optimization is relatively complicated and not well suited to architectures including noise or parameter sharing.

Proximal policy optimization (Schulman et al. [2017]) aims at data efficiency and reliability of TRPO performance while using only first-order optimization. The authors propose a novel objective with clipped probability ratios, which provides a pessimistic lower bound on the performance of the policy.

Let $r_t(\theta)$ denote the probability ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}.$$

Then the "surrogate" objective of TRPO can be expressed again in the form:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right]$$

CPI refers to conservative policy iteration (Kakade and Langford [2002]), where this objective was originally proposed. Maximizing L^{CPI} without any constraint would, as discussed in the previous section, lead to an excessively large policy update. Thus, the authors propose a new modified objective, called the clipped surrogate objective, which penalizes policy adjustments that move $r_t(\theta)$ far from 1.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right],$$

where the value $\epsilon = 0.2$ is empirically suggested. The objective can be intuitively explained as following. The first term in min is the previous L^{CPI}. And the second term modifies the surrogate objective by a clipping probability ratio that prevents r_t from escaping the interval $[1-\epsilon, 1+\epsilon]$. Finally, taking the minimum of the clipped and unclipped objectives makes the final objective pessimistically bounded on the unclipped objective.

The training process of PPO then proposes to alternately sample data from the policy and then perform several epochs of optimization steps on the sampled data. Note here that for each first training epoch it holds that $r_t(\theta) = 1$, so the objective in the first epoch always equals L^{CPI}.

Alternatively, the authors propose a second version of PPO that includes the KL penalty as part of the objective, making it even more similar to the idea proposed in TRPO. However, we will not cover more details here, as the authors themselves prefer the version including clipping of the surrogate objective.

When using a neural network architecture that shares parameters between the policy and the value function, a combined objective function that includes both the policy surrogate and the value function error term must be applied.

$$L_t^{CLIP+VF}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) \right],$$

where c_1 is the coefficient and L_t^{VF} is a squared error loss $(V_\theta(s_t) - V_t^{targ})^2$.

Algorithm 2.5: Proximal Policy Optimization

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for *k = 0,1,2,...* **do**

Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2,$$

typically via some gradient descent algorithm.

end for

Lastly, one problem we haven't explicitly addressed is that of exploitation and exploration. In general, during training we always try to strike a balance between exploiting learned experience and exploring new possibilities. In the case of Q-learning, this is most often done artificially by using the ϵ-greedy approach, where with probability $1 - \epsilon$ we exploit our knowledge by taking $\arg\max_a$ action according to the Q value. And with probability ϵ we explore by taking a random action.

Sampling according to the policy π_θ in policy optimization methods solves this problem more naturally. Typically, at the beginning of the training process, when the parameters are initialized, the parametrized probability distribution is close to uniform, which implicitly makes the action selection mechanism exploratory. And as the policy is updated, it becomes more specialized towards the optimal policy, effectively forcing the action selection to be more exploitative. Unfortunately, the exploration described in policy optimization methods is often insufficient, and policies tend to get stuck at bad local optima despite an initial uniform distribution. On this account, an exploration mechanism in the form of bonus entropy(Williams [1992]) is often employed and also recommended by the PPO authors. By adding a small bonus for policy entropy, the policy is slightly forced in the direction of uniform distribution, shifting towards exploratory behavior. Thus, the final PPO objective may have a form as follows:

$$L_t^{CLIP+VF}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right],$$

where c_2 is another coefficient and S denotes an entropy bonus.

Q-Learning and Policy optimization

So far we have covered the simplest possible division rule between types of reinforcement learning algorithms. However, there are actually several algorithms that combine both Q-learning and policy optimization. In these algorithms, both the Q approximator and the policy approximator are learned simultaneously. To name just a handful of the more popular ones: Deep Deterministic Policy Gradient (DDPG, Lillicrap et al. [2015]), Twin Delayed DDPG (TD3, Fujimoto et al. [2018]), Soft Actor-Critic (SAC, Haarnoja et al. [2018]). We won't cover them in detail, as they are beyond the scope of our needs.

3. Multi-agent environment

In this chapter, we revisit concepts from the previous section and extend them to multi-agent settings. We introduce theoretical definitions for various types of settings and provide possible schemes for concurrent learning of multiple agents. And at the end, we mention two popular RL approaches in the field of MARL.

3.1 Multi-agent Markov Decision Process

So far, all the theory and algorithms that have been built have revolved around environments where there is a single agent. However, this is rarely the case in real-world problems. Much more often we encounter environments with multiple agents operating within them. In the general case, the agents are heterogeneous, which means that the agents may have different goals. Nevertheless, we will mostly focus on environments that are fully cooperative, meaning that the utility of any given state of the system is equivalent for all agents.

With this setting we can extend the definition from the MDP section 1.1:

Definition

Multiagent Markov Decision Process(Boutilier [1996])is a tuple $\langle n, S, \mathcal{A}, T, R \rangle$

- n is the number of agents
- S is the set of all valid states,
- $\mathcal{A}$ is the set of joint actions
- $T : S \times \mathcal{A} \times S \to [0, 1]$ is the transition function
- $R : S \to \mathbb{R}$ is a real-valued reward function, where reward determined by the reward function $R(s)$ is received by the the entire collection of agents, or alternatively, all agents receive the same reward.

3.2 Decentralized Partially Observable MDP

Unfortunately, we cannot be satisfied with this definition, since MMDP provides a description that is far too idyllic for real-world problems. The MMDP model presumes that the state of the environment can be globally accessed by all agents. However, this is rarely the case in multi-agent environments. In general, the world can be of high complexity, and combined with limited sensory capabilities, the agent may perceive only limited observations describing only a part of the entire environment state. Therefore, we extend our model definition to include the concept of partial observability.

Definition

Decentralized partially observable Markov decision process (Dec-POMDP, Oliehoek et al. [2008]) is tuple $\langle n, S, \mathcal{A}, P, R, \mathcal{O}, O, h, b^0 \rangle$ where,

- n is the number of agents
- S is the set of all valid states,
- $\mathcal{A}$ is the set of joint actions
- $P : S \times \mathcal{A} \times S \to [0, 1]$ is the transition function
- $R : S \to \mathbb{R}$ is the immediate reward function
- $\mathcal{O}$ is the set of joint observations
- $O : \mathcal{A} \times S \to \mathcal{P}(\mathcal{O})$ is the observation function
- h is the horizon of the problem
- $b^0 \in \mathcal{P}(S)$, is the initial state distribution at time $t = 0$

We define $\mathcal{A} = \times_i \mathcal{A}^i$, where $\mathcal{A}^i$ is the set of actions available to agent i. Similarly, $\mathcal{O} = \times_i \mathcal{O}^i$, where $\mathcal{O}^i$ is the set of observations available to agent i. Note that even this definitional extension does not provide us with a model capable of describing an environment where agents have different reward functions, which is needed for situations where agents have different goals or are even competing. This extension is possible with the definition of Stochastic game (Shapley [1953]). However, we don't need to provide the formal definition and extend our models, since the primary goal of this work will focus mainly on the cooperative setting where all agents have a common goal.

If the observation satisfies the condition that the individual observation of all agents uniquely identifies the true state of the environment, the environment is considered fully observable and such a Dec-POMDP can be reduced to MMDP.

Notation

To denote common entities, we will use bold: $\boldsymbol{a} = (a^1, ..a^n) \in \mathcal{A}$. The common policy $\boldsymbol{\pi}$ induced by the set of individual policies $\{\pi^i\}_{i \in n}$ gives the mapping from states to common actions. Now we can use the similar notation as in the first chapter by using the bold symbols:

$$\begin{aligned}
\text{Trajectory :} \quad & \boldsymbol{\tau} = (s_0, \boldsymbol{a_0}, s_1, \boldsymbol{a_1}, ...) \\
\text{Return :} \quad & R(\boldsymbol{\tau}) = \sum_{t=0}^{\tau} r_t \\
\text{Probability of trajectory :} \quad & P(\boldsymbol{\tau}|\boldsymbol{\pi}) = \rho_0(s_0) \prod_{t=0}^{|\boldsymbol{\tau}|} P(s_{t+1}|s_t, \boldsymbol{a_t}) \boldsymbol{\pi}(\boldsymbol{a_t}|s_t) \\
\text{Expected return :} \quad & J(\boldsymbol{\pi}) = \int_{\boldsymbol{\tau}} P(\boldsymbol{\tau}|\boldsymbol{\pi}) R(\boldsymbol{\tau}) = \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \boldsymbol{\pi}} [R(\boldsymbol{\tau})]
\end{aligned}$$

Given the joint utility function, it is useful to think of the collection of agents as a single agent whose goal is to produce an optimal joint policy. The problem with treating MMDP as a standard MDP where actions are distributed lies in coordination. In general, there are several different optimal joint policies. However, even if all agents choose their individual policies according to some optimal policy, there is no guarantee that they all pick from the same optimal joint policy. Such a final joint policy may be nowhere near the optimal one, and most likely even produce significantly worse performance. In theory, there are two simple ways to ensure optimal coordination. First, there could exist a central control mechanism 3.3 that can compute the joint policy and then communicate the result actions to all individual agents. Or second, each agent can communicate its choice of individual policy to the others. However, neither of these approaches are feasible in the real application.

3.2.1 Nash Equilibria

Alternatively, we can look at the MMDP from the perspective of an n-person game. Then the problem of determining an optimal joint policy can be viewed as a problem of optimal equilibrium selection. A Nash equilibrium (Nash [1950]) for a $\boldsymbol{\pi}^*$ can be defined as:
A set of policies π^* is a Nash equilibrium if:

$$\forall i \in n, \forall \pi^i : J(\boldsymbol{\pi}^{*-i}, \pi^{*i}) \geq J(\boldsymbol{\pi}^{*-i}, \pi^i)$$

However, not all Nash equilibria are optimal joint policies, as some may have lower utility than others. This makes multi-agent environments more sensitive to convergence to local suboptimal policies, as the only way to escape such an equilibrium is through coordinated modifications of all individual policies.

3.3 Learning schemes

Centralized scheme

From a theoretical point of view, we could consider MMDP as an instance of single-agent MDP, where the goal would be to learn a central joint policy, which would then be distributed among individual agents. With this idea, we could solve MMDP problems using the same single-agent RL algorithms from the second chapter, only instead of learning a single policy, we would learn a joint policy. However, this approach has several shortcomings in real-world cases.

First, from a practical point of view, agents in this scenario would be somewhat passive entities whose job would only be to report perceived observations to some central authority. After all the observations have been collected by the central unit, a joint policy is constructed and distributed to the agents, who then blindly act as instructed by the central control. This has several serious problems. Imagine that some failure of the central mechanism occurs. Suddenly, the whole system of agents collapses because all agents lack individual autonomy.

Second, such an intensive communication between all agents and the central control mechanism may be too demanding or not even plausible for technical reasons.

Finally, from a theoretical point of view, the representation of such a joint policy grows exponentially with the number of agents with respect to both observations ($\prod_{i=0}^{n} |\mathcal{O}^i|$) and actions ($\prod_{i=0}^{n} |\mathcal{A}^i|$), which makes it unscalable.

Concurrent scheme

At the other end of the spectrum is the concurrent scheme. Here, all kinds of global information are omitted and agents rely solely on their local observations. The training of such an agent is then completely independent.

Centralized training with decentralized execution

And finally, somewhere in between the concurrent and centralized schemes, we can identify the so-called centralized training with decentralized execution. Here we consider the two life stages of agents. First, we train our agents in safe laboratory conditions, and only after the training is complete are they deployed in the real world.

During training, we can take advantage of the fact that we presumably have more information about the state of the environment at our disposal. Whether it is the true global state of the environment, local observations of other agents, or actions taken by others. All of this additional information can be incorporated into the training process to capture the true state to act upon. In other words, we want to make the training process as simple and accurate as possible.

Once the agents are deployed, they again depend solely on their local observations.

This learning scheme is often used in RL algorithms where both actors and critics are employed. For example, this is reflected in the PPO 2.5 algorithms mentioned above. During training, both actor and critic are trained in parallel, and the value estimate obtained by the critic can be used to provide more accurate information about the real value of the given state, making the actor's policy update more stable and accurate. Once training is complete, the agents, provided with local observations, are asked to take action based solely on the actor.

3.4 Non-stationarity

In addition to all the problems of partial observability, local equilibria, and policy distribution that have been mentioned so far there is another important problem that is not present in single-agent settings. Although we are able to reproduce the value function expressions:

$$
\begin{aligned}
V^{\boldsymbol{\pi}}(s) &= \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\boldsymbol{\pi}}[R(\boldsymbol{\tau})|s_0 = s] \\
Q^{\boldsymbol{\pi}}(s, \boldsymbol{a}) &= \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\boldsymbol{\pi}}[R(\boldsymbol{\tau})|s_0 = s, \boldsymbol{a_0} = \boldsymbol{a}] \\
A^{\boldsymbol{\pi}}(s, \boldsymbol{a}) &= Q^{\boldsymbol{\pi}}(s, \boldsymbol{a}) - V^{\boldsymbol{\pi}}(s)
\end{aligned}
$$

due to the non-stationarity problem, we cannot obtain the optimal value using Bellman equations as it was done in the single agent setting. Non-stationarity refers to the fact that changing policies of other agents as a consequence changes,

from a fixed agent's perspective, the state transition and the reward function of the environment. Using the idea of Bellman equations for optimality is still possible. However, in a multi-agent setting, we lose the theoretical basis that promises convergence to the optimal policy, and training using the Q-learning update rule must be accompanied by some mechanisms to overcome the non-stationarity problem.

3.5 RL algorithms

Having extended our mathematical model to settings that satisfy the conditions for a multi-agent environment, we can continue with a brief mention of the multi-agent variants of reinforcement learning algorithms discussed in the previous chapter.

MADDPG

The first of the algorithms mentioned is the Multi Agent Deep Deterministic Policy Gradient (MADDPG, Lowe et al. [2017]) algorithm. The authors extend DDPG (briefly mentioned in 8) by using centralized learning with decentralized execution. To address the problem of non-stationarity, the authors propose sampling from the ensemble of policies for each agent to obtain more robust multi-agent policies. This algorithm has been extensively experimented in academia with respect to multi-agent coordination environments.

MAPPO

Finally, we want to mention the recent success in the form of Multi Agent Proximal Policy Optimization (MAPPO, Yu et al. [2021]) variant of the PPO 2.5 algorithm. The authors revisit the use of PPO in multi-agent settings. In recent years, MADDPG has usually been the first choice of algorithm when dealing with multi-agent environments, leaving PPO ommitted. The authors hypothesize that this is due to two reasons: first, the belief that PPO is less sample-efficient than off-policy methods, and second, the fact that common implementation and hyperparameter tuning techniques when using PPO in single-agent settings often do not yield strong performance when transferred to multi-agent settings.

The authors propose five important changes with respect to adaptation to multi-agent settings.

- Value normalization:
  employing value normalization using Generalized Advantage Estimation (Schulman et al. [2015b])
- Input Representation to Value Function:
  enhanced global state with agent-specific features are used for critic, where such representation does not have substantially higher dimension
- Training Data Usage:
  using smaller amount of trainig epochs to avoid non-stationarity problem and not splitting data to mini-batches to make policy update more accurate

- PPO Clipping:
  maintain a clipping ratio ϵ under 0.2, within this range, tune ϵ as trade-off between training stability and fast convergence

- PPO Batch Size:
  utilize a large batch size

Using these five concepts, they were able to achieve comparable or superior results to off-policy algorithms on several benchmark frameworks without any other domain or structural modifications.

4. Overcooked environment

4.1 Overcooked game

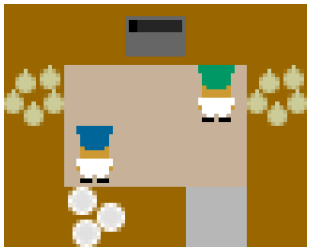
Before we get into our problems with cooperation, let us first examine the environment. We will work with an environment based on a popular cooking video game (Games [2016]). Overcooked is a multiplayer cooperative game where the goal is to work in a kitchen as a team with partner cooks and prepare various dishes together within a limited time. However, the game is highly dynamic. In many maps, the kitchen itself is not static and can change during a run. Random events such as pots catching fire add to the chaos. The challenge is to coordinate with the rest of the team and divide up subtasks efficiently.

The aforementioned game was simplified and reimplemented in a simpler environment (Carroll) to serve a purpose of scientific common ground for studying multi-agent cooperation in somehow complex settings. Many additional features of the original game were removed, leaving only essential coordination aspects. In its simplest form, the environment takes place in a small static kitchen layout, where the only available recipe is onion soup, which can be prepared by putting three onions into a pot and waiting for a given period of time. Somewhere in the kitchen there is an unlimited source of onions and a dish dispenser where the player can grab a dish to carry cooked onion soup to the counter. The team of cooks is rewarded as a team by an abstract reward of value 20 every time cooked soup is delivered to the counter. It may seem like a simple task. However, players face problems on several levels.

4.2 Basic layouts

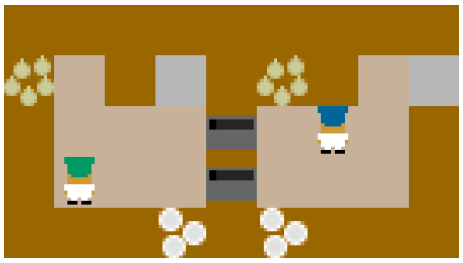
Although the Overcooked implementation has its own generator that can be used to generate new random kitchen layouts, most of the related scientific work to date has experimented with a fixed set of predefined layouts, each of which captures some important aspect of coordination.

Cramped Room



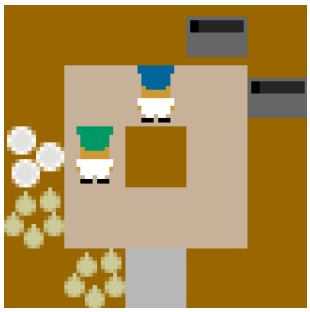
Cramped Room, as the name suggests, represents a cramped kitchen layout where all the important places are relatively easy to reach. The challenge lies in the low level of coordination of movement with the other partner, as there is no free space around.

Assymetric advantages



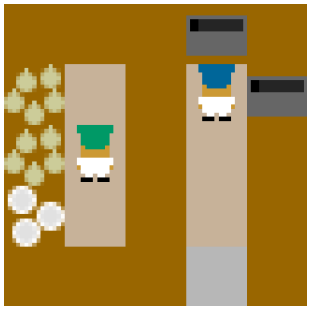
In Assymetric Advantage, both players are in separate regions where each region is completely self-sufficient. However, each region has better potential for a specific subtask. And only when both players make the best use of their own region's potential, the maximum joint efficiency is achieved.

Coordination Ring



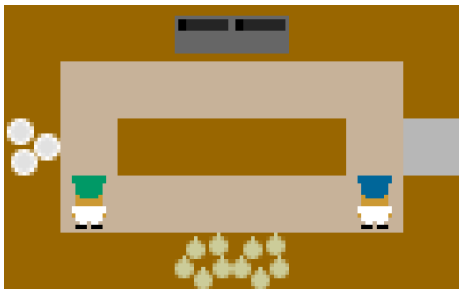
The Coordination Ring is another example of a layout that requires skillful coordination, as the only possible movement around the kitchen is along a narrow circular path that can be used in only one direction. For example, if one player chooses to move in a clockwise direction, the other player would automatically get stuck if they tried to move in a counterclockwise direction.

Forced Coordination



The Forced Coordination kitchen layout is quite different from the others. In this layout, each player is located in an isolated region where no player has all the resources necessary to prepare a complete onion soup alone. Thus, players are forced to cooperate with each other using the resources they have at their disposal.

Counter Circuit



In the last layout, the situation may look similar to the Coordination Ring. In this case, however, carrying onions around the entire kitchen is highly suboptimal no matter which direction the players choose. To deliver onions efficiently, players must pass them over the counter to shorten the distance. However, the cooks still need to decide who is responsible for bringing the plates.

4.3 Environment description

4.3.1 Action space

The action space is quite trivial, since it contains only the essential actions needed to operate in this environment.

- Go north
- Go south
- Go east
- Go west
- Stay
- Interact

4.3.2 State representation

There are two state representation functions predefined by the authors, namely `featurize_state` and `lossless_state_encoding`.

The first of the two extracts manually designed features into a one-dimensional vector of ones and zeros. These features can be interpreted as a partial observable representation of the environment state, as the majority of features are computed in relation to the nearest point of interest. For example, only the closest source locations of onions and dishes related to the player's current location are included, resulting in the loss of global information about the entire layout.

The second state representation, as the name suggests, provides lossless global information about the state of the environment. As in the previous case, environment variables are also represented by ones and zeros. However, unlike the previous function, the resulting representation is not a one-dimensional vector.

The result is formed by stacked masks, where each mask represents some feature of the environment in the form of a two-dimensional vector corresponding to the layout width and height. For instance, the mask representing the location of player i is a vector of shape $(width, height)$ filled with zeros at all positions except for the value one at the coordinates where player i is located. Similarly, the mask of the same shape representing onion sources is filled with zeros, and at all coordinates where there is an onion source in the environment, there are ones instead. We can see that this representation goes beyond the nearest locations and provides global information.

4.3.3 Rewards

As we said in the introduction, the environment is purely cooperative in the sense that players share the common reward of value 20 each time a soup is delivered to the counter location. And this reward is player independent. The cycle of the environment is virtually infinite, since there is no a priori end state. Thus, it would be theoretically easy to get a cumulative sum reward of infinity. This is prevented by setting a finite time horizon, which by convention is set to the limit of 400 steps. With this constraint, it makes sense to compare the results of different runs.

In addition to the general reward for delivering soup, there may be player-dependent partial rewards that are not accounted for in the total sum of rewards. These rewards are mainly used for learning purposes of the agents, since especially on some particular maps it is highly unlikely that by following the initially random policy the whole process of soup making including delivery will be completed. Using predefined partial rewards

- Dish pickup reward - Dish is picked in useful situation, e.g. pot is ready or cooking.
- Soup pickup reward
- Placement in pot reward - Useful ingredient is added to the pot, in our setting only onion soup recipe is used, so this corresponds to the action where onion was added to the pot.

allows agents to learn subtasks first. It is important to eventually ignore these partial rewards during training, as agents could focus on these subtasks and ignore the main goal thus failing the main task altogether. This is usually implemented by linearly decreasing the weight of partial rewards over some finite time horizon.

4.3.4 Initial state

As there are two functions predefined by the authors to represent the state of the environment, there are also two ways to create the initial state of the environment.

The first of the two uses fixed layout initial player locations, which always creates an identical initial state including both players' initial locations. However, learning a single agent by always placing it in the same initial location will likely cause it to fail when starting in the opposite location. For this reason, *player_index* is introduced to specify which agent is interpreted as player number

one and player number two. This index is randomized every time the environment is reset, ensuring that the agent encounters both starting locations during learning. Using this index introduces a minor chaos into the environment, as various parts of the environment control have to address the player index problem, and both actions and state observations have to be switched to the correct order, making it a bit opaque. We assume that this kind of initial state is suitable for some kind of benchmarking where the initial conditions are always the same.

In our experiments, however, we use the second approach to state initialization, where initial positions are always randomly sampled for both players. We claim that this is consistent with our intuition, since robustly cooperative agents should be able to cooperate well regardless of their initial position, which has been partially demonstrated (Knott et al. [2021]). However, this may come at the cost of reduced average performance. Besides random positions, the randomized initialization function also provides a probability threshold argument that can be used to randomly initialize some additional random objects in the environment.

5. Related work

A common approach to two-player games is to train an agent by confronting it with a set of other AI agents. This has been shown to perform impressively well against human experts in several complex games such as Dota (OpenAI [2019]) or Starcraft (Deepmind [2019]). The authors of the overcooked environment (Carroll et al. [2020]) believe that the distributional shift from AI training to human evaluation is successful due to the fact that the nature of this environment is strongly competitive. This is illustrated by the canonical case of a two-player zero-sum game. If humans take a branch in the search tree that is inadvertently suboptimal while in the role of minimizer, this only increases the outcome for the maximizer.

However, the situation is not so ideal in the case of common payoff games such as overcooked environment. If a self-play trained AI agent is paired with another sub-optimal partner, the result can be abysmal. In this case, both agents play a maximizing role in the search tree. If the self-play AI agent expects it's sub-optimal partner to be the same agent, it can choose branches that lead to maximum joint payoff. However, since the sub-optimal partner may behave sub-optimally, it may happen that it mistakenly chooses some worse branches that are unforeseen by the self-play agent, which may lead to a significantly worse result. However, since the common payoff is shared between the two agents, this is no longer an advantage for the self-play maximizing agent. Rather, it leads to failure for both agents.

5.1 Human cooperation

Most previous work in this area has focused on one of two types of coordination. The first is coordination between a human and an AI partner. And second, focusing solely on the fully AI-driven pair.

While perfect AI-human coordination is generally a more desirable goal to achieve in all sorts of domains, it will not be our main focus. Several previous scientific papers have addressed this issue. A particularly noteworthy contribution is the paper On the Utility of Learning about Humans for Human-AI Coordination (Carroll et al. [2020]). They collected several human-human episodes and incorporated these experiences into training. This human data was used to create human-based models using behavior cloning and Generative Adversarial Imitation Learning (Ho and Ermon [2016]) and then incorporated into training. One of the important conclusions was that when self-play and population-based agents were paired with human models, the overall results were substantially worse than when paired with agents designed to play with human models.

5.2 AI-AI cooperation

We have already mentioned the problem when a self-play agent is paired with a suboptimal (e.g. human) partner. However, a similar problem can also occur when two different agents trained in self-play mode are paired together. A com-

mon approach used in competitive settings is to introduce diversity by exposing the trained agent to a diverse set of partners during the training phase. There are several popular methods for partner selection during training.

5.2.1 Self-play

The self-play method has already been mentioned, and it is arguably the simplest and most straightforward method for partner sampling. As the name indicates, the idea of this approach is that the agent currently being learned is paired with the copy of itself during training. This method doesn't introduce any kind of diversity into the learning process, as it only learns from its own current policy.

5.2.2 Partner Sampling

Partner sampling is an extension of the previous method. However, instead of playing only with the current policy, the partner is sampled from previously periodically saved policies of this given training period. The diversity in this method comes from the theoretical point of view that previous policies correspond to different behaviors.

5.2.3 Population-Based Training

The population-based training method is based on an evolutionary algorithm that focuses on hyperparameter training and model selection. The population consists of agents parameterized by a neural network and trained by a DRL algorithm. During each iteration, agents are drawn from the population and trained using the collected trajectories. Pair-wise results are recorded, and at the end of the iteration, the worst agents are replaced by a copy of the best agents with their hyperparameters mutated.

5.2.4 Pre-trained Partners

Finally, a often employed method is to employ pre-trained partners. Diversity is expected here due to the fact that different runs of reinforcement learning algorithms often yield different agent behaviors.

5.2.5 Result

The problem with fully cooperative games is that when agents are trained together, they tend to exploit the common knowledge they have learned during training, which often makes them unable to cooperate with unseen agents. It has been shown (Charakorn et al. [2020]) that this is true for the first three methods mentioned, where, when performing a cross-play evaluation of the set of different agents obtained by the same method, only those pairs of agents that were explicitly trained together performed well, while the remaining pairs failed.

Nemam si tady od nich z toho clanku pujcit primo ty obrazky ukazujici ten SP off-diagonal failure, abych se na to pak mohl vizualne odkazat ve svych experimentech? Nebo krast obrazky se nedela? The authors conclude with results

showing that incorporating different pre-trained agents for training robust agents was significantly more successful than other methods.

5.3 Problem of robustness

Another important issue related to common payoff is evaluating the performance of trained agents. As discussed earlier, we cannot rely on training performance alone, no matter what metric is chosen, because training sets of agents exploit shared knowledge and thus make training performance appear excellent. As noble as it sounds, it is quite complicated to define which agent behavior is actually robust. Obviously, it's a desirable goal to create a trained agent that can cooperate well with all possible behaviors of its partner. However, since it is difficult to come up with a diverse set of agents for partner sampling during training, the same is true for evaluation agents.

5.3.1 Average performance

The first obvious choice of evaluation metric could be to look at the average evaluation performance over the evaluation set of agents. It is questionable, though, whether this answers the question of robustness. In one scenario, we could theoretically achieve a result where our agent fully fails to cooperate with half of the evaluation agents, while performing excellently with the other half. In another case, we could achieve some kind of intermediate performance with the whole set of evaluators, where cooperation does not fail completely, but performance is also nowhere near the optimal values. In both cases, the average evaluation performance reaches similar values.

5.3.2 Threshold performance

Another possible approach for the given layout could be to define our own threshold, which could be practically equivalent to a certain number of delivered soups. The evaluation metric could then simply be expressed in terms of the number of evaluation agents that managed to reach such a threshold when paired with our trained agent.

5.3.3 Edge case testing

The problem with previous metrics is that they are reasonable only in a situation where the agent's evaluation set is complex enough in terms of behavioral diversity, which also implies a condition on the sufficient size of such a set. However, creating an evaluation set that satisfies this condition is practically impossible.

Instead of trying to construct such an evaluation set and trying to evaluate whole episodes in an effort to test cooperation on the run, we could break the interesting cooperation-challenging situation into a number of separate tests (Knott et al. [2021]). Suddenly, our thinking about robustness shifts from looking at cooperation from the point of view of the outcome of the entire episode to just separate small modular situations where the desired correct behavior of both agents can be described. This approach is inspired by unit testing in software

development, where even if the program behaves correctly in the vast majority of cases, it can still produce undesirable responses in edge cases.

Similar to unit testing, we can define a wide set of situations (e.g., instances of overcooked environment states) and pair them with a set of acceptable consequent behaviors of agents that can be labeled as robust response. Evaluation using such a metric can then be expressed as the number of tests that have passed.

However, there are several challenges with this metric as these edge cases require manual design, in many scenarios it is also difficult to know with certainty what the correct expected behavior is, and finally, similar to unit testing in software development, there are never enough defined edge cases to cover all possible situations.

6. Our work - Preparation

Before we get into our own experiments with training a robust agent, which will be the main focus of the next chapter, we want to prepare our solution first and reimplement some of the previously mentioned results regarding self-play agents.

6.1 Framework

Writing all of our code on top of the overcooked environment from scratch is not necessary, as there are already several frameworks that implement deep reinforcement learning algorithms. Although in general there are many more DRL frameworks that could be suitable for our purposes, most related projects using the overcooked environment have so far used either the RLib library or Stable Baselines.

As the authors themselves say, RLlib provides support for production-level and highly distributed RL workloads. This framework is also suitable for multi-agent learning. The downside here is that it is less suitable for smaller projects and development on a local machine, as there is a fair amount of overhead due to it's parallelization capabilities towards cluster computing. Personally, I found the framework a bit intimidating and the documentation a bit confusing.

Another option was the Stable Baselines framework, which in my opinion offers documentation that is clearer and easier to understand. Also, in my humble opinion, the code base structure is more transparent and basic API is quite light. To demonstrate its simplicity, once you have an environment that implements the standard RL OpenAI environment interface, you can perform all training with default hyperparameters and policy represented by a multi-layer perceptron network using the PPO algorithm as a simple as follows:

```
env = make_vec_env("CartPole-v1", n_envs=4)
model = PPO("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=25000)
```

The downside here is that the framework does not have native support for simultaneous multi-agent learning.

6.1.1 Modifications

After considering both frameworks and the scope of our project, we decided to build our project within the Stable Baselines framework. We will not cover all the details of our diverse population approach, as there is plenty of room for this in the next chapter. Here, we just want to mention some of the major framework modifications that had to be carried out in order to bend the framework to suit our purposes.

Embedding partner

First, as mentioned above, the Stable Baselines framework does not support multi-agent learning. In our approach, we will make a slight simplification by theoretically transforming the environment from a multi-agent to a single-agent

setting. Of course, we will not reduce the number of agents in the environment, as it will still be multi-agent. Rather, we will look at the situation from the point of view of the single agent that is learning. In this way, we can look at the environment as if the partner cook was only embedded as part of the system.

And exactly the same change is necessary in the framework. During a training run, the partner is embedded as part of the environment, and at each step of the loop, the actions of the partner are sampled. The actions of the trained agent are sampled in the usual way, as expected, while the actions of the partner cook are sampled according to the policy of the embedded partner. The resulting two actions are concatenated and passed to the environment as expected. This mechanism allows us to use any partner sampling method (5.2), including self-play, where the same instance of the parameterized policy is used as the one being learned.

Convolution policy

By default, Stable Baselines provides several types of parameterization policy representations, including fully connected dense multi-layer perceptron (MLP) and convolutional neural network (CNN). However, regarding the CNN representation, when using this wrapper, the framework expects the inputs to be strictly in a standard image format (RGB, RGBD, GrayScale). Unfortunately, this is accompanied by the use of inadequate assumptions throughout the code base, heuristically expecting the inputs to be in a certain format. For example, the function to detect if the image space is in the first format only checks if the first dimension is the smallest. This erroneously returns true for some overcooked layouts using lossless state representation (4.3.2) where width is less than both height and definitely the number of stacked masks. In lossless encoding, masks are equivalent to channels and are represented by the last dimension. Authors might argue that with such a state representation, the inputs are technically not images, and therefore such sanity checks expecting images should not be used in the first place. However, this is the problematic part as I was unable to find out whether the framework allows some options to bypass these constraints. Eventually, after disabling these assertions in certain critical places, the code managed to work flawlessly with our convolutional representation without any further modifications.

Loss and rewards augmentation

Finally, two important aspects of our experiments will revolve around augmenting rewards and extending the objective loss function. These modifications are not truly related to Stable Baselines specifically, as these would have to be made regardless of the framework chosen, we just want to mention them here to make the list of changes complete. Fortunately, these modifications were also quite straightforward, as both of these elements are nicely separated in transparent, easily extensible locations.

6.2 Self-play

Before we dive into our experiments, we want to make sure that our solution is ready to be used with the environment. As a reasonable setup, we'll start by reimplementing basic agent training using self-play (5.2.1) and try to demonstrate its allegedly poor cooperation abilities.

Default setting

We started with the simplest default settings. For initial experiments we chose to start with the first layout Cramped Room (4.2) using the more lightweight of the two, partially observable state representation (4.3.2). As we recall, this representation uses a one-dimensional vector of features that is out-of-the-box compatible with prefabricated Stable Baselines MLPPolicy wrapper, which contains two shared hidden layers before separate layers for actor and critic are applied. Despite several attempts to tweak the hyperparameters, we were not able to come close to the reported results of cooperation failure (5.2.5). In all of our experiments, different self-playing agents managed to cooperate reasonably well. While there were some pairs of agents that did not cooperate at all, this was rather rare compared to the overall cross-play evaluation (Figure 6.1). However, we believe that the problem of uncooperative behavior may be exacerbated by this evaluation scheme in some layouts more than others, since different layouts present different challenges to cooperative behavior. To make matters worse, while we were able to obtain similar cooperative behavior in the Asymmetric Advantage and Coordination Ring layouts, the same training procedure failed completely in the remaining Forced Coordination and Counter Circuit layouts. Here, the self-play training procedure was unable to learn how to perform the main task at all.

Lossless state representation

To both overcome this problem and make our experiments as relatable as can be to previous related work, we tried to adapt our solution as closely as possible to the settings used in previous work. We modified our solution based on the original overcooked environment project (Carroll et al. [2020]), whose specific settings are described in detail in Appendix A. Instead of the partially observable state representation, the global lossless representation was adopted. This implied the need to change the policy parameterization as well. The Stable Baselines framework provides a prefabricated convolutional CNN policy wrapper. However, the convolutional part of the network had to be manually modified to match the described structure. mam se poustet do detailniho popisu zmen, nebo staci ze jsem zminil appendix A, kde to lze dohledat? In addition, several PPO hyperparameters have been changed compared to the default stable baseline PPO setting.

Hyperparameters search

With these settings adjusted, we attempted to run the self-play agent training on all layouts as before. On the basic Cramped room layout, the training seemed

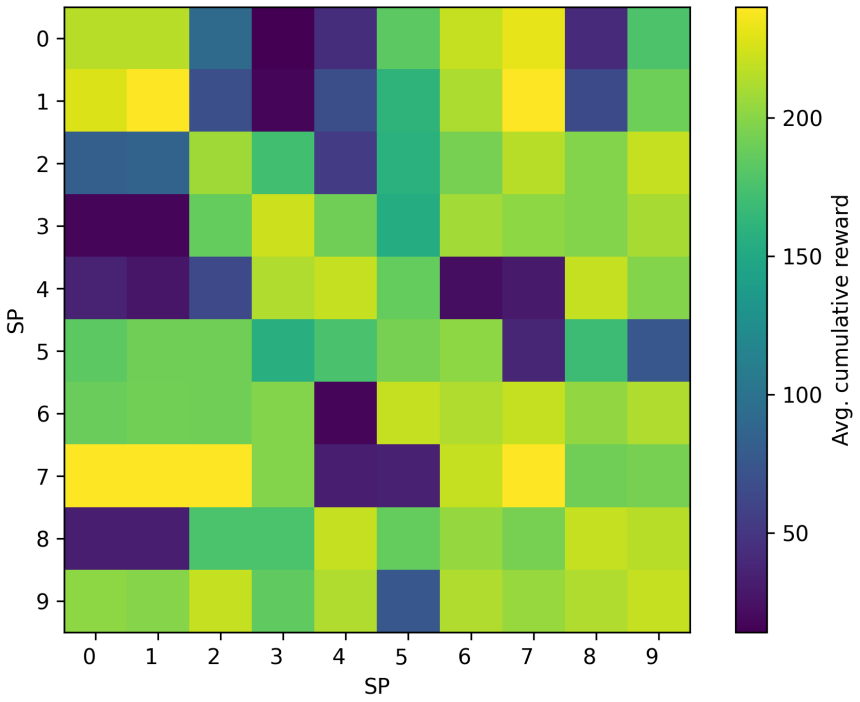


Figure 6.1: Self-play MLP cross play evaluation

Cross-play evaluation of 10 agents trained via self-play method on cramped_room layout, where policy is parametrized by MLP

to progress successfully. However, when cross-play was evaluated, similar pairwise cooperative behavior was achieved as in the previous state representation. However, training failed on all other layouts.

We performed hyperparameter search by running the training process with several hundred randomly different hyperparameters to find a suitable stable configuration. With the resulting configuration (table 6.1) we were able to train a self-play agent on all layouts in most of the runs.

Preemptive divergent check

Unfortunately, even this configuration is not completely stable. In some runs, training on more complex layouts such as Forced Coordination and Counter Circuit fails to achieve the main goal. In these cases, the agents always learn how to successfully acquire partial rewards by performing the subtasks, but fail to learn the final soup delivery. We suspect that the combination of layout complexity and likely policy exploitation prevents the agent from experiencing the last part of the task as often as necessary to adapt such behavior. If this is the case, it is highly unlikely that the agent will discover the soup delivery behavior once the time steps reach the partial reward shaping horizon. To account for this fact, we perform a preemptive divergence check at the $3 \cdot 10^6$ time step, where we measure the average number of soups delivered during training. We have heuristically found that the average soup reward threshold of 3 over the collected episodes is sufficient to ensure that the agent will learn the main goal in the remaining training process.

Name	Originally proposed	**Applied**
PPO Hyperparameters		
Discount factor γ	0.99	0.98
GAE factor λ	0.98	0.95
Learning rate	10^{-3}	$4 \cdot 10^{-4}$
VF coefficient	0.5	0.1
PPO clipping	0.05	0.1
Maximum gradient norm	0.1	0.3
Gradient steps per minibatch	8	8
Minibatch size	2000	2000
Number of parallel environments	30	30
Total timestemps	$6 \cdot 10^6$	$5.5 \cdot 10^6$
Entropy bonus start coefficient	0.1	0.1
Entropy bonus end coefficient	0.1	0.03
Entropy bonus horizon	none	$1.5 \cdot 10^6$
Preemptive divergent time step	$3 \cdot 10^6$	$3 \cdot 10^6$
Environment settings		
Episode length	400	400
Soup delivery shared reward	20	20
Dish pickup partial reward	3	3
Onion pot placement partial reward	3	3
Soup pickup partial reward	5	5
Partial reward shaping horizon	$2.5 \cdot 10^6$	$2.5 \cdot 10^6$
Initial state player locations	static	random

Table 6.1: Hyperparameters for self-play agent training on Cramped room layout

6.2.1 Random state initialization

Despite these adjustments, there remained problems on two particular layouts. On the Asymmetric Advantages layout, the resulting performance of the trained agent was significantly worse than in the presented results (Carroll et al. [2020]). Whereas on the Forced Coordination layout, training converged to zero performance suspiciously often. We found that there was a missing constraint on the sampled initial positions of the agents, which allowed the initial positions to be in the same isolated layout region. While this is not a problem for the remaining layouts, it is an important obstacle for the aforementioned maps.

In both Asymmetric Advantages and Forced Coordination layouts, not all locations are globally reachable. As a result, in Assymetric Advantages, the agent was forced to learn how to operate within its own region, rather than relying on beneficial cooperation from its partner using its region advantage. And this led to a significantly lower performance value.

In the Forced Coordination layout, the situation was even worse, because if both players started in the same isolated region, they would not be able to finish the soup at all.

Lastly, the authors (Knott et al. [2021]) believe that introducing random initial states can lead to better robustness, which could be an explanation for our better measured cross-play cooperation. Nevertheless, we ran the same training with fixed initial positions and the results were moreless the same.

6.2.2 Focus on Forced coordination

With all the observations we have made thus far, we can state that there are vast differences between the different layouts. Therefore, for our subsequent experiments, we are primarily limiting our attention to one particular layout. We believe that the Forced Coordination layout provides a good balance between complexity and also emphasizes the importance of cooperation, which is the main point of our interest. Furthermore, this layout shows signs of a cooperation problem that resembles the reported result (5.2.5) when different self-play agents are evaluated using cross-play (6.2).

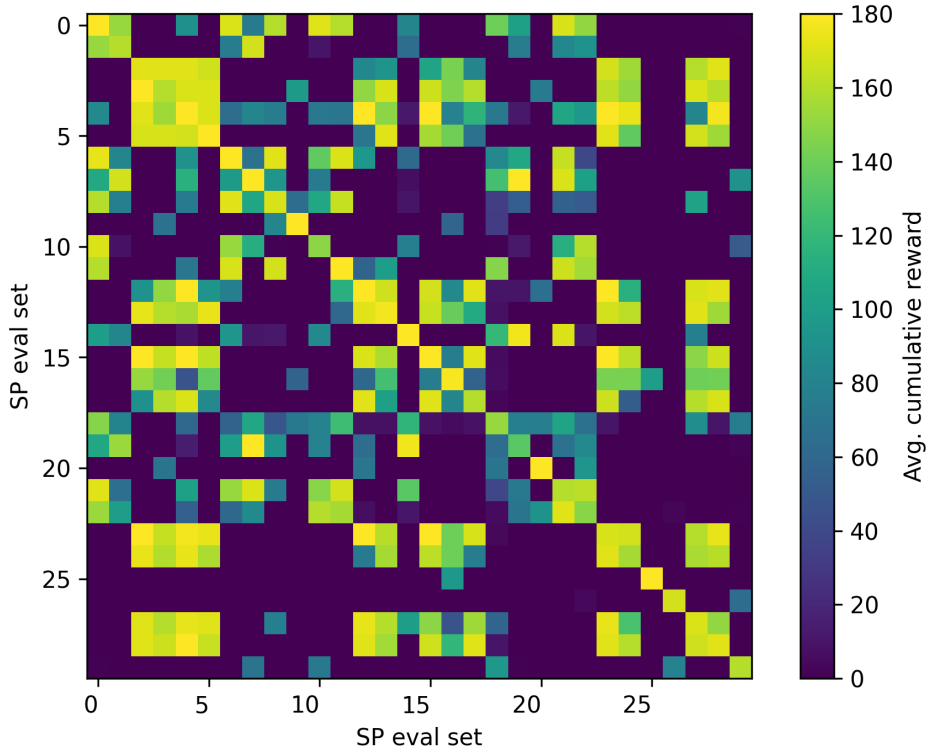


Figure 6.2: Self-play CNN cross play evaluation on Forced coordination

Cross-play evaluation of 30 agents trained via self-play method on Forced coordination layout, where policy is parametrized by convolutional network

7. Our work - Contribution

In this chapter, we propose a novel approach to introduce explicit diversification into agent behavior by training diverse populations. The traditional population-based methods mentioned above were not effective enough to add the necessary diversification to the population. We try to use population training in a slightly different way, focusing on two important aspects. First, we try to address well-known problems associated with multi-agent systems by changing our perspective to single-agent settings and transforming to an iterative process. And second, we try to address the lack of explicit diversification in standard population-based learning methods by pursuing population policy diversification.

In the following sections, we describe several different aspects of our population building approach. These parts can be seen as a soft guide to the diverse population training algorithm that will be presented at the end of this chapter.

7.1 Multi-agent setting simplification

7.1.1 Single-agent perspective

As we have already seen (chapter 3), multi-agent systems can be much more intimidating to tackle. One of the suggestions was to try to simplify the situation by looking at the multi-agent environment from a single-agent point of view, where the goal is to learn a common policy. As discussed above, this is often impossible and mostly impractical.

Nevertheless, in a sense, we try to use a similarly motivated approach. We propose a single-agent learning approach, where only one agent is learned at a time. Of course, we cannot simply ommit the other agent from the environment because we cannot deny the multi-agent nature of the environment. Rather, we consider the environment as an instance of a single-agent environment in which the partner is embedded and in some sense forms an integral part of the system.

7.1.2 Non-stationarity

We propose population training as an iterative process where only one agent is trained and only after its training is complete is it added to the population. Once an agent is added to the population, it's policy is fixed and no further updates are applied for the rest of the population training process. Only agents from the already trained population are used in the environment embedding. We claim that this effectively solves the non-stationarity problem we encountered in multi-agent settings, since from the point of agent learning, the behavior of its partners never changes.

Unfortunately, there is a downside to this approach. While before the environment had a purely deterministic state transition function, now with the partner embedded in the system it becomes stochastic. However, we hypothesize that this may also be beneficial for the final robustness, since by experiencing stochastic state transitions, the agent could theoretically be more robust as a result, since it will have to learn how to behave in a more stochastic exploratory environment.

7.1.3 Population exploitation

We hypothesize that by using an iterative process where past agents are fixed, it could also help with population exploitation. If a set of agents is trained together at the same time, they can all exploit their behavior towards the shared experience. It is likely that the new agents will still try to overfit their behavior to the population agents. However, by fixing the previous past agents, we try to break this relationship from at least one side of the pair. We are optimistic that this, combined with other diversification factors, will greatly reduce exploitation of the population.

7.2 Population building

A mere change in our perception of single-agent settings would probably not be enough to improve agent robustness. Therefore, we propose several ideas to promote population diversification.

7.2.1 Inicialization

First, inspired by the original motivation behind all population-based methods, we follow the idea that populations in general should always be more diverse than a single agent. Thus, when utilized in training, this diverse robustness should be passed on to the next learned agent. In order for this to be true, the previous condition about the non-existence of population exploitation must hold. Additionally, we have already proposed an iterative training process where we assume the pre-existence of a set of agents used for partner embedding.

We propose that in order for other diversification factors to work, we need to start with an initial population for partner embedding sampling that is already somewhat diverse. One could argue that if we had a method for creating such a diverse initial population, we could just stop there and expand that population using such a method. However, we do not consider perfect diversity and robustness of the initial population to be critical. Rather, we expect the initial population to introduce a slightly different set of behaviors just to provide a reasonable starting point for the rest of the population training.

7.2.2 KL divergence

We hypothesize that the pre-existence of a diverse population is only a soft constraint. We propose that the main ingredient behind our diversity building process lies in attempting to shift the behavior of each successively trained agent to be as different as possible from the behaviors in the existing population. Since the sampled population partners are embedded in the environment, we can think of this approach as a domain randomization technique.

To emphasize the difference between two behaviors, we must first be able to measure it. There are arguably several possible ways to express the difference between two behaviors, including methods discussed in evaluation approaches (5.3). However, since the agent's behavior is represented by a parameterized

policy, we suppose that one of the more straightforward methods for low-level difference might be to look at the difference between the two policy distributions.

The Kullback-Leibler divergence (KLD) often comes to mind when measuring the distance between probability distributions.

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log \left(\frac{P(x)}{Q(x)} \right)$$

Although it is not a metric since it is not symmetric in the two distributions and it does not satisfy the triangle inequality, we can still utilize it since we mostly are interested in relative distribution differences rather than absolute values. We propose the idea of incorporating the maximization of the KL divergence as a tool to differentiate new agents from those already contained in the population by taking the average of the KL divergence of the trained policy and the policies from the population.

We propose that there are two different appropriate ways to incorporate KL divergence maximization into the process. First, KL divergence between the currently trained policy and the policies in the population can be used during episode sampling by interpreting it as another additional partial reward based on a current state of the environment. And second, a similar approach can be applied at the level of the PPO objective function, where another additional term based on maximizing the KL divergence can be added.

It may seem that these two approaches have the same effect. However, we believe they work in slightly different ways. When applied in the PPO objective, the term attempts to differentiate current policy regardless of the current state of the environment. While applying the KL divergence bonus as a partial reward based on state might work as an exploration-enhancing technique since it should push the agent to visit more often the states where its current policy differs the most, potentially leading to state trajectories that agents in the population have not seen.

KL divergence coefficients

Applying the absolute value of the KL divergence, as calculated by the formula, would not work in our favor, since we have to consider the value ranges with respect to the environmental properties. The KL divergence is not negative, but it can easily reach infinity (jde to od nuly do nekonecna) if it is not bounded. Additionally, we need to consider a suitable multiplicative coefficient to scale the values to a reasonable range.

When a new parameterized policy p is initialized, it produces the probability distribution that is close to uniform. After sampling several already trained self-play agents and evaluating them in the same initial states, we obtained specialized policies yielding the probability distributions q_1, q_2, q_3 (see table 7.1).

These values of KL divergence can give us an idea of the absolute values when new policies are compared to already trained agents. We believe that the probability distributions p and q_1 are different enough in terms of what we are trying to achieve when training a population of diverse agents. By considering the absolute value of the KL divergence of these two distributions, we try to derive appropriate KL divergence coefficients for loss and reward augmentation.

	Probability distribution	$D_{KL}(p\|\|q)$
p	$[0.1679, 0.1647, 0.1644, 0.1663, 0.1713, 0.1654]$	
q_1	$[0.0336, 0.0729, 0.0467, 0.5419, 0.1431, 0.1617]$	0.449
q_2	$[0.0453, 0.1906, 0.1270, 0.2221, 0.0942, 0.3207]$	0.183
q_3	$[0.1875, 0.1851, 0.1663, 0.1369, 0.1414, 0.1828]$	0.009

Table 7.1: KL divergence values ilustrated on trained policies on same random inital state

We hypothesize that the KL divergence term, which we add to the PPO objective, should have comparable weight to the entropy bonus (recall 2.5), as this bonus is also designed to be a mild exploratory pressure rather than a main objective. Entropy in the early stages of training corresponds to a uniform probability distribution, yielding absolute values of 1.8. This combined with the initial entropy bonus coefficient of 0.1 (hyperparameter table 6.1) yields absolute values of cca 0.18. In the final stage, when the policy is already trained, the entropy usually decreases to absolute values of about 1.2, which combined with the entropy bonus end coefficient of 0.03 produces a final value of 0.036.

As far as reward augmentation is considered, we believe that when all KL divergence bonuses are summed over the 400 steps of the episode, they should correspond to only a fraction of what can be obtained by following the main objective of soup delivery. We propose to design such reward clipping in equivalence to the number of soups delivered (0.5, 1, 1.5 (R0, R1, R2, respectively)).

After performing some trivial experiments, we came up with the following experiment settings (table 7.2), which we want to evaluate in our following experiments.

Experiment Name	Reward coefficent	Reward clip	Loss coefficent	Loss clip
No divers.	0.0	0.0	0.0	0.0
R0	0.08	0.025	0	0
R1	0.15	0.05	0	0
R2	0.1	0.075	0	0
L0	0	0	0.08	0.03
L1	0	0	0.12	0.07
L2	0	0	0.1	0.15
R0L0	0.08	0.02	0.08	0.02
R1L1	0.1	0.04	0.1	0.03

Table 7.2: Summary of experiments utilizing KL divergence

7.2.3 Population structure

Finally, we need to provide a description of the entire proposed population building process. We have already dealt with population initialization, where we proposed how the initial agents could be constructed, and we also designed diversification tools to enhance the behavioral diversity of the population.

However, our ultimate goal is to provide a procedure that produces a final agent or set of final agents that should be robust and highly cooperative. By applying diversification techniques, we effectively train agents that do not fully optimize towards the main objective, but their performance may be degraded due to diversification.

After training several diversified population agents, we propose to train final agents by exposing them to the already built population, but without the diversification pressure. We propose to train the first final agent by exposing it to a population that includes the agents that were part of the initialization. and then train the second agent with the same population, but without the initial agents.

To ensure that the final agents have enough space to learn how to cooperate with the entire population, we increase the number of training time steps by a factor of 1.5.

7.3 Simple convolution experiments

Initial experiments visualised

After the initial population training experiment (Figure 7.1), we can see that the first three self-playing agents used to initialize the population play an important role. In three seeds (10,11,13) the initial population seems to represent strongly the same one set of behaviors and lacks diversity towards other behaviors. It seems that as population training progresses, agents learn to fixate only on this type of behavior, as it promises maximum available outcome, and ignore other types of behaviors.

To address this issue, we propose two suggestions. First, in these initial experiments, the first three trained self-play agents were automatically selected for population initialization. Instead, we propose to perform the initial population agent selection manually by first training five self-play agents, evaluating them with any kind of evaluation method (in our case evaluating them with another set of self-play agents), and finally selecting from these five such TODO: proc ne? three agents that are both successful and also pairwise different as much as possible.

And second, during training, all available agents from the current population were present at each training epoch, as they were equally distributed across 30 parallel environments. This guarantees that the dominant type of behavior from the population initialization was always present during training, which allowed newly trained agents to move towards this type of behavior. Alternatively, we propose not to sample all agents from the current population, but to sample only a few to give a chance to omit the dominant ones and allow the trained agent to focus also on the remaining types. After performing a simplified experiment, we concluded with a suitable value of three for the number of sampled partners.

Average outcome evaluation

As discussed in previous chapters (section 5.3), it is not very intuitive to perform the evaluation of the obtained population using the usual methods. When the overall cross-play average outcome metric is used (Figure 7.2), there are only

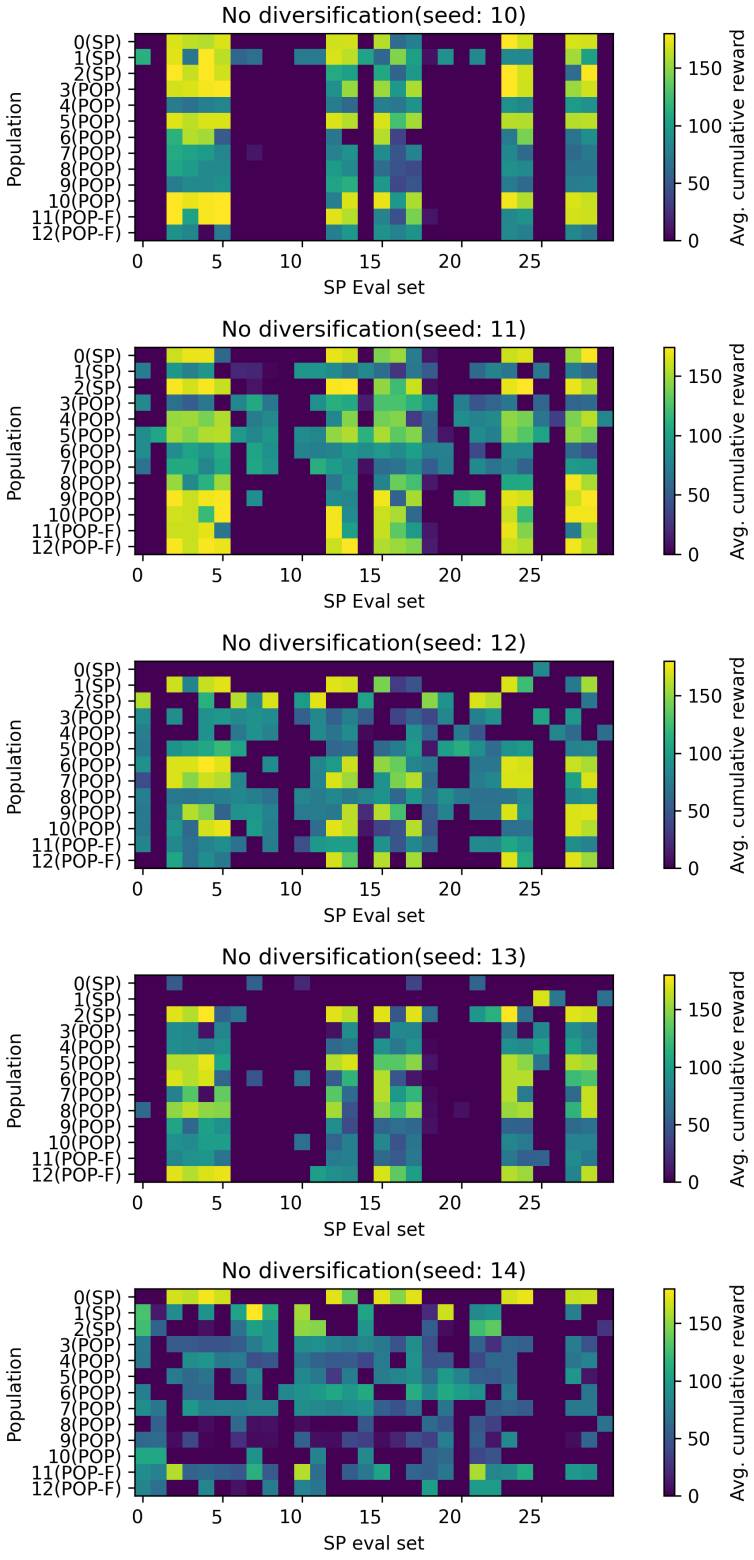


Figure 7.1: Diversity free populations cross-play evaluation

Cross play evaluation of five populations trained with no diversification pressures against self-play agents

slight improvements over self-play agents, but the results vary widely, which we

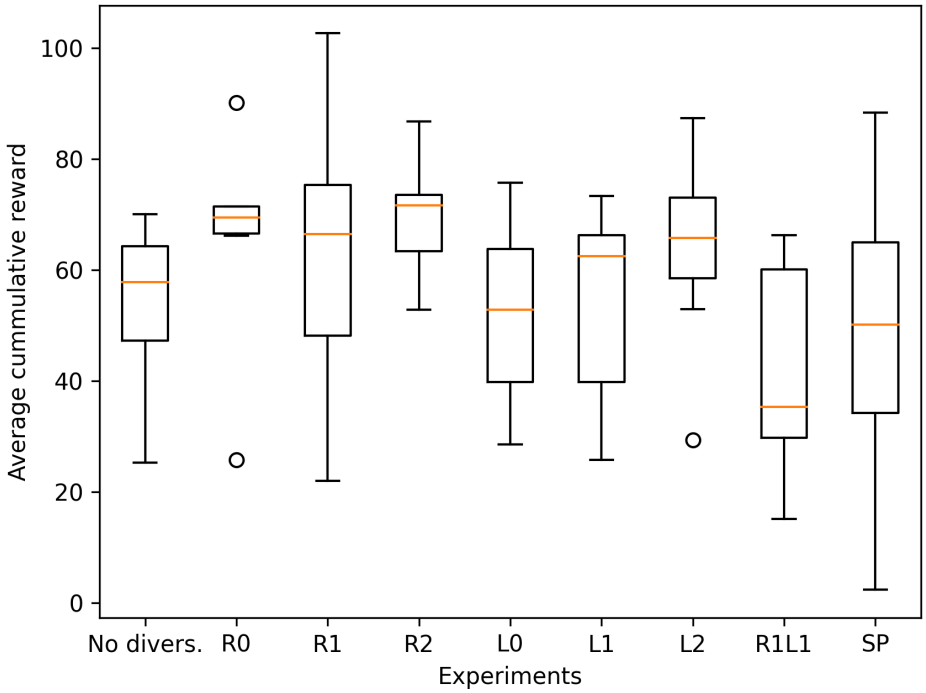


Figure 7.2: Average cummulative reward of final agents

Average cummulative reward of the two final agents trained in last phase of the population building. Includes values also for cross-play evaluation of self-play evaluation set for reference.

believe is due to two factors. First, as we just discussed, the overall cross-play average score of agents in the population initializations also varies a lot between different seeds. And second, even when the final population agents are trained, there is no guarantee that they will not end up in some suboptimal local optima with respect to the current population.

Ordered average outcome evaluation

We propose to use a different metric to evaluate the final population of agents. After evaluating the cross-play results, we suggest to order them in ascending order. By using this ordering, we lose the information about the variety of specific pairwise cooperation in terms of whether different agents managed to cooperate with different set of evaluation agents. On the other hand, this gives us an idea of how many evaluation agents our evaluated agents are able to cooperate with at a specific cooperation level. For this evaluation metric, the best of the two final population agents is used.

Using this evaluation metric and averaging over the results of three different seeds, we can see an evident difference in the cooperation curve (see figure 7.3). Looking at the cooperation level with an outcome value of 20, we can see that all agents from all different experiment settings managed to cooperate with a much larger set of evaluation self-play agents. Similar results can be seen at the cooperation level with a value of about 120. The most important change in

the curve progression can be seen in the last part of the ordered results, where the self-play agents start to catch up and overcome the results of the diversified population agents. We believe that this result is expected, since the final part of the results corresponds to the pairwise result of self-play agents with similar learned strategy, which should lead to a strong cooperative result.

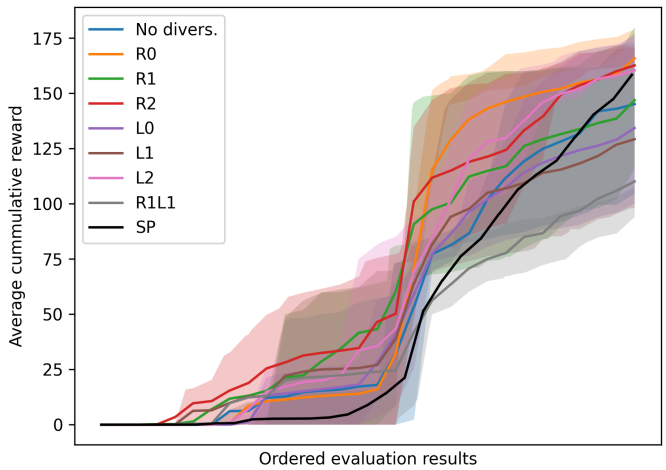


Figure 7.3: Average, ordered cross-play outcomes

Average of ordered results of cross-play evaluation of best population final agents against self-play agents. Includes values also for cross-play evaluation of self-play evaluation agent set for reference, where same agent pairs are ommited. (Average, 0.25Q, 0.75Q)

Another interesting result can be seen if we apply the same metric, but instead of considering the average result, we consider the 0.15 quantile value. By lowering the quantile, we are looking at the worse case cooperation results obtained on different seeds. Looking at the visualized results (Figure 7.4), we can see that in the worse case there were far fewer self-play agents that managed to cooperate with other agents at a cooperation level value of at least 80. This result can be seen as an argument for the cooperative capabilities of our approach, since it shows greater robustness with other agents at lower, but reasonable, cooperation levels.

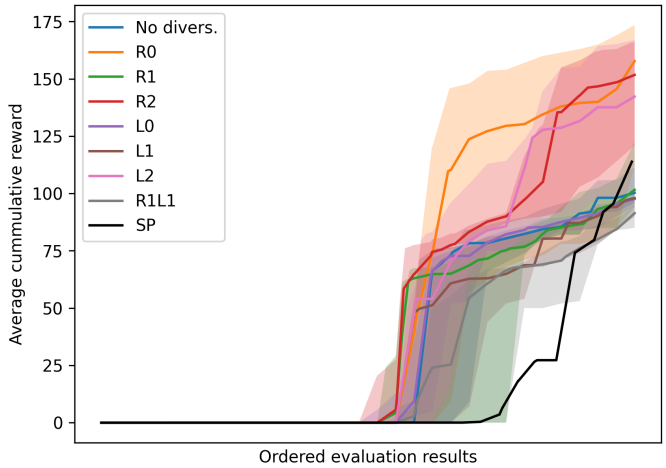


Figure 7.4: 0.15 Quantile, ordered cross-play outcomes

0.15 quantile of ordered results of cross-play evaluation of best population final agents against self-play agents. Includes values also for cross-play evaluation of self-play evaluation agent set for reference, where same agent pairs are ommited. 0.15Q, Min, 0.3Q

7.4 Frame stacking

After performing initial experiments and observing visualized pair-wise results with a set of self-play agents, we often witnessed final agent results that were both unable to cooperate well with the dominant behavior of population initialization agents, but also unable to learn how to cooperate with the remaining behavior types. As a result, we questioned whether the potential capabilities of our agent architecture were not limited.

So far, we have tried to expose our new agent to various potentially different types of behaviors represented by the previous population agents, in an attempt to teach it to be capable of cooperating with each of them. In other words, we are trying to learn a policy that responds best to all previous policies.

However, we claim that this might not be possible if the agent has to decide its policy based on a single current state of the environment only. To illustrate a simple counterexample, we can think of any kind of situation where agents get stuck by repeating the same actions, resulting in the same state as before. By considering only such a particular state, we cannot be certain whether agents are stuck since we do not know what action our partner will take. However, if we expect our partner to do an action that, when combined with our action, will lead to a non-stuck state, while our agent may expect us to try to do the same, then we are effectively stuck.

Similarly, we believe that our partner's understanding of the current high-level goal can be significantly improved if its several previous steps are available.

To solve this problem, we propose that using some kind of temporal features about previous states is necessary. We try to propose two slightly different modifications incorporating temporal features that we will try to use in the rest of

our experiments. While we could arguably wish to preserve the entire history of our partner's behavior, we believe that this leads to a significantly more difficult problem where we are trying to model our partner at a much higher level of complexity. We seek to find a middle ground by considering only a limited number. Inspired by similar approach applied in successful Atari AI project (Mnih et al. [2015]), we assume that considering the last four states for temporal features should be sufficient for our goal.

7.4.1 Channels

The first of our two approaches is based on extending the global lossless state representation (see section 4.3.2). The 22 channels in this representation can generally be divided into three groups. The first group consists of constant values describing the physical properties of the given kitchen layout. This includes layout terrain, source locations of onions and dishes, and also pot and dispenser locations. The next group consists of locations of currently existing items such as onions, dishes and soups. And lastly, state consists of mask representing player's location and orientation.

We therefore introduce the idea of extending the current state representation by appending only the player location and orientation masks from the previous three states. Appending the static layout information channels from the previous states does not add any new information, as they are already present in the current state representation. Furthermore, we also omit the information about the current instances of objects from the previous states, since these can be inferred in terms of important temporal features by combining their current state, which is present in the current state representation, with the information about the past movements of the agents.

Since there are five channels describing an agent's position and orientation, by including information about the last three previous states, we arrive at our final state representation consisting of 52 channels.

While we are aware that this state representation construction does not follow the same pattern as the original representation, since in the original representation there were no temporal relations between different channels, we are hopeful that the network will be able to develop such a representation.

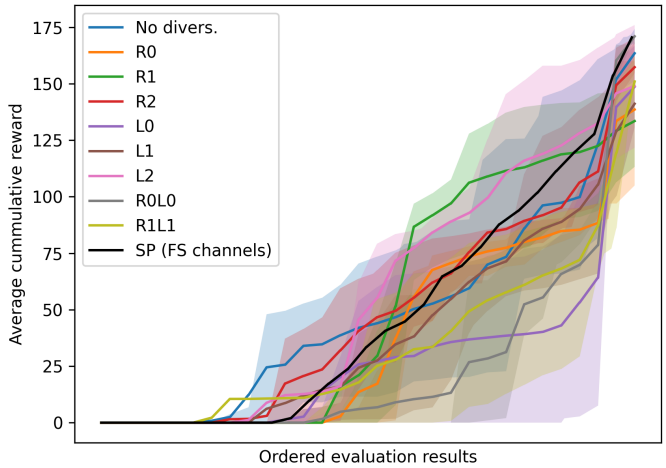


Figure 7.5: Average cummulative reward of final agents ordered by evaluation results

Average cummulative reward of the best final agent using channels frame stacking extension for temporal features. (Average, 0.25Q, 0.75Q)

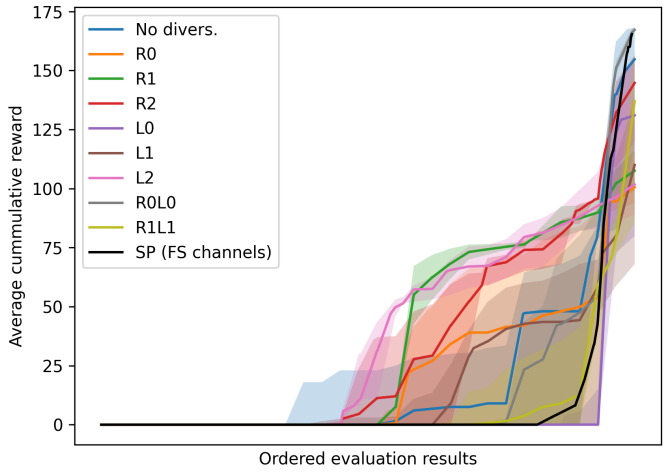


Figure 7.6: Average cummulative reward of final agents, 0.15quantile

Average cummulative reward of the best final agent using channels frame stacking extension for temporal features. (0.15Q, Min, 0.3Q)

7.4.2 States tuple

In our second design, we rely on the idea that the initial convolutional part of the network needs to learn how to extract state feature representation that hopefully could be reused for previous states without significant modifications. Therefore, instead of modifying the state representation as we did in the previous case, we always store three previous lossless state representations without any changes and we slightly modify the network architecture. The input of the network will now

consist of a tuple of four most recent states, including the current state.

For all states of the tuple, the state representation is passed into the shared convolutional block of the same architecture as before. However, we add an additional shared small dense layer after the convolutional block, which is applied to each part of the tuple input that corresponds to one of the previous states. Only then are all four vector parts concatenated back together before being passed to the shared hidden layers as before.

The size of the additional dense layer is intentionally smaller. First, the idea is to extract only those features that, when combined with the same features from other previous states, can construct the temporal information. Second, we want to limit the throughput of global state information through this part of the computational graph.

Note that this additional layer is not applied to the part of the tuple input corresponding to the current state. Therefore, this part of the computation graph is still responsible for computing the overall complex state representation as before.

Note also that both the convolutional block and the additional dense layer are shared for different parts of the input. The goal is not to train the modules responsible for a particular task multiple times. The convolutional block should be able to learn such general feature extraction that should be invariant to the particular single input state. Similarly, we assume that the additional features of the dense layer should be time-invariant, and the construction of the temporal features should be the responsibility of the following dense layer working with concatenated results.

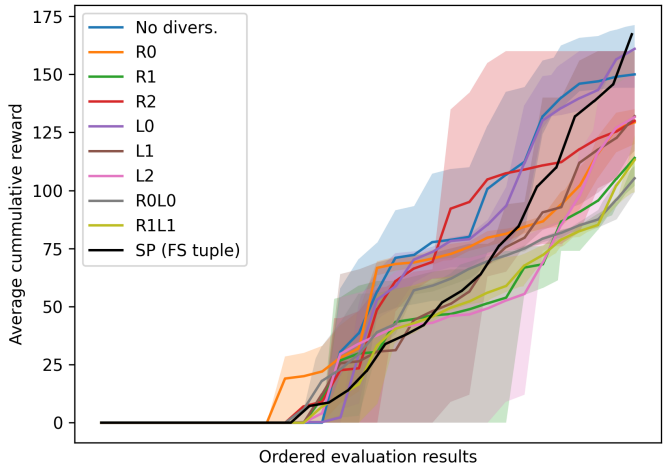


Figure 7.7: Average cummulative reward of final agents ordered by evaluation results

Average cummulative reward of the best final agent using tuple frame stacking extension for temporal features. (Average, 0.25Q, 0.75Q)

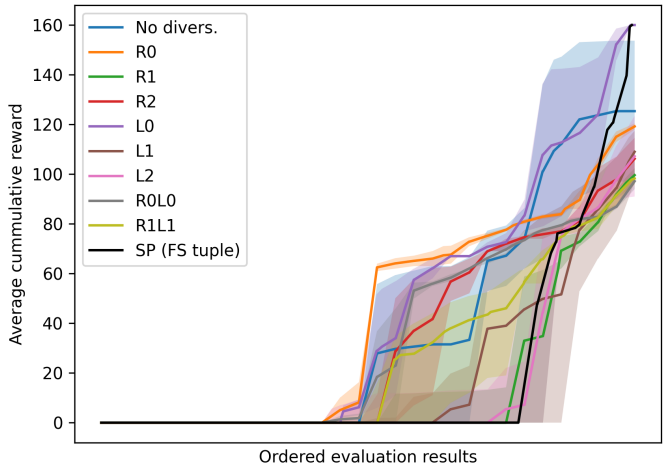


Figure 7.8: Average cummulative reward of final agents, 0.15quantile

Average cummulative reward of the best final agent using tuple frame stacking extension for temporal features. (0.15Q, Min, 0.3Q)

7.4.3 Comparison

After implementing these temporal features modifications, we performed the same series of diversity population building experiments as before with the simple single-state convolutional representation. Trained population agents are always evaluated against self-play agents trained with the same frame stacking modifications, eliminating any potential advantages. In addition, in the initial phase of the experiments, the diversity combination R1L1 (see table 7.2) was both problematic in terms of learning, as the training diverged more often than in other settings, and also achieved significantly worse cooperation results (see evaluations 7.3, 7.4). Therefore, for the following experiments we added another slightly weaker combination of parameters R0L0.

Using the same evaluation scheme as before (see description 7.3), the ordered evaluation results of the population training did not turn out to be as better for the channels frame stacking technique as it was for the simple CNN approach (see figure 7.5). Similarly, the improvement is debatable for the tuple frame stacking variant (Figure 7.5). Nevertheless, we argue that for several experimental settings (No divers, R0, R2, and L0) the improvements can be demonstrated at lower values of the average cumulative reward.

However, it is important to mention the comparison of the different sets of self-play agents (Figure 7.9). This comparison shows that the self-play agents using temporal features managed to cooperate at a lower level of cooperation with a significantly larger set of agents. On the one hand, this is a great result, since with these extensions even simple self-play agents became better at cooperating. On the other hand, the improvement in cooperation seems to be less noticeable in our population training.

While the results seem rather bland for the average value of the ordered scores, it still holds that the population training approach shows significant improvement

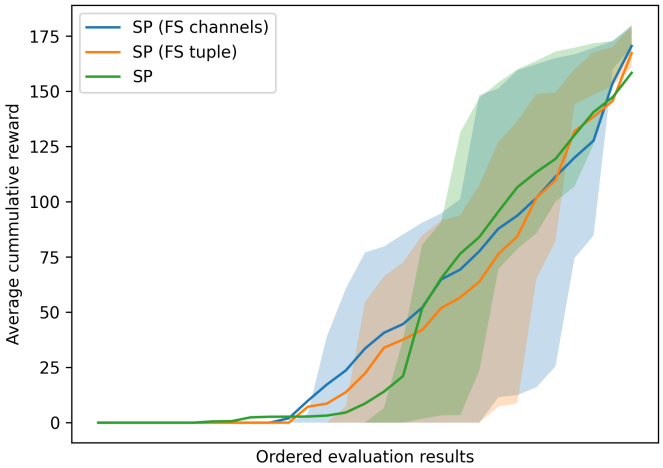


Figure 7.9: Average cummulative reward of final agents

Average cummulative reward of all agents from evaluation set. All three types of self-play agent designs included.
Average, 0.25Q, 0.75Q

when considering the 0.15 quantile value. Here (figures 7.6, 7.8) the population agents managed to vastly outperform the self-play agents.

7.5 Population evaluation

So far, we have measured how well our trained population agents perform when paired with self-play agents and compared this measure to how well self-play agents perform when paired with other self-play agents. However, this evaluation approach may favor self-play agents because they have the theoretical advantage of being trained with the same learning technique.

To make the comparison of these two training methods more fair, instead of evaluating final population agents against different self-play agents, we will now evaluate them against other final population agents from different diversity experiments. In previous experiments, we considered the best of the two final population agents according to their cross-play results with the self-play set. However, in the vast majority of cases, the best agent was the one that was trained with the agent training set, which included all agents from the population initialization. Therefore, for the following evaluation we will always consider the first of the final agents.

Our evaluated set in the following experiment will consist of one final agent for each diverse population experiment configuration and for each seed, giving us a total of 27 agents for frame stacking extensions (and 40 for original simple CNN agents). We compute the cross-play evaluation matrix for all pairs and group the results by the rows corresponding to the same diversity experiment. We then order these results in the same way as in the previous experiments.

With this evaluation setting, our population-trained agents were able to sig-

nificantly outperform the cooperation performance of the self-play agents when using both the original simple convolutional and channel frame stacking representations (Figures 7.10, 7.11). The most significant improvement can be seen in the ability to cooperate with a much wider set of agents. Some, albeit very low, level of cooperation was achieved with almost all of the paired agents.

Unfortunately, the results were not as convincing for the tuple temporal extensions (Figure 7.12).

TODO: Jeste me napada ze jsou ty vysledky asi zkreslene, protoze vlastne pro jeden seed ale jine parametry experimentu byla populace inicalizovana stejne, tedy vychazeli ze stejnych zakladnich chovani, tj. vzdy 9 z 27 agentu vychazelo ze stejneho zdroje

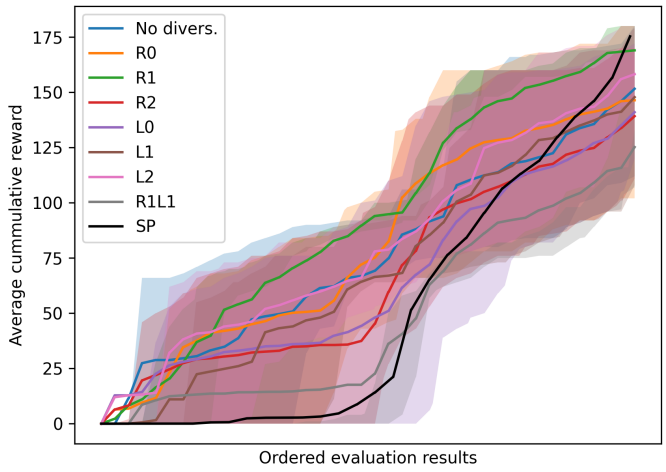


Figure 7.10: Evaluation of final population agent using simple CNN

Diversity experiments final population agents trained with simple convolutional state representation against final population agents from different experiments. Average, 0.25Q, 0.75Q

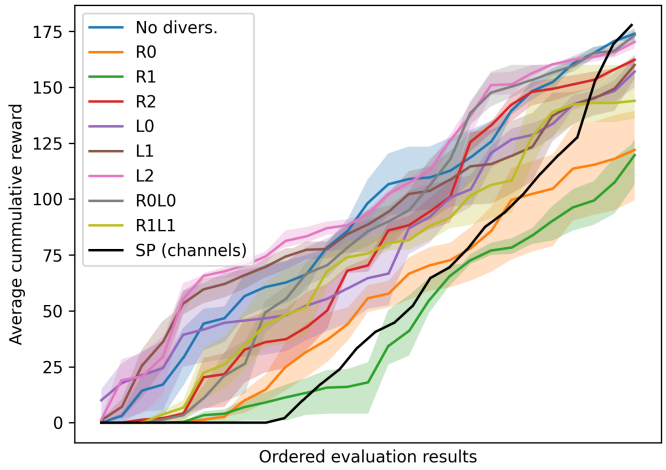


Figure 7.11: Evaluation of final population agent using channels frame stacking

Diversity experiments final population agents trained with channel frame stacking representation against final population agents from different experiments. Average, 0.25Q, 0.75Q

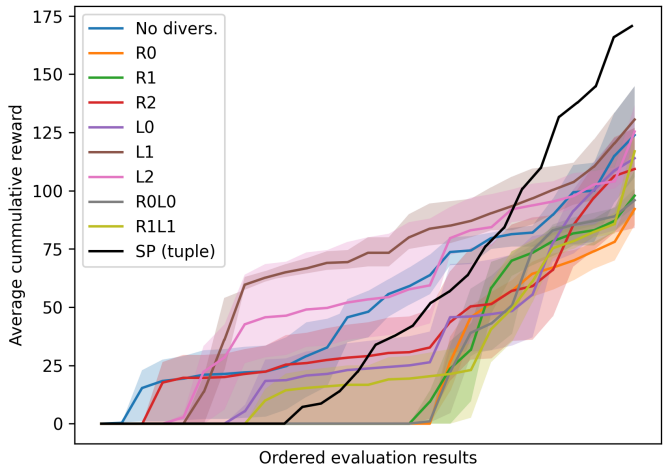


Figure 7.12: Evaluation of final population agent using tuple frame stacking

Diversity experiments final population agents trained with tuple frame stacking representation against final population agents from different experiments. Average, 0.25Q, 0.75Q

7.6 Other layouts

So far, all of our observations and experiments have revolved around the same Forced Coordination layout. In the following section, we examine the smaller set of similar experiments on two other layouts. First, we demonstrate the difference in the cooperation of self-playing agents with respect to temporal features. And second, we try to apply our diversity population generation method.

7.6.1 Cramped Room

On this layout, we can see (Figure 7.13) that adding temporal features had a huge impact on the overall performance of the self-play agents, both in terms of the level of cooperation with other agents and in terms of the maximum common outcome reached. This may be explained by the characteristics of this cramped layout, as observing the last partner's locations can greatly help in predicting the partner's movement pattern.

While our diverse population showed some signs of improvement in most of our experiment settings when using the tuple frame stacking representation (Figure 7.15), there was no improvement for the channel frame stacking representation (Figure 7.14), where many of the experiment settings even showed worse performance.

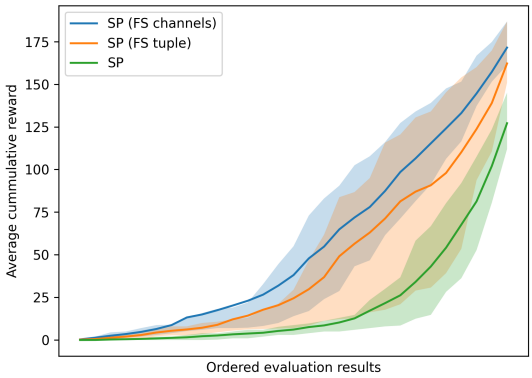


Figure 7.13: Average cummulative reward of final agents

Average cummulative reward on the Cramped room layout of all self-play agents. All three types of self-play agent designs included.
Average, 0.25Q, 0.75Q

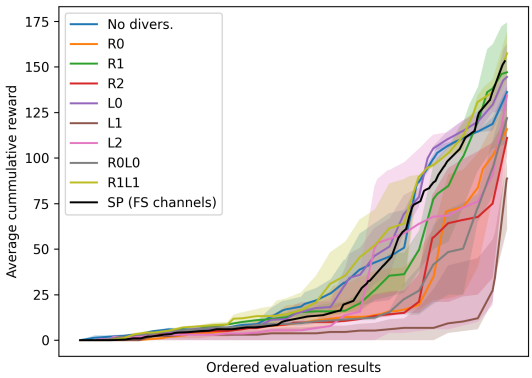


Figure 7.14: Average cummulative reward of final agents ordered by evaluation results

Average cummulative reward on the Cramped room layout of the best final agent using channels extension for temporal features. (Average, 0.25Q, 0.75Q)

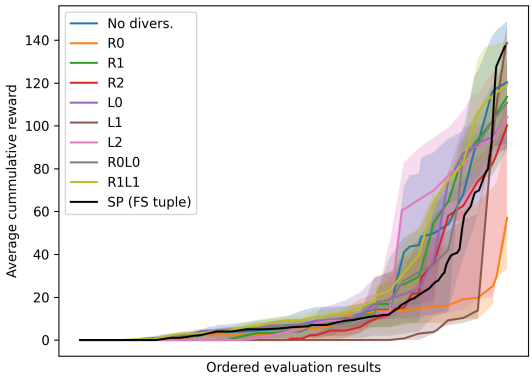


Figure 7.15: Average cummulative reward of final agents, 0.15quantile

Average cummulative reward on the Cramped room layout of the best final agents evaluated against self-play agents using tuple frame stacking for temporal features. (0.15Q, Min, 0.3Q)

7.6.2 Counter circuit

Similarly, the application of temporal features had a significant impact on the Counter Circuit layout(Figure 7.16). Interestingly, the evaluation curve of channels and tuple is different. It seems that the use of the channels frame stacking representation makes agents cooperative with virtually all other agents at a non-significantly low level of cooperation. However, at some point the increase in cooperation level is outperformed by a states tuple representation.

On this layout, the diversity population method managed to outperform the self-play agents in most of the experiment settings (Figures 7.17, 7.18). However, we cannot say that the improvement is very convincing, as the cooperation failed noticeably in a few experimental settings.

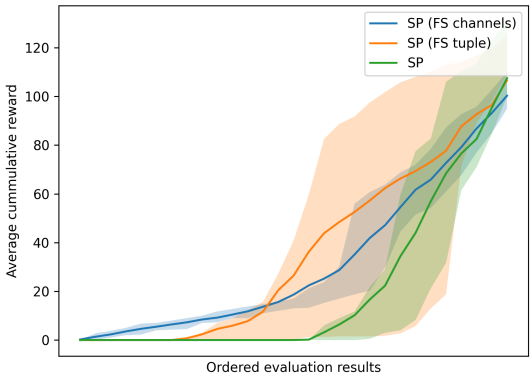


Figure 7.16: Average cummulative reward of final agents

Average cummulative reward on the Counter circuit layout of all self-play agents. All three types of self-play agent designs included.
Average, 0.25Q, 0.75Q

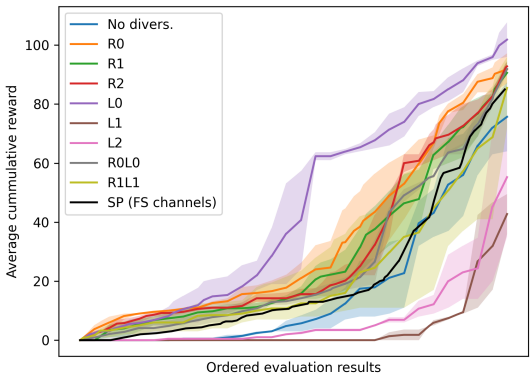


Figure 7.17: Average cummulative reward of final agents ordered by evaluation results

Average cummulative reward on the Counter circuit layout of the best final agent using channels extension for temporal features. (Average, 0.25Q, 0.75Q)

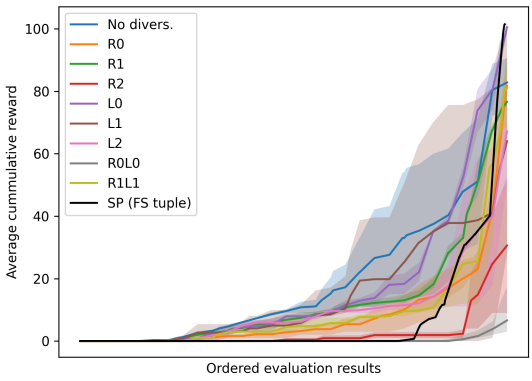


Figure 7.18: Average cummulative reward of final agents, 0.15quantile

Average cummulative reward on the Counter circuit layout of the best final agents evaluated against self-play agents using tuple frame stacking for temporal features. (0.15Q, Min, 0.3Q)

Conclusion

So far it looks that our method works if we set threshold for cooperation to smaller value (cca 70 shared reward) and look for 0.30 quantil.

KL divergence as additional partial reward seems to be reasonable exploration ehancing technique that did not have that significant negative effect on trained policy. But when applied in objective function it seems to force the learned policy the way that is worsening the main objective.

TODO: Even when no diversification was applied, it managed in some cases to learn to cooperate with some self-play agents that none from pop init could.

While populatino agents are more cooperative at lower level, there are still quite a lot self-play agents that our trained agents cannot cooperate with.

Bibliography

Kullback leibler divergence. URL https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence.

Josh Achiam, 2018. URL https://spinningup.openai.com/en/latest/.

Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*, 16(1):1 – 3, 1966. doi: pjm/1102995080. URL https://doi.org/.

Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge*, TARK '96, page 195–210, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1558604179.

Micah Carroll. URL https://github.com/HumanCompatibleAI/overcooked_ai.

Micah Carroll, Rohin Shah, Mark K. Ho, Thomas L. Griffiths, Sanjit A. Seshia, Pieter Abbeel, and Anca Dragan. On the utility of learning about humans for human-ai coordination, 2020.

Rujikorn Charakorn, Poramate Manoonpong, and Nat Dilokthanakul. Investigating partner diversification methods in cooperative multi-agent deep reinforcement learning. In Haiqin Yang, Kitsuchart Pasupa, Andrew Chi-Sing Leung, James T. Kwok, Jonathan H. Chan, and Irwin King, editors, *Neural Information Processing*, pages 395–402, Cham, 2020. Springer International Publishing. ISBN 978-3-030-63823-8.

Deepmind, 2019. URL https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018. URL https://arxiv.org/abs/1802.09477.

Ghost Town Games, 2016. URL https://ghosttowngames.com/overcooked/.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications, 2018. URL https://arxiv.org/abs/1812.05905.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017. URL https://arxiv.org/abs/1710.02298.

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *NIPS*, 2016.

Sham M. Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *International Conference on Machine Learning*, 2002.

Paul Knott, Micah Carroll, Sam Devlin, Kamil Ciosek, Katja Hofmann, A. D. Dragan, and Rohin Shah. Evaluating the robustness of collaborative agents, 2021.

Solomon Kullback. *Information Theory and Statistics*. Wiley, New York, 1959.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015. URL `https://arxiv.org/abs/1509.02971`.

Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments, 2017. URL `https://arxiv.org/abs/1706.02275`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518: 529–33, 02 2015. doi: 10.1038/nature14236.

John F. Nash. Equilibrium points in ¡i¿n¡/i¿-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950. doi: 10.1073/pnas.36.1.48. URL `https://www.pnas.org/doi/abs/10.1073/pnas.36.1.48`.

F. A. Oliehoek, M. T. J. Spaan, and N. Vlassis. Optimal and approximate q-value functions for decentralized POMDPs. *Journal of Artificial Intelligence Research*, 32:289–353, may 2008. doi: 10.1613/jair.2447. URL `https://doi.org/10.1613%2Fjair.2447`.

OpenAI, 2019. URL `https://openai.com/blog/openai-five-finals/`.

R. Tyrrell Rockafellar. *Convex analysis*. Princeton Mathematical Series. Princeton University Press, Princeton, N. J., 1970.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015a. URL `https://arxiv.org/abs/1502.05477`.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015b. URL `https://arxiv.org/abs/1506.02438`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL `https://arxiv.org/abs/1707.06347`.

L. S. Shapley. Stochastic games*. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953. doi: 10.1073/pnas.39.10.1095. URL https://www.pnas.org/doi/abs/10.1073/pnas.39.10.1095.

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

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games, 2021. URL https://arxiv.org/abs/2103.01955.