Opponent Awareness at all levels of the Multiagent Reinforcement Learning Stack

Daniel Hernandez
Doctor of Philosophy

UNIVERSITY OF YORK

Computer Science

August 11, 2022

UNIVERSITY OF YORK

Abstract

Computer Science

Doctor of Philosophy

Opponent Awareness at all levels of the Multiagent Reinforcement Learning Stack

by Daniel Hernandez

Multiagent Reinforcement Learning (MARL) has experienced numerous high profile successes in recent years in terms of generating superhuman gameplaying agents for a wide variety of videogames. Despite these successes, MARL techniques have failed to be adopted by game developers as a useful tool to be used when developing their games, often citing the high computational cost associated with training agents alongside the difficulty of understanding and evaluating MARL methods as the two main obstacles. This thesis attempts to close this gap by introducing an informative modular abstraction under which any Reinforcement Learning (RL) training pipeline can be studied. This is defined as the MARL stack, which explicitly expresses any MARL pipeline as an environment where agents equipped with learning algorithms train via simulated experience as orchestrated by a training scheme. Within the context of 2-player zero-sum games, different approaches at granting opponent awareness at all levels of the proposed MARL stack are explored in broad study of the field.

At the level of training schemes, a grouping generalization over many modern MARL training schemes is introduced under a unified framework. Empirical results are shown which demonstrate that the decision over which sequence of opponents a learning agent will face during training greatly affects learning dynamics. At the agent level, the introduction of opponent modelling in state-of-the art algorithms is explored as a way of generating targeted best responses towards opponents encountered during training, improving upon the sample efficiency of these methods. At the environment level the use of MARL as a game design tool is explored by using MARL trained agents as metagame evaluators inside an automated process of game balancing.

Declaration of Authorship

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. I use *nosism* throughout this thesis for stylistic pourposes, which is is the practice of using the pronoun *we* to refer to oneself.

The following publications arose from the original work presented in this thesis:

1. Hernandez, D., Denamganai, K., Devlin, S., Samothrakis, S., & Walker, J. A. (2020). A Comparison of Self-Play Algorithms Under a Generalized Framework. arXiv preprint arXiv:2006.04471.

2. Hernandez, D., Denamganaï, K., Gao, Y., York, P., Devlin, S., Samothrakis, S., & Walker, J. A. (2019, August). A generalized framework for self-play training. In 2019 IEEE Conference on Games (CoG) (pp. 1-8). IEEE.

3. Hernandez, D., Gbadamosi, C. T. T., Goodman, J., & Walker, J. A. (2020, August). Metagame Autobalancing for Competitive Multiplayer Games. In 2020 IEEE Conference on Games (CoG) (pp. 275-282). IEEE.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction & Motivation

Learning via trial and error through action inside of an environment is an integral part of the nature of humans and animals and also one of the main mechanisms by which they develop new skills. The intuition that expertise over a skill is obtained via repeated practice rings true for an immense number of scenarios. Cooks will rehearse a recipe until a desired taste is obtained, athletes will train to reduce the time required to finish a race and professional e-sport players will practice to become unrivaled in their game of choice. Artificial intelligence captures this idea of learning a skill through repeated attempts inside of an environment within the field of Reinforcement Learning (RL). Reinforcement Learning is a formulation that frames problems as the iterative learning via trial and error of an agent acting in an environment with the objective of maximizing a reward function [Mitchell, 1997, Sutton and Barto, 1998]. The ultimate goal of any RL algorithm is to find a so-called policy, a function that dictates the behaviour of an agent at any given situation.

This already very general framework is augmented by Multiagent Reinforcement Learning (MARL) which expands RL by allowing multiple agents to act in the same environment. Due to the significant generality of the MARL framework, designing algorithms that successfully find high quality policies for a wide set of environments can have an impact on a vast number of applications. Recent successes in MARL even go as far as to hypothesise that RL could constitute a necessary and sufficient bed for any emergent intelligent behaviour [Silver et al., 2021]. In multiagent scenarios, having awareness of the other agents in the environment receives extra importance. Cooks fine-tune a recipe to cater for customer preferences, athletes adapt their technique to differentiate themselves from other contestants and e-sport players change their strategies in response to that of their opponents. This thesis studies the addition of agent awareness within various aspects of MARL as a means of increasing the quality of the policies trained by MARL algorithms.

1.1 MARL in the Games Industry

The video-game industry, or games industry for short, is ripe with titles featuring a myriad of environments where multiple agents interact in many different ways. It is precisely due to this variety in environments that video games have a long-standing tradition of being used as testbeds to drive MARL research [Yannakakis and Togelius, 2018]. Yet, albeit games being ubiquitous in MARL academic circles, the promises and advances of MARL techniques have yet to bear fruit and see widespread use within the games industry, where game developers still spend large amounts of effort into hand-crafting gameplaying agents to be used during game deployment, or using slow and expensive humans during Quality and Assurance processes [Wilkins et al., 2020a, Wilkins et al., 2020b]. Studies aiming at closing the gap between this industry and MARL [Jacob et al., 2020] accentuate a number of issues that industry practitioners struggle with or altogether prevent them from successfully using MARL as a framework to create solutions to their game design problems. Amongst the most salient issues, practitioners highlight the prohibitive computational cost associated with training agents and the lack of interpretability of evaluation methods. The temporal and economic costs associated with generating high quality agents using MARL is an ongoing issue within the field. Even with ample computational resources it is not uncommon to see training runs taking days [Silver et al., 2018] for traditional board games such as Shogi, or an entire month [Silver et al., 2016] for games like Go. Such computational and economical cost are increased even further when MARL was applied to modern videogames such as Dota 2[1] [Berner et al., 2019] and StarCraft 2[2] [Vinyals et al., 2019]. The interpretability of evaluation methods stems from the inherent complexity of MARL methods and the lack of knowledge transfer about useful and informative abstractions about MARL training pipelines from academical experts to game developers.

This thesis shares the view that MARL can be a powerful asset to the games industry, and thus addresses the aforementioned issues of training costs and interpretability of evaluation methods. The common theme of this thesis is to present clarifying abstractions for representing problems within MARL and to accentuate the usefulness of opponent awareness as a means to enhance evaluation methods and reduce the computational cost of training agents. By targeting improvements that mitigate the these issues, we progress MARL research while specially benefiting its use within the games industry.

1.2 The MARL Stack

Despite being conceptually simple to reason about at an intuitively level, MARL algorithms are notoriously difficult to implement due to the large amount of moving elements that need to be accounted for. One of the key hardships is that there are various levels of abstraction at which these elements interact in complex ways. Borrowing terminology from software design, we propose a descriptive formulation of MARL problems as the inner workings and relationships between three stacked modules. These are (1) an environment (2) a set of agents equipped with learning algorithms and (3) a training scheme, as depicted in Figure 1.1. This thesis proposes that a problem formulated under MARL requires the explicit specification of an environment where agents equipped with learning algorithms train via simulated experience as orchestrated by a training scheme. Albeit not fully orthogonal, these modules feature a large degree of independence between them. Understanding their outline sheds light on

[1] https://www.dota2.com/home

[2] https://starcraft2.com

both its enormous descriptive power and the areas where each independent module can be improved. We briefly introduce them here and explore them in Chapter 2.

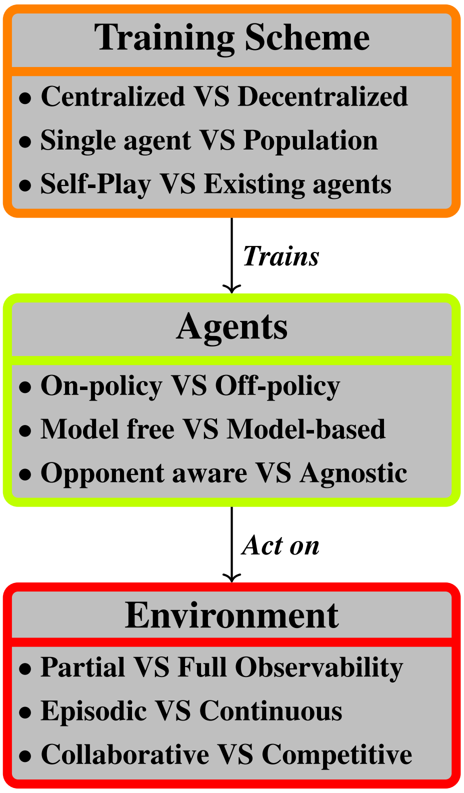

Training Scheme
• Centralized VS Decentralized
• Single agent VS Population
• Self-Play VS Existing agents
Trains
Agents
• On-policy VS Off-policy
• Model free VS Model-based
• Opponent aware VS Agnostic
Act on
Environment
• Partial VS Full Observability
• Episodic VS Continuous
• Collaborative VS Competitive

Figure 1.1: MARL abstraction stack

The environment is the world containing the task for which the agents will learn on via simulated trial and error. An environment constitutes both the dynamics of the world being simulated and a reward or scoring function which the agents will try to optimize against. The nature of these dynamics and the properties of the reward functions induce different types of environment categorizations.

The underlying environment processes actions from the agents acting on it. Some agents can be static, with no learning happening when deployed in the environment. Others can learn by collecting experiences from which updates to the agents' internal systems are generated, with the hope that this will help them to perform better at the target task. What these internal systems are and how they are used and updated is what differentiates between agent algorithms.

Finally, training schemes determines the relationship between the agents that will act in an environment. This involves decisions like what information the agents can access about each other, or which agents are allowed to learn instead of remaining static. Training schemes usually assume learning agents to be oracle functions which, if deployed against other agents, will eventually improve its performance against them.

1.3 Research Questions

We identify three main points of improvement within the MARL stack, with each enhancement being preceded by a corresponding research question which motivated it, presenting a broad study of MARL.

Training Scheme

Training schemes are some of the least rigorously studied parts in recent research in the field of MARL and their importance is usually secondary in modern publications. These are often described in passing and without a standard notation, obscuring whether improvements of a specific MARL algorithm come from novelties at the agent level or the level of the training scheme. We focus on disambiguating a famous family of training schemes known as self-play (SP). Self-play training schemes often train a single agent and assume a pool of possible opponents that grows over time to play against, normally initialized with a random policy. With the lack of structured comparisons between SP training schemes, we posit the first research question:

> **Research question 1:** Does the choice of opponents that a learning agent plays against have a measurable effect on the learning dynamics of such agent?

The study of self-play and its effect on several aspects of MARL algorithms will be a prominent recurrent topic within the thesis beyond this research question.

Agents

The second research question descends down the MARL stack and draws from research question 1's idea that being aware of other agents is a core part of a MARL algorithm's design. We focus specifically on model-based agents. These are agents that can simulate episodes using internal environment models without affecting the real environment in which they are playing. During these internal simulations the simplifying assumption is often made that the model-based agent is playing against itself. We argue that in many cases of interest we can do without this assumption, learn an opponent policy model by playing against such opponent in the real environment, and then simulate play against this learnt opponent model instead of the agent's own policy. Thus:

> **Research question 2:** Is the sample efficiency of model-based algorithms improved by learning opponent models during play and then using them inside model simulations?

Environment

MARL research on video-games overly emphasizes on generating superhuman level agents as its ultimate goal, as if the environment presented by these videogames were merely a task to solve, rather than using MARL methods as tools to aide in game design tasks. It is this latter case that we want to shed light on. Assume an environment with different agent types, such that an individual policy has to be computed for each agent (i.e a game with different character classes, each one with a different strategy). The game designers will have an intended design for the classes' relative strengths, also known as game balance. However, the intended game balance often fails to emerge once humans start playing the game. This usually happens because it is very hard to evaluate what is the strategic equilibrium that will emerge from a given environment purely from the environment's specification. Furthermore, if the emergent balance is far away from the target balance, there is often no informed formal algorithm to decide how to change the underlying environment so that rational play will yield the target balance. This task is tedious, time-consuming and requires hand-tailored knowledge. Hence we posit:

> **Research question 3:** Can agents trained via MARL be used as proxy players as part of the automation of game balancing?

1.4 Thesis Overview & Contributions

Chapter 2 features a comprehensive review of the existing research that this thesis is built upon. Covering single and multiagent RL, game theory and model-based planning methods, it contains all intuition and technical concepts required to understand the remaining parts of the thesis. Each of the research chapters that follow focuses on adding opponent awareness on a different module of the MARL stack from Figure 1.1.

Chapter 3 focuses on training schemes with its main contribution being a novel generalized framework which generalizes over many state-of-the-art self-play training schemes, showing its capabilities by putting under its umbrella various state-of-the-art self-play algorithms. A novel visualization metric is developed to qualitatively measure how different training schemes have different effects on the learning dynamics of the underlying learning agents. Finally, various quantitative experiments are carried out to develop on the qualitative findings. The findings and insights of this chapter regarding self-play are used to derive further discussion on the content of future chapters. This chapter focuses on research question 1.

Chapter 4 studies the agent level of the MARL stack by introducing Best Response Expert Iteration (BRExIt), an improvement upon state-of-the-art model-based MARL algorithms. It enhances Expert Iteration (ExIt) [Anthony et al., 2017] and AlphaGo [Silver et al., 2016] style algorithms by adding opponent modelling at various stages of the algorithm's execution. To aid in the comparison between BRExIt and ExIt, a novel evaluation metric of agent skill is developed, MCTS equivalent strength, and is used in the process of determining that BRExIt has a higher sample efficiency than ExIt through a series of experiments in the game of Connect4. This chapter focuses on research question 2.

Chapter 5 is concerned with environments and presents an approach for automating game balancing efforts. The main contribution is an algorithm for automating the balancing of video games to match designer intent. This is done by first exposing an intuitive graph-based definition of game balance. This definition is then used by an algorithm to find the underlying game parameters that, by training MARL agents inside of it, the resulting balance between these agents aligns with an intended target game balance. We demonstrate the capabilities of our algorithm in a game of our own making. This chapter focuses on research question 3.

Chapter 6 concludes the thesis, summarizing all presented contributions. These are accompanied by discussions on the extent to which we have answered the three research question posited in this introduction and the limitations of the approaches taken. Finally, a compilation of thoughts on future work directions is explored.

1.5 Implementation

All of the experiments carried out throughout this thesis used an open source framework developed over the course of the author's PhD: Regym[3]. Regym is an open source framework

[3] Original: `https://github.com/Danielhp95/Regym`. There also exists a fork with a focus on natural language emergence [Denamganaï and Walker, 2020].

for RL, MARL and game theory research built on top of Numpy [Harris et al., 2020] and PyTorch [Paszke et al., 2019].

Chapter 2

Literature review

2.1 Introduction

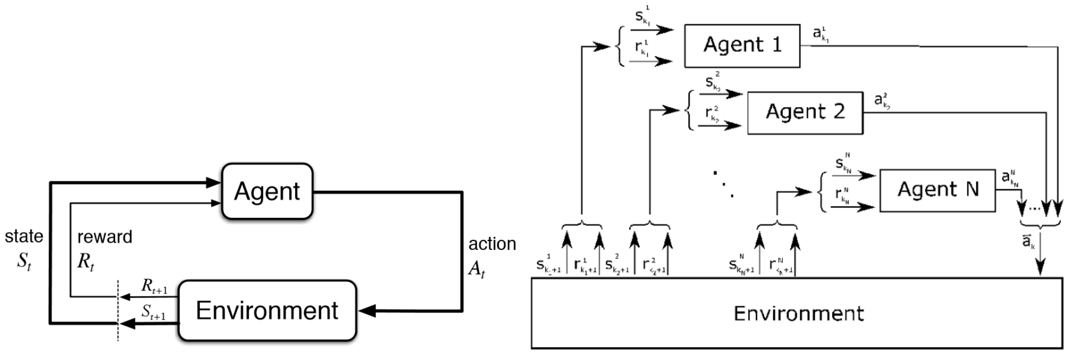
This chapter covers through a bottom-up approach the fundamental and current state of research pertinent to understanding the topics of this thesis. Because the research presented in this document combines notions from MARL, Game Theory and Monte Carlo Tree Search (MCTS), an introduction to all of these fields is required. We first cover the three layers of the MARL stack, as depicted in Figure 1.1. Section 2.2, introduces different mathematical formulations for defining environments in single and multiagent RL respectively. Section 2.3 explains the fundamental building blocks used for the making of agents to act in these environments. Finally, Section 2.4 covers training schemes, as a final abstraction layer to coordinate training. These are followed by an introduction to Game Theory for the purpose of multiagent evaluation in Section 2.5, finishing on an introduction to MCTS in Section 2.6.

On notation: Cursive lowercase letters denote scalars (n). Bold lowercase, vectors ($\boldsymbol{\pi} \in \mathbb{R}^n$). Bold uppercase, matrices ($\boldsymbol{A} \in \mathbb{R}^{n \times n}$).

A problem can be considered a (MA)RL problem if it can be framed in the following way: Given an environment in which an agent (or agents) can take actions, receiving a reward for each action, find a policy (or policies) that maximizes the expected cumulative reward that (each of) the agent (agents) will obtain by acting in the environment. All experiments and analyses in this thesis will be framed under this lens unless otherwise stated. Figures 2.1a and 2.1b capture the interaction described in this statement in a graphical manner.

2.2 The Environment

We now describe different formulations for formally defining environments in which agents can act. These definitions are accompanied by their pseudocode code description and equations dictating the goal for learning agents acting on them. We begin with the simpler single-agent environments, and slowly build up to the more complex multiagent environments which shall be used in the later chapters of this thesis. The purpose of presenting all of these different environment models is twofold. On the one hand the presentation shows step-by-step generalizations of mathematical models which can help in framing future problems within



(a) Single-agent Reinforcement Learning loop (b) Multiagent Reinforcement Learning loop

Figure 2.1: Reinforcement Learning Loops

a correct environment model. On the other hand, Chapter 3 will define a family of training schemes which generalizes over all presented environment models, and therefore their complete description is given here for completeness. As a summary, and for readers familiar with the content of this chapter, Table 2.1 features all the environment models discussed in this section.

Model	Partial observability	Multiagent	Multiple Reward Signals	Solution	Algorithm
MDP	×	×	×	Eq. 2.1	1
POMDP	✓	×	×	Eq. 2.2	2
MMDP	×	✓	×	Eq. 2.3	3
Dec-POMDP	✓	✓	×	Eq. 2.4	4
Markov Game	×	✓	✓	Eq. 2.5	5
POSG	✓	✓	✓	Eq. 2.6	6

Table 2.1: Properties of the environment models discussed in this section. Ticks ✓ correspond to the presence of the column property and crosses × otherwise.

2.2.1 Markov Decision Process (MDP)

The most famous mathematical structure used to represent single-agent RL environments are Markov Decision Processes (MDP) [Bellman, 1957]. Bellman introduced the concept of a Markov Decision Process as an extension of the famous mathematical construct of Markov chains [Norris, 1998]. Markov Decision Processes are a standard model for sequential decision making and control problems.

An MDP is fully defined by the 6-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}(s'|s,a), \mathcal{R}(s,a,s'), \gamma, \rho_0(s))$:

- $\mathcal{S}$ is the set of states of the underlying Markov chain, where $s_t \in \mathcal{S}$ represents the state of the environment at time t. As an example, take a maze-like gridworld, where an agent needs to navigate from an initial position to a goal location. Here the state space could be comprised of all of the possible locations the agent could find itself in.

- $\mathcal{A}$ is the set of actions available to the agent. $A_t \subset \mathcal{A}$ is the subset of available actions in state s_t at time t. If an state s_t has no available actions, it is said to be a *terminal* state. Terminal states are equivalent to absorbing states in the Markov chain literature. Following the previous gridworld example, the set of available actions at time t, A_t, would correspond to the cardinal directions that the agent could move towards (up, down, left, right), with some of these being unavailable if an obstacle was present (i.e, a wall). A terminal state would be one where the agent has reached the goal location.

- $\mathcal{P}$ is the transition probability function[1]. $\mathcal{P}(s'|s,a) \in [0,1]$, where $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$ expresses a distribution over states. It defines the probability of transitioning to state s' from state s after performing action a. Thus, we have the function signature: $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$. Given a state s_t and an action a_t at time t, we can find the next state s_{t+1} by sampling from the distribution $s_{t+1} \sim P(s_t, a_t)$. The *dynamics of the world* are captured by $\mathcal{P}$. In our gridworld, this would correspond to updating the location of the agent once it has chosen to move in a given direction.

- $\mathcal{R}(s,a,s') \in \mathbb{R}$, where $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$. $\mathcal{R}$ is the reward function, which represents the immediate reward the agent will obtain after performing action a in state s and ending in state s'. If the environment is deterministic, the reward function can be rewritten as $\mathcal{R}(s,a)$. For games like Chess, a simple reward function would yield -1 upon taking an action that leads to a loss, and +1 if it leads to a victory. A more opinionated reward function might give smaller rewards upon taking or losing pieces to facilitate learning. Although this can reduce the complexity of the task, it can also bias the agents to learn suboptimal behaviours.

- $\gamma \in [0,1]$ is the discount factor, which represents the rate of importance between immediate and future rewards. For any given γ, an agent will value the cumulative reward up to time t' as $\sum_{t=0}^{t'} \gamma^t r_t$. If $\gamma = 0$ the agent will care only about reward coming from the following state transition, if $\gamma = 1$ all following rewards $\sum_t^\infty r_t$ are taken into account.

- $\rho_0(s_0) \in [0,1]$, where $s_0 \in \mathcal{S}$, is the initial state distribution, representing the probability of starting an episode on a given state s_0. It is from this distribution that the initial state is sampled from $s_0 \sim \rho_0$. In our gridworld, this would correspond to the distribution over possible initial agent positions. If there was a single initial position, ρ_0 would be deterministic.

Acting inside of the environment, there is the agent, and through its actions the transitions between the MDP states are triggered, advancing the environment state and generating reward signals. For any given MDP, the goal of this agent is to find a policy, a (potentially stochastic) mapping from states to actions $\pi : \mathcal{S} \times \mathcal{A} \to [0,1]$, which maximizes the observed discounted cumulative reward. We can postulate this search into a single equation:

$$\pi^* = \max_{\pi} \mathbb{E}_{s_0 \sim \rho_0, s_{t+1} \sim \mathcal{P}(\cdot|s_t,a_t), a_t \sim \pi(\cdot|s_t)} [\sum_{t=0}^{\infty} \gamma^t r_t(s_t, a_t)] \tag{2.1}$$

[1] The function $\mathcal{P}$ is also known in the literature as the transition probability kernel, the transition kernel or the dynamics of the environment in model-based contexts. The word kernel is a heavily overloaded mathematical term that refers to a function that maps a series of inputs to a value in $\mathbb{R}$.

Algorithm 1: Agent / Environment interaction loop: MDP

```
Input: Environment: $E = (\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot), \gamma, \rho_0(\cdot))$
Input: Agent Policy: $\pi(\cdot)$
Sample initial state from the initial state distribution $s_0 \sim \rho_0$ ;
$\texttt{t} \leftarrow \texttt{0}$ ;
repeat
    Sample action $a_t \sim \pi(s_t)$;
    Sample successor state from the transition probability function $s_{t+1} \sim P(s_t, a_t)$ ;
    Sample reward from reward function $r_t \sim R(s_t, a_t, s_{t+1})$ ;
    $\texttt{t} \leftarrow \texttt{t} + \texttt{1}$ ;
until $s_t$ is terminal;
```

Even though Markov Decision Processes are the most famous mathematical structure used to model an environment in reinforcement learning, there are other types of possible models for RL environments which act as extensions to vanilla MDPs. These mainly relax assumptions about the state space, the action space, the observability of the environment state, the reward function, and the number of agents present in the environment.

2.2.2 Partially Observable Markov Decision Process (POMDP)

In an MDP, the internal representation of the environment is the same representation that the agent observes at every timestep. POMDPs introduce the idea that what the agent observes at every timestep t is only a partial representation o_t of the real environment state s_t [Kaelbling et al., 1998]. This partial observation o_t alone is not enough to reconstruct the real environment state s_t, which entails that $o_t \subset s_t$. This is the case in many 3D games of interest, in which the agent can only observe the environment around itself but won't be able to observe the entire game space. A Partially Observable Markov Decision Process is defined by a 8-element tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}(s'|s,a), \mathcal{R}(s,a,s'), \Omega, \mathcal{O}(o|s,a), \gamma, \rho_0(s))$:

- $\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}$ and γ express the same concepts as in classical MDPs.
- Ω is the set of all possible agent observations.
- $\mathcal{O}(o|s,a)$, where $o \in \mathcal{O}, s \in \mathcal{S}, a \in \mathcal{A}$, represents the probability of the agent receiving observation o after executing action a in state s.
- $\rho_0(s_0, o_0) \in [0,1]$, where $s_0 \in \mathcal{S}$ and $o_0 \in \Omega$, is the initial state distribution, which extends the same concept from MDPs so as to be defined over the joint space of environment states and agent observations.

The objective of a POMDP is to find an optimal policy π conditioned on the *history* of environment observations, $a_t \sim \pi(\boldsymbol{o}_{\leq t})$, where $\boldsymbol{o}_{\leq t} = [o_0, \ldots, o_t]$, which will maximize the expected cumulative reward. Informally, we ask the agent to maximize expected reward given the information that it has received, which means that this policy can perform worse than if it had observed the actual game state throughout an episode, instead of only partial state observations. This goal is formalized as:

Algorithm 2: Agent / Environment interaction loop: POMDP

```
Input: Environment: $E = (\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot,\cdot), \Omega, \mathcal{O}(\cdot|\cdot,\cdot), \gamma, \rho_o(\cdot,\cdot))$
Input: Agent Policy: $\pi(\cdot)$
Sample initial state and observation from the initial state distribution $s_0, o_o \sim \rho_0$ ;
t ← 0 ;
repeat
    Sample action $a_t \sim \pi(o_t)$;
    Sample successor state from the transition probability function $s_{t+1} \sim P(s_t, a_t)$ ;
    Sample successor observation from observation function $o_{t+1} \sim \mathcal{O}(s_t, a_t)$;
    Sample reward from reward function $r_t \sim R(s_t, a_t, s_{t+1})$ ;
    t ← t + 1 ;
until $s_t$ is terminal;
```

$$\pi^* = \max_{\pi} \mathbb{E}_{s_0, o_0 \sim \rho_0, s_{t+1} \sim \mathcal{P}(\cdot|s_t,a_t), o_{t+1} \sim \mathcal{O}(\cdot|s_t,a_t), a_t \sim \pi(\cdot|\boldsymbol{o}_{\leq t})} [\sum_{t=0}^{\infty} \gamma^t r_t(s_t, a_t)] \tag{2.2}$$

A POMDP can be reduced to an MDP iff, for all timesteps t the agent's observation o_t and the environment state s_t are equal $o_t = s_t$. That is, if there is a one-to-one correspodance between observations and states.

2.2.3 Multiagent Markov Decision Process (MMDP)

A major shortcoming of the previous two environments is that they assume stationary environments (stationarity assumption), which by definition entails that the environment does not change over time. This assumption makes MDPs and POMDPs unsuitable for modelling multiagent environments. From the perspective of any given agent, other agents must be considered as non-stationary parts of the environment, because the policies that define other agents behaviours can change over time through the course of learning, breaking the environment stationarity assumption [Laurent et al., 2011] which is paramount for proving the convergence of many single-agent RL algorithms on the aforementioned environments [Watkins, 1989, Sutton and Barto, 1998].

[Boutilier, 1996] introduces Multiagent Markov Decision Processes (MMDPs) as a framework to study coordination mechanisms. MMDPs feature multiple agent policies, each of them submitting an individual action every timestep, which is executed as a joint action by the environment, producing a new state via the transition probability function. A *single* reward function is shared amongst all agents, making MMDPs fully collaborative.

A Multiagent Markov Decision Process featuring k agents is defined by a 6-element tuple $(\mathcal{S}, \mathcal{A}_{1,\ldots,k}, \mathcal{P}(s', |s, \boldsymbol{a}), \mathcal{R}(s, \boldsymbol{a}), \gamma, \rho_0(s))$:

- $\mathcal{S}, \gamma$ and ρ_0 express the same concepts as in classical MDPs.
- $\mathcal{A}_{1,\ldots,k}$ is a collection of action sets, one for each agent in the environment, with $\mathcal{A}_i$ corresponding to the action set of the ith agent.
- $\mathcal{P}(s'|s, \mathbf{a})$, where $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents the probability transition function. It states the probability of transitioning from state s to

state s' after executing the *joint* action $\mathbf{a}$. The joint action is a vector containing the action performed by every agent at a certain timestep.

- $\mathcal{R}(s, \mathbf{a}) \in \mathbb{R}$, where $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \dots, a_k\}$ and $a_1 \in \mathcal{A}_1, \dots, a_k \in \mathcal{A}_k$, represents the reward function, mapping states and joint actions to a single scalar reward, which will be propagated to all agents.

We can let the vector $\boldsymbol{\pi} = \{\pi_i, \dots, \pi_k\}$ denote all of the policies that act in an MMDP environment with k agents. For notational convenience, we can use standard mathematical notation to define $\boldsymbol{\pi}_{-i} = \{\pi_1, \dots, \pi_{i-1}, \pi_{i+1}, \dots, \pi_k\}$ as the vector of all policies except for policy π_i. Both $\boldsymbol{\pi}$ and $\boldsymbol{\pi}_{-i}$ can be conceived as joint probability distributions from which to sample joint actions, such as: $\boldsymbol{a} \sim \boldsymbol{\pi}(\cdot|s)$. Similarly, we can break the joint action vector $\boldsymbol{a} = \{a_i, \boldsymbol{a}_{-i}\}$ as the union between ith agent's action, and the actions from all other agents.

Algorithm 3: Agent / Environment interaction loop: MMDP

```
Input: Environment: $(\mathcal{S}, \mathcal{A}_{1,\dots,k}, \mathcal{P}(\cdot,|\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \gamma, \rho_0(\cdot))$
Input: Policy vector: $\boldsymbol{\pi}(\cdot)$
Sample initial state and observation vector $s_0 \sim \rho_0$;
$t \leftarrow 0$;
repeat
    Sample action vector $\boldsymbol{a_t} \sim \boldsymbol{\pi}(s_t)$;
    Sample successor state $s_{t+1} \sim \mathcal{P}(s_t, \boldsymbol{a_t})$;
    Sample scalar reward signal $r_t \sim \mathcal{R}(s_t, \boldsymbol{a_t})$;
    $t \leftarrow t + 1$;
until $s_t$ is terminal;
```

With this, we can denote the MMDP objective for any agent i:

$$\pi_i^* = \max_{\pi} \mathbb{E}_{s_0 \sim \rho_0, s_{t+1} \sim \mathcal{P}(\cdot|s_t, \{a_t, \boldsymbol{a}_t^{-i}\}), a_t^i \sim \pi(\cdot|s_t), \boldsymbol{a}_t^{-i} \sim \boldsymbol{\pi}_{-i}(\cdot|s_t)} [\sum_{t=0}^{\infty} \gamma^t r_t(s_t, \{a_t^i, \boldsymbol{a}_t^{-i}\})] \quad (2.3)$$

2.2.4 Decentralized Markov Decision Process (Dec-POMDP)

A Dec-POMDP is a natural extension of POMDPs to multiple agents, by reintroducing partial observability to MMDPs. Now not all agents share the same full description of the environment. Instead each has its own observation, which only expresses a subset of the environment's full state. Dec-POMDPs form a framework for multiagent planning under uncertainty [Oliehoek and Amato, 2014]. This uncertainty comes from two sources. The first one being the partial observability of the environment, the second one stemming from the uncertainty that each agent has over the other agent's policies, as the actions of other agents might not be observed.

The term *decentralized* here dictates that each of the present agents receives its own observation. If it were centralized, a singular (partial) observation of the environment would instead be shared amongst all agents. Hence, in Dec-POMDPs the agents do not have the explicit ability of sharing their observations with each other. Every agent bases its decision purely on its individual observations. On every timestep each agent chooses an action simultaneously and they are all collectively submitted to the environment.

A Dec-POMDP with k agents is defined as a 9 element tuple $(I, \mathcal{S}, \mathcal{A}_{1,\ldots,k}, \mathcal{P}(s'|s,\mathbf{a}), \Omega_{i,\ldots,k}, \mathcal{O}(\boldsymbol{o}|s,\boldsymbol{a}), \mathcal{R}(s,\mathbf{a}), \gamma, \rho_0(s,\boldsymbol{o}))$:

- $\mathcal{S}$, $\mathcal{A}_{1,\ldots,k}$, $\mathcal{P}$, $\mathcal{R}, \gamma$ and ρ_0 express the same concept as in MMDPs.

- I is the set of all agents .

- $\Omega_{1,\ldots,k}$ represents the joint set of all agent observations, with Ω_i representing the set of all possible observations for the ith agent.

- $\mathcal{O}(\mathbf{o}|s,\mathbf{a})$, where $\mathbf{o} = \{o_1,\ldots,o_k\}$, $o_1 \in \Omega_1,\ldots,o_k \in \Omega_k$, $s \in \mathcal{S}$, $\mathbf{a} = \{a_1,\ldots,a_k\}$ and $a_1 \in \mathcal{A}_1,\ldots,a_k \in \mathcal{A}_k$, represents the probability of observing the joint observation vector $\mathbf{o}$, containing an observation for each agent, after executing the joint action $\mathbf{a}$ in state s.

Algorithm 4: Agent / Environment interaction loop: Dec-POMDP

```
Input: Environment: $(I, \mathcal{S}, \mathcal{A}_{1,\ldots,k}, \mathcal{P}(\cdot|\cdot\cdot), \Omega_{i,\ldots,k}, \mathcal{O}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \gamma, \rho_0(\cdot,\cdot))$
Input: Policy vector: $\boldsymbol{\pi}(\cdot)$
Sample initial state and initial vector observation $s_0, \boldsymbol{o_0} \sim \rho_0$;
$t \leftarrow 0$;
repeat
    Sample action vector $\boldsymbol{a_t} \sim \boldsymbol{\pi_e}(\boldsymbol{o_t})$;
    Sample successor state $s_{t+1} \sim \mathcal{P}(s_t, \boldsymbol{a_t})$;
    Sample successor observation vector $\boldsymbol{o_{t+1}} \sim \mathcal{O}(s_t, \boldsymbol{a_t})$;
    Sample scalar reward $r_t \sim \boldsymbol{R}(s_t, \boldsymbol{a_t})$;
    $t \leftarrow t + 1$;
until $s_t$ is terminal;
```

Overloading previous notation, we can use $\boldsymbol{\pi}$ as a joint distribution over a joint action space conditioned on a vector of observations, where each agent gets its corresponding observation: $\{1_1 \sim \pi_1(\cdot|o_1),\ldots,a_k \sim \pi_1(\cdot|o_k)\}\} \sim \boldsymbol{\pi}(\cdot|\boldsymbol{o})$. Thus, we get the following problem formulation for an arbitrary agent i:

$$\pi_i^* = \max_{\pi} \mathbb{E}_{s_0 \sim \rho_0, s_{t+1} \sim \mathcal{P}(\cdot|s_t,\{a_t^i, \boldsymbol{a}_t^{-i}\}), \boldsymbol{o_{t+1}} \sim \mathcal{O}(\cdot|s_t,\boldsymbol{a_t}), \boldsymbol{a_t} \sim \boldsymbol{\pi}(\cdot|\boldsymbol{o}_{\leq t})} [\sum_{t=0}^{\infty} \gamma^t r_t(s_t, \{a_t, \boldsymbol{a}_t^{-i}\})] \tag{2.4}$$

A Dec-POMDP featuring a single agent, $|I| = 1$, can be treated as a POMDP. When the environment features full observability, the term Dec-MDP is used [Bernstein et al., 2002].

2.2.5 Markov Game

The idea of a Markov Game was first introduced through the field of game theory as stochastic games [Shapley, 1953], and began receiving MARL recognition in later work [Owen and

Owen, 1982, Littman, 1994a]. A Markov Game with k different agents is denoted by a 6-element tuple $(\mathcal{S}, \mathcal{A}_{1,...,k}, \mathcal{P}(s'|s, \boldsymbol{a}), \mathcal{R}_{1,...,k}(s, \boldsymbol{a}), \gamma, \rho_0)$:

- $\mathcal{S}$, $\mathcal{A}_{1,...,k}$, $\mathcal{P}, \gamma$ and ρ_0, express the same concepts as MMDPs.
- $\mathcal{R}(s, \mathbf{a}) \in \mathbb{N}^k$, where $s \in \mathcal{S}$, $\mathbf{a} = \{a_1, \ldots, a_k\}$ and $a_1 \in \mathcal{A}_1, \ldots, a_k \in \mathcal{A}_k$, represents the reward function. $\boldsymbol{r} \in \mathbb{N}^k$ is the reward vector obtained at the environment transition number t. The reward $r_i \in \boldsymbol{r}$ denotes the reward for the ith agent after the joint action vector $\mathbf{a}$ is executed in state s.

Algorithm 5: Agent / Environment interaction loop: Markov Game

```
Input: Environment: $(\mathcal{S}, \mathcal{A}_{1,...,k}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}_{1,...,k}(\cdot,\cdot), \rho_0(\cdot), \gamma)$
Input: Policy vector: $\boldsymbol{\pi}(\cdot)$
Sample initial state and observation vector $s_0 \sim \rho_0$;
$t \leftarrow 0$;
repeat
    Sample action vector $\boldsymbol{a_t} \sim \boldsymbol{\pi}(s_t)$;
    Sample successor state $s_{t+1} \sim \mathcal{P}(s_t, \boldsymbol{a_t})$;
    Sample reward vector $\boldsymbol{r_t} \sim \mathcal{R}(s_t, \boldsymbol{a_t})$;
    $t \leftarrow t + 1$;
until $s_t$ is terminal;
```

Each agent independently tries to maximize its own expected discounted cumulative reward signal. This allows for reward functions that go beyond fully collaborative environments like those of MMDPs. The objective formulation for agent i in Markov Games is thus:

$$\pi_i^* = \max_{\pi} \mathbb{E}_{s_0 \sim \rho_0, s_{t+1} \sim \mathcal{P}(\cdot|s_t, \{a_t, \boldsymbol{a}_t^{-i}\}), a_t^i \sim \pi(\cdot|s_t), \boldsymbol{a}_t^{-i} \sim \boldsymbol{\pi}_{-i}(\cdot|s_t)} [\sum_{t=0}^{\infty} \gamma^t r_t^i(s_t, \{a_t^i, \boldsymbol{a}_t^{-i}\})] \tag{2.5}$$

Markov Games have several important properties [Owen and Owen, 1982,Littman, 1994b]. For the two-player zero-sum case every Markov game features an optimal policy for each agent. Unlike MDPs, these policies may be *stochastic*. An intuitive advantage of stochastic policies stems from the agent's uncertainty about the opponent's pending moves. On top of this, stochastic policies make it difficult for opponents to anticipate the agent's action, which can make the policy less exploitable. The main contributions of Chapter 4 follow this idea of trying to model the other agents' policies based on their historical actions, to further inform another agent's policy in its pursuit to maximize reward.

When the number of agents in a Markov Game is exactly 1, the Markov Game can be considered an MDP. If all agents shared the same reward function, the Markov Game is reduced to an MMDP.

2.2.6 Partially Observable Stochastic Games (POSG)

One of the most generalized environment models are Partially Observable Stochastic Games, which come from game theoretical literature [Hansen et al., 2004]. These are multiagent environments with agent-independent reward functions and partial observability. All previous environment models are concrete instantiations of POSG. A POSG with k agents is denoted by a 9 element tuple:
$(\mathcal{I}, \mathcal{S}, \mathcal{A}_{1,\dots,k}, \mathcal{P}(s'|s, \boldsymbol{a}), \Omega_{i,\dots,k}, \mathcal{O}(\boldsymbol{o}|s, \boldsymbol{a}), \mathcal{R}_{1,\dots,k}(\cdot, \cdot), \gamma, \rho_0(s, \boldsymbol{o}))$:

- $\mathcal{I}, \Omega_{1,\dots,k}$ and $\mathcal{O}$ expresses the same concept as Dec-POMDPs.
- $\mathcal{S}, \mathcal{A}_{1,\dots,k}, \mathcal{P}, \mathcal{R}_{1,\dots,k}, \gamma$ and ρ_0 express the same concepts as Markov Games.

Algorithm 6: Agent / Environment interaction loop: POSG

```
Input: Environment:
       $(\mathcal{I}, \mathcal{S}, \mathcal{A}_{1,\dots,k}, \mathcal{P}(\cdot|\cdot,\cdot), \Omega_{i,\dots,k}, \mathcal{O}(\cdot|\cdot,\cdot), \mathcal{R}_{1,\dots,k}(\cdot,\cdot), \gamma, \rho_0(\cdot,\cdot))$
Input: Policy vector: $\boldsymbol{\pi}(\cdot)$
Sample initial state and initial vector observation $s_0, \boldsymbol{o_0} \sim \rho_0$;
$t \leftarrow 0$;
repeat
    Sample action vector $\boldsymbol{a_t} \sim \boldsymbol{\pi_e}(\boldsymbol{o_t})$;
    Sample successor state $s_{t+1} \sim \mathcal{P}(s_t, \boldsymbol{a_t})$;
    Sample successor observation vector $\boldsymbol{o_{t+1}} \sim \mathcal{O}(s_t, \boldsymbol{a_t})$;
    Sample reward vector $\boldsymbol{r_t} \sim \boldsymbol{R}(s_t, \boldsymbol{a_t})$;
    $t \leftarrow t + 1$;
until $s_t$ is terminal;
```

The formulation of the solution for an arbitrary agent i in a POSGs is:

$$\pi_i^* = \max_{\pi} \mathbb{E}_{s_0 \sim \rho_0, s_{t+1} \sim \mathcal{P}(\cdot|s_t, \{a_t^i, \boldsymbol{a_t^{-i}}\}), \boldsymbol{o_{t+1}} \sim \mathcal{O}(\cdot|s_t, \boldsymbol{a_t}), \boldsymbol{a_t} \sim \boldsymbol{\pi}(\cdot|\boldsymbol{o_{\leq t}})} [\sum_{t=0}^{\infty} \gamma^t r_t^i(s_t, \{a_t, \boldsymbol{a_t^{-i}}\})] \tag{2.6}$$

This concludes the introduction and review of environments for single and multiagent scenarios. The environment for every experiment described in this thesis can be cast as one of the these environments. We remind the reader of Table 2.1, which summarizes the contents of this section.

2.3 The Agent

Moving upwards in the MARL stack, we find the agents. Agents can be separated into two component parts: a *policy* and a *learning algorithm* or *learning function* [Laurent et al., 2011]. The policy, as defined earlier is a function which maps environment states (or more generally, observations) to actions. A learning algorithm is a more complex function which maps state-action-reward sequences, also known as paths or trajectories, to policies. Essentially all learning agents in RL iteratively apply a learning algorithm over paths sampled from the environment via a policy, with the hope of improving this policy over time and eventually getting closer to the optimal policies stated as goals in Section 2.2. This section defines what main modules and building blocks are used to construct both single and multiagent RL agents, focusing on how to learn a high reward policy. We do not dwell into the inner workings of any particular algorithm, and instead focus on a framework by which most RL algorithms can be studied.

2.3.1 Value Functions and Bellman Equations

There are two functions of special relevance in RL which capture a notion of how much reward an agent is expected to obtain from a given point in an episode. Intuitively, being able to ascertain how much reward a policy will accrue from a certain observation onwards should be helpful in deciding how to update such policy to increase its cumulative reward. For instance, given a parameterized policy, one could take the gradient of such function with respect to the parameters of the policy in the direction of positive improvement. We first focus on the single agent scenario. For simplicity, we assume a stationary MDP with a discount factor $\gamma = 1$, although these definitions can be extended to non-deterministic and discounted cases.

- **State value function**: $V^{\pi \in \Pi}(s \in \mathcal{S}) \in \mathbb{R}$ under a policy π from policy space Π, represents the expected sum of rewards obtained by starting in state s and following the policy π until termination. Formally: $V^{\pi}(s) = \mathbb{E}_{a_t \sim \pi(\cdot|s_t)}[\sum_{t=0}^{\infty} r(s_t, a_t)|s_0 = s]$. With a recursive definition:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s)(r(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)V^{\pi}(s')) \tag{2.7}$$

- **State-action value function**: $Q^{\pi \in \Pi}(s \in \mathcal{S}, a \in \mathcal{A}) \in \mathbb{R}$ under a policy π, represents the expected sum of rewards obtained by performing action a in state s and then following policy π. It is a one-step look-ahead version of $V(s)$, which peeks one action into the future. Formally: $Q^{\pi}(s,a) = \mathbb{E}_{a_t \sim \pi(\cdot|s_t)}[r(s_0, a_0) + \sum_{t=1}^{\infty} r(s_t, a_t)|s_0 = s, a_0 = a]$. With a recursive definition:

$$Q^{\pi}(s,a) = r(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)(\sum_{a' \in \mathcal{A}} \pi(a'|s')Q^{\pi}(s',a')) \tag{2.8}$$

Similarly, we can define the value functions for multiagent environments, only for Markov Games for simplicity. The only difference being that in a multiagent setting we need our equations to account for the presence of other agent policies in the environment. Let's take a policy vector $\boldsymbol{\pi} = \{\pi_i, \boldsymbol{\pi}_{-i}\}$, for which we define the recursive state value function for agent i:

$$V_i^{\{\pi_i, \boldsymbol{\pi}_{-i}\}}(s) = \sum_{\boldsymbol{a} \in A} \boldsymbol{\pi}(\boldsymbol{a}|s)(r_i(s, \{a_i, \boldsymbol{a}_{-i}\}) + \sum_{s' \in S} \mathcal{P}(s'|s, \{a_i, \boldsymbol{a}_{-i}\})V_i^{\boldsymbol{\pi}}(s')) \tag{2.9}$$

Equation 2.9 assumes that all other policies π_{-i} remain fixed (stationary). Thus, finding a policy π_i for agent i which maximizes this equation will yield the best performance against such other agents, at the potential expense of not being robust to changes in the opponents' policies. In other words, small variations in opponent policies could cause big changes in the agent i's performance, as measured by the state value function. This idea is further explored in Section 2.5 and Chapter 4.

2.3.2 The Optimality Principle and Actor-Critic Algorithms

The optimality principle, found in [Bellman, 1957] and furthered in [Borkar, 1988] for MDPs, states the following:

> *An optimal policy π^* has the property that given any initial state s_0 and initial action a_0, the remaining actions $a_{t>0}$ must constitute an optimal policy π^* with regard to the state s_1 resulting from the initial action a_0.*

The notion of *optimal* is of course defined with respect to the cumulative reward obtained by a policy as it acts in an environment, which is precisely what is captured by the aforementioned value functions. Therefore, the optimality principle gives rise to *optimal* value functions:

$$V^*(s) = V^{\pi^*}(s) = \arg\max_{\pi} V^{\pi}(s) \tag{2.10}$$

And the optimal state-action value function:

$$Q^*(s,a) = Q^{\pi^*}(s,a) = \arg\max_{\pi} Q^{\pi}(s,a) \tag{2.11}$$

The optimal value functions determine the best possible performance an agent can obtain in an environment. This gives a link between the optimal policy and the optimal value functions. If we have access to the optimal policy π^*, we can use Monte Carlo evaluation by deploying π^* in an environment for a large number of episodes to compute the empirical cumulative reward that it obtains at every state-action pair $(s \in \mathcal{S}, a \in \mathcal{A})$, allowing us to approximate Equations 2.7 and 2.8. Symmetrically the optimal policy can be found by inspecting the optimal value functions:

$$\begin{aligned}
&\forall s \in \mathcal{S}: \ \pi^*(s) = \arg\max_{a} \ r(s,a) + \sum_{s'} \mathcal{P}(s' \mid s,a) V^{\pi^*}(s') && \text{(Deriving policy from } V^{\pi^*}(s)) \\
&\forall s \in \mathcal{S}: \ \pi^*(s) = \arg\max_{a} Q^{\pi^*}(s,a) && \text{(Deriving policy from } Q^{\pi^*}(s,a))
\end{aligned} \tag{2.12}$$

The equations above allow us to extract a policy given any arbitrary value function, not just the optimal value functions. This is a key step in most RL algorithms. Note how, in order to extract a policy from a state value function $V(s)$, an environment model $\mathcal{P}$ is required, as we need to inspect all of the possible successor states where an agent might end up. Once we know the most valuable successor state, we construct a policy that maximizes the probability of taking an action which transitions the environment to it. Fortunately, state-action value functions abstract this problem away, as we only need to select from a set of available actions instead of successor states, foregoing the need of an environment model.

Algorithms which both keep track of a parameterized value function and policy are known as *actor-critic methods*. The actor is the policy, taking actions in the environment, and the critic is the value function, which judges the performance of the actor. Actor-critic methods combine the strong points of both policy-only [Williams, 1992] and value function-only [Tamar et al., 2017] based algorithms, overcoming some of their individual weaknesses. The critic is used to update the actor's policy parameters in a direction of improvement. This dynamic leads to faster convergence than actor-only methods, as the critic can be used as an efficient substitute for Monte Carlo policy evaluations to inform actor parameter updates. A key observation is that that in actor-critic algorithms the actor parameterization and the critic parameterization should *not* be independent [Konda and Tsitsiklis, 2000]. The choice of critic parameters should be directly prescribed by the choice of the actor parameters. That is why most real world applications that implement an actor-critic algorithm use a single parameterized model (e.g. a neural network) to represent both the policy (actor) and the value function approximation (critic). This is the most straight forward way of sharing parameters between actor and critic.

At the beginning of training, we have access to neither an optimal policy nor its corresponding optimal value function, and thus we cannot perform an optimal policy or value function extraction as in Equations 2.12. The optimal actor and corresponding critic need to be discovered during training, via trial and error in true RL fashion. In order to do that, most RL algorithms follow the idea of *policy iteration* [Howard, 1960].

2.3.3 Policy Iteration

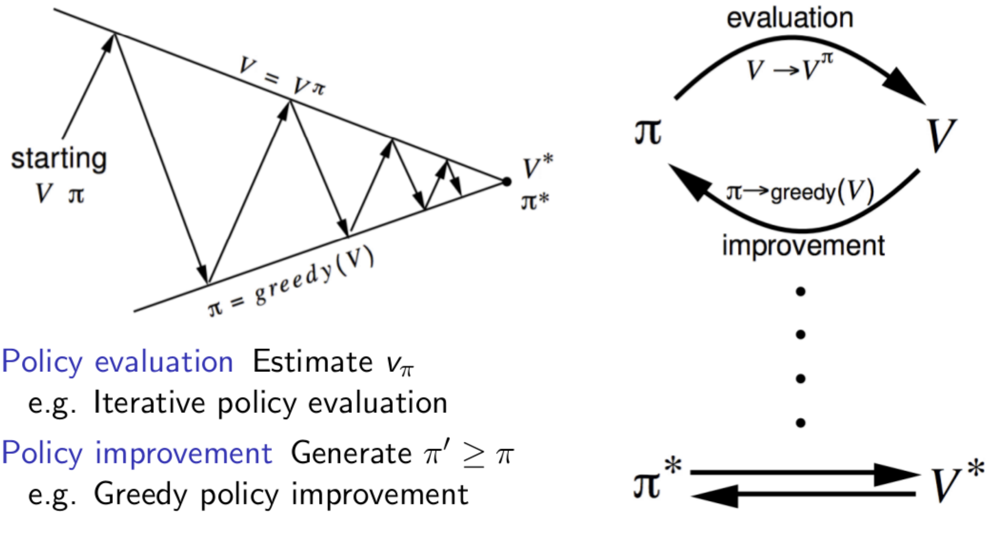

$V = V\pi$
starting
V π
V^*
π^*
π = greedy(V)
Policy evaluation Estimate v_π
e.g. Iterative policy evaluation
Policy improvement Generate $\pi' \geq \pi$
e.g. Greedy policy improvement
evaluation
$V \to V^\pi$
π
V
$\pi \to$greedy(V)
improvement
π^*
V^*

Figure 2.2: Policy iteration diagram. Switching between policy evaluation and policy improvement, an optimal policy and its corresponding value functions can be found. Image taken from [Sutton and Barto, 1998].

Most RL algorithms follow a form of policy iteration, an iterative two-step process for converging to both an optimal policy π^* and an optimal value function V^{π^*} from randomly initialized ones π^0, V^0. As depicted in Figure 2.2. We use the state value function V as a running example, but the state-action value function Q could be used as well.

Policy iteration goes back and forth between *policy evaluation* and *policy improvement*. At iteration t, the policy evaluation step computes a value function $V^{\pi^t}(s)$ even for a given policy π^t. The policy improvement step extracts a new policy π^{t+1} from the pre-computed value functions $V^{\pi^t}(s)$ using Equation 2.12, with the hope that this newly extracted policy is of higher quality[2]. These two processes are carried out one after the other, looped *ad nauseam*, until both the policy and the value function converge, or until a specified computational budget runs out.

Both steps of this process can take a large amount of time. For policy evaluation, stochasticity in both the environment and the policy might require an elevated number of samples to estimate the value of a policy π^t at a single specific state for $V^{\pi_t}(s)$ or state-action pair for $Q^{\pi_t}(s,a)$. Due to this computational complexity an approximated version of this process is often carried out, where the value function is approximated for a subset of the state-action space, and the policy is updated only for this subset. How this subset of environment transitions is chosen, what value function is approximated, and how is a policy extracted from it are the main factors that differ between RL algorithms.

2.3.4 On-Policy and Off-Policy Algorithms

A very informative division of RL algorithms is that of *on-policy* versus *off-policy* methods:

- **On-policy algorithms** [Williams, 1992, Schulman et al., 2017, De Asis et al., 2017, Sutton and Barto, 1998] use the policy that they are learning about to sample actions in the environment. A policy π is both being improved over time and also used to take actions inside of the environment. To perform policy evaluation and improvement, a dataset of environment transitions is generated using π. Once π has been updated, this dataset is discarded, as further data generated from this updated policy might not represent environment trajectories generated by the previous version.

- **Off-policy algorithms** [Mnih et al., 2013, Silver et al., 2014, Lillicrap et al., 2015, Espeholt et al., 2018] use a *behavioural* policy μ to take actions in the environment, $a \sim \mu(s)$, to improve a target policy π. A dataset of environment transitions is constructed using the behavioural policy μ, which we no longer need to discard upon first use due to our assumption that $\pi \neq \mu$. The learning that takes place in off-policy algorithms can be regarded as learning from somebody else's experience, whilst on policy algorithms focus on learning from an agent's own experience.

Most on-policy algorithms dedicate all computational power on learning and using a single policy π. These methods focus on spending the computational resources on applying

[2] The notion of improvement over time is expressed as a monotonic increase in the expected reward of an episode $\mathbb{E}_{a\sim\pi_0}[\sum_{t=0}^{\infty} r_t] < \mathbb{E}_{a\sim\pi_1}[\sum_{t=0}^{\infty} r_t]$

a learning function for the sole benefit of improving this policy. With off-policy algorithms it is possible to dedicate computational time to modifying both the behavioural policy μ and the policy π being learnt. The motivation behind this being that the paths sampled from the environment using π won't necessarily yield the best paths to learn from. However, by spending some of the computational resources to using, or even changing, the behavioural policy μ, it is possible to generate more "informative" trajectories with which we can improve π through a learning function. For instance [Badia et al., 2020, Espeholt et al., 2018] deploy many behavioural policies in a decentralized fashion, with each featuring different levels of exploratory behaviour to generate different types of trajectories to learn a single target policy.

By freeing computational time from directly improving the target policy π, it is possible to tackle many other auxiliary tasks. An auxiliary task is defined as a learning objective which is being optimized in conjunction with more traditional objectives such as reward maximization or value function estimation. The idea being that by learning these auxiliary tasks it might be possible to discover underlying structure in the environment which can be beneficial to learning an optimal policy. [Jaderberg et al., 2016] brought this idea to the spotlight by showing the benefits of learning auxiliary tasks for off-policy algorithms. These tasks were as abstract as: immediate reward prediction[3] and learning a separate policy that maximizes the change in the state representation[4]. However, these auxiliary tasks are highly environment specific and thus not applicable to a large set of environments. In contrast, Chapter 4 explores a particular type of auxiliary tasks which are generalizable to all MARL environments.

2.3.5 Experience Replay

Modern off-policy algorithms heavily rely on *experience replays*, also known as *replay buffers* [Lin, 1993], made famous after the success of [Mnih et al., 2013] on the Arcade Learning Environment [Bellemare et al., 2013]. The experience replay is a fixed-size buffer that holds the most recent transitions or experiences collected by a behavioural policy, used during policy iteration. More importantly, these transitions are not discarded upon use. At the time of updating the function approximators, an experience (or batch of experiences) is sampled uniformly from the experience replay. Because these sampled experiences may have been generated using either an explicitly different behavioural policy or a previous version of the target policy deployed in the environment, experience replay buffers allow for policy updates to happen in an off-policy fashion.

The most basic representation of a transition or experience is a 4 element tuple $\{s_t, a_t, r_t, s_{t+1}\}$, although these can be augmented via any algorithm-specific environment signal, as we shall see in Chapter 4. Apart from the obvious usage of keeping data used to update the actor or the critic for an RL algorithm, there are some other reasons for using a replay buffer:

[3] This is auxiliary task is different from value function estimation, as the former is trying to predict expected immediate reward and the latter expected future cumulative reward.

[4] Given a matrix of pixels as input, the authors define “pixel control” as a separate policy that tries to maximally change the pixels in the following state. The reasoning behind this approach is that big changes in pixel values may correspond to important events inside of the environment. Alternatively it could be considered as an indirect measure for *empowerment* [Mohamed and Rezende, 2015].

1. It greatly improves the sample efficiency of the algorithm by enabling data to be reused multiple times for training, instead of throwing away data immediately after the first time it is used, as done in on-policy algorithms.

2. Function approximators such as neural networks rely on the assumption that the data they are trained on is independent and identically distributed in order to properly learn. This property is not immediately available in RL settings, where data is gathered sequentially, and thus experiences are heavily temporally correlated. By keeping a replay buffer from which data is sampled in a different manner to how it was collected, we break this correlation. This is imperative to stabilize training.

Since their inception in [Lin, 1993], there has been vast amounts of research done on experience replays, and even small properties of these have proved to be of great importance to stabilize learning in off-policy algorithms. For instance, the *replay ratio*, the ratio at which agents switch between acting in an environment to updating their policies and value functions, was discovered to be a driving factor in determining the learning rate of off-policy algorithms [Fedus et al., 2020].

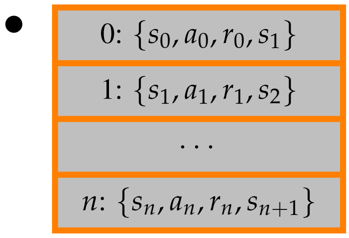

0: $\{s_0, a_0, r_0, s_1\}$
1: $\{s_1, a_1, r_1, s_2\}$
$\dots$
n: $\{s_n, a_n, r_n, s_{n+1}\}$

Figure 2.3: Experience replay of size $n+1$. Replay buffers follow the first-in-first-out principle, newer datapoints will overwrite old ones once the buffer is full. In this figure, the introduction of the $n+1$ transition will replace the 0th entry, denoted by the black dot.

Lists are normally used to represent experience replays, where the oldest transition in the buffer is removed to make room for the latest transition if the buffer is already full, in a circular fashion, as in Figure 2.3. Apart from the mentioned uniform sampling, whereby each transition in the buffer is sampled with equal probability, other sampling strategies exist. Some of these different sampling mechanisms elicit different underlying data structures, such as prioritized experience replay [Schaul et al., 2015], which biases sampling towards experiences where policy evaluation has the largest error. There is also extensions to distributed experience replays [Horgan et al., 2018] and for multiagent scenarios [Foerster et al., 2017].

In Section 2.3.3 we described how policy iteration encapsulates the spirit of many RL algorithms, and in Section 2.3.4 we saw how on-policy and off-policy methods partition the space of possible algorithms. We now move to one final categorization of agents, whether or not they are capable of accessing an environment model to run "side simulations" alongside the real environment where they are being deployed.

2.3.6 Model-Based and Model-Free Algorithms

The types of algorithms covered thus far have been model-free. This means that, in order to use their policies and apply their learning functions, model-free agents use only trajectories sampled from the environment during simulation. Model-based algorithms break from this mold by having explicit access to an environment model or *world model*. As explained in

Section 2.2.1, the model or the dynamics of an environment are the transition function $\mathcal{P}$ and reward function $\mathcal{R}$. Model-based algorithms make use of such environment models to both define their policies and / or for their learning functions. How these models are actually used is algorithm specific, with Section 2.6 covering in-depth one such algorithm, namely Monte Carlo Tree search. Chapters 3 and 4 train a multitude of agents with model-free algorithms, Chapters 4 and 5 heavily use model-based algorithms to generate gameplaying agents.

One of the major advantages of having a model is that it allows for forward planning, which is the main method of learning for search-based artificial intelligence, as we shall see in Section 2.6. Forward planning methods are able to run simulations of the environment aside from the real game being played. These allows them to *look ahead* deeper into the game and observe outcomes of imagined trajectories without affecting the real game being played. Figure 2.4 shows a 2-player game being played in a POSG between a model-free agent and a model based agent. During the model-based agent's turn in the real environment, the agent runs multiple simulations aside from the real game being played to decide what action to take.

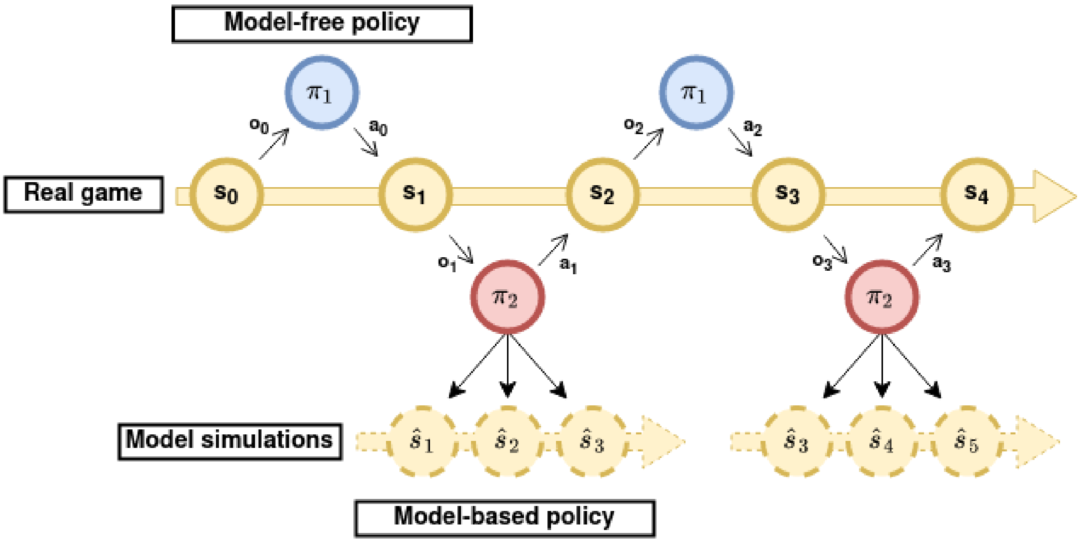


Figure 2.4: Depiction of a sequential two player POSG environment with a model-free agent (top, blue), and a model-based agent (bottom, red). Both players receive an observation on their respective turns based on the real game state (middle, yellow). Top player acts using a model-free policy, for instance, by passing the observation through a neural network, outputting an action which is used by the environment to update its state. The bottom player uses a model-based policy to compute what action to take for given environment observations. Using its model of the environment, it can run many simulations starting at the given environment observation (yellow dotted lines), without affecting the real game being played. It decides on which action to take based on metrics aggregated over these simulations.

Model based algorithms are either given a prior model that they can use for planning [Browne et al., 2012, Soemers, 2014], or they learn an approximated model $\hat{\mathcal{P}}, \hat{\mathcal{R}}$ via their own interaction with the environment [Sutton, 1991, Guzdial et al., 2017, Deisenroth and Rasmussen, 2011]. Some algorithms learn from a combination of real trajectories and trajectories simulated on their models. However, there is an ongoing trend of algorithms which learns entirely from *imagined* trajectories inside of their own models [Hafner et al., 2020], and use the real environment merely to learn a dynamics model. Even though having a model of the environment given a priori intuitively sounds like the superior option, there are advantages to learning an environment model. One benefit is that this model can be specifically tailored for an agent's decision process. For instance, all video game environments arguably feature elements which are irrelevant for optimal decision making, such as aesthetic elements like falling leaves. These take computational power to model and to simulate, but are meaningless for an agent's policy. As a notable example, the model-based algorithm MuZero [Schrittwieser et al., 2020] learns an environment model which embeds environment states into a latent space. The only learning signal used to shape the state embedding function is the agent's cumulative reward on the real game it acts on. This allows the agent to learn a state representation specially suited to maximize its performance on the real environment[5]. All model based algorithms used in this thesis assume a given and perfectly accurate environment model.

This concludes the high level overview and categorization of RL agents. Where we have covered fundamental distinctions of how agents use data generated from environments to update their policies in order to achieve optimality with respect to a reward function. We now move to the final part of the MARL stack, which heavily influences both what will be the trajectories sampled on from an environment and what information agents have about the other agents around them.

2.4 Training Schemes

Training schemes sit at the highest level of abstraction in the MARL stack. Environments represent the task or suite of tasks to be solved by agents. Agents define the logic of how actions will be taken in an environment and how trajectories from these will be handled to generate policy updates. Training schemes alter the interaction between the lower two components of the MARL stack and implicitly define assumptions made about how both the environment and the agents acting in it can be manipulated. They define the interface between the agents and the environment, and between the agents and other agents, in the case of multiagent environments. Arguably, training schemes in multiagent settings have a bigger part to play due to many MARL algorithms requiring access to information about other agents for their learning functions, with this access being mediated by a learning scheme.

In the same fashion that (model-free) agents are unaware of the underlying structure of the environment they act on, training schemes are in principle agnostic to the structural workings of the underlying agents. This means that training schemes treat agents as a blackbox function or *oracle*, which given a policy π and a task t within the set of possible tasks prescribed by the environment $t \in T$, it returns a relatively higher performing policy for the given task:

[5] We emphasize that even though this approach sounds really attractive, in practice, it requires a lot of training data which can be computationally intensive. Although these issues can be mitigated with further additions [Ye et al., 2021].

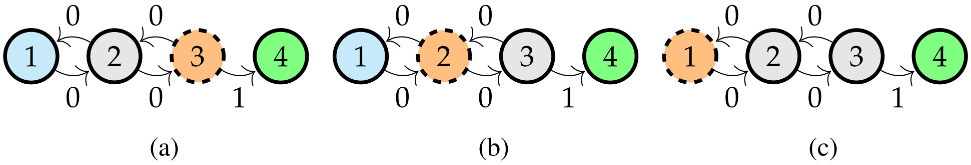


Figure 2.5: Corridor environment. The objective is to reach the goal at the rightmost node (green state) from the initial state prescribed by the training scheme (orange, dashed). The leftmost state (blue)(blue) is environment's original initial state. Edges denote the reward associated with the transition. The only reward signal is obtained reaching the goal, the rightmost state.

$$\pi' = oracle(\pi, t) \quad \text{such that: } V(\pi', t) \geq V(\pi, t) \tag{2.13}$$

Where $V(\pi \in \Pi, t \in T) = \mathbb{E}_{s_0 \sim \rho_0}[V^{\pi}(s_0)]$ defines a notion of value of for policy π on a task t with initial state distribution ρ_0. Note how this definition uses π's state-value function as a metric of performance which implicitly relies on the environment's reward function. Training schemes need not be constrained by this, and can decide to define the value of a policy under other performance metrics *even if* the underlying agents are optimizing against the environment's reward functions. This is one of the benefits of having a separate level of abstraction.

The descriptions of training schemes are often more subtle than that of environments or agents, as they are a less established level of abstraction in the literature. To shed light on their somewhat nebulous nature and to demonstrate their utility, we introduce an MDP toy environment and training scheme in Figure 2.5. This environment features only 4 states, with the leftmost one being the initial state, and the rightmost one being the goal state. Its reward function is very sparse, and only rewards an action that reaches the goal state, giving a 0 reward to every other action. Imagine that we would like to train an agent for this environment. As it is often the case, its policy will be uniform random at the beginning. For a random policy, the expected number of steps required to reach the goal state from the environment's initial state is 9 (This can be readily computed by treating the environment as a Markov chain, and computing its fundamental matrix [Ge, 2016]). This expected number of steps before reaching a rewarding transitions increases exponentially with the number of states. Keeping in mind that an agent's behaviour will only be shaped through the processing of rewarding transitions, training an agent on longer versions of this environment quickly becomes too computationally costly. It is here where training schemes can be of help. We can use a training scheme to modify the environment's initial state to the third one, as in Figure 2.5a, which reduces exponentially the expected number of steps required for the agent to reach a rewarding state. Once the agent can consistently solve this modified environment, we can move the initial state further from the goal, as in Figure 2.5b. It is easy to see how this progressive modification of the environment can reduce the overall complexity of the problem, while still yielding a final policy that behaves optimally with respect to the original environment. This type of training schemes, in which a task is broken into easier subtasks which slowly increase in difficulty, are part of the field of Curriculum Learning [Bassich et al., 2020, Narvekar et al., 2020].

One would not be wrong in stating that the exemplified training scheme modifies with the actual environment, by adding hand-crafted a priori information, which goes against common practices of AI which exacerbate the value of learning everything from scratch and with minimal supervision. Although such a view can be of value for theoretical studies, we have already expressed how such a simple environment, if unmodified, can require vast amounts of compute to solve. If we want RL based methods to be adopted by the games industry, it is of paramount importance to find tools which greatly reduce training time. The challenge for training scheme designers is to obtain this without modifying the original environment's optimal policy [Devlin and Kudenko, 2012], or equilibria, for multiagent scenarios [Devlin and Kudenko, 2016].

There is a valid point in saying that the changes that a training scheme brings can already be captured, for instance, by changing the environment itself. Instead of an environment and a training scheme, Figure 2.5 could as well be represented by 3 different environments, one for each subfigure. For a multiagent scenario, Take the game of Connect4[6] with two independent learning agents and two training schemes. The first training schemes doesn't modify either the environment or the agent interactions, the other augments agent observations with the distribution over actions from their opponent's last move (i.e. the output of the policy $\pi(\cdot|s) \in P(\mathcal{A})$). These two instances can be thought of as being two separate environments. However, the dynamics of the game (the rules) are the same, and the set of optimal policies remains largely the same for both cases. One just happens to offer more information to the agents. Siding with the current literature at large, we argue that it makes sense to abstract some of the interactions between the agents and between the environment via training schemes without considering these different interactions to be distinct environments altogether.

2.4.1 Multiagent training schemes

Training schemes for multiagent settings can be broadly categorized into *decentralized* and *centralized* training schemes. There are two clearly defined categories which differ in the assumptions made about the accessibility and capacity to modify agents' policies in an environment. In decentralized schemes, each agent is assumed to be independent from other agents, learning a policy by processing only their own local environment signals (observations and rewards), with no external global module to coordinate policy learning between agents. In centralized training, there is an additional entity overlooking the entire system of agents, which can take many forms.

Decentralized Training Schemes

These training schemes assume control over a single agent, and are often used to model interactions were agents do not have access to the internal state, observations, actions or even the

[6] Game explanation can be found here: `https://en.wikipedia.org/wiki/Connect_Four`

reward obtained by the other agents. Having a less constraining set of assumptions, it is possible to train agents on a wider set of tasks. Studies on decentralized training schemes focus on the global properties that emerge from agents which maximize their individual reward functions. Due to the dynamical parts of these systems only requiring local info agents trained under them tend to be resilient to errors in the system, such as communication dropouts between agents [Tan et al., 2021], or by being agnostic to the total number of agents in the environment [Arslan and Yüksel, 2016].

Due to the low amount of restrictions posed by decentralized schemes, they have been used to model many real world scenarios where there is normally little to no centralized control that can be imposed over agents in an environment. This also means that it is possible to gather further data to continue training after deployment, due to the low requirements that the learning algorithms pose on the learning process.

There has been extensive work on using decentralized training schemes for auctions and inter company operations modelling [Chang et al., 2020], as in these cases it is hard to concretely capture agent specific reward functions beyond individual desires to maximize them. In certain scenarios, like traffic light control [Zhou et al., 2020b], fault tolerance of individual elements and low individual compute capacity are requirements of the system, which demands decentralized solutions. Even videogames, which traditionally have been perceived to be highly centralized settings, can benefit from decentralized training schemes. For instance, it is possible to have an agent learn by being plugged to an online matchmaking hub featuring an active player base, where it could gather training data in real time from its interactions with many simultaneous players. But this latter usage has yet to receive significant attention and research effort.

Centralized Training Schemes

Centralized training schemes accept more constraining assumptions about elements in the system in favour of stronger learning signals for the underlying agent's learning functions. Some of these more constraining assumptions may include the availability of communication channels between agents [Eccles et al., 2019], access to other agent's internal states [Ossenkopf, 2019], etc.

Perhaps the most common type of centralized training scheme is *centralized training, decentralized execution* [Zhou et al., 2020a]. This approach tries to take advantage of both types of training schemes, by requiring centralizing assumptions to generate datasets to train on, but relaxing these to allow policies to act in a decentralized manner, based only on local information [Fan et al., 2020]. In [Wen et al., 2020, Vinyals et al., 2019], a centralized state-value function is learnt which takes as input the joint observation of all agents. As seen in Section 2.3.1, state value functions for multiagent settings require taking into account the policies of all agents. Thus, by allowing the value function to be augmented with joint observations, it can better estimate the value of each policy in the environment. This augmented centralized critic is in turn used to compute updates for the individual agent policies.

Another way in which multiagent training schemes can affect learning agents is by altering the policies which act in the environment. An agent's policy updates depend on the trajectories sampled from an environment, and these trajectories can be heavily influenced by the other agents acting in it. Training schemes can leverage this by selectively choosing agents will inhabit the environment to influence the learning path of a subset of the agents in the environment. This is the main focus of Chapter 3, and so we delay further elaboration on this topic.

On a closing remark, game studios have full control over the games they develop, and thus have the capacity to train RL agents in-house or using rented cloud compute. This can save them from the aforementioned decentralized complications, such as sets of unknown users in the training environments, inaccessibility to information about other agents, major latency during agent communication and other issues can be cast aside. This thesis shares the opinion that the future for training RL agents in the games industry necessitates centralized training schemes. Thus, further innovations in these training schemes will bring the largest payoff to the games industry.

This ends the overview of the MARL stack. The remaining of this chapter introduces the fields of Game theory in Section 2.5 and Monte Carlo Tree Search in Section 2.6.

2.5 Game Theory

On Notation: Albeit close in spirit, game theory and MARL feature different sets of notations for similar objects, we have tried to homogenize the notation on the important element of our analyses to avoid potential confusion. Most notably, we use the notion of policy, strategy and agent interchangeably.

Game theory is the study of the behaviour and dynamics of agents trying to maximize a notion utility obtained via taking actions in structured environments or games. The work in this thesis uses game theory as an evaluator, as a means to the end of measuring agent performance, rather than as a definition of the environments being played. MARL is used as a framework to postulate the problem of generating gameplaying agents, and MARL algorithms are used to find policies for these agents via simulation. We will use game theory as a tool to measure the progress of running MARL algorithms, to qualitatively and quantitatively contrast performance of different agents, or populations of agents trained by different MARL algorithms.

The notion of *game* in game theory is very similar to that of finite horizon environments in RL, and for every RL environment in Table 2.1 there is a game-theoretical game equivalent. For instance MMDPs can be thought as k-person stochastic games [Shapley, 1953] in which the reward function, or payoff function in Game theoretical nomenclature, is the same for all agents. Similarly, a Markov game with a single state $|\mathcal{S}| = 1$ and discount factor[7] $\gamma = 0$ reduces a Markov game to a normal form game.

2.5.1 Normal Form Games

[7] In repeated games (normal form games that are repeated overtime) the discount factor can be thought of as the probability of the last repetition happening next round.

A *normal form game* is a tuple (Π, U, n) where n is the number of players with $\Pi = (\Pi_1, \ldots, \Pi_n)$ being the set of joint policies, one for each player. $U : \Pi \rightarrow \mathbb{R}^n$ is a payoff table mapping each joint policy to a scalar utility, equivalent to reward, for each player. The RL notion of state does not apply to normal form games, as these have a single initial state at which agents take actions simultaneously and are immediately rewarded, finishing the game.

$$\begin{array}{cc} & \begin{array}{cc}\mathbf{T} & \mathbf{S}\end{array} \\ \begin{array}{c}\mathbf{T}\\ \mathbf{S}\end{array} & \begin{pmatrix} (-2,-2) & (\ \ 0,-3) \\ (-3,\ \ 0) & (-1,-1)\end{pmatrix}\end{array}$$

Figure 2.6: The prisoners' dilemma, a famous general sum two-player normal form game. Two prisoners independently need to decide whether to talk (action T), blaming a crime on the other, or remain silent (action S). The values on each tuple entry represent the 1st and 2nd player payoffs respectively, each year in prison being associated with a utility of -1.

We define rationality as the objective of maximizing a utility function. Thus, rational players try to maximize their own expected utility. Each player i does so by selecting a so-called *pure strategy* from $\pi_i \in \Pi_i$ or equivalently by sampling from a mixture (distribution) over them $\pi_i \in \Delta(\Pi_i)$, also known as *mixed strategies*. A vector containing a strategy for each agent is known as a *strategy profile*, equivalent to policy vectors in MARL. The *value* v_i for a player i, given a policy vector $\boldsymbol{\pi}$ is the expected payoff obtained by player i if all players play their corresponding strategy prescribed by the strategy profile $\boldsymbol{\pi}$: $v_i = U_i(\boldsymbol{\pi})$. In Figure 2.6, each player has 2 pure strategies, T and S, and an infinite amount of mixed strategies. The notion of a mixed strategy can also be conceived as if a player was made up of a population of agents, with the probability weight on each pure strategy representing the proportion of agents in the population using such strategy. This idea will be further explored in Chapter 3.

A game is *zero-sum* if $\forall \boldsymbol{\pi} \in \Pi,\ \mathbf{1} \cdot U(\mathbf{\Pi}) = \mathbf{0}$, otherwise it is a *general-sum* game. A game is *symmetric* if all players feature the same policy set ($\Pi_1 = \ldots = \Pi_n$) and the payoff associated to each joint policy depends only on the policies and not on the identity of the players. Take Connect4 as an example of an asymmetric game, as all players have the same policy space, but because it is a turn based game the first player always moves first, and thus the in-game identity of the policy being followed affects the outcome of following a given strategy.

two-player normal form games ($n = 2$) are typically defined by a tuple ($\boldsymbol{A}$, $\boldsymbol{B}$), where $\boldsymbol{A} \in \mathbb{R}^{|\Pi_1| \times |\Pi_2|}$ gives the payoff for player 1 (row player), and $\boldsymbol{B} \in \mathbb{R}^{|\Pi_1| \times |\Pi_2|}$ gives the payoff for player 2 (column player). If $\boldsymbol{B} = \boldsymbol{A^T}$ the game is symmetric. Most importantly for us, if $\boldsymbol{B} = -\boldsymbol{A}$ the game is zero-sum. Exploiting this equality, two-player zero-sum games are often represented by a single matrix $\boldsymbol{A}$ containing the payoffs for player 1.

2.5.2 Best Responses and Nash Equilibriums

Game theory presents a *best response* as a notion of optimality against fixed agents. A best response denotes the best possible strategy that can be followed by one agent against a fixed set of other agents. If the other agents' strategies remain static, then their presence can be absorbed as part of the environment dynamics (its transition $\mathcal{P}$ and reward functions $\mathcal{R}_{1,\ldots,n}$), and thus we are presented with a similar framework as single agent RL. In Figure 2.6, the best response against a column player that always decides to remain silent, is to talk. This will yield the row agent payoff of 0, and no time spent in jail, instead of the alternative 1 year sentence. Formally, a (possibly mixed) policy π_i^{br} is a best response for player i against all other players' policies $\boldsymbol{\pi_{-i}}$ if playing π_i^{br} yields player i the highest possible payoff against strategies $\boldsymbol{\pi_{-i}}$:

$$\pi_i^{br} \in BR(\boldsymbol{\pi_{-i}}) \iff \forall \pi_i \in \Pi_i : \quad U_i(\{\pi_i^{br}, \boldsymbol{\pi_{-i}}\}) \geq U_i(\{\pi_i, \boldsymbol{\pi_{-i}}\}) \tag{2.14}$$

Despite the enticing simplicity of the idea of a best response, it does not account for the fact that other agents may change their strategies. And thus it is not suitable as an optimization objective in multiagent settings with multiple learning agents or scenarios where we want a strategy that is good against a large set of possible combinations of agents. In Figure 2.6, the previously computed best response of choosing to talk against a column agent that remains silent can be exploited by the latter. If the column agent decides to change its strategy to talking, then both agents will serve a sentence of 2 years. This necessitates a guideline for choosing strategies that is robust to changes to the other agent's policies. For this, game theory puts forth the definition of *Nash equilibrium* [Nash et al., 1950]. A Nash equilibrium is a strategy profile (one strategy for each player) such that each player's policy is a best response against all other player policies. For a game G of n players:

$$NE(G) = \{\pi_i \in BR(\boldsymbol{\pi_{-i}}) : \forall i \in \{n\}\} \tag{2.15}$$

In symmetrical games, all of the policy spaces are the same, like in the prisoners dilemma where both agents can only talk or remain silent. In these cases, instead of defining a strategy for each agent, we can describe the full Nash equilibrium by simply stating the (potentially mixed) strategy to follow by the first agent. Note that, depending on the game, there can be multiple or even infinite Nash equilibria.

For the prisoners' dilemma described in Figure 2.6, there is a single Nash equilibrium (NE) which prescribes each agent to talk. This is because it is the only combination of policies for which neither agent can unilaterally deviate its strategy to obtain individual improvement. In other words, there is no incentive to individually change strategies. This is unfortunate, because both agents would strictly obtain higher payoff if they were to bilaterally change their behaviour to remain silent. Definitions of equilibria such as Nash often support themselves on worst-case scenario options, such as all agents playing optimally with respect to their utility functions. This is clearly disadvantageous in most practical MARL applications, where agent strategies always feature sub-optimalities which can exploited. This is why in practical settings reward maximization might lead to a better policy than than one prescribed by a Nash equilibrium. Chapter 4 further builds on this idea.

Finally, take the famous game of Rock Paper Scissors (RPS), which is a symmetrical two-player zero-sum game. It can trivially be shown that there is a unique Nash equilibrium, this being a mixed strategy, the uniform distribution over all three actions: $NE(RPS) = \{\text{R} : \frac{1}{3}, \text{P} : \frac{1}{3}, \text{S} : \frac{1}{3}\}$. Figure 2.7 depicts an extension of RPS, where a copy of the Rock action is made. This simple modification has an interesting effect on the set of Nash equilibria. The original NE for RPS puts on R a probability weight of $\frac{1}{3}$, and thus by adding another action which is strategically equivalent, we can still compute a NE by distributing that $\frac{1}{3}$ probability weight over R_1 and R_2 indistinctively. Thus we can construct infinitely many Nash equilibria by choosing a $p \in [0, \frac{1}{3}]$ in $\{R_1 : p, R_2 : \frac{1}{3} - p, P : \frac{1}{3}, S : \frac{1}{3}\}$.

$$\begin{array}{c|cccc} & \mathbf{R_1} & \mathbf{R_2} & \mathbf{P} & \mathbf{S} \\ \hline \mathbf{R_1} & 0 & 0 & -1 & 1 \\ \mathbf{R_2} & 0 & 0 & -1 & 1 \\ \mathbf{P} & 1 & 1 & 0 & -1 \\ \mathbf{S} & -1 & -1 & 1 & 0 \end{array}$$

Figure 2.7: Rock-Rock-Paper-Scissors, an extension of Rock-Paper-Scissors with an extra action that is identical to playing Rock. It is a zero-sum two-player normal form game.

As will be explained shortly, sometimes it can be useful to unambiguously choose from a set of Nash equilibria. To this effect we have *maximum entropy Nash equilibria*, or maxent Nash. A Nash equilibrium is maximally entropic if the policies prescribed to each player are maximally indifferent between actions with the same empirical performance. For the case of Figure 2.7, the unique maxent Nash is: $\{R_1 : \frac{1}{6}, R_2 : \frac{1}{6}, P : \frac{1}{3}, S : \frac{1}{3}\}$. [Balduzzi et al., 2018] shows that symmetrical two-player zero-sum games with an antisymmetric[8] payoff matrix A have a unique maxent Nash. Throughout this thesis, to compute a maxent Nash we use the algorithm presented in [Ortiz et al., 2007].

2.5.3 Agent Evaluation via Winrate Matrices

This thesis adopts a form of *Behaviourism*. Agents are operationally characterized by all the ways that they can interact with all of the other agents and the environment itself. This interaction is made via the agent's policy, which means that we can fully define the end behaviour of an agent via its policy. However complex an agent's internal structure might be, during test time an agent's policy fully captures their behaviour inside an environment. Thus, to evaluate an agent performance it suffices to have it play against all available agents for a sufficiently large number of episodes.

We now present the tools required to answer the following questions:

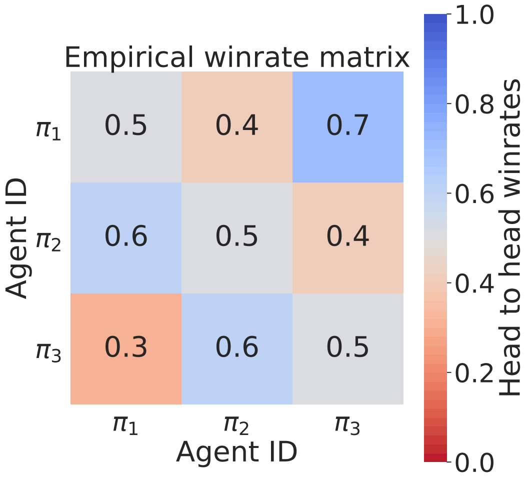
1. Given a single population of agents $\boldsymbol{\pi}$ how can we measure their relative strength?

2. Given two populations $\boldsymbol{\pi}^1$ and $\boldsymbol{\pi}^2$ which one is stronger?

The former requires an inter-population metric of performance while the latter a cross-population metric. Traditional metrics for measuring agent performance such as ELO [Elo, 1978] or its Bayesian derivation TrueSkill [Herbrich et al., 2007] suffer from undesirable effects. For instance, the ELO ratings of an agent can be artificially inflated by instantiating many copies of other agents that lose against it, and similarly for TrueSkill.

$$\begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix}$$

(a) Population



	π_1	π_2	π_3
π_1	0.5	0.4	0.7
π_2	0.6	0.5	0.4
π_3	0.3	0.6	0.5

(b) Single population empirical winrate matrix metagame

$$\begin{bmatrix} 0.245 \\ 0.51 \\ 0.245 \end{bmatrix}$$

(c) Maxent Nash

$$\begin{bmatrix} 2.043 \\ 2.61 \\ 0 \end{bmatrix} \times 10^{-7}$$

(d) Nash averaging

Figure 2.8: Given a population of agents (a), we might be interested in knowing what are the relative skills of such agents. First, we compute an empirical winrate matrix metagame (b), approximating each entry by simulating many head to head matches. From this, we can compute both the maxent Nash (c) and the Nash averaging (d). The former tells us that the second agent π_2 is the strongest, with the others being equally good. Note that Nash averaging yields very small values when computed on winrate matrices, and thus looking at the maxent Nash is often sufficient.

Inter-Population Evaluation Metrics

Let's assume that we have a set of n agents $\boldsymbol{\pi}$ which we would like to evaluate according to (1). How these agents were created is not relevant to us; these agents could use hand-crafted heuristics, be trained with reinforcement learning, evolutionary algorithms or any other method for developing game-playing agents. As an example of a method that will be used in Chapters 3 and 4, let $\boldsymbol{\pi}$ be a vector containing n different checkpoints of a learning agent trained under a MARL algorithm. For simplicity, let's assume a two-player zero-sum game like Connect4 with a single reward signal given at the end to each agent depending on whether they win (+1) or lose (-1).

Let $\boldsymbol{W_\pi} \in \mathbb{R}^{n \times n}$ denote an *empirical winrate matrix*. The entry $w_{i,j}$ for $i, j \in \{n\}$ represents the winrate of many head-to-head matches[9] of policy $\pi_i \in \boldsymbol{\pi}$ when playing against policy $\pi_j \in \boldsymbol{\pi}$. Such a mathematical abstractions of the underlying game, which is derived from emerging interactions of strategies being played in an environment is known as *metagame*. An interesting metagame definition that has recently received attention in multiagent system analysis [Omidshafiei et al., 2019] defines a two-player normal form game over a population of agents $\boldsymbol{\pi}$ such that the action set of each player corresponds to choosing an agent $\pi \in \boldsymbol{\pi}$ from the population to play the game for them. These players are sometimes referred to as meta-players, as they only play the game by proxy. This is intuitively similar to the previously mentioned equivalent nomenclature for normal form games, strategic form games.

Another flavour of a metagame are *evaluation matrices* [Balduzzi et al., 2019], a generalization of empirical winrate matrices. Instead of representing the win/loss ratio between strategies, it captures the payoff or score obtained by both the winning and losing strategy. That is, instead of containing win-rates for a given set of agents, an entry in an evaluation matrix $a_{ij} \in \boldsymbol{A}$ can represent the score obtained by the players or some other metric derived from simulated trajectories. This is useful for games that feature reward functions that go beyond rewarding only wins and losses.

By definition, evaluation matrices are represented by antisymmetric matrices in two-player zero-sum games [Balduzzi et al., 2019]. One can turn an empirical winrate matrix $\boldsymbol{W}$ into an antisymmetric matrix $\boldsymbol{A}$ by performing the element wise operation $a_{i,j} = w_{i,j} - \frac{1}{2}$, shifting the range of each entry from $[0, 1]$ to $[-\frac{1}{2}, \frac{1}{2}]$. As previously mentioned symmetrical two-player zero-sum games represented by an antisymmetric matrix $\boldsymbol{A}$ conveniently feature a unique maxent Nash equilibrium [Balduzzi et al., 2018].

Using all of these properties, *Nash averaging* is introduced as an inter-population evaluation method [Balduzzi et al., 2018]. Take an evaluation matrix $\boldsymbol{A_\pi}$ with its corresponding *unique* maxent Nash maxent-$NE(\boldsymbol{A_\pi}) = (\boldsymbol{p}, \boldsymbol{p})$. The Nash averaging for $\boldsymbol{A_\pi}$ is a vector

[8] A matrix A is antisymmetric iff: $-\boldsymbol{A} = \boldsymbol{A}^T$

[9] As previously mentioned, Connect4 is not a symmetrical game. This is due to the sequential nature of the environment, and thus for any pairwise match-up we have to separately compute the winrate when agent 1 plays first, and when it plays second. We can solve this by averaging the winrate of both positions. This slightly changes the formal definition of Connect4 to add a constraint that both agents are equally likely to start first, which we argue is a common sense rule.

$\boldsymbol{p_{na}} = \boldsymbol{A_{\pi}} \cdot \boldsymbol{p}$ containing one entry for each agent in the population. Each entry i in $\boldsymbol{p_{na}}$ indicates what is the value of agent i as an average over its performance against all other agents. Nash averaging features some useful properties. It is invariant to redundant agents, unlike ELO. Due to the uniqueness of a maxent-NE on antisymmetric matrices, it achieves certain levels of interpretability. This uniqueness means that given any evaluation matrix, there will be exactly one Nash averaging. It can evaluate both cyclic and transitive behavioural dynamics, which shall be described in Chapter 3. This is the main method of inter-population evaluation used in this thesis, depicted in Figure 2.8.

Cross-Population Evaluation Metrics

We can readily extend the notation of empirical winrate matrices to define it over multiple populations. Let $\boldsymbol{A_{\pi_1,\pi_2}} \in \mathbb{R}^{|\pi_1| \times |\pi_2|}$ be a matrix whose entries $w_{i,j}$ denote the average result from many head-to-head matches between agents $\pi_i^1 \in \boldsymbol{\pi_1}$ and $\pi_j^2 \in \boldsymbol{\pi_2}$. In this case, each meta-player chooses agents from a different population.

From this, we can introduce the *relative population performance* [Balduzzi et al., 2019], a cross-population measure of performance. Given two populations $\boldsymbol{\pi_1}, \boldsymbol{\pi_2}$, it yields a single scalar value comparing the performance of $\boldsymbol{\pi_1}$ against $\boldsymbol{\pi_2}$. It is computed by generating an evaluation matrix of one population versus the other A_{π_1,π_2} which is then treated as a two-player zero-sum game. A Nash equilibrium is then computed $NE(\boldsymbol{A_{\pi_1,\pi_2}}) = (\boldsymbol{p_{\pi_1}}, \boldsymbol{p_{\pi_2}})$ for the zero-sum game defined by A_{π_1,π_2}. The relative population performance is the value v for the meta-player 1: $v = \boldsymbol{p_{\pi_1}} \cdot A_{\pi_1,\pi_2} \cdot \boldsymbol{p_{\pi_2}^T}$. A positive v indicates that $\boldsymbol{\pi_1}$ wins on average against population $\boldsymbol{\pi_2}$, with the opposite being true if v is negative, $v = 0$ indicates both populations are equivalent. This is the main cross-population evaluation metric used in this thesis, depicted in Figure 2.9.

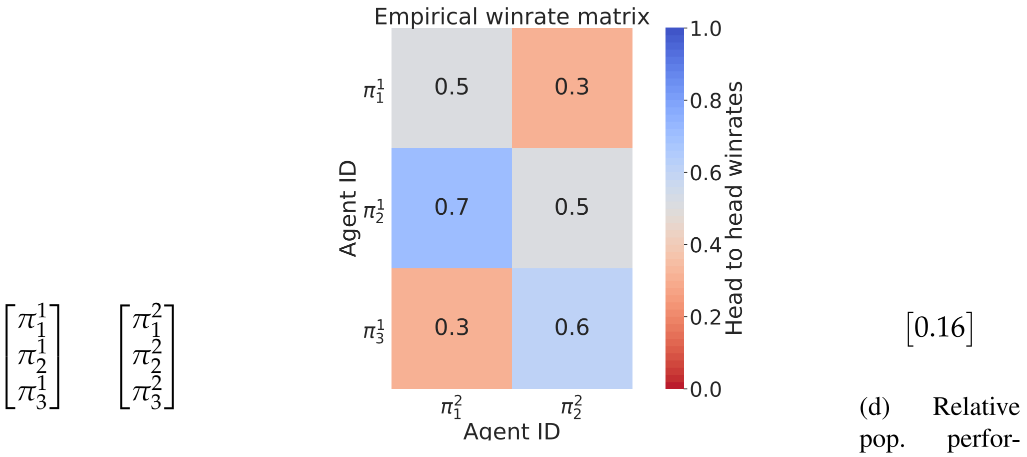
Now we are in a position to answer the question posited at the beginning of this Section 2.5.3. Nash averaging allows us to give an answer to question (1), and similarly, relative population performance gives us a way of reaching a solution for (2). We are now equipped to use game theory as a performance evaluator for future MARL trained agents.

We close the chapter by reviewing Monte Carlo Tree Search, an algorithm that will be heavily used in Chapters 4 and 5.

2.6 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is model-based, forward planning algorithm, developed for optimal decision making [Coulom, 2006]. It has enjoyed vast amounts of further work over the years, with some of its major early extensions surveyed in [Browne et al., 2012]. There have also been efforts trying to bridge the gap between MCTS and RL literature, understanding the former through the latter [Vodopivec et al., 2017]. In essence, MCTS is used to construct an ad-hoc policy for a *single* environment state: $a = \pi_{MCTS}(s \in \mathcal{S})$.

This will be the only model-based algorithm explored in this thesis. The decision to use MCTS was founded on it currently presenting state-of-the-art results in a wide variety of scenarios, from board-games [Silver et al., 2016, Silver et al., 2017a] to continuous action



(a) Pop. 1 (b) Pop. 2 (c) Cross-population empirical winrate matrix metagame (d) Relative pop. performance

Figure 2.9: Given two populations (a-b) of agents, we might be interested in obtaining a single scalar value denoting the relative performance of the first population against the second. First, we compute an empirical winrate matrix metagame $\boldsymbol{W}_{\pi^1,\pi^2}$ (c), where entry (i,j) denotes the winrate of agent π_i^1 against π_j^2, approximated by simulating many head to head matches. The relative population performance (d) tells us that, if we have two players whom want to maximally exploit their population in order to beat the opponent's population, player 1 will win on average 16% more often than player 2.

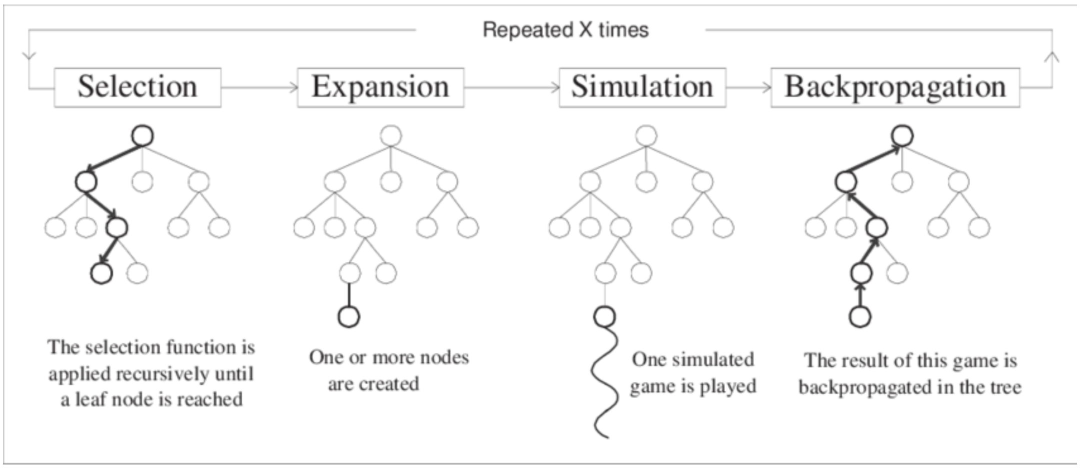

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 2.10: Overview of the Four Main Phases of Monte Carlo Tree Search. Taken from [Chaslot et al., 2006].

control environments [Mao et al., 2020]. Its vast applicability coupled with good performance entails that bringing improvements to this algorithm can have a greater impact in the field than potentially any other model-based algorithm.

Environments are represented in a tree graph structure, where nodes correspond to game states and edges represent a conjunction of actions and the corresponding transition reward. Hence, they are also called *game trees*, which is the primary names that will be used throughout this thesis. This is very similar to representing a game as a Markov chain with a tree structure, with the depth of the tree representing the dimension of time. Game trees for any game of interest will be unwieldy large, and therefore exhaustively traversing them to find out the most valuable action for a given state is unreasonable. MCTS proposes a solution which asymmetrically explores the areas of the game tree in an iterative fashion. It explores the game tree by choosing areas of perceived high value, while selectively pruning (i.e ignoring from further search) the least valuable ones. We now turn to the explanation of its inner workings, Figure 2.10 depicts the 4 main phases by which MCTS operates, while Algorithm 7 describes it in pseudocode.

```
Algorithm 7: MCTS Loop
    Input: $rootstate \in mathcalS$
    Input: $budget \in \mathbb{N}$
1   $rootnode \leftarrow$ Node(rootstate);
2   $i \leftarrow 0$;
3   repeat
4       $node \leftarrow rootnode$;
5       $state \leftarrow$ clone($rootstate$);
6       $node \leftarrow$ selection($node, state$);
7       $node \leftarrow$ expansion($node, state$);
8       $outcome \leftarrow$ rollout($state$);
9       backpropagation($node, outcome$);
10      $i \leftarrow i + 1$
11  until $i \geq budget$;
12  $a_{best} \leftarrow$ action_selection($rootnode$);
    Result: $a_{best}$
```

Starting from a single node representing the state for which MCTS is tasked to find an optimal action for (Line 1), it begins its main loop to construct a game tree representing the areas that it has already explored of the real environment. At every iteration of the loop a fresh copy of the environment is made (Line 5). Then, a selection strategy (Line 6) is used to traverse the cloned game tree towards its perceived most promising areas, acting accordingly in the cloned environment. This selection strategy is tasked with dealing with the exploration vs exploitation dilemma. That is, choosing to explore areas that have not been sampled often for hopes of further gains versus reaffirming the relative superiority of the current most promising areas via further simulation. Once a leaf node has been selected, a simulation rollout is carried out using a rollout policy to act within the cloned environment (Line 8) until a terminal state (or other condition) is reached. The reward associated with this final state is backpropagated upwards through the tree (Line 10). This entire process is repeated until a given computational budget is spent. Finally, the most promising action to take, according to the statistics stored in the game tree, is chosen and executed in the real environment. The entire search tree is discarded at the end, entailing that there is no learning happening in between MCTS procedures.

Being a model-based algorithm, MCTS requires the ability to generate clones of the environment and evaluate arbitrary environment transitions in order to play simulations outside of the real environment. It is a best-first algorithm, iteratively expanding the tree by choosing the most promising node chosen according to a specified rule. It is an any-time algorithm, as it can produce a final outcome with any amount of compute. With minimal computational resources it tends to perform poorly. With an infinite computational budget and under certain conditions and hyperparameter choices, the action chosen by MCTS has been proven to be the same as the minimax action [Bouzy, 2007, Kocsis and Szepesvári, 2006], the action that minimizes losses in worst case scenario. The concept of computational budget normally takes the shape of a fixed amount of iterations, or a fixed amount of computational time. The former is used throughout this thesis.

MCTS features many moving parts, and there are a plethora of possible implementations, each one of them making slightly different assumptions about either the agents in the environment or the environment itself. Thus, for completeness, we now briefly describe each of the phases of MCTS and the design choices made for the MCTS implementation in this thesis.

2.6.1 Search Game Tree

We use an open-loop MCTS implementation [Silver et al., 2017b], each node in the tree represents a sequence of actions taken from the root node s_{root}, instead of the environment state itself, which would be the case for closed-loop implementations. For notational convenience, we use s to represent such sequence of actions, as it can be analogously be thought of as the state of the tree, even if it does not correspond to the state of the actual simulated environment. Each edge (s,a), represents taking action a at state s. Each edge stores a set of statistics:

$$\{N(s,a), Q(s,a), P(s,a), i, A_n\} \tag{2.16}$$

$N(s,a)$ is the number of times that edge (s,a) has been visited. $Q(s,a)$ is the combined mean action-value for that edge, aggregated from all simulations that have traversed the edge (s,a) so far. $P(s,a)$ is the prior probability of following edge (s,a), which is only required for some selection strategies. In sequential games, the index i denotes either the player who will act in node s. Finally, A_n indicates the actions available from node n.

2.6.2 Selection Phase

The selection phase chooses which part of the tree will be explored next, based on the previous information collected during the MCTS run. Hence, this phase is most responsible for the direction of growth of the tree. Beginning in the root node with a fresh clone of the real environment's current state, it recursively selects which edge to follow according to a *selection strategy*. As edges are being traversed, the action corresponding to that edge is executed in the simulated environment. The recursion stops upon reaching either a terminal node or a node that has not been fully expanded. A node is considered fully expanded if all of its child nodes have been visited at least once.

The most commonly used selection strategy is Upper Confidence bound for Trees (UCT) [Kocsis and Szepesvári, 2006]:

$$UCT(n) = \arg\max_{a \in A_n} Q(s,a) + c_{UCT}\sqrt{\frac{\log \sum_{a'} N(s,a')}{N(s,a)}} \tag{2.17}$$

Where c_{UCT} is a constant that allows a trade-off between exploration and exploitation. A lower value for c increases the importance of exploitation, and a higher value biases the selection towards exploration. Many other equations for selection have been proposed in the literature, and we would like to focus on PUCT [Gelly and Silver, 2007, Rosin, 2011], an extension of UCT which takes into account a prior value for each edge to follow. This allows us to add offline[10] information and thus opening the potential to add learning capabilities to MCTS. This plays a crucial role in Chapter 4, where we bring opponent awareness to the computation of the action priors $P(s,a)$, to explore the effects of biasing MCTS towards a best response against the opponents in an environment.

$$PUCT(n) = \underset{a \in A_n}{\arg\max}\ Q(s,a) + c_{PUCT} \frac{P(s,a)\sqrt{\sum_{a'} N(s,a')}}{1 + N(s,a)} \tag{2.18}$$

2.6.3 Expansion Phase

There are many potential expansion strategies [Chaslot et al., 2008], and the choice of expansion strategy often depends on the chosen selection strategy. We align our choice of expansion strategy with the most common used ones in the literature. When using the PUCT selection strategy as in Chapter 4, we expand all children of the leaf node selected via the selection strategy. That is, we initialize all child nodes with default values for the statistics from Equation 2.16 and a further action prior $P(s,a)$ for each edge. When using UCT as in Chapter 5 one of the leaf node's unvisited child nodes is chosen at random, expanding the tree by adding one more node and initializing it with default values for the statistics from Equation 2.16.

2.6.4 Rollout Phase

A rollout is a Monte Carlo (i.e random) tree traversal that begins in a node of the game tree and ends in a terminal node. It is a game simulation that takes place in a cloned version of the real game being played where a rollout policy is used to take actions in it. The simulation of the rollout phase begins in the last node expanded in the expansion phase and actions are taken for all players according to a *rollout strategy*. The vanilla version of MCTS uses a rollout strategy that plays randomly.

2.6.5 Backpropagation Phase

Once a terminal node is reached during the rollout phase, the result of the simulation is observed. For vanilla MCTS, the result is a binary random variable, either a win or a loss. In Chapter 5 this is expanded to arbitrary score values. This result is backpropagated through both the newly expanded node, and all the nodes selected during the selection phase. To backpropagate the result means to update the statistics described in Equation 2.16 of all nodes on the path from the root of the tree to the node where the rollout began.

2.6.6 Action Selection Phase

Once the computational budget has run out, the last step in any MCTS algorithm is to select an action to take in the actual game being played via a *final selection strategy*. The most common ones are to select the action corresponding to either (1) the edge with the higher

[10] By offline we mean information that is not computed *during* search.

visitation count $N(s,a)$ or (2) the edge with the higher $Q(s,a)$. Arguably, the former is closer to the notion of best-first heuristics. The most visited child node must correspond to the most valuable action explored by MCTS. During the experiments involving MCTS in Chapters 4 and 5 we found no significant difference between (1) or (2).

2.7 Summary

In this chapter we have thoroughly explored the three levels of the MARL stack presented in Chapter 1, furthering our understanding of what are the relevant goals and challenges presented at each level of abstraction. As a summary, the overarching goal of MARL approaches is to use a training scheme to guide learning agents towards high quality behaviours in an environment, with modern MARL approaches using game theory as a means to benchmark and evaluate agent and population performance, aiding during training to find better policies and during testing to verify their strength in a grounded fashion.

Efforts put into clarifying the fundamental levels of abstraction used by MARL approaches already reduce the technical understanding required by practitioners to adopt MARL as a way of generating gameplaying agents, and we hope that the efforts put to that end in this chapter help towards such goal. Yet a much bigger problem preventing MARL from being used in the games industry is the prohibitive computational cost required to train agents. The running message throughout the remaining chapters of this thesis is to use opponent awareness as a means of identifying the most efficient training schemes and agent level algorithms between existing approaches and to incorporate opponent awareness into these so as to improve on their computational efficiency. Chapter 3 will make a comparison under a novel generalized framework of many state-of-the-art training schemes, informing on the quality of existing training schemes and providing a tool to generate new opponent aware training schemes. Chapter 4 will introduce opponent awareness within MCTS as a means of improving the sample efficiency of state-of-the art model-based MARL algorithms. Finally, Chapter 5 uses MARL as a tool to aid game designers, taking opponent awareness as an approach to define the game design task of game balancing.

Chapter 3

Generalized Self-Play framework

3.1 Introduction

In the classical single agent reinforcement learning (RL) scenarios described by [Sutton and Barto, 1998], where a stationary environment is modelled by an MDP or POMDP, a solution concept readily presents itself, as showcased by Table 2.1 in Section 2.2 in the previous chapter. These environments are solved by computing a policy which yields the highest possible episodic reward. However, it is not clear how to define a pragmatic solution concept when training a single policy in a multiagent system. On the agent level, it is simple, for an agent only aims to maximizing its observed reward. However, it is not as clear for the higher level of training schemes. As game theory tells us, an agent's optimal strategy is dependent on behaviours of the other agents that inhabit the environment, such as a Nash equilibrium in zero-sum games. Therefore, a potential solution is to find a policy that maximizes its expected reward obtained against the *entire* set of all possible other policies in the environment. A formalization of this idea for an arbitrary Markov Game E, for an agent i, is to compute a policy (π_i^*) which maximizes its expected reward obtained when acting in an environment across the *entire* set of all possible other policies Π_{-i} in the environment:

$$\pi_i^* = \arg\max_{\pi_i \in \Pi_i} \int_{\pi_{-i} \subseteq \Pi_{-i}} \mathbb{E}_{a_t \sim \pi; s_{t+1}, r_t \sim P(s_t, a_t)} [\sum_{t=0}^{\infty} \gamma^t r_t] \tag{3.1}$$

This is, of course, intractable in all but toy scenarios. Furthermore, this gives equal importance to any combination of opponent policies, which is undesirable. Also, because this integration will take place over a large amount of weaweak policies that other players would possibly never play. Traditionally, to solve this issue in the context of competitive games, the focus of this chapter, MARL methods would use a centralized training scheme. With an opponent policy-deciding module which would train a single policy by matching it against a set of *preexisting* and *fixed* policies, using as a success metric the relative performance against these fixed agents. By preexisting it is meant that these agents are assumed as input to the entire training procedure; For a policy to be fixed means that there is no learning happening, it does not change over time. These methods rest on two assumptions. Firstly, the availability of benchmarking policies to train and test against. Secondly, these existing policies dominate, in a game theoretic sense, most of the policy space. Thus, it would not be necessary to compute the expectation over the entire policy space, using as a proxy an expectation over the preexisting policies. However, this approach has drawbacks. If this benchmarking set is too small, the trained policy may overfit to the behaviour of the agents it was trained with, and thus prone to being exploitable by other policies.

Another question to answer is, where do these preexisting agents come from? The field of RL offers multiple methods for computing these benchmarking policies which must be available before training commences. To name a few, these can be generated using supervised learning on datasets of expert human moves to bias learning a policy towards expert human play via imitation learning [Silver et al., 2016] [Tesauro, 1992] or further improve upon human experience with Offline RL [Yu et al., 2021, Kumar et al., 2020, Levine et al., 2020]; they can be tree-search based algorithms using hand-crafted evaluation functions or MCTS based approaches if an environment model is present [Browne et al., 2012]. Some methods are as creative as deriving a strong policy by using off-policy methods on video replays [Aytar et al., 2018, Malysheva et al., 2018]. This is by no means an exhaustive list, even within the field of RL, but it showcases the wide reach and creativity that RL can bring to the generation of game-playing agents for the games industry.

What about the cases in which we don't have access to these learning resources? Such as when developing a new game for which no prior expert information is known, without access to a fast enough environment model and for which any hand-crafted evaluation function yields a fruitless policy. A priori methods such as optimistic policy initialization are still permitted [Machado et al., 2014], but these only prime untrained policies into a potentially beneficial learning path, without generating valuable policies themselves. Thus, under such constraints, there is little room to compute a set of good benchmarking policies, let alone a set of dominating ones. In these cases one viable option is to proactively discover this set of policies.

As early as the 1950s, authors such as Samuel [Samuel, 1959] began experimenting on self-play (SP) as a training scheme for MARL. An SP which training scheme trains a learning agent *purely* by simulating play with a copy of itself, or fixed policies generated during training. These generated policies can dynamically build a set of benchmarking policies during training. Such a set can potentially be curated to remove dominated (policies which are always beaten by other policies) or redundant (i.e duplicated) policies. Thus, SP can be considered as an open-ended centralized training scheme for MARL, a it poses and masters a sequence of objectives (the policies chosen to be used by other agents in the environment) rather than optimizing against a pre-specified objectives [Balduzzi et al., 2019]. Once we leave behind the limiting approach of training against a fixed and known set of policies in favour of SP, it is of paramount importance to define meaningful metrics to inform this open-ended learning process as the responsibility of searching the policy space falls within the realm of the training scheme. Fortunately, recent years have seen the introduction of metrics for multiagent evaluation, stemming from game theory [Balduzzi et al., 2019] or dynamical systems analysis [Omidshafiei et al., 2019], which serve as guiding mechanisms towards the exploration of the policy space. The place of SP within the wider context of MARL training schemes is visually captured in Figure 3.1.

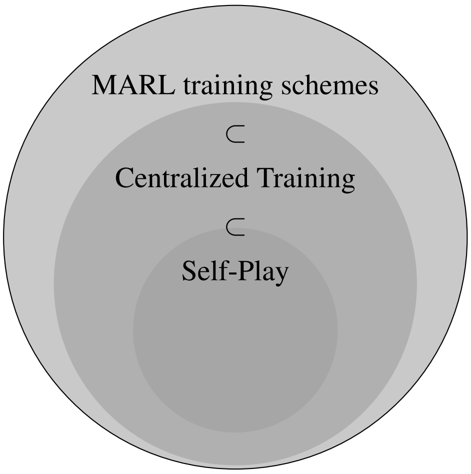


Figure 3.1: Self-play's place in the landscape of MARL training schemes visualized via an inclusion graph.

3.1.1 Defining Self-Play

The notion of self-play, albeit often cited in multiagent Reinforcement Learning as a process by which to train agent policies from scratch, has received little efforts to be taxonomized within a formal model. Even worse than that, authors normally skip explanations regarding the inner workings of the self-play training schemes they use or brush them off in passing, preferring to dedicate more detailed explanations about how the underlying learning agents transform sampled trajectories into policy updates. As a notable example, consider this passage from the now famous paper introducing AlphaGo [Silver et al., 2016], which we will improve upon in Chapter 4. The algorithm they introduce is of a complicated nature, combining model-based tree search with an actor-critic RL algorithm, and it receives a richly detailed explanation. In contrast, here is the description of their training scheme, in terms of deciding which are the fixed opponents that the training agent will play against:

> *We play games between the current policy network and a randomly selected previous iteration of the policy network. Randomizing from a pool of opponents in this way stabilizes training by preventing overfitting to the current policy.*

Not only is this description of their training scheme brief, but it also is *the only mention of their entire self-play procedure* in the publication. This quote features the strong claim that randomizing over a pool opponents prevents overfitting, which in their defense is indeed correct, as we shall see in Sections 3.7 , 3.8 and 3.9. Yet, without a formal specification of the training scheme, it is hard to reproduce this claim, verify it and contrast it against potential alternatives.

Historically, SP lacks a formal definition, and notation is often not shared among researchers. This has led to isolated, and sometimes conflicting, conceptions of what constitutes SP as a training scheme in MARL. Undoubtedly, a formally-grounded framework with rigorous and unified notation will strengthen the field of SP MARL and allow for incremental efforts on existing and future contributions to be captured on a shared language. In this vein, this chapter introduces a formalized framework in Section 3.3, with clearly defined assumptions, encapsulating the meaning of self-play as abstracted from a variety of commonly used self-play algorithms. This framework is cast as an approximation to a theoretical solution concept for multiagent training. To showcase that the choice of self-play algorithm does indeed affect the learning dynamics of the underlying agents, a novel qualitative visualization metric is introduced in Section 3.5. On a simple environment, we show that the choice of self-play algorithm affects the generation of episode trajectories leading to different explorations of the policy space by the learning agents. These empirical findings are further developed by a quantitative study in Section 3.6.2. On two environments, we analyze the learning dynamics of policies trained under different self-play algorithms captured under our framework and perform cross self-play performance comparisons. The outcome of this experiment in Sections 3.7, 3.8 and 3.9 shows that, throughout training, various widely used self-play algorithms exhibit cyclic policy evolutions and that the choice of self-play algorithm significantly affects the final performance of trained agents.

3.2 History of Self-Play

The notion of SP has been present in the game-playing AI community for over half a century. [Samuel, 1959] discusses the notion of learning a state-value function to evaluate board positions in the game of checkers[1], to later inform a one step look-ahead tree search algorithm to traverse more effectively the search space. This learning process takes place as the opponent uses the same state-value function, both playing agents updating simultaneously the shared state-value function. This training scheme was named self-play. The TD-Gammon algorithm [Tesauro, 1995] featured SP to learn a policy using TD(λ) [Sutton and Barto, 1998] to reach expert level backgammon play. This approach surpassed previous work by the same author, which derived a backgammon playing policy by performing supervised learning on expert datasets [Tesauro, 1990]. More recently, AlphaGo [Silver et al., 2016] used a combination of SP and supervised learning on a dataset of expert moves to beat the world champion of Go. This algorithm was later refined [Silver et al., 2018], removing the need for expert human moves. A policy was learnt purely by using a mix of supervised learning on moves generated by an agent which combined MCTS with an actor-critic RL architecture trained under SP, similarly introduced in [Anthony et al., 2017]. These works echo the sentiment that superhuman AI need not be limited or biased by preexisting human knowledge.

In the game of Othello, [Van Der Ree and Wiering, 2013] experimented with training single agent RL algorithms using two different training schemes: SP and training versus a fixed opponent. Their results show that, depending on the RL algorithm used, learning by SP yields a higher quality policy than learning against a fixed opponent. Concretely, TD(λ) learnt best from self-play, but Q-learning performed better when learning against a fixed opponent. Similarly, [Firoiu et al., 2017] found that deep Q-Network (DQN) [Mnih et al., 2013], a deep variant of Q-learning, did not perform well when trained against other policies which were themselves being updated simultaneously, but otherwise performed well

[1] https://simple.wikipedia.org/wiki/Checkers

when training against fixed opponents in the fighting game *Super Smash Brothers: Melee*. We note that that the environments in their experiments differ significantly to draw parallel conclusions.

It is often assumed that a training scheme can be defined as SP if, and only if, all agents in an environment follow the same policy, corresponding to the latest version of the policy being trained. Meaning that, when the learning agent's policy is updated, every single agent in the environment mirrors this policy update. We refer to this SP method as *naive* SP, as it is the simplest instantiation of an SP algorithm as we shall see in Section 3.4. [Bansal et al., 2017] relaxes this assumption by allowing some agents to follow the policies of "past-selves". Instead of replicating the same policy over all agents, the policy of all of the non-training agents can *also* come from a set of *fixed* "historical" policies. This set is built as training progresses, by taking *checkpoints*[2] of the policy being trained. At the beginning of a training episode, policies are uniformly sampled from this "historical" policy set and define the behaviour of some of the agents in the environment. The authors claim that such a version of SP aims at training a policy which is able to defeat random older versions of itself, ensuring continual learning. This notion of "choosing policies from a historical set" allows for two decision points: (1) Which agents will be added into this "historical" set of policies and (2) which of these agents will populate the environment at any given time. Different takes on (1) and (2) spawn different SP algorithms.

From this scenario, consider the following: each *combination* of fixed policies sampled as opponents from the "historical" dataset can be considered as a separate MDP[3]. This is because by leaving a single agent learning in a stationary environment, the fixed agents' influence on the environment is stationary [Laurent et al., 2011]. This is of genuine importance, given the convergence properties of most RL algorithms heavily rely on the assumption of a stationary environment. SP algorithms can leverage the assumption that they are using SP, so they can provide the learning agent with a label denoting which combination of agent behaviours inhabits the environment, a powerful assumption in transfer learning [Sutton et al., 2007] and multi-task learning [Taylor and Stone, 2009]. In fact, there are already multitask meta-RL algorithms, which assume knowledge of a distribution over MDPs that the agent is being trained on, such as RL^2 [Duan et al., 2016]. Note that a SP algorithm featuring a growing set of "historical" policies will introduce a non-stationary distribution over the policies that will inhabit the environment during training. It ensues that the distribution over the set of MDPs encountered by the training agent becomes non-stationary. In other words, if self-play decides which policies will define the behaviour of other agents in the environment, by choosing a set of fixed strategies to play against, we are essentially fixing an environment for the agent under policy π to act on. The SP module presents the learning agent π with a non-stationary sequence of environments throughout the duration of the training.

Recently, Berner *et al.* [Berner et al., 2019] trained a team of RL agents using SP to achieve superhuman level performance in the competitive team-based game of Dota 2. During training, the team would play 80% of the games using *naive* SP while the remaining 20%

[2] For deep RL, this is equivalent to freezing the weights of the neural networks which represent an agent's policy.

[3] This can be trivially extended to POMDPs if there is partial observability.

were played against "past-selves". The probability of facing any of these previous policies depends on a per-policy metric, which is updated during training, evaluating how much there is to learn from a policy. AlphaStar [Vinyals et al., 2019] reached Grandmaster level in StarCraft II with various policies by using a combination of various SP algorithms [Jaderberg et al., 2019]. Part of their training pipeline relied on training a set of “exploiter” policies, which focus on exploiting specific policies under training, relaxing the need for them to be robust to all opponents.

Lanctot *et al.* [Lanctot et al., 2017] defines the Policy-Space Response Oracles (PSRO) family of algorithms, unifying various game theoretical algorithms for multiagent training. PSRO algorithms tackle this problem by iteratively generating relative to an existing set of policies. These algorithms iterate over the following loop: a meta-game (as defined in Section 2.5) is defined over the current set of policies, for which a "solution" is computed, and from this solution one or more policies are added to the set of policies. The choice of solution concept and the procedure to generate new policies from this concept is the differentiating factor between PSRO algorithms. There are current efforts to show convergence properties of some PSRO algorithms [Muller et al., 2019,Balduzzi et al., 2019] towards existing multiagent solutions [Omidshafiei et al., 2019]. The framework proposed within this chapter shares the spirit of creating a generalised framework to encompass existing algorithms, but with a focus on MARL literature instead of game theory.

3.3 Generalized Self-Play Algorithm

Here we present the mathematical formulation and required assumptions for a formal framework which encapsulates the notion of self-play in the context of MARL. It allows for the creation and comparison of existing and future SP algorithms.

Self-play training schemes can be conceived as modules which extend the MARL loop by introducing a functionality prior to, and after, every episode. Let π be the only policy being trained throughout the MARL loop. An SP scheme envelops the MARL loop by first deciding which policies $\boldsymbol{\pi'}$, taken from a set of *fixed* policies $\boldsymbol{\pi'} \subseteq \boldsymbol{\pi^o}$, will define the agents’ behaviour for the next episode. This *excludes* the agent whose behaviour is defined by π, the learning agent. Once the episode ends, a function G decides whether or not a frozen copy of the possibly updated policy π will be introduced in the pool of available policies $\boldsymbol{\pi^o}$. This intuition is formally captured in Algorithm 8, which presents our SP framework inside a Partially Observable Stochastic Game (POSG) loop, which as a reminder is an *n-player*, *general-sum*, *partially-observable* environment. The steps belonging to the SP scheme have been highlighted in orange. For completeness and in agreement with Table 2.1 from the previous Chapter, Algorithm 9 showcases our SP framework on dec-MDPs, Algorithm 10 for dec-POMDPs and finally Algorithm 11 for MMDPs. It is clear that our framework is agnostic to the environment model, as the core loop of all environments are unaffected by our additional logic, which is not surprising given that training schemes and environments sit on different level of abstractions on the MARL stack proposed on this thesis in Figure 1.1.

The attentive reader will realise that the presented algorithms only update policy π once at the end of each episode, and this does not allow for Temporal Difference (TD) learning [Sutton and Barto, 1998] algorithms to be trained, as these rely on updating policy π any arbitrary number of environment steps. One can readily move the *update* functionality to be called on the inner loop (during gameplay). We have decided on keeping it on the outer loop for readability purposes.

3.3.1 Framework Definition

We define a SP module or training scheme by formalizing the notation of the *menagerie* $\boldsymbol{\pi^o}$, the *policy sampling distribution* Ω, and the *gating function* G. Specified by the tuple $< \Omega(\cdot|\cdot,\cdot), G(\cdot|\cdot,\cdot) >$:

- $\boldsymbol{\pi^o} \subseteq \Pi_i$; the *menagerie*. A set of policies from which agents' behaviour will be sampled. This set always includes the currently training policy π. A constraint is placed over $\boldsymbol{\pi^o}$. All of its elements must be derived, at least indirectly, from π, the policy being trained. Hence, all policies in the menagerie are elements of π's policy space. Due to the fact that all policies in the menagerie come from policy space Π_i, we dropped the i index in the menagerie's for improved clarity in the notation. The menagerie *can* change as training progresses by the curator function described below.
- $\Omega(\boldsymbol{\pi'} \in \Pi_{-i}|\boldsymbol{\pi^o} \subseteq \Pi_i, \pi \in \Pi_i) \in [0,1]$; where $\boldsymbol{\pi'} \subseteq \boldsymbol{\pi^o}$; The *policy sampling function*. It chooses which policies, apart from π, will inhabit the environment's agents. Conditioned on the menagerie $\boldsymbol{\pi^o}$ and the current policy π being trained, it outputs a probability distribution over the menagerie $\boldsymbol{\pi^o}$ from which agents will be sampled to act as fixed agents to train against. We denote the set of agents sampled from the policy sampling distribution given a menagerie and a policy used by the learning agent thus: $\boldsymbol{\pi'} \sim \Omega(\boldsymbol{\pi^o}, \pi)$.
- $G(\boldsymbol{\pi^{o\prime}} \subseteq \Pi_i|\boldsymbol{\pi^o} \subseteq \Pi_i, \pi \in \Pi_i) \in [0,1]$; The *curator* of the menagerie or gating function. Conditioned on the menagerie $\boldsymbol{\pi^o}$ and the current policy π being trained, it outputs a probability distribution over menageries. We denote the menagerie sampled from the curator given an input menagerie and a policy used by the learning agent thus: $\boldsymbol{\pi^{o\prime}} \sim \Omega(\boldsymbol{\pi^o}, \pi)$. For deterministic menageries, we simply write $\boldsymbol{\pi^{o\prime}} = \Omega(\boldsymbol{\pi^o}, \pi)$. The curator serves two purposes, which complex curators could break into two functions:
  - G decides if the current policy π will be added to in the menagerie. This decision could be made on any arbitrary measure such a quantified metrics of behavioural difference with respect to agents in the menagerie [Kanervisto et al., 2020].
  - G decides which policies in the menagerie, $\pi \in \boldsymbol{\pi^o}$, will be discarded from the menagerie. For instance [Berner et al., 2019] discards policies in the menagerie if it is deemed that there is nothing left to learn by playing against them. We note that discarding a policy from the menagerie is equivalent to giving a 0 probability weight by the opponent sampling distribution Ω.

The definitions above capture the minimal structure of all SP training schemes. However, it is possible to condition both the policy sampling distribution Ω and curator G on any other variables. For instance, it could be interesting to define an SP algorithm whose components are conditioned on episode trajectories, which has proved useful in RL research [Schaul et al., 2015].

Finally, to highlight similarities between fields, we note that the curator and menagerie bear resemblance with the notion of Hall of Fame from evolutionary algorithms [Nogueira et al., 2013, Liskowski, 2013]. A mechanism that keeps a set of *elite* policies, agents with the highest fitness[4] from a population of agents. Hall of Fame algorithms also consider the problem of curating a policy set over time.

3.3.2 Defining Assumptions

As it was established in Chapter 2, multiagent training schemes define a set of assumptions made about the relationship between agents and environment. These assumptions both constrain and enable different types of agent algorithms to be developed within the scope of such training scheme. Our SP framework is no exception. Thus, we explicitly posit the following assumptions for any SP algorithm covered by our framework:

Assumption 1: *The policies present in the environment can either be exact copies of the policy being trained, or policies derived indirectly from it, taken from the menagerie.*

Assumption 2: *Prior, during and after a training episode, the SP module has access to the agents' policy representations*[5]*. Allowing any-time read and write rights for all policies.*

3.3.3 SP as a MARL Solution Concept

Assumption 3: *There exists a set of policies, $\boldsymbol{\pi} \subseteq \Pi$, significantly smaller than the entire original policy space, $|\boldsymbol{\pi}| \ll |\Pi|$, which we can use in place of Π in equation 3.1 such that a solution to the modified equation yields a policy which remains optimal in the environment. If so, the integration over the proxy policy space becomes computationally tractable, making equation 3.1 computationally solvable.*

The assumption above is very similar to the vision of [Czarnecki et al., 2020], which states that to solve an environment is to find the "Nash of the game", defined by the authors as the set of policies which collectively dominate the rest of the policy space. The policy sampling distribution Ω and the gating function G are tools by which a menagerie $\boldsymbol{\pi^o}$ can be computed and curated over time. Self-play can be conceived as a bottom up approach towards computing a set of policies, $\boldsymbol{\pi^o}$, to be used as a proxy for the entire policy space Π in equation 3.1 which would in turn be the "Nash of the game". The obvious fact that an agent cannot act according to a policy outside its policy space means that a menagerie can only contain policies of a single policy space. Consequently, for environments with disjoint policy spaces, SP may be unable to serve as an approximate solution to equation 3.1. A

[4] Fitness in evolutionary computation is largely similar to the expected reward of a policy in a given environment, which we have previously denoted as $V(\pi, t)$ for a policy π and task or environment t.

[5] If the policies are being represented by a neural network. Access to the policy representation means access to the neural network topology and weights.

potential approach that lies outside the studies presented in this thesis could be to keep a separate SP procedure running for each separate policy space as in [Lanctot et al., 2017]. For completeness, one could expand the use of this framework to assymetric games by adding a chance node at the beginning of every episode which decides what side an agent will play on (in the case of boardgames).

[Balduzzi et al., 2019] introduces the notion of the *gamescape*, a polytope which geometrically encodes interactions between agents for zero-sum games. They derive a set of algorithms whose goal is to grow and curate an approximation to this polytope. We draw parallels between their work and the idea of using SP algorithms to compute a proxy for a target policy space.

3.3.4 Transitivity and Cycles in 2-player Zero-sum Games

In [Balduzzi et al., 2019] the authors show that 2-player zero sum games can be broken down into transitive components (skill levels) and cyclic components (viable strategies within a given skill level). By analyzing the effect of SP algorithms on a heavily cyclic game we can observe how an algorithm deals with the obstacle of catastrophic forgetting, learning to beat new policies by unnecessarily deteriorating its performance against known policies. Analyzing SP algorithms on a heavily transitive games gives us information about the speed at which these algorithms increase the skill of the learning agent [Czarnecki et al., 2020]. We use this notions to guide our experimental designs in Section 3.6.

Algorithm 8: POSG Loop with Self-Play

```
Input: Environment: $(S, A, O, \mathcal{P}(\cdot,\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \rho_0)$
Input: Self-Play Scheme: $(\Omega(\cdot|\cdot,\cdot), G(\cdot|\cdot,\cdot))$
Input: Initial policy: $\pi \in \Pi_i$
$\pi^o = \{\pi\}$ ;                                  // Menagerie initialization
for $e = 0, 1, 2, \dots$ do
    $\pi' \sim \Omega(\pi^o, \pi)$ ;                  // Sample from menagerie
    $\pi = \pi' \cup \{\pi\}$;
    $s_0, \boldsymbol{o_0} \sim \rho_0$;
    for $t = 0, \dots, termination$ do
        $\boldsymbol{a_t} \sim \boldsymbol{\pi}(\boldsymbol{o_t})$;
        $s_{t+1}, \boldsymbol{o_{t+1}} \sim P(s_t, \boldsymbol{a_t})$;
        $\boldsymbol{r_t} \sim \boldsymbol{R}(s_t, \boldsymbol{a_t})$;
        $t \leftarrow t + 1$;
    end
    $\pi \leftarrow update(\pi)$;
    $\pi^o \sim G(\pi^o, \pi)$ ;                      // Curate menagerie
    $e \leftarrow e + 1$;
end
return $\pi$;
```

Algorithm 9: dec-MDP Loop with Self-Play

```
Input: Environment: $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \boldsymbol{\mathcal{R}}(\cdot,\cdot), \gamma, \rho_0)$
Input: Initial policy: $\pi(\cdot)$
$e \leftarrow 0$;
$\boldsymbol{\pi}_0^o = \{\pi\}$
for $e = 0, 1, 2, \ldots$ do
    $\boldsymbol{\pi}' \sim \Omega(\boldsymbol{\pi}_e^o)$;
    $\boldsymbol{\pi}_e = \boldsymbol{\pi}' \cup \{\pi\}$;
    $s_0 \sim \rho_0$;
    $t \leftarrow 0$;
    repeat
        $\boldsymbol{a_t} \sim \boldsymbol{\pi_e}(s_t)$;
        $s_{t+1} \sim P(s_t, \boldsymbol{a_t})$;
        $\boldsymbol{r_t} \sim \boldsymbol{R}(s_t, \boldsymbol{a_t})$;
        $t \leftarrow t + 1$;
    until Episode termination;
    $\boldsymbol{\pi}_{e+1}^o = G(\boldsymbol{\pi}_e^o, \pi)$;
    $e \leftarrow e + 1$;
end
```

Algorithm 10: dec-POMDP Loop with Self-Play

```
Input: Environment: $(S, A, O, \mathcal{P}(\cdot,\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \rho_0)$
Input: Initial policy: $\pi(\cdot)$
$e \leftarrow 0$;
$\pi_e \leftarrow \pi$;
$\boldsymbol{\pi}_0^o = \{\pi_e\}$;
for $e = 0, 1, 2, \ldots$ do
    $\boldsymbol{\pi}'_t \sim \Omega(\boldsymbol{\pi}_e^o)$;
    $\boldsymbol{\pi}_e = \boldsymbol{\pi}'_t \cup \{\pi_e\}$;
    $t \leftarrow 0$;
    $s_t, \boldsymbol{o_0} \sim \rho_0$;
    repeat
        $\boldsymbol{a_t} \sim \boldsymbol{\pi_e}(\boldsymbol{o_t})$;
        $s_{t+1}, \boldsymbol{o_{t+1}} \sim P(s_t, \boldsymbol{a_t})$;
        $\boldsymbol{r_t} \sim \boldsymbol{R}(s_t, \boldsymbol{a_t})$;
        $t \leftarrow t + 1$;
    until Episode termination;
    $\pi_{e+1} \leftarrow update(\pi_e)$;
    $\boldsymbol{\pi}_{e+1}^o \sim G(\boldsymbol{\pi}_e^o, \pi_e)$;
    $e \leftarrow e + 1$;
end
return $\pi_e$;
```

Algorithm 11: MMDP Loop with Self-Play

```
Input: Environment: $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot|\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \gamma, \rho_0)$
Input: Initial policy: $\pi(\cdot)$
$e \leftarrow 0$;
$\pi_0^o = \{\pi\}$
for $e = 0, 1, 2, \ldots$ do
    $\boldsymbol{\pi'} \sim \Omega(\pi_e^o)$;
    $\boldsymbol{\pi_e} = \boldsymbol{\pi'} \cup \{\pi\}$;
    $s_0 \sim \rho_0$;
    $t \leftarrow 0$;
    repeat
        $\boldsymbol{a_t} \sim \boldsymbol{\pi_e}(s_t)$;
        $s_{t+1} \sim P(s_t, \boldsymbol{a_t})$;
        $r_t \sim R(s_t, \boldsymbol{a_t})$;
        $t \leftarrow t + 1$;
    until Episode termination;
    $\pi_{e+1}^o = G(\pi_e^o, \pi)$;
    $e \leftarrow e + 1$;
end
```

3.4 Self-Play Algorithms

We demonstrate the generalizing capabilities of our framework by presenting four prevalent SP schemes from MARL literature. Let π be a policy being trained, and $\boldsymbol{\pi^o}$ a menagerie:

3.4.1 Naive Self-Play

Originating in [Samuel, 1959], this is the oldest and simplest SP algorithm, which is also what most researchers refer to when using the term self-play. The premise is that every agent in the environment is populated with the latest version of the policy being trained. All agents share the same behaviour. To capture this, the policy sampling distribution Ω puts all probability weight to the latest π.

$$\Omega(\boldsymbol{\pi'}|\boldsymbol{\pi^o}, \pi) = \begin{cases} 1 & \forall \pi' \in \boldsymbol{\pi'} : \ \pi' == \pi \\ 0 & \text{otherwise} \end{cases}$$

In this extreme scenario the gating function G always deterministically inserts the latest version of the training policy into the menagerie, discarding the previous menagerie entirely.

$$G(\boldsymbol{\pi^o}, \pi) = \{\pi\}$$

3.4.2 δ-Uniform Self-Play

Introduced by [Bansal et al., 2017] and mentioned in Section 3.2. This SP scheme treats the menagerie as a time-ordered set of "historical" policies. The authors wanted to create an SP scheme that ensured continual learning by training a policy which could consistently beat random older versions of itself.

Let $M = |\boldsymbol{\pi^o}|$ be the size of the menagerie, and let $\delta \in [0, 1]$ denote the percentage threshold of the oldest policy to be considered as a potential candidate to be sampled from the menagerie $\boldsymbol{\pi^o}$ by Ω. As an example, if $M = 100$, with $\delta = 0$ all 100 policies in the menagerie would be considered as candidates, with $\delta = 0.7$ only the most recent 30 policies would be considered as candidates and with $\delta = 1$ only the last policy introduced in the menagerie could be sampled by Ω. After computing the set of candidate policies following this criteria, the authors use a uniform distribution to sample from it. Note that the authors originally use a shorthand notation, in which the policy sampling distribution samples the index of the policy in the menagerie (understanding the menagerie as a time-ordered list of policies) rather the policies themselves.

$$\Omega(\boldsymbol{\pi'}|\boldsymbol{\pi^o}, \pi) = Uniform(\delta M, M)$$

The gating function G used in δ-uniform SP is fully inclusive and deterministic. After every episode, it always inserts the training policy into the menagerie.

$$G(\boldsymbol{\pi^o}, \pi) = \boldsymbol{\pi^o} \cup \{\pi\}$$

In practical implementations, the agent can also be added only when the policy has been updated, instead of at the end of an episode. Most modern RL algorithms, especially on-policy ones, may require thousands of new sampled transitions before performing a single update. As a notable example, for a particular 3D multiagent game the authors of [Baker et al., 2019] found that *batch sizes*[6] of less than 16.000 datapoints yielded their learning algorithms unable to converge to useful policies, needing multiple episodes trajectories to perform policy evaluation and improvement. Hence, adding a policy at the end of every episode if it already exists within the menagerie results in a waste of compute and space.

3.4.3 Population Based Training Self-Play

As introduced in [Jaderberg et al., 2017, Jaderberg et al., 2019], Population Based Training SP is a parallel SP algorithm influenced by evolutionary algorithms. Each agent is independently learning on their own SP augmented MARL loop. The menagerie, initialized with a population of random policies, is shared amongst all learning agents. The menagerie is treated as the population of an evolutionary algorithm. The policy sampling distribution chooses opponents from the menagerie which are similar in skill to the currently training agent. Agent skill is measured by Elo ratings, but this metric could be replaced by any other principled metric of agent skill. The gating function is analogous to the selection, crossover and mutation phases of an evolutionary algorithm. It modifies and changes the menagerie by dropping low performing agents and introducing evolved versions of the existing population.

[6] This is a large batch size, given the computing capacity of most researchers and practitioners. This indicates that there is a large room for improvement within the field.

Population based training is a good example of how to extend SP to multiple simultaneous learning agents.

3.4.4 Policy-Spaced Response Oracles (PSRO) and SP-PSRO

A family of algorithms introduced in [Lanctot et al., 2017]. Such algorithms maintain a tensor which denotes the expected scores that different policy combinations would obtain when facing each other. We focus on the 2-player case with environments where the scores indicate which agent wins or losses in an episode. In such cases, PSRO maintains an empirical winrate matrix $\boldsymbol{W}_{\boldsymbol{\pi}^o}$ generated from a menagerie $\boldsymbol{\pi}^o$, and are parameterized via the choice of two functions:

- $\mathcal{M}(\boldsymbol{W}_{\boldsymbol{\pi}^o} \in \mathbb{R}^{|\boldsymbol{\pi}^o| \times |\boldsymbol{\pi}^o|}) \in \Delta(\boldsymbol{\pi}^o)$. The meta-game solver, which takes a meta-game and outputs a "meta-game solution"", a distribution over the policies of the menagerie.
- $\mathcal{O}(\pi \in \Pi, \boldsymbol{\pi}' \in \Delta(\boldsymbol{\pi}^o)) \in \Pi$. The oracle, which takes a starting policy π and a distribution over policies $\boldsymbol{\pi}'$ to derive a new policy π^* which performs better against $\boldsymbol{\pi}'$ than π. This role of the oracle is usually taken by an agent-level RL algorithm, as mentioned in Section 2.4 in the previous chapter.

The function of the meta-game solver $\mathcal{M}$ is to compute a distribution per player using game theoretic analysis over the metagame, which is captured by captured by our policy sampling distribution Ω, as they both output a probability distribution over a set of policies, the menagerie. After the oracle discovers a new policy, it is added to the meta-game, and the empirical winrate matrix $\boldsymbol{W}_{\boldsymbol{\pi}^o}$ is updated via game simulations. $\mathcal{M}$ operates on a meta-game generated by doing head-to-head matches between all policies in the menagerie, whereas a policy sampling distribution Ω operates directly on the menagerie, allowing for an extra degree of generality. For the later experiments featuring PSRO, we use $\mathcal{M}$ = maxent-Nash [Balduzzi et al., 2018].

$$\Omega(\boldsymbol{\pi}' | \boldsymbol{\pi}^o, \pi) = \mathcal{M}(\textit{meta-game}(\boldsymbol{\pi}'))$$

The functionality of the oracle can be anything that generates a new policy, such as an RL algorithm or evolutionary algorithm amongst other options. Upon completion of the oracle function, a new policy is added to the meta-game. To this extent, the oracle $\mathcal{O}$ partially encapsulates the functionality of our curator function G insofar as both functions decide when a policy is introduced in the menagerie. The curator has the advantage of being able to remove or modify the policies from the menagerie. To finalize the description of PSRO within our framework, we define the curator as a function which always accepts the policy discovered by the oracle function.

$$G(\boldsymbol{\pi}^o, \pi) = \boldsymbol{\pi}^o \cup \{\mathcal{O}(\pi, \Omega(\cdot | \boldsymbol{\pi}^o, \pi))\}$$

We point to an important difference between PSRO as originally introduced in [Lanctot et al., 2017] and its usage within this thesis. In the original PSRO presentation, every time that a new distribution of agent policies has been selected as opponents, the oracle $\mathcal{O}$ resets the learning agent's policy to a random policy, so that learning might begin from scratch for this new set of opponents. Retraining agents from scratch can be very computational intensive. We instead do not reset the learning agent's policy when a new set of opponents is chosen by $\mathcal{M}$ and instead continue using the latest set of parameters found during the optimization process. We denote this modification of PSRO as Self-Play PSRO or SP-PSRO.

3.4.5 δ-Limit Uniform Policy Sampling Distribution

We present a novel policy sampling distribution that attempts to alleviate the shortcomings of the δ-Uniform sampling distribution. A minimal incremental change to the existing method of δ-Uniform SP, showing how it is possible to make quantitative changes to existing SP algorithms within the context of a general framework.

In traditional supervised learning approaches, training datasets are fixed before training, rendering a stationary distribution over training samples. this is not the case in RL settings. On the abstraction level of agents, they collect data sequentially by acting in an environment, creating a high degree of correlation among collected data samples which needs to be accounted for with tools like experience replays. On the abstraction level of training schemes, we also observe that SP algorithms like δ-Uniform sequentially include policies into their menageries throughout training, on which both the curator G and the opponent policy sampling distribution Ω operate. Keeping this in mind, we analyze a property of the δ-Uniform SP algorithm. As stated earlier, it aims to generate an agent which can defeat *random* past versions of itself. However, this is affected by the sequential data collection curse of RL methods. By sampling uniformly at random from a menagerie, we observe a bias of the policies sampled from Ω towards earlier policies, as we will prove in Section 3.4.6. Intuitively, earlier policies are sampled more often by virtue of being electable to sampling more times than recently added policies. Computing a policy which generalizes against a broad set of policies is desirable. However, we worry that by sampling earlier policies too often the learning policy will be biased towards interacting with, often random, initial agents. This worry is furthered by empirical evidence stating that, in certain board games, the quality of the fixed policies being used during training is directly proportional to potential quality of the policy being trained [Van Der Ree and Wiering, 2013]. Over sampling earlier policies, which we can reasonably assume to be of weak quality, might slow down learning compared to other approaches.

We present a novel policy sampling distribution, named δ-Limit Uniform, that gives increased probability to later policies within the candidate window, without collapsing to naive SP, as an attempt to amend the δ-Uniform bias towards earlier policies. Figure 3.2 shows the histograms of the number of samples per policy for both $\delta = 0$-Uniform and $\delta = 0$-Limit Uniform, clearly showing how the δ-Limit Uniform distribution avoids biasing towards earlier policies.

Let $|\boldsymbol{\pi}_n^o|$ be the size of the menagerie at the beginning of the nth episode. π_e is the eth policy to have entered the menagerie (asserting $e \leq n$). The logit probability ρ_e^n and normalized probability p_e^n of sampling π_e for the nth SP episode are computed as:

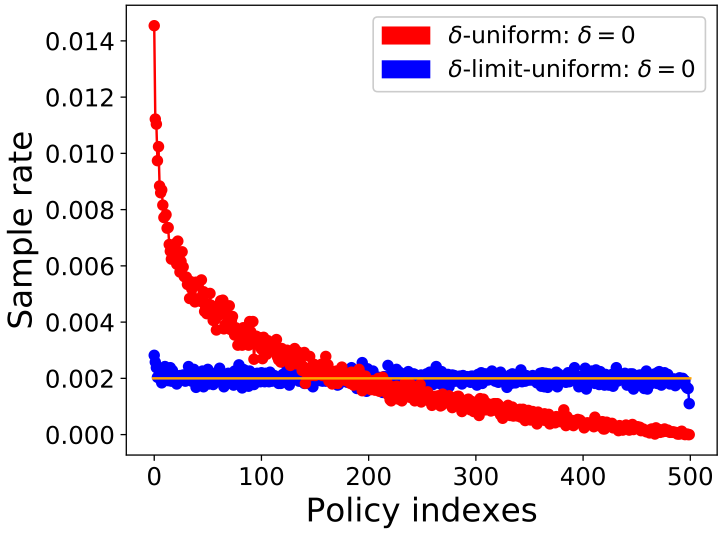


Figure 3.2: Histograms of sample rates (i.e percentage of times each policy index has been sampled) for policies indexes inside a menagerie for two sample training runs of 500 timesteps, one run for each policy sampling distribution. On each timestep (1) a policy index equal to the timestep is added to the menagerie and (2) the policy sampling distribution is called to sample an index from the menagerie. Note that earlier policy indexes eligible to be chosen more times than later ones. The horizontal line represents a $Uniform(0,500)$ distribution to show that asymptotically δ-Limit Uniform approximately places uniform probability over the policy indexes.

$$\rho_e^n = \frac{1}{|\boldsymbol{\pi_n^o}|(|\boldsymbol{\pi_n^o}| - e)^2} \qquad (3.2) \qquad\qquad p_e^n = \frac{\rho_e^n}{\sum_{i=0}^{|\pi_n^o|} \rho_i^n} \qquad (3.3)$$

We finish the section with a proof that δ-uniform SP biases the opponent selection to earlier policies.

3.4.6 Proof of δ-Uniform's Policy Sampling Distribution Biasing Towards Earlier Policies

This proof demonstrates that δ-Uniform's policy sampling distribution induces a higher sample rate on earlier policies compared to policies that have been added later during training, as stated in Section 3.4.5. This self-play scheme alternates between adding a single policy to the menagerie, and sampling exactly one policy from it on every iteration of the MARL loop specified in Section 3.3.1.

Let us assume that $\delta = 0$, meaning that the entire policy history is kept in the menagerie. Let $\boldsymbol{\pi^o}$ be a menagerie, where $\boldsymbol{\pi_e^o}$ represents the menagerie at episode e. Because a policy is added after every episode, the size of the menagerie at episode e is $|\boldsymbol{\pi_e^o}| = e$. Let policy π_i be a policy introduced in the menagerie at the end of episode $i \in [0 : e]$. Thus, a policy π_i is contained in a menagerie, $\pi_i \in \boldsymbol{\pi_e^o}$, if $e \geq i$.

The δ-uniform policy sampling distribution samples uniformly from a menagerie. Given n episodes, sampling the policy π_i at episode $e \in [0 : n]$ can be modelled as a Bernoulli trial with probability of success being $p_i^e = \frac{1}{|\pi_e^o|} = \frac{1}{e}$. At each episode e, sampling is done with replacement in an independent fashion. Moreover, given that the probability p_i^e changes at each episode, those Bernoulli trials are not identically distributed. Thus, we can define the mean sample count of policy π_i over n episodes as a Poisson binomial distributed variable, S_i. By definition, we have:

$$S_i = \sum_{k=i}^{n} \frac{1}{k} \tag{3.4}$$

In order to show that the δ-Uniform self-play scheme biases toward sampling earlier policies in a menagerie, we need to show that:

$$S_i > S_j \qquad \forall i, j \; i < j \wedge j \leq n$$

The derivation goes as follows:

$$\begin{aligned} S_i - S_j &= \sum_{k=i}^{n} \frac{1}{k} - \sum_{k=j}^{n} \frac{1}{k} && \text{(from equation (3.4))} \\ &= \sum_{k=i}^{j-1} \frac{1}{k} \\ \text{Hence,} \quad \sum_{k=i}^{j-1} \frac{1}{k} &> 0 && \text{(from } i < j\text{)} \\ \implies S_i &> S_j \end{aligned}$$

Thus, we have proven that δ-Uniform self-play biases towards sampling earlier policies throughout training. ■.

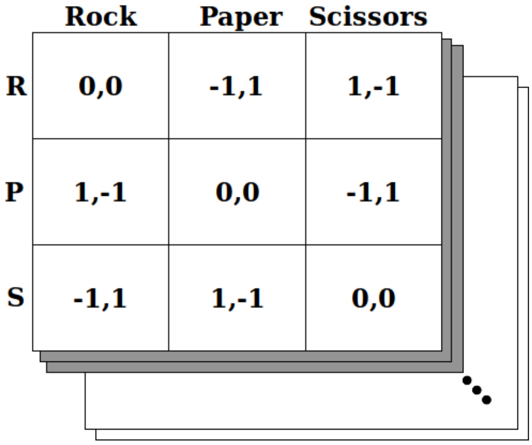
3.5 Qualitative Metric: Exploration of Policy Space

We have discussed the definition of different SP algorithms, but we still have a glaring question in front of us: do different SP algorithms meaningfully alter the learning dynamics of the underlying agents trained under them?

To begin constructing an answer to this question, we introduce a qualitative visual metric, aimed at understanding how much of the policy space does an agent explore during training. If we observe that different SP algorithms yield different exploration patterns, we will have qualitative confirmation of the effect of SP algorithms. We first introduce the environment in which agents will be trained to test this hypothesis.

3.5.1 Repeated Imperfect Recall Rock Paper Scissors (RirRPS)

We present Repeated imperfect recall Rock Paper Scissors (RirRPS), as introduced in [Hernandez et al., 2019], an environment especially amenable to the visualization we are presenting. RirRPS is a repeated, imperfect information, simultaneous version of Rock Paper Scissors (RPS), which can be modelled as a 2 agent, zero-sum POSG. The agent which obtains the highest cumulative reward by the end of the last repetition is considered the winner.



	Rock	Paper	Scissors
R	0,0	-1,1	1,-1
P	1,-1	0,0	-1,1
S	-1,1	1,-1	0,0

Figure 3.3: Visualization of RirRPS, a repeated version of RPS. The square matrix on top of the stack denotes the traditional payoffs for the RPS game. Gray squares indicate the repetitions which are part the observation. The subsequent white squares indicate previous repetitions outside of the shared observation.

We will always use 10 repetitions. The environment state is defined to be the last n joint actions, where n is the recall, which we always set to 3. This means that, as input to their policies, agents only observe the last 3 joint actions that were taken, partially obscuring the environment state. Ties are broken uniformly at random. Figure 3.3 shows a visualization of a possible episode within the RirRPS environment.

Due to RirRPS being based on RPS, it is a highly cyclic game, with a small amount of transitive skill coming from the *repeated* aspect of the game, which introduces exploitability that increases with the number of repetitions. Its partial observability entails that agents featuring a memory module which can help in recalling previous repetitions, unavailable to memoryless agents. We are interested in seeing whether agents featuring memory are affected differently than purely reactive agents under the same SP algorithms.

3.5.2 Identifying Policy Space Exploration via 2D Policy Space Projection

An indirect way of characterising a policy is by the environment trajectories it generates throughout simulation. Conversely, a set of trajectories could be marked as coming from a specific agent policy. Thus, studying the span of the state trajectories induced by an agent learning under an SP training scheme enables an assessment of both the span of the policies generated by an SP scheme, and the policies that can live inside their respective menageries.

This can be achieved in a human-readable manner by projecting individual trajectories into a 2D space which can be visualized. Different policies will yield trajectories in different parts of the embedded projected space. This projection must be structured insofar as to hold semantic value that can be used to discern policies apart. This is only possible providing we can label some subsets of the projected space with high-level understanding of what is happening throughout the state trajectories.

For example, take Figure 3.4a, which shows a 2D projection of RirRPS trajectories computed between a random policy against 3 other policies, one that plays only rock (RockAgent), one that plays only paper (PaperAgent), and one that plays only scissors (ScissorsAgent). Each point in the figure corresponds to a single trajectory. The aggregation of all trajectory projections creates 3 different clusters that we can easily identify as trajectories for which there are pure strategies being followed by at least one of the playing agents.

We now discuss the input to the dimensionality reduction algorithm. A trajectory τ can be defined as a sequence of state-action-rewards: $\tau = [(s_o, a_o, r_0), (s_1, a_1, r_1), \ldots]$. For the simple game of RirRPS, we drop the actions and rewards from the representation, as the state already contains information about which actions have been taken and because having the reward did not alter the final projections. Thus each trajectory is just a sequence of states: $\tau = [s_0, s_1, \ldots]$. Assume that states have a constant size denoted by $|s| \in \mathbb{R}$, and that all trajectories have the same number of timesteps, namely $T \in \mathbb{R}$. Then a trajectory can be represented as a concatenation of state vectors, itself a vector of size $(|s|T)$. Similarly, a series of n trajectories can be considered as a matrix of stacked trajectories of dimensions $(|s|T) \times n$. We use t-SNE [Maaten and Hinton, 2008] to project the multi-dimensional, environment specific representation of trajectories unto a 2D space. The algorithm t-SNE requires a matrix, as previously specified, and a class label for each row in the matrix, which we touch upon later. We note that other dimensionality reduction algorithms can be used [McInnes et al., 2020]. Once a 2D projection has been computed, we propose two visual cues to help discern the effects of SP training schemes on an agent's learning dynamics:

- **Density Heightmap**: Visualization of the density function yielded by the embedded state trajectories, computed via a kernel density estimation (KDE) method. Intuitively, it gives insight towards understanding where, inside the embedded state trajectory space, the agent has spent most time on during training.

- **Time Window-Averaged SP induced trajectories**: Visualization of the temporal evolution of the average embedded trajectory for an agent during training. Computed by uniformly dividing the time-sorted embedded trajectories in buckets, with the window-averaged trajectory being the median trajectory of each bucket, computed in the 2D embedding space. We link the median trajectory of each bucket with a color gradiant, to show what is the evolution of the median trajectory during training Intuitively, it displays which parts of the embedded trajectory space the agent has traversed throughout training. This cue can be used to visually assess to what extent an agent is prone to revisit some areas of the trajectory space, which can help identify catastrophic forgetting and cyclic policy evolutions.

The output of dimensionality reduction algorithms such as t-SNE vary depending on the data used as input. For our purposes it means that if we were to separately embed two sets of different state trajectories, we might not be able to meaningfully compare both separate embeddings. We tackle this problem with two measures:

(1) We compute a basis of possible state trajectories using some environment-specific heuristics that enables the basis to span over most of the whole state trajectory space. As

a practical note, the number of basis state trajectories computed is of the same order as the number of state trajectories generated during training.

(2) When comparing two or more sets of state trajectories generated by different algorithms, we compute the embeddings of each algorithm-induced state trajectories all at once via an *aggregated* set of state trajectories. Thus, it allows for meaningful comparisons across state trajectory embeddings from different algorithms.

Finally, this metric does not come without issues, computing a basis of trajectories is a non-trivial, environment specific task. Furthermore, noise coming from either environment dynamics or stochastic policies. Thus, this metric is most suitable for simple and deterministic environments.

3.5.3 Computing the Qualitative Visualization Metric for RirRPS

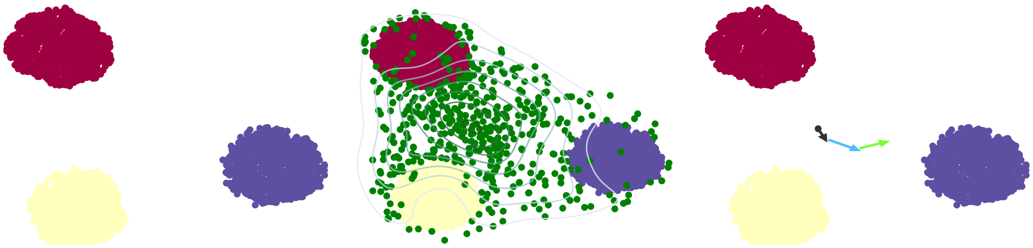

(a) (b) (c)

Figure 3.4: Plot of all basis trajectories (a). 500 embedded trajectories from training with their respective kernel density estimation (b). Evolution of median trajectories, each point computed from 500 trajectories (c).

We present the step by step procedure for generating the aforementioned visualizations for the RirRPS environment. We follow the following procedure:

1. **Compute basis trajectories**: We wanted the basis to showcase three equidistant clusters, each one representing trajectories that heavily favour one of the pure actions (rock, paper or scissors). We use 3000 thousand trajectories as basis trajectories: 1000 from RockAgent vs RandomAgent, 1000 from PaperAgent vs RandomAgent and 1000 from ScissorsAgent vs RandomAgent. Each combination has its own class label, required to guide t-SNE's clustering. Because the observation space of RirRPS is a sequence of joint actions it is possible to represent a RirRPS trajectory as the concatenation of all joint executed actions without any information loss. We denote the matrix of all basis trajectories as $\boldsymbol{B}$.

2. **Compute training trajectories**: Save all training trajectories for a given agent being trained under a SP training scheme. These trajectories receive a separate label from the basis trajectories. We denote the matrix of all training trajectories as $\boldsymbol{T}$.

3. **Use t-SNE to embed all trajectories to 2D space**: Use scikit-learn's [Pedregosa et al., 2011] t-SNE implementation with a PCA[7] initialization, as it is common practice to generate more legible t-SNE projections. The input matrix that is passed to t-SNE is the concatenation of both basis and training trajectories: $[\boldsymbol{B}, \boldsymbol{T}]^T$. This reduces every trajectory to a 2D coordinate.

4. **Plot basis**: Plot all embedded basis coordinates corresponding to $\boldsymbol{B}$ as a scatter plot. See Figure 3.4a.

5. **Plot training trajectories**: Plot some or all embedded training trajectories as a scatter plot. Use Seaborn's [Waskom, 2021] kernel density estimator to clearly see denser areas denoting the more visited parts of the trajectory space. See Figure 3.4b.

6. **Plot evolution of trajectories over time**: Divide the set of all training trajectories in buckets of equal size. Find their centroids and link them with arrows to show the evolution of mean trajectories as the agent trains. See Figure 3.4c, where 2000 episodes where broken in 4 groups of 500 episodes each.

To compare multiple SPs, one only needs to repeat step (2) for each SP training scheme, and concatenate all training trajectory matrices together with the matrix of basis trajectories on step (3).

3.6 Experimental Details

Equipped with the aforementioned qualitative visualization tool and with both inter-population and cross-population performance metrics presented in the previous chapter in Section 2.5.3, we carry out qualitative and quantitative experiments to find out whether or not SP schemes affect the learning dynamics of the underlying learning agents. We now present the evaluation metrics, environments, SP choices, algorithm used by the learning agents and agent architectures used in these experiments.

3.6.1 Environments

We want to explore how different SP schemes react to both cyclic and transitive environments, and so we choose two environments which, albeit not fully cyclic or fully transitive, represent two different near edge cases of 2-player symmetrical zero-sum games, as detailed in [Balduzzi et al., 2019]:

- **RirRPS**: As previously presented, we use it both due to it being amenable for our qualitative metric and for it's highly cyclic policy space.

- **Connect4**: A sequential game with a high degree of transitivity as empirically demonstrated in [Czarnecki et al., 2020]. Connect4 is not a symmetrical game, when training

[7] Principal Component Analysis (PCA) is found in any linear algebra university degree, however we present a link to its explanation for completeness: `https://en.wikipedia.org/wiki/Principal_component_analysis`

or benchmarking agents we randomly assign agent positions to enforce this symmetry. We developed and open sourced an OpenAI Gym compliant implementation of the game [8].

3.6.2 Quantitative Evaluation Metrics

Winrate matrices: SP algorithms train / modify a policy π overtime. We can consider an SP scheme sp as an stochastic generative process, which we can query at any time t to obtain the latest version of π being trained by sp, $\pi_t \sim sp$. This is analogous to creating checkpoints during training at which to freeze a copy of the policy π being trained. We only freeze π_t and not the menagerie $\boldsymbol{\pi_t^o}$. Thus, we can generate a population which represents the evolution of the policy training under an SP algorithm overtime, $\boldsymbol{\pi_{sp}} = [\pi_{t_0}, \pi_{t_1}, \dots]$. By examining the evaluation matrix generated from this population, $\boldsymbol{W_{\pi_{sp}}}$, we can quantitatively examine if different SP algorithms (1) suffer from catastrophic forgetting by presenting cyclic policy evolutions within the winrates in $\boldsymbol{W_{\pi_{sp}}}$ and (2) the speed at which the learning agent is improving upon previous versions.

Evolution of relative population performances: We use the relative population performance as a direct meassure of the relative quality between the populations spawned by two different SP algorithms, to measure which SP algorithm yields the strongest set of agents within a fixed number of training episodes. We are also interested in how this relative performance evolves overtime. Below we describe the algorithm to obtain such evolution: Given a set of SP training algorithms SP:

1. For each $sp \in SP$ sample a population $\boldsymbol{\pi_{sp}}$ of n agents. Each agent is sampled after a fixed number of episodes, with all sp receiving the same computational budget.
2. For each population pair $(\boldsymbol{\pi_{sp_1}}, \boldsymbol{\pi_{sp_2}})$, $sp_1, sp_2 \in SP$, compute an evaluation matrix $\boldsymbol{A_{\pi_{sp_1}, \pi_{sp_2}}}$ between both populations.
3. Compute $\boldsymbol{A_{sub}} = \{\boldsymbol{A_{1 \dots i \times 1 \dots i}} : i \in \{n\}\}$, which represents all submatrices of $A_{\pi_{sp_1}, \pi_{sp_2}}$.
4. Compute the evolution of relative population performance associated with each submatrix $\forall i \in [0, n]$, $\boldsymbol{A_i} \in \boldsymbol{A_{sub}}$, $\boldsymbol{v_{sp_1, sp_2}} = [v_{\boldsymbol{A_i}}] \in \mathbb{R}^n$. The vector $\boldsymbol{v_{sp_1, sp_2}}$ denotes the evolution of relative population performance overtime between two growing populations of agents.

Evaluation matrices are expensive to compute: $O(n^2)$ where n is the population size, making this experiment very computationally costly. There is current research on reducing the computational load of generating evaluation matrices [Rowland et al., 2020]. Fortunately, the procedure outlined above uses a single evaluation matrix to compute the relative population performances for all submatrices, meaning that we can recycle the empirical winrate

[8] https://github.com/Danielhp95/gym-connect4

matrix used to generate the evaluation matrices. We compute the winrate for an entry $w_{i,j}$ in an empirical winrate matrix $\boldsymbol{W}$ we use 50 simulations.

3.6.3 Self-Play Choices

We present the SP training schemes that we will take into consideration. We extend the MARL training loop as described by the following schemes introduced earlier in Section 3.4:

- Naive SP.
- δ-Uniform and δ-Limit Uniform, where the value of δ is specified each time.
- SP-PSRO($\mathcal{M}$ = maxent-Nash, $\mathcal{O}$ = Approximate Best Response). The *best response* oracle is governed by two hyperparameters, which play a role in determining whether the training agent has converged to an approximate best response: (1) The winrate $w \in [0, 1]$ at which it is considered that the current agent has converged and (2) the number of episodes $n_{matches}$ that will be used to compute the aforementioned winrate. For all experiments, we used $w = 72\%$, $n_{matches} = 50$.

For all SP training schemes, the initial menagerie contains a copy of the initial policy, with randomly initialized weights.

3.6.4 Algorithm for Learning Agents

For our qualitative studies we used Proximal Policy Optimization (PPO) [Schulman et al., 2017]. PPO is a famous actor-critic, on-policy algorithm which features widespread usage within the field. One of the main reasons behind using PPO is that during policy improvement, it heavily discourages policy updates which would bring big changes to the policy. This enables small, stable and gradual policy changes, which is a desirable property as we aim to study progressive changes over time.

PPO's hyperparameters used for the experiments are shown in Table 3.1, with minor modifications from the original publication [Schulman et al., 2017]. The quantitative study features smaller values for some hyperparameters to produce smaller changes to the policy on every update, providing a smoother continuum of generated policies. For Connect4 a larger horizon and minibatch sizes were chosen to compute less noisy gradient updates, compared to hyperparameters from the quantitative RirRPS experiments, as Connect4 is a more complex environment. We refer the reader to the original publication for a thorough explanation of each hyperparameter [Schulman et al., 2017].

We emphasize the fact that the agent learns uninterruptedly and that the training scheme does not modify the agent's model parameters (for instance by way of parameter resetting as in the original PSRO [Lanctot et al., 2017]). The agent updates its policy only via PPO's update rule, which takes place after a number of timesteps have been elapsed and a sufficiently large dataset of recent transitions has been obtained. The number of timesteps after which a policy update in PPO will take place (line 12 in Algorithm 8) is governed by the *horizon* hyperparameter. Note that due to the separation between modules in the MARL stack, the agent using PPO as a learning algorithm is unaware of the workings of the SP training scheme choosing the opponents that the agent is facing. In turn this means that the dataset of collected experiences that the agent will use to update its policy might contain a mixture of different opponents chosen by different calls to the opponent sampling distribution.

Hyperparameter	(Qual) RirRPS	(Quant) RirRPS	(Quant) Connect4
Horizon (T)	2048	128	512
Adam stepsize	3×10^{-4}	10^{-5}	10^{-5}
Num. epochs	10	10	10
Minibatch size	64	16	32
Discount (γ)	0.99	0.99	0.99
GAE parameter (λ)	0.95	0.95	0.95
Entropy coeff.	0.01	0.01	0.01
Clipping parameter (ϵ)	0.2	0.2	0.2

Table 3.1: PPO hyperparameters used for all experiments.

3.6.5 Agent Architectures

The actor-critic architecture of PPO agents requires a 2-headed neural network, with one head representing the policy distribution π, and the other denoting the corresponding critic V^{π}. The actor and the critic share all weights except for the final fully connected head layers. For the qualitative study, we experiment with both having a feed forward neural network (MLP-PPO) and a recurrent neural network (RNN-PPO). For our quantitative studies we only use MLP-PPO.

1. **RirRPS**: The MLP architecture is composed of 2 fully connected layers of 30 and 64 neurons. The output of this last layer is routed into two heads, a policy head of 3 neurons (one for each action in RirRPS) and a value head of a 1 neuron (estimating a state value function) [Schulman et al., 2017]. The RNN architecture augments the MLP architecture by adding at the beginning of the network an LSTM cell of 64 neurons.

2. **Connect4**: The input are 3 channels of dimensions 7×6, which we feed to 5 convolutional layers with constant stride and padding of 1, 3×3 kernels and channels $[3, 10, 20, 20, 20]$. For every consecutive pair of layers, we add a residual connection from the first to the last. The output of the convolutional layers is fed into the same MLP architecture used in the RirRPS experiment, with the difference that the policy head features 7 neurons, representing putting a chip down each of the columns in the game board.

We now present the results of two experiments. The first qualitatively inspects in Section 3.7 how different SP schemes induce different explorations of the policy space in the game of RirRPS. The second experiment, divided in Sections 3.8 and 3.9, quantitatively inspects both the intra-population policy evolution of agents under different SP schemes, and

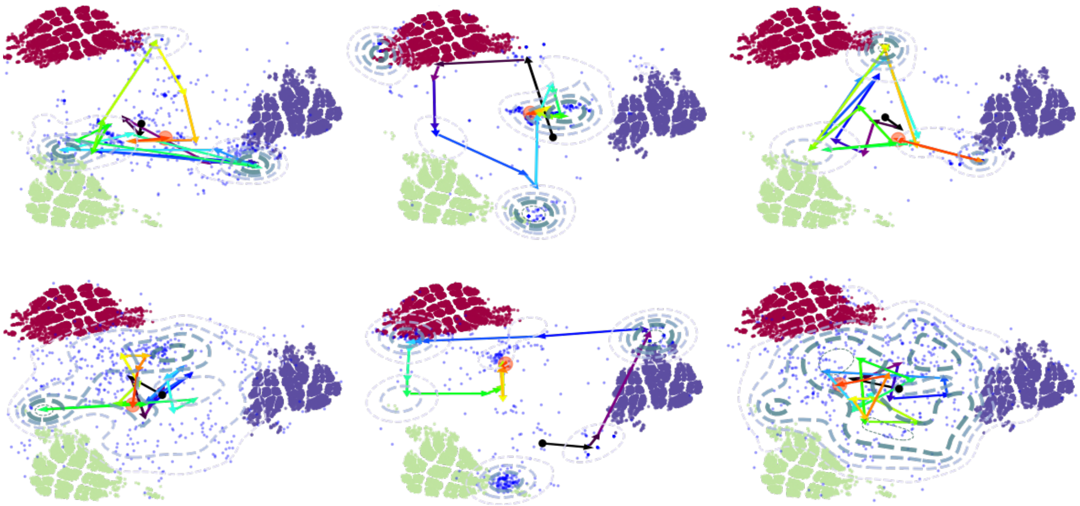


Figure 3.5: Density Heightmap and Time Window-averaged SP-induced of episode trajectories in a 2D t-SNE trajectory embedding space. The three clusters denote the basis trajectories. **Red:** scissors, **Green:** rock, **Purple:** paper. Left column shows training trajectories from Naive SP, centre for δ=0-Uniform and right for δ=0-Limit Uniform. Top row shows agents with feedforward architectures, bottom row shows agents with recurrent architectures. **Top-Left:** Naive SP with MLP-PPO. **Bottom-Left:** Naive SP with RNN-PPO. **Top-Centre:** δ = 0-Uniform SP with MLP-PPO. **Bottom-Centre:** δ = 0-Uniform SP with RNN-PPO. **Top-Right:** δ = 0-Limit Uniform SP with MLP-PPO. **Bottom-Right:** δ = 0-Limit Uniform SP with RNN-PPO. RirRPS environment, $1e4$ SP training episodes. The scattered blue dots represent the individual projection of each one of the $1e4$ trajectories. Their density heightmaps are represented through dashed contours. The time-sorted training trajectories experienced by the SP agents were divided into 20 time-windows, and a centroid (median trajectory) was computed for each. Consecutive centroids have been linked by arrows, creating the Time Window-averaged SP-induced episode trajectories. Starting at the black dot, their progression is highlighted via the rainbow colour transitions.

cross-population evolution of their relative population performances with respect to other SP schemes, in both RirRPS and Connect4.

3.7 Qualitative Analysis

Figure 3.5 shows the 2D t-SNE state trajectory embeddings for Naive, δ = 0-Uniform and δ = 0-Limit Uniform SP schemes, for both agent architectures introduced in the previous section. Each training session lasted for a $1e4$ episodes on the RirRPS environment.

We begin by observing δ = 0-Uniform (Figure 3.5 middle column). δ = 0-Uniform's Time Window-averaged SP episode trajectories visit each fixed agent clusters one by one. After behaving like a RockAgent, as seen by the KDE that gathers around the rock-playing trajectories (bottom green cluster), the trained policy starts to behave like a PaperAgent (right purple cluster). This transition occurs because, after behaving like a RockAgent for a few episodes, RockAgent-like policies enter the menagerie and progressively start being sampled as opponents, eliciting the learning agent to generate a best response against them, yielding PaperAgent-like policies. Both MLP-PPO and RNN-PPO agents exhibit that cyclic and ordered exploration of the embedding space.

In contrast, agents trained using naive SP and $\delta = 0$-Limit Uniform exhibit erratic exploration of the embedded space, as depicted by the sharp turns in their time window-averaged trajectories. Regardless of their erratic exploration, the KDE on the projected trajectories indicates that during training the agents do visit all three basis clusters of the policy space.

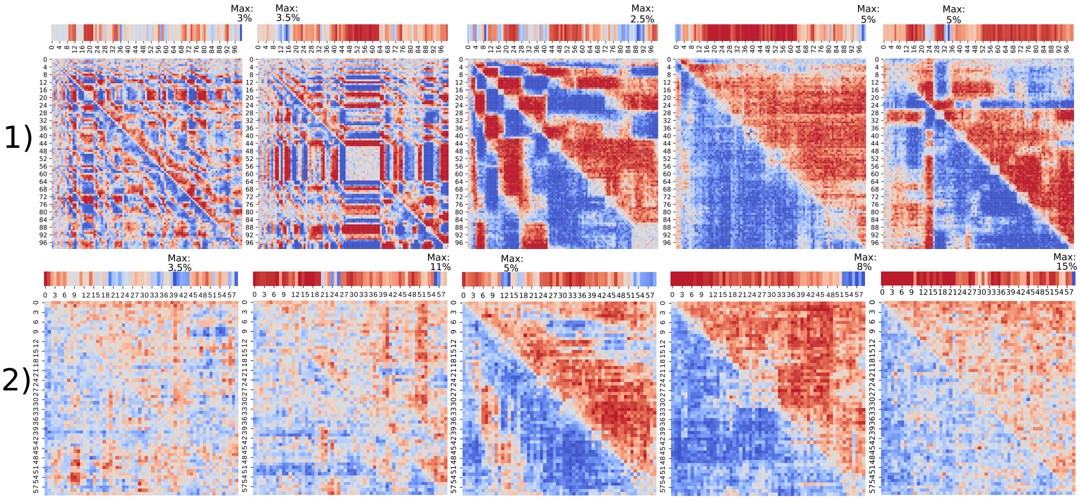
Let's compare both agent architectures, as Figure 3.5 divides its top row for MLP architectures and bottom row for RNN ones. Specially, we want compare $\delta = 0$-Limit Uniform and Naive SPs, as the architecture does not seem to affect $\delta = 0$-Uniform. The density of heightmap of RNN-PPO seem to be made of plateaus whereas the ones of MLP-PPO are made of sharp peaks, indicating that recurrent policies seem to further spread the menagerie over the whole policy space to some greater extent compared to feedforward policies. Whether one property is desirable over the other is not relevant, but rather the fact that the architecture of the underlying agents can have an effect on an agent's learning dynamics similar to that of the training scheme.

Albeit non-exhaustive, based on this qualitative projections, we have gathered evidence that the choice of SP training scheme does affect the way that a learning agent explores the joint policy space. This brings further weight to the argument that SP schemes deserve not only more detailed explanations in MARL work, but also a generalised framework under which to compare results.

3.8 Quantitative Analysis 1: Winrate Matrices

The results from Figure 3.6 are metrics gathered on a representative training run for the quantitative evaluation metrics described in Section 3.6.2. Note, different agents, on different training runs and different hyperparameters, were used to generate Figure 3.5 and Figure 3.6, thus we advise caution when comparing both figures and we ourselves refrain from doing so.

As a reminder, each row i of winrate matrix $\boldsymbol{W}$ represents the winrates of policy at checkpoint i against all other policies generated during training. Thus, for any given row i, the entries left of the diagonal ($w_{i,j}, \forall j < i$) indicate winrates against policies from earlier checkpoints in training, or older policies. Conversely, entries right of the diagonal ($w_{i,j}, \forall j > i$) denote winrates of policy i against later checkpoints, or newer policies. Diagonal entries represent the winrate of a policy against itself, which we trivially set at 50%. An ideal training scheme which would always compute monotonically better policies as training progressed would yield a winrate matrix where the lower triangular indices would show positive winrates (higher than 50%) and the upper triangular would show negative winrates (lower than 50%). In other words, a policy would always win against previous versions of itself, and lose against newer ones. Similarly, from an ideal winrate matrix, we would get a Nash support which places little to no probability mass on the earlier policies, and increases for later policies.



(A) Naive (B) δ = 0-Limit Uniform (C) δ = 0.5-Uniform (D) δ = 0-Uniform (E) PSRO

Figure 3.6: Empirical winrate matrices showing the evolution of 5 policies where each one is being trained via a different SP algorithm, as indicated at the bottom of each column. **Row 1)** RirRPS. **Row 2)** Connect4. For every SP training process, we sample a policy after every 1 policy update on RirRPS (for greater granularity) and 2 updates for Connect4 (for greater variety), totalling 100 checkpoints for RirRPS and 60 for Connect4. Treating each matrix as the payoff matrix for a symmetrical 2-player zero-sum game, we present on top of of each matrix the support received by each policy on the maxent-Nash equilibrium of such game. **Blue / red** indicates **higher / lower** support relative to the other policies. This support gives a measure of quality of each individual policy with respect to the other policies in the population. The agent with the largest support is marked alongside its support. For the empirical winrate matrices below each of the Nash support, **Blue / red** indicates **positive / negative** winrates for the column player.

3.8.1 Naive & $\delta = 0$-Limit Uniform

We turn our focus to the winrate matrices from Figure 3.6. As discussed, Naive SP uses as opponent an identical version of the policy being trained, and thus the underlying RL algorithm tries to compute a best response against itself. This is clearly manifested in the winrate matrix for RirRPS in Figure 3.6.1.A. The entries just left of the diagonal show positive winrates, and those just right of the diagonal show negative winrates. The latter meaning that the training policy learns how to beat the last version of itself. As expected, as we move further from the diagonal, most checkpoints obtained during naive SP training cycle between losing and winning against previous and future checkpoints. These are indications that as the training policy progresses through training, it sacrifices its knowledge on how to win against previous agents in favour of winning against the immediately next agents encountered as opponents. Due to the fact that previous agents quickly disappear from the menagerie, there is no learning pressure put on the agent to retain knowledge about beating them. This is further evidenced by the support under Nash from Figure 3.6.1.A, where under Nash equilibrium many policies share the highest amount of support (around 3%). We observed similar cyclic behaviour in $\delta = 0$-Limit Uniform in Figure 3.6.1.B and in $\delta = 0.5$-Limit Uniform (not shown). Which may entail that δ-Limit Uniform SP training schemes over-correct the bias towards earlier policies, matching our observations on the previous qualitative analysis which also showcased similar behaviours between naive SP and δ-Limit Uniform. Looking at Naive SP for Connect4, with its winrate matrix depicted in Figure 3.6.2.A, with most winrates laying around 50%, they show slow policy improvement during training. There *is* a small gradual internal improvement, as the lower triangular matrix features an average entry of 53%. This shows that there is some learning happening, which is also present across other training runs. $\delta = 0$-Limit Uniform yields very similar results, as seen in Figures 3.6.(1/2).B. Combined with the similarities with Naive SP found in our qualitative analysis, we believe that our proposed δ-Limit Uniform training schemes are indeed over-correcting the bias towards sampling earlier policies and putting too much emphasis on sampling later ones, leading to behaviours that resemble Naive SP. This over correction could perhaps be resolved by having different values of δ, which we leave for future work.

3.8.2 $\delta = 0.5$-Uniform

Figure 3.6.1.C shows the evolution of the policy training under $\delta = 0.5$-Uniform. This policy attempts to compute a best response against the later half of its history. Figure 3.6.1.C features more pronounced winrates (entries are either closer to 100% or 0% winrate) than the equivalent winrate matrix from $\delta = 0$-Uniform's Figure 3.6.1.D. This is a result of $\delta = 0.5$-Uniform's menagerie being smaller than $\delta = 0$-Uniform's counterpart. By having relatively fewer potential opponents, the learning policy is not forced to generalize as much as in $\delta = 0$-Uniform, leading to higher overfitting against the policies in the menagerie. This in turn seems to open the learning agent to exploitation by earlier policies. We see that on average, for a given row i, the corresponding policy tends to win against policies $j \in [\frac{i}{2}, i-1]$. Interestingly, any agent checkpoint i features negative winrates against the policies immediately outside of the menagerie's moving window, as determined by the choice of $\delta = 0.5$. This suggests that the training policy not only does not generalize to opponents outside of the menagerie, but that it also very quickly forgets how to beat them. This is again because, as these policies fall out of the menagerie, there is no pressure coming from the optimization process to maximize the learning agent's performance against them. This causes the underlying RL algorithm to update the policy parameters in a way that could be detrimental with respect to the performance of the training agent against policies outside of the menagerie. We were surprised to find that same effect is also present Connect4 as shown by the red band in the winrate matrix of Figure 3.6.2.C. Theoretically, in purely transitive games one does not need to maintain a large pool of varied policies to ensure skill improvement [Balduzzi et al., 2019]. We propose two explanations. First, that this effect showcases the cyclic components of Connect4 if Connect4 is understood as a functional form game [Balduzzi et al., 2019]. Secondly, the use of function approximators (the neural networks which represent the policies) are introducing non transitivities into the learning process [Chen et al., 2021]. Focusing on Figure 3.6.2.C, this exploitability by earlier policies is present in the Nash support, where policies with index [12-19] add up to 27% of the support under Nash.

3.8.3 $\delta = 0$-Uniform

Figures 3.6.(1/2).D, whose underlying policy attempts a best response against its entire history, show a close-to-ideal empirical winrate matrices insofar as any given policy beats most previous versions of itself and loses against later ones. Also, for any subgame of Figures 3.6.(1/2).D the largest concentration of support under Nash consistently lays on the latest policies. This is specially the case for the case of Connect4, where the last 7 policies accrue 49% of the support under Nash. $\delta = 0$-Uniform does not exhibit a cyclic policy evolution. Instead it tends towards generating monotonically better policies.

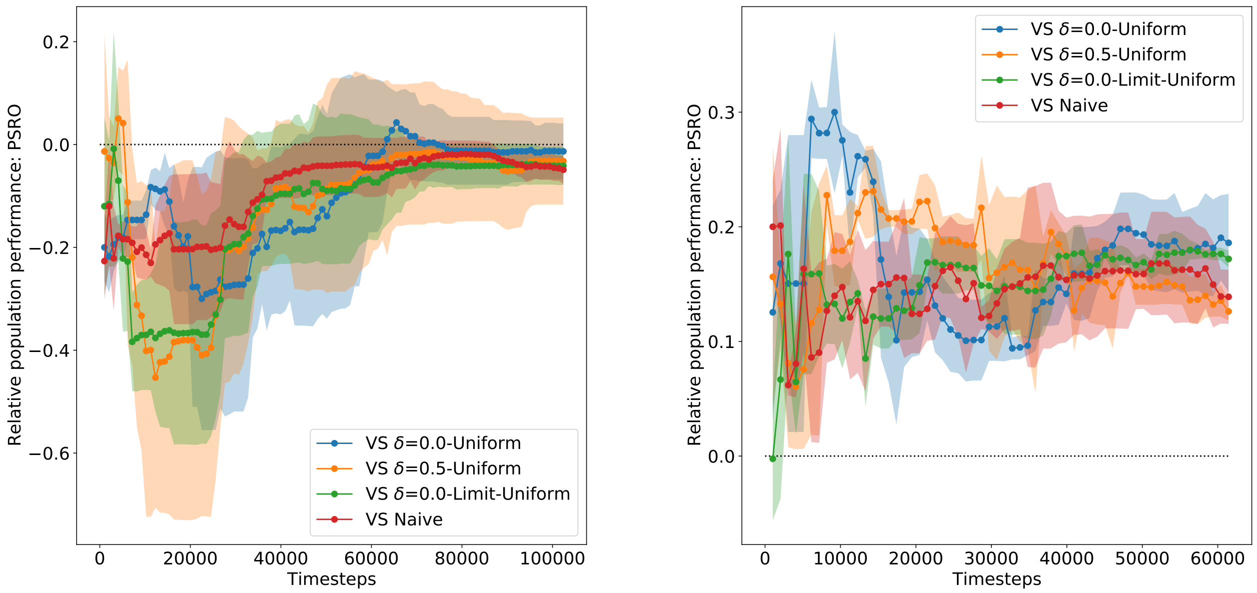
3.8.4 SP-PSRO

The winrate matrix for RirRPS, depicted in Figure 3.6.1.E, does follow a positive trend, although less so than $\delta = 0$-Uniform, as checkpoints beyond the 37th lose against policies 20 to 26, which worsens as later policies are introduced, which could be due to maxent-Nash not mixing enough exploration in its choice of opponents. Interestingly, the policy featuring the largest support under Nash is the 34th checkpoint. This does *not* necessarily mean that all 66 checkpoints that came after it were weaker in comparison. The quality of a policy (in terms of support under Nash) can vary greatly when policies are added or dropped from the population. For instance, if we consider a subgame of Figure 3.6.1.E taking only the first 92 checkpoints, we would find that the 92nd policy features the largest support under Nash around 3%, yet it falls around 1% on the final winrate matrix Figure 3.6.1.E. For the game of Connect4, in most subgames from Figure 3.6.2.E the policy with the largest support under Nash is always the last one, with the last one featuring 15%. Note how the average winrate on the lower triangular indices from Figure 3.6.2.E is not as high as Figure 3.6.2.D. This could mislead the reader into thinking that $\delta = 0$-Uniform generates higher quality policies, but such conclusions should be drawn from metrics that make comparisons not on the internal policy progression, but rather on comparisons across SP schemes, which is precisely what shall be discussed next.

3.9 Quantitative Analysis 2: Relative Population Performances

We now use the relative population performance metric to make a quantitative comparison across different SP algorithms. Figures 3.7a and 3.7b, computed from the agents trained for Figure 3.6, show the evolution of the relative population performance between SP-PSRO and the other 4 SP algorithms for RirRPS and Connect4 respectively. Figure 3.6 shows the aggregated metrics over 3 training runs SP scheme. As a reminder, in the context of winrate matrix meta-games, the relative population performance between two populations π_1 and π_2 is 0 if both populations are equal in strength, +1 if π_1 always beats π_2 and viceversa for -1.

In Figure 3.7a we notice that the relative population performance seems to converge near zero for all SP algorithms in the RirRPS environment. This implies that the populations generated by all SP training processes are of similar quality, furthering the idea that in highly cyclic games individual policy improvement is not meaningful [Balduzzi et al., 2019], even



(a) **RirRPS**. SP-PSRO features negative relative population performances against all other SP algorithms, meaning that it trained a weaker sequence of policies.

(b) **Connect4**. SP-PSRO outperforms all other SP schemes.

Figure 3.7: Evolution of relative population performances of SP-PSRO versus all other SP schemes. Positive values indicate that SP-PSRO performs better against an SP scheme, negative values indicate the opposite. This experiment was repeated 3 times to obtain an estimate of the standard variation (shaded regions).

when there is potential for exploitation due to repetitions[9] the in RirRPS. However, we are surprised to find Naive SP performing better than $\delta = 0$-Uniform and SP-PSRO, which is not obvious by just looking at the winrate matrices from Figure 3.6.1. A possible reason is that the cyclic behaviour of naive SP quickly discovers how to play Rock, Paper and Scissors, which are enough to generate a Nash equilibrium. Conversely, a SP algorithm that scarcely introduces a new policy into its menagerie, such as SP-PSRO, slows down the variety of trajectories experienced by the learning agent, taking longer to explore the policy space. This is also backed by the findings that some SP-PSRO configurations[10] contract (instead of expand) the space of policies covered by the menagerie [Balduzzi et al., 2019, Liu et al., 2021].

Figure 3.7b showcases how SP-PSRO is better suited for transitive environments, as it shows a positive relative population performance versus all other SP algorithms at all points during training. This difference in performance stabilizes on an average of 0.15 across all SP algorithms. We hypothesize that this difference would increase overtime, meaning that the rate of policy performance improvement induced by SP-PSRO is higher. This would require experiments with longer training times. However, with our computational budget of around

[9] By repetitions we mean the multiple rounds of RPS within the game of RirRPS.

[10] By configuration we mean the choice of meta-game solver $\mathcal{M}$ and oracle $\mathcal{O}$.

60K environment timesteps, we can say that $\delta = 0.5$-Uniform and Naive SP were the most promising training schemes behind SP-PSRO (lower relative performances against SP-PSRO means that they were closer to being as strong as SP-PSRO).

We want to emphasize that even though the winrate matrices present in Figure 3.6 perhaps show more promising results for $\delta = 0$-Uniform, we see in Figure 3.7 that this is not the case.

3.9.1 Fragility of SP-PSRO's Oracle Hyperparameters

Hyperparameters	$\mathcal{M}$	W_{π^o}	Training time (days, hours, minutes)	$\|\pi^o\|$
(60%, 30)	95%	2%	>2d	332
(70%, 30)	95%	5%	1d 12h 44m	294
(75%, 30)	70%	25%	1h 38m	139
(80%, 30)	59%	32%	55m	118
(85%, 30)	0%	0%	4m	1
(70%, 45)	69%	24%	58m	112
(75%, 45)	2%	9%	5m	19
(80%, 45)	0%	0%	4m	1
(70%, 50)	67%	26%	1h 7m	123

Table 3.2: Hyperparameter sweep time profiling for 12k episodes in RirRPS. Columns $\mathcal{M}$ and W_{π^o} represent the percentage of training time spent on computing a meta-game solution and updating the meta-game respectively.

We wanted to highlight one of the perils of MARL, which is hyperparameter selection, as these often play a large role in eliciting the best results for most algorithms [Henderson et al., 2018]. Unfortunately, choosing the right values is often a heuristic process which depends on intuition and is often environment specific. During our experiments, we found this to be the case especially for SP-PSRO.

Small changes in SP-PSRO's best response oracle hyperparameters (winrate threshold w, window size of match outcomes $n_{matches}$) can quickly lead to unfeasibly long training times (too many policies added to the menagerie) or arguably degenerate behaviour by the SP algorithm (the curator never introduces new policies into the menagerie). In the worst case scenario for highly cyclic environments such as RirRPS, the training policy will never converge towards a best response against the initial (randomly initialized) policy in the menagerie. We show in Table 3.2 a sweep over both hyperparameters for our choice of SP-PSRO in RirRPS. Most of the training time is spent inside of the meta-game solver $\mathcal{M}$, which leads us to believe that a less computationally intensive meta-game solver should be used. We used maxent-Nash in the hopes that it would help with debugging operations and post-training interpretability, but given its computational cost, we would now suggest the use of computationally cheaper alternatives such as using convex optimizers to find arbitrary Nash equilibria for 2-player zero sum games[11]. This would trade off the aforementioned properties in favour

[11] For an implementation of a Nash finder using convex optimization for 2 player zero sum games we refer to our open sourced implementation: `https://github.com/Danielhp95/Regym/blob/master/regym/game_theory/solve_zero_sum_game.py`

of much quicker training times.

As a final point on this, a Nash Equilibrium in RirRPS is to act randomly, which we argue is likely the behaviour of the training policy at the beginning of training. Hence, it is highly unlikely that a policy will obtain a high enough winrate against this random policy to be added to the menagerie, making it difficult for the learning policy to discover policies which differ from random play. This is especially the case if the required winrate is high and the moving window large. This means that the learning policy will not discover how to exploit policies beyond random play.

3.10 Conclusion

We now conclude and summarize the presentation of our general framework for defining SP training schemes. This is done by formalizing the notion of a menagerie, a policy sampling distribution and a curator (gating) function. This framework is posited as theoretical approximation to a solution concept in MARL, under stated assumptions. The framework's generalizing capabilities have been showcased by capturing existing SP algorithms within it. We have also identified shortcomings of some of the captured algorithms, and have proposed methods which could potentially overcome said issues, although the theorized benefits failed to materialize in practice. Through a qualitative study we have showcased that, on a simple environment, different SP algorithms differ in how the joint policy space is explored. We have also carried out a quantitative analysis on (1) the evolution of policies being trained under different SP algorithms to discover cyclic policy evolutions and (2) the relative performance between various SP algorithms, on both a highly cyclic and highly transitive environment, with empirical results strongly suggesting that training schemes have a big role in the structure of the learning dynamics of the underlying agents.

Future work will study other possibilities presented within the expressive capabilities of our SP framework. For instance, there is no research exploring which policy sampling distribution works best for different types of environments. Furthermore, it may even be possible to *learn* a policy sampling distribution or curator during training using meta RL.

Acknowledging the limitations of our experiments, it must be noted that both RirRPS and Connect4 are *2-player*, *zero-sum* and *simultaneous* games. Our experimental results may not extend to *n-players* or *general-sum* games. Moreover, following the dimensionality-based definition of the complexity of a game [Balduzzi et al., 2019], the lower-bound on the complexity of RPS and, incidentally, RirRPS are rather low. Therefore, it would be interesting to compare current results with games of verifiably greater lower-bound on their complexity. However, these experiments will put a heavy computational requisites on already computationally expensive experiments.

3.10.1 The Significance of Training Schemes for Game Studios

A lot of games studios already struggle to use MARL in their games [Jacob et al., 2020], and normally will spend their engineering efforts trying to replicate existing algorithms within their games. There current state of SP literature unfortunately features two main problems.

The first is the lack of literature emphasising the importance of SP and training schemes more generally, which can make MARL practitioners within game studios overlook their implementation. If ignorant of its implications, these practitioners might not discern the consequences between design decisions such as allowing all agents to update their policies agents in a MARL environment versus allowing a single agent to train while keeping all other agents stationary. The second problem is the high abstraction level required to understand that state-of-the art findings on training schemes such as [Balduzzi et al., 2019], which was heavily referenced in this chapter. Due to this, only studios with experts in the field might be able to lead their MARL implementations in the direction prescribed by the current state-of-the-art.

Our hopes are two fold. Firstly, that the work developed within this chapter clears the still nebulous concept of training schemes in a fashion approachable for non experts, specifically that of self-play training schemes. Secondly, that the provided empirical evidence and formal framework will be able to guide future research within the field of SP.

Chapter 4

BRExIt: Opponent Modelling in Expert Iteration

4.1 Introduction

The centralized MARL training schemes explored in the previous chapter can be considered as an outer loop which decides on elements (the opponents) which parameterize a target task for the inner loop to be optimized against by underlying RL agents. These training agents act in the inner loop collecting experiences to update their internal models to improve at the current task. Having explored different training schemes that make up the outer loop in the previous chapter, this chapter descends one level in the MARL stack to focus on the algorithms used by the RL agents within the inner optimization loop.

Following the trend of modern menagerie based MARL training schemes based on Double Oracle [McMahan et al., 2003] and the parallel Nash memory [Oliehoek et al., 2006] which inspired PSRO [Lanctot et al., 2017] and many of its derivatives [Balduzzi et al., 2019, Nieves et al., 2021, Liu et al., 2021], we treat agents as learning oracles. This idea was briefly introduced in Section 2.4 and similarly explored in our definition of the PSRO training scheme within our generalized self-play framework in the previous chapter. Learning oracles have also been called policy improvement operators or best response operators, depending on whether the lens under which these concepts are studied comes from RL or game theory. In our description of PSRO we defined an oracle as an improvement operator over opponent policies, as each opponent (or combination of opponents) can be considered as a different task. Using standard notation, an oracle for an environment E, $\pi' = \mathcal{O}(\pi, \pi_o)$ is a function which given an initial policy π and an opponent policy (or distribution over policies) sampled from a menagerie $\pi_o \in \boldsymbol{\pi_o}$, it returns a relatively higher performing policy π' against π_o than π. For our purposes, this improvement is done via MARL methods. One of the main properties of interest for any oracle is sample efficiency, measured in terms of either environment steps, environment episodes or wall-clock time. This is often the primary practical consideration for deciding between different algorithms to solve a task at hand, and it is the main metric that we will use within this chapter to compare algorithmic efficiency.

The large number of training episodes required to train modern RL algorithms as policy improvement operators makes the inner loop the computational bottleneck of modern MARL training schemes. In order to alleviate this computational burden we take a look at the findings from the last chapter, where we have seen that at the abstraction level of training schemes there are tangible benefits for being aware of which agents are present in an environment. This was because being able to decide which opponent to play against allows us to decide on the objective that learning agents will optimize against, which in turn allows us to shape the exploration of the policy space towards areas of higher skilled policies. Yet, this awareness was lacking at the agent level of the MARL stack in the previous chapter. The agents themselves neither conditioned their policies on other opponents policies, either real or predicted, nor did their learning functions discriminate between trajectories drawn from one opponent or another.

In this chapter we focus on one approach to grant opponent awareness at the agent level to explore efficient policy improvement operators. We study the introduction of opponent awareness in the state-of-the-art model-based Expert Iteration (ExIt) [Anthony et al., 2017], most known for its use in AlphaGo [Silver et al., 2016], as a policy improvement operator within a centralized training scheme's inner loop. Further exploiting centralized training assumptions that hold within these training approaches, we introduce Best Response Expert Iteration (BRExIt), an opponent aware enhancement of ExIt which uses opponent models at various stages of the algorithm. These models are used both as feature shaping mechanisms and to bias MCTS's search around the vicinity of a best response against the opponent policies, making it robust to local improvements by the policy it is responding against. For the case where given opponent policies are too computationally demanding for search, but which can be used to generate training games, we also test a variant of BRExIt that uses learned surrogate opponent models instead. In the game of Connect4, we find that BRExIt learns to achieve target winrates with statistically significantly fewer episodes compared to ExIt, alleviating the computational bottleneck in training towards a best response.

BRExIt stands at the confluence of two streams of literature. On the one hand, it draws from recent efforts of incorporating opponent models within DRL, shown in Section 4.2.1. On the other, it builds on the existing corpus of work which brings improvements to the sample efficiency of the ExIt framework, explored in Section 4.3. The algorithm is presented in depth in Section 4.4. An experiment to measure BRExIt's efficacy as a policy improvement operator is explained in Section 4.5 with results on Section 4.6. Section 4.7 closes the chapter's main contents, with several optional implementation details explained in Section 4.8.

4.2 Opponent Modelling as an Auxiliary Tasks in DRL

Opponent modelling focuses on reasoning about the behaviour of other agents by constructing models which make predictions of various properties of interest of the modelled agents [Albrecht and Stone, 2018]. We focus on a specific type of opponent modelling known as policy reconstruction, which is the task of learning the policy of other agents in the environment, typically from observed actions. That is, we want to approximate a policy $\hat{\pi} \approx \pi$ where the left hand side is the approximated opponent model and on the right is the ground truth policy we are trying to model.

A complementary motivation for learning opponent models is to cast this process as an auxiliary task. As introduced in Section 2.3.4, auxiliary tasks exploit the underlying structure of an environment to generate learning signals derived from environment observations or other agent policies, providing learning targets which go beyond predicting future reward signals. Despite their prevalence and reported empirical benefits in single-agent literature [Jaderberg et al., 2016, Pathak et al., 2017, Mirowski et al., 2016], multiagent specific auxiliary tasks have received relatively scarce attention.

Auxiliary tasks tend to be environment specific, as they make assumptions about the underlying structure of the game being played. Yet we argue that opponent modelling stands as a notable exception, as by definition every multiagent environment will feature other policies acting in them that can be modelled. Furthermore, the optimality notion of a best response that is core to the literature of policy improvement operators requires a set of target policies to be optimal *against*. This hints at the fact that an agent which models aspects of other agents can learn valuable information about its own future rewards. Hence, by using opponent modelling as an auxiliary task we can exploit an ever-present structure in the dynamics of multiagent environments. Interestingly, opponent modelling has been absent from recent successes in multiagent systems [Vinyals et al., 2019, Berner et al., 2019]. In the field of natural science, opponent modelling and social awareness has been observed in mammals [Preston and Jacobs, 2009].

4.2.1 Opponent Modelling in DRL

Policy reconstruction methods have been shown to be beneficial in collaborative [Carroll et al., 2019], competitive [Nashed and Zilberstein, 2022] and mixed settings [Hong et al., 2018]. One of the first algorithms that combined DRL with opponent modelling was Deep Reinforcement Opponent Modelling (DRON) [He et al., 2016]. The authors used two networks, one that learns Q-values using Deep Q-Network (DQN) [Mnih et al., 2013], and another that learns an opponent's policy by observing hand-crafted opponent features, such as the frequency of environment specific events. Their key innovation is to combine the output of both networks to compute a Q-function that is *conditioned* on (a latent encoding of) the approximated opponent's policy. This accounts for a given agent's Q-value dependency on the other agents' policies. Based on this, Deep Policy Inference Q-Network (DPIQN) [Hong et al., 2018] brings two further innovations: (1) merging both modules into a single neural network, and (2) baking the aforementioned Q function conditioning into the neural network architecture by reusing parameters from the opponent model module in the Q function. This approach was then extended to on-policy algorithms [Hernandez-Leal et al., 2019]. We incorporate this approach into the off-policy ExIt framework.

DPIQN combines two loss functions, one to approximate a Q-function and another to perform policy reconstruction to learn opponent models. The former is learnt by minimising the standard DQN loss function $\mathcal{L}_Q$ from [Mnih et al., 2013], regressing a state-action value critic Q against observed rewards. As an auxiliary task opponent models are learnt by fitting a neural net to observed agent actions, $\hat{\pi}_j \in \hat{\boldsymbol{\pi}}_{-i}$, with one policy reconstructed for each agent in $\boldsymbol{\pi}_{-i}$. A dataset of state-actions is collected $(s, \boldsymbol{a}_{-i})$, and the opponent models are trained by minimising the cross-entropy between one-hot encoded observed agent actions, $\boldsymbol{a}_{-i}$, and the predicted opponent policies at state s, $\hat{\pi}_j(\cdot|s)$. This is defined as the *policy inference* loss $\mathcal{L}_{PI}$. Equation 4.1a defines such loss for a set of N opponents. Thus, DPIQN's network is trained by combining both losses into Equation 4.1c with an *adaptive weight* λ which they claim significantly improves learning stability in Equation 4.1b.

$$\mathcal{L}_{PI} = -\frac{1}{N}\sum_{j=0}^{N} \boldsymbol{a}_j log(\hat{\pi}_j(\cdot|s)) \tag{4.1a}$$

$$\lambda = \frac{1}{\sqrt{\mathcal{L}_{PI}}} \tag{4.1b}$$

$$\mathcal{L}_{DPIQN} = \lambda\mathcal{L}_Q + \mathcal{L}_{PI} \tag{4.1c}$$

BRExIt makes use of both the policy inference loss for its opponent models and the adaptive learning weight to regularize its critic loss. However, instead of learning a Q-function as a critic, BRExIt learns a state-value function V as both in the original ExIt algorithm and as suggested by previous work [Hernandez-Leal et al., 2019].

4.3 Expert Iteration and Improvements

ExIt was introduced in [Anthony et al., 2017] in the context of training policies in multiagent environments. An algorithmic framework that combines planning and learning during training to generate a parameterized policy $\pi_{\boldsymbol{\theta}}$, where $\boldsymbol{\theta} \in \mathbb{R}^n$, which maximizes an observed reward signal. Even though an environment model is required during training to generate learning targets, the trained policy is represented as a neural network that performs no planning, and thus can act as a stand alone module during deployment. We note that there are variants of ExIt which try to forego this training constraint by learning an environment model [Schrittwieser et al., 2020, Ye et al., 2021].

ExIt's two main components are (1) the *expert*, traditionally an MCTS algorithm, and (2) the *apprentice*, a parameterized neural network policy $\pi_{\boldsymbol{\theta}}$. In short, the expert takes actions in an environment using MCTS, populating a replay buffer with good quality moves. The apprentice updates its parameters $\boldsymbol{\theta}$ to better predict both the expert's actions and future rewards. The expert in turn uses the apprentice inside MCTS to bias its search. Updates to the apprentice yield higher quality expert searches, yielding new and better targets for the apprentice to learn. This iterative improvement constitutes the main ExIt learning loop, captured in the top half of Figure 4.1.

We refer to our in-depth explanation of MCTS in an earlier chapter in Section 2.6. We briefly describe MCTS again in the context of ExIt. We use an open-loop MCTS implementation [Silver et al., 2017b], each node in the tree represents an environment state s, and each edge (s, a), represents taking action a at state s. Each edge stores a set of statistics:

$$\{N(s,a), Q(s,a), P(s,a), i, A_n\} \tag{4.2}$$

$N(s,a)$ is the number of visits to edge (s,a). $Q(s,a)$ is the mean action-value for (s,a), aggregated from all simulations that have traversed (s,a). $P(s,a)$ is the prior probability of following edge (s,a) which biases the initial search over the node. ExIt uses the apprentice policy to compute priors, $P(s,a) = \pi_\theta(a|s)$. Index i denotes the player to act in the node, for sequential games, or the index of the player who is calling the MCTS procedure for simultaneous games. Finally, A_n indicates the available actions.

For every state s encountered by an ExIt agent during an episode, the expert takes an action a yielded by running MCTS rooted at state s and selecting the action corresponding to its action selection phase, which in our case is choosing the action from the root node's most visited edge (s,a). During search, the tree is traversed using the same selection strategy as AlphaZero [Silver et al., 2017a]. Edges (s,a) are traversed following the most promising action according to the PUCT formula:

$$a = \arg\max_a \; Q(s,a) + C_{PUCT} \frac{P(s,a)\sqrt{\sum_{a'} N(s,a')}}{1 + N(s,a)} \tag{4.3}$$

Where C_{PUCT} is a tunable constant. The apprentice π_θ is a distillation of previous MCTS searches, and provides $P(s,a)$, thus biasing MCTS towards actions that were previously computed to be promising.

Upon reaching a leaf node with state s', the original ExIt [Anthony et al., 2017] used a random rollout policy for the rollout phase. The famous algorithm of AlphaGo [Silver et al., 2016] instead opted for training a policy on well-known Go patterns to act as the rollout policy. Due to the large variance in value estimations and the computational cost of such rollout policies, their use was eventually discarded in later iterations of the algorithm [Silver et al., 2017b, Silver et al., 2017a]. We follow the same footsteps. Instead, upon reaching a leaf node with state s', we backpropagate a value given by a learnt state value function $V_{\boldsymbol{\phi}}(s')$ with real valued parameters $\boldsymbol{\phi} \in \mathbb{R}^n$ trained to regress against observed episodic returns using a mean squared error loss (MSE). Note how this makes ExIt an actor-critic algorithm, with both the actor policy π_θ and critic value function $V_{\boldsymbol{\phi}}$ being represented within the apprentice's neural network, as shown in the top left corner of Figure 4.1.

After completing a search from a root node representing state s, a policy $\pi_{MCTS}(\cdot|s)$ can be extracted from the statistics stored on the root node's edges. This policy is stored as a training target for the apprentice's policy head to imitate:

$$\pi_{MCTS}(a|s) = \frac{N(s,a)}{\sum_{a'} N(s,a')} \tag{4.4}$$

As shown in the top right corner of Figure 4.1 At every timestep t, ExIt builds a dataset, which can be implemented via a replay-buffer, containing:

$$\{s_t, \pi_{MCTS}(\cdot|s_t), G_t\} \tag{4.5}$$

Where G_t is the return for such player i, the undiscounted sum of rewards until the end of the episode. A cross-entropy loss is used to bias the actor towards the expert's moves, and a mean-square error loss is used to update the critic's state value function towards observed returns G_t.

Significant amount of research has been aimed at improving ExIt. The authors of [Willemsen et al., 2020] explore what are the most valuable learning targets that can be extracted from the search tree constructed by MCTS. [Soemers et al., 2020] equips ExIt with prioritized experience replay [Schaul et al., 2015] and informed exploratory measures. On auxiliary task learning, Wu *et al* [Wu, 2019] introduce the Go specific auxiliary tasks of predicting (1) ownership of each board position and (2) final game score, leading to improvements in sample efficiency. The use of auxiliary tasks to learn opponent models to be used within search has not been studied, even though using opponent models within ExIt's search has already proven beneficial. We highlight the work of [Timbers et al., 2020] which assumes that a ground truth opponent model is given at search time in order to have the search compute an approximate best response the assumed opponent, in a very similar fashion to how BRExIt handles its search. A main difference between BRExIt and the work of [Timbers et al., 2020] is that BRExIt uses auxiliary tasks to learn an opponent model. This has two effects, firstly that learning an opponent model acts as a feature shaping mechanism for the policy network, which partially shares a parameterization with the learnt opponent model, and also that having such learnt model allows for ablations of the algorithm where a ground truth opponent model is not given. Although a perfect opponent model is intuitively preferable over a partially erroneous approximation Goodman *et al* [Goodman and Lucas, 2020] empirically show that even partially incorrect models can be useful, although very inaccurate models will be detrimental to the quality of the search. These models were used to mask the opponent's actions as part of the environment dynamics, increasing the depth of the search, which yielded actions specifically targeted to respond to the modelled policies. This makes such actions brittle responses in the face of changes by the modelled policies. In contrast, we propose using opponent models as priors in BRExIt, such that planning also improves upon the opponent policy, as this induces learning targets that are robust to local improvements of the opponent policy. Although this brittle response could still be set up within BRExIt if potentially more fragile but more exact best responses are desired.

4.4 BRExIt: Opponent Modelling in ExIt

We present Best Response Expert Iteration (BRExIt), an improvement over ExIt that uses opponent modelling for two purposes:

1. To allow the MCTS expert to approximate a best response against a known set of opponents robust to local improvements: $\pi_{MCTS} \in BR(\boldsymbol{\pi}_{-i})$. See Subsection 4.4.3.

2. To enhance the apprentice π_θ with learnt opponent models. These opponents models are not directly used outside of MCTS to select actions in an environment and serve only as feature shaping mechanisms to improve the quality of the actor-critic architecture. See subsection 4.4.1.

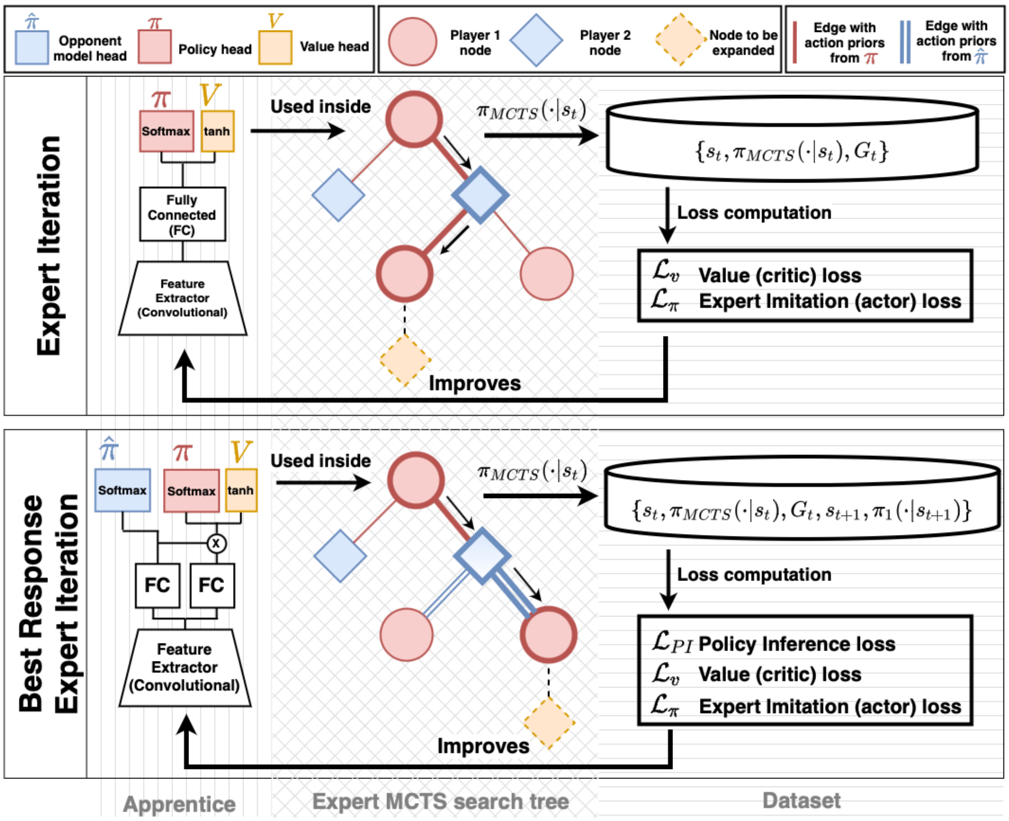

Opponent model head
Policy head
Value head
Player 1 node
Player 2 node
Node to be expanded
Edge with action priors from π
Edge with action priors from $\hat{\pi}$
Expert Iteration
Softmax
tanh
Fully Connected (FC)
Feature Extractor (Convolutional)
Used inside
$\pi_{MCTS}(\cdot|s_t)$
$\{s_t, \pi_{MCTS}(\cdot|s_t), G_t\}$
Loss computation
$\mathcal{L}_v$ Value (critic) loss
$\mathcal{L}_\pi$ Expert Imitation (actor) loss
Improves
Best Response Expert Iteration
FC
$\{s_t, \pi_{MCTS}(\cdot|s_t), G_t, s_{t+1}, \pi_1(\cdot|s_{t+1})\}$
$\mathcal{L}_{PI}$ Policy Inference loss
Apprentice
Expert MCTS search tree
Dataset

Figure 4.1: Graphical depiction of the partial inner workings of both ExIt (top half) and BRExIt (bottom half) for a 2-player game, portraying differences between both approaches. **Apprentice:** ExIt's apprentice architecture follows a standard actor-critic architecture, with a policy head and a value head. BRExIt has a more intricate composition adapted from [Hong et al., 2018] which features opponent models and intertwined internals, as explained in Section 4.4.2. **Expert:** During MCTS action priors are computed as part of its expansion phase, as explained on Equation 4.3, to bias future searches that traverse the tree. ExIt uses its apprentice policy head π to compute priors for all nodes regardless of the player index stored within the node statistics, as shown by the red single-lined edges on ExIt's MCTS search tree above. BRExIt bases its decision of how to compute action priors on whose turn a node corresponds to. For opponent nodes, during training, these action priors are computed from the true or modelled opponent policy $\hat{\pi}$, as shown by the double blue lines on BRExIt's tree edges. This opponent aware computation is formalized in Equation 4.7. **Dataset:** For every state s_t encountered during training MCTS generates a policy target $\pi_{MCTS}(\cdot|s_t)$ for the apprentice policy and a proxy of the episodic return G_t as target for the apprentice value function, these are added to the dataset. ExIt's dataset only requires the states observed during training, the policy targets π_{MCTS} and episode returns G_t, from which two losses are computed to improve the apprentice actor and critic heads respectively. BRExIt adds a third opponent modelling loss by also gathering the states observed by the opponent s_{t+1} and the output of their policy $\pi_1(\cdot|s_{t+1})$ (centralized training) or one-hot encoded observed action (decentralized training).

Our goal is to train a system which can both correctly identify opponent policies $\boldsymbol{\pi}_{-i}$ and also compute a best response against these policies $\pi_i \in BR(\boldsymbol{\pi}_{-i})$.

4.4.1 Learning Opponent Models in Sequential Games

Previous approaches aimed at learning opponent models used observed actions as learning targets, coming from a game theoretic tradition where individual actions can be observed but not the policy that generated them [Brown, 1951]. However, the centralized population-based training schemes which motivate our research already require access during training to all policies in the environment both for the training agent and the opponents policies (Assumption 2 from Section 3.3.2). We further exploit this assumption by allowing the use of full distributions over actions, computed by the opponents during play, as learning targets instead of one-hot encoded observed actions in the policy inference loss from Equation 4.1c.

In prior work these approaches have mostly been applied to fully observable *simultaneous* games [Hernandez-Leal et al., 2019,Hong et al., 2018], where a shared environment state is used by all agents to compute an action at every timestep, much like the RirRPS environment used in the previous chapter. Thus, if agent i wanted to learn models of its opponents' policies, storing (a) each of i's observed shared states and (b) opponent actions, was sufficient to learn opponent models. We extend this to sequential games, where agents take turns acting based on individually observed states. BRExIt augments the dataset collected by ExIt, specified in Equation 4.5, by adding (1) The state which each opponent agent $j \neq i$ observed and (2) the ground truth action distributions given by the opponent policies in their corresponding turn. Formally, assuming n_o opponents and that turns are taken in a cyclic fashion without change in turn order, at every timestep t where BRExIt acts, it builds a dataset on a replay buffer, where π_n, s_{t+n} correspond to the nth opponent policy and its observed state.

$$\{s_t, \pi_{MCTS}(\cdot|s_t), \{s_{t+j}, \pi_j(\cdot|s_{t+j})\}_{j=1}^{n_o}, G_t\} \tag{4.6}$$

4.4.2 Apprentice Representation

BRExIt's three headed apprentice neural network architecture extends ExIt's actor-critic representation with opponent models as per DPIQN's design [Hernandez-Leal et al., 2019], depicted on the bottom left of Figure 4.1 for a single opponent. As explained in Section 4.2.1, this architecture reuses parameters from the opponent model to act as feature shaping mechanisms for both the actor and the critic. This neural net takes as input environment states $s_t \in \mathcal{S}$ for a timestep t, which can correspond to the observed state of any agent. It features three outputs (1) The apprentice's actor policy $\pi_{\boldsymbol{\theta}}(s_t)$ (2) The state-value critic $V_{\boldsymbol{\phi}}^{\pi_\theta, \hat{\boldsymbol{\pi}}_{\boldsymbol{\Psi}}}(s_t)$ (3) The opponent models $\hat{\pi}^{\psi_j}(s_t) \in \hat{\boldsymbol{\pi}}^{\boldsymbol{\Psi}}$, where $\boldsymbol{\Psi}$ contains the parameters for all opponent model and $\psi_j \subset \boldsymbol{\Psi}$ the parameters for opponent model head j. We drop the superscript from the learnt state-value function for notational clarity, $V_{\boldsymbol{\phi}}$.

Note that on sequential games the distribution of states encountered by each agent might differ, and so the actor-critic head and each of the opponent model heads might each be trained on different state distributions. In practice this means that the output of each head might only be safe to use in states corresponding to the turns of agents that said head is modelling. This does not affect BRExIt, which only uses opponent models during search in nodes corresponding to opponent turns, and the modelling heads have been trained from states observed by the opponent, so there is no risk for evaluating the model on out of distribution states. Simultaneous games are not affected by this, as all observed environment states are shared among all agents.

We now describe in detail the architecture presented in [Hong et al., 2018, Hernandez-Leal et al., 2019], showed at the bottom left corner of Figure 4.1. Let's assume that agent i is represented by such network. An input state s_t goes over a feature extractor module, which forks into two paths, an opponent modelling path (left) and actor-critic path (right). On the left, the extracted features are processed by a sequence of fully connected layers[1] which compute a latent vector z of opponent features. These features z act as the input to each of the opponent modelling heads, $\hat{\pi}^{\psi_j} \subset \hat{\pi}^{\psi}_{-i}$, used to reconstruct the other agent's policies. These opponent modelling heads are represented by a fully connected layer followed by a final softmax layer that outputs the estimated action distributions. On the right side of the architecture, after extracting actor-critic specific features, the latent z is integrated in the bodies of the actor-critic path. This integration acts as a statistical conditioning. This means that the last two heads compute a policy $\pi_\theta(\cdot|z)$ and value function $V_\phi(\cdot|z)$ conditioned on the latent vector used to inferred opponent policies z[2].

With an understanding of the internal workings of the apprentice's architecture and how to interpret its outputs, we discuss its use within the experts MCTS to bias its search around the vicinity of a best response against given or modelled opponents.

4.4.3 Opponent Modelling Inside MCTS

BRExIt follows Equation 4.7 to compute the action priors $P(s,a)$ for the PUCT selection formula in Equation 4.3 for a node with player index j, notably using opponent models in nodes within the search tree that correspond to opponent turns. This process is exemplified in the lower middle half of Figure 4.1. Such opponent models can be either the ground truth opponent policies, the real opponent policies π_{-i}, by exploiting centralized training scheme assumptions, or otherwise the apprentice's learnt opponent models $\hat{\pi}^{\psi}_{-i}$. This is a key difference from ExIt, which always uses the apprentice's own policy π_θ to compute $P(s,a)$.

$$P(s,a) = \begin{cases} \pi_\theta\ (a|s) & j == \text{BRExIt player index} \\ \pi^{j}_{-i}(a|s) & \text{If using ground truth models} \\ \hat{\pi}^{\psi_j}(a|s) & \text{If using learnt opponent models} \end{cases} \tag{4.7}$$

By using either ground truth or learnt opponent models, we initially bias the search towards a best response against the actual policies in the environment. In contrast, if the apprentice π_θ was used to compute $P(s,a)$ at *every* node in the tree, we would bias the search to compute a best response against π_θ. That is, MCTS assumes that all agents follow the same policy

[1] The original publication [Hong et al., 2018] uses an LSTM layer, The choice of the exact architecture is not the focus of our work, so we used fully connected layers for simplicity.

[2] This latent vector z could be interpreted as an agent type modelling [Hong et al., 2018].

as the apprentice, whereas in reality agents might follow any arbitrary policy. However, note that $P(s,a)$ will eventually be overridden by the aggregated simulation returns $Q(s,a)$, as with infinite compute MCTS converges to best response against a perfectly rational player, whose actions may deviate from the underlying opponent's policy. Thus, BRExIt's search maintains MCTS's asymptotic behaviour while biasing short searches, as they are typical in ExIt, towards the game sub-tree which is likely to be visited by given opponents.

For completeness we now present pseudocode for implementing BRExIt. Algorithms 14 & 15 show pseudocode for sequential games, and similarly for simultaneous games in Algorithms 12 & 13. Finally, Algorithm 16 depicts the training loop. Colored lines represent our contributions with respect to ExIt. Figure 4.1 visually compares BRExIt and ExIt.

Algorithm 12: BRExIt Single Match Data Collection (Simultaneous Game)

```
Input: Three head network: $NN = (\text{apprentice } \pi, \text{opponent models } \hat{\pi}^{\psi}_{-i}, \text{critic } V_{\phi})$
Input: Opponent policies: $\boldsymbol{\pi}_{-i}$
Input: Environment: $E = (\mathcal{P}, \rho_0)$
Initialize dataset: $D = [\,]$;
Sample initial state $s_0 \sim \rho_0$;
for $t = 0, 1, 2, \ldots$ do
    Search using apprentice and opponent models: $a_t, tree_t = MCTS(\pi, \boldsymbol{\pi}_{-i}, V)$;
    Derive from tree: $\forall a \in \mathcal{A}\ \pi_{MCTS}(s_t, a) = \frac{N_{root}(s_t,a)}{\sum_{a'} N_{root}(s_t,a')}$;
    Act in the game: $s_{t+1}, \boldsymbol{r}_t \sim \mathcal{P}(s_t, ([a_t] \cup [a^t_j \sim \pi_j(s_t) \quad \forall \pi_j \in \boldsymbol{\pi}_{-i}])$;
    Add datapoint: $D = D \cup (s_t, \boldsymbol{r}_i, \pi_{MCTS}(s_t, \cdot), r_i, \boldsymbol{\pi}_{-i}(s_t))$;
end
return $D$;
```

Algorithm 13: BRExIt Model Update (Simultaneous Game)

```
Input: Three head network: $(\text{apprentice } \pi_{\boldsymbol{\theta}}, \text{opp. models } \hat{\pi}^{\psi}_{-i}, \text{critic } V_{\boldsymbol{\phi}})$
Input: Dataset: $D$
for $t = 0, 1, 2, \ldots$ do
    Sample batch of size $n$: $(s_t, \pi_{MCTS}(s_t), \boldsymbol{\pi}_{-i}(s_t), v)_{1,\ldots,n} \sim D$;
    MSE value loss: $\mathcal{L}_v = (v - V_{\boldsymbol{\phi}}(s_t))^2$
    CE policy loss $\mathcal{L}_\pi = \pi_{MCTS}(s_t) \log(\pi_{\boldsymbol{\theta}}(s_t))$ ;
    CE policy inference loss: $\mathcal{L}_{PI} = \frac{1}{|\boldsymbol{\pi}_{-i}|} \boldsymbol{\pi}_{-i}(s_t) \log(\hat{\boldsymbol{\pi}^{\psi}}_{-i}(s_t))$ ;
    Policy inference weight: $\lambda = \frac{1}{\sqrt{\mathcal{L}_{PI}}}$;
    Compute weighted sum of losses $\mathcal{L}_{total} = \lambda(\mathcal{L}_v + \mathcal{L}_\pi) + \mathcal{L}_{PI}$ ;
    Backpropagate gradients $\nabla \mathcal{L}_{total}$ to update $\boldsymbol{\theta}, \boldsymbol{\psi}, \boldsymbol{\phi}$ ;
end
```

Algorithm 14: BRExIt Single Match Data Collection (Sequential Game)

```
Input: Three head network: (apprentice $\pi_{\boldsymbol{\theta}}$, opp. models $\hat{\boldsymbol{\pi}}^{\psi}_{-i}$, critic $V_{\boldsymbol{\phi}}$)
Input: Opponent policies: $\boldsymbol{\pi}_{-i}$
Input: Environment: $E = (\mathcal{P}, \rho_0)$
Initialize dataset: $D = [\,]$;
Initialize time $t \leftarrow 0$;
Sample initial state $s_0 \sim \rho_0$;
while $s_t$ is not terminal do
    Search: $a_t, tree = MCTS(s_t, \pi_{\boldsymbol{\theta}}, \boldsymbol{\pi}_{-i}, V_{\psi})$;
    Act in the game $s_{t+1}, r_t \sim \mathcal{P}(s_t, a_t)$;
    Get from tree: $\pi_{MCTS}(s_t, a) = \frac{N_{root}(s_t,a)}{\sum_{a'} N_{root}(s_t,a')}$;
    for $j = 1, \dots, |\boldsymbol{\pi}_{-i}|$ do
        Sample opp. action: $a_{t+j} \sim \boldsymbol{\pi}^{j}_{-i}(s_{t+j})$;
        Act in the game $s_{t+j}, \_ \sim \mathcal{P}(s_{t+j}, a_{t+j})$
    end
    Add datapoint: $D = D \cup \{s_t, \pi_{mcts}(s_t), \{s_{t+j}, \boldsymbol{\pi}^{j}_{-i}(s_{t+j})\}_{j=1}^{|\pi_o|}, r_i\}$;
    $t \leftarrow t + |\boldsymbol{\pi}_{-i}|$;
    if episode is over then
        $v$ = env.score();
        Propagate $v$ over all tuples in $D$;
    end
end
return $D$;
```

Algorithm 15: BRExIt Model Update (Sequential Game)

```
Input: Three head network: (apprentice $\pi_{\boldsymbol{\theta}}$, opp. models $\hat{\boldsymbol{\pi}}^{\psi}_{-i}$, critic $V_{\boldsymbol{\phi}}$)
Input: Dataset: $D$
for $t = 0, 1, 2, \dots$ do
    Sample $n$ datapoints from $D$: $(s_t, r_t, \pi_{MCTS}(s_t, \cdot), r_i, \{s_{t+j}, \boldsymbol{\pi}^{j}_{-i}(\cdot|s_{t+1})\}_{j=1}^{|\hat{\boldsymbol{\pi}}^{\psi}_{-i}|})$;
    MSE value loss: $\mathcal{L}_v = (v - V_{\boldsymbol{\phi}}(s_t))^2$ ;
    CE policy loss $\mathcal{L}_\pi = \pi_{MCTS}(s_t) \log(\pi_{\boldsymbol{\theta}}(s_t))$;
    CE policy inference loss: $\mathcal{L}_{PI} = \frac{1}{|\boldsymbol{\pi}_{-i}|} \sum_{j=1}^{|\boldsymbol{\pi}_{-i}|} \pi^{j}_{-i}(s_{t+j}) \log(\hat{\pi}^{\psi_j}_{-i}(s_{t+j}))$;
    Policy inference weight: $\lambda = \frac{1}{\sqrt{\mathcal{L}_{PI}}}$;
    Weighted final loss $\mathcal{L}_{total} = \lambda(\mathcal{L}_v + \mathcal{L}_\pi) + \mathcal{L}_{PI}$;
    Backpropagate $\nabla \mathcal{L}_{total}$ through $\boldsymbol{\theta}, \psi, \boldsymbol{\phi}$ ;
end
```

Algorithm 16: BRExIt Training Loop

```
Input: Three head network: (apprentice $\pi_\theta$, opp. models $\hat{\pi}^{\varphi}_{-i}$, critic $V_\phi$)
Input: Opponent policies: $\pi_{-i}$
for training iteration $= 0, 1, 2, \dots$ do
    Get dataset $D = DatasetCollection(NN, \pi_{-i})$ (Algorithm 14);
    Update $NN = UpdateApprentice(NN, D)$ (Algorithm 15);
end
return NN
```

4.4.4 MCTS as an Agent Evaluation Tool: MCTS Equivalent Strength

RL agents tend to generate *reactive* policies. That is, policies which exploit immediate weaknesses of their opponents in order to maximize rewards. This is mostly due to a couple of reasons. The first one is an epistemic [3] short-sightedness, they *must* act based on individual observations, with no look-ahead, and can therefore only react to immediate observations. This could be partially addressed by adding memory modules to agents by using, for instance, LSTMs [Sak et al., 2014] or differentiable memory [Beck et al., 2019]. They capture all their understanding of the consequences of their actions purely within their policy parameters. The second reason is schematic, based on the training scheme by which these policies arise. During the training of RL agents, these agents only face other RL agents, and therefore only learn to play against reactive agents. The lack of example planning policies to learn against, coupled with the lack of pressure to learn to play against such type of policies means that the agents will not learn how to become proficient against planning based policies. Thus, the relative performance between two populations of reactive agents might not be informative about which population is the strongest against planning based methods.

We propose a direct usage of MCTS as an evaluator for agent skill, which we now describe and motivate. The call of an MCTS procedure has a few parameters, such as the exploration factor C_{UCT} used in Equation 2.17, which can greatly affect its internal behaviour. Yet the most important one is the *computational budget* allocated to the procedure. That is, the number of iterations of the inner loop in Algorithm 7, which cycles over MCTS's four main phases. Increasing this budget is loosely equivalent to improving the skill of an agent[4]. This convenient positive correlation between the computational budget and MCTS' performance has lead some authors to use MCTS as a way of generating gameplaying agents of targeted skill levels. For instance, in [Czarnecki et al., 2020] the authors generate various agents of similar skill level by running MCTS with a fixed computational budget, with different strategies being generated by using different seeds for their random number generation. Each seed can be thought of as a possible strategy within a skill level, which links different seeds to a notion of behavioural diversity. By repeating this with a different computational budget, we

[3] Coming from the mathematical models used to represent the policies.

[4] This excludes *pathological game trees*, which refer to game trees for which searching deeper is actually detrimental, unless a terminal node is found [Abramson, 2013]. None of the games used in the experiments in this thesis are pathological, so we ignore such cases in our reasoning.

can get sets of diverse agent policies at different levels of skill. MCTS equivalent strength can serve as a somewhat objective measure of transitive skill against planning based methods. This can be specially useful to complement the performance study for an agent or population of agents when conjoined with other metrics such as Nash averaging.

MCTS Equivalent Strength Definition

Echoing this sentiment, we extend this usage of MCTS to generate a scalar transitive performance metric for agent skill. We define the *MCTS equivalent strength* of an agent policy π at a winrate $w\%$ as the computational budget required for an MCTS agent to have a winrate of $w\%$ against policy π.

Algorithm 17: MCTS Equivalent Strength

```
Input: Agent to benchmark: $\pi$
Input: $target\_winrate \in [0,1]$
Input: $initial\_budget \in \mathbb{N}$
Input: $max\_budget \in \mathbb{N}$
Input: $budget\_step \in \mathbb{N}$
for $budget \leftarrow initial\_budget$ to $max\_budget$ by $budget\_step$ do
    $\pi_{MCTS} \leftarrow MCTS(budget)$;
    winrate $\leftarrow compute\_winrate(\pi, \pi_{MCTS})$;
    if $winrate \geq target\_winrate$ then
        Result: $budget$
    end
end
Result: $budget$
```

Time Complexity

We now informally estimate the time complexity of Algorithm 17. For simplicity, we will compute the complexity of this metric for 2 player games, and assume that the non-MCTS agent takes constant time to compute its actions. This is often the case when comparing MCTS agents versus agents using neural networks to represent their policies, as a single policy forward pass on a neural network takes negligible time when compared against a full MCTS procedure. We make some further simplifying assumptions, presented alongside the variables that are required to compute the time complexity:

1. o_{MCTS}: The time complexity for an *average single iteration* of the MCTS loop. For simplicity, we keep this value constant, although the time complexity of the rollout phase of MCTS varies on many factors such as the depth of the MCTS tree or the average game tree depth at a given state.

2. b: The computational budget given to the MCTS agent.

3. b_{min}, b_{max}: Lower and upper budget bounds contemplated by the algorithm, denoting the "strength" search space. The larger space covered, the more likely the algorithm is

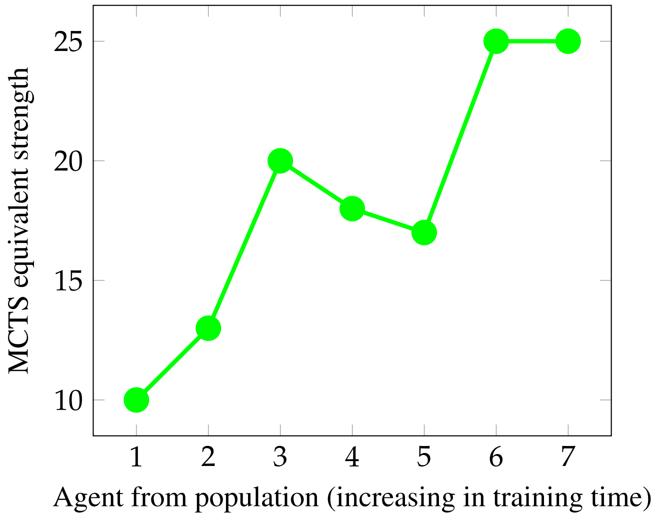


Figure 4.2: Exemplary MCTS equivalent strength for a 50% winrate for a population of 7 agents. The horizontal axis denotes the index of the agent within the population, with a higher value indicating longer training time. The vertical axis denotes the MCTS equivalent strength. It can be the case that sometimes longer training time does not lead to a higher MCTS equivalent strength, as with agents 3-6. Or sometimes there is an stagnation, as with agents 6-7, even if agent 7 hypothetically beats agent 6 on average. Note that this data is fictitious and used only for illustrative purposes, it does not correspond to any training run on an actual environment.

to find the correct agent strength. For practical purposes, however, it is preferable to center such values around the perceived strength of the agent.

4. l: The average length of an episode for a target task. Conversely, the average depth of the search tree for current environment.

5. n: The number of episodes used to estimate the winrate between the MCTS agent and the target agent. Larger values will yield more faithful approximations of the true winrate between the MCTS agent with a given budget and the agent being assessed, at the expense of longer compute.

The time complexity of a single episode of length l with budget b is $O(\frac{l}{2}o_{mcts}b)$, as we treat the opponent moves as constant time, we ignore its moves, halving the length of the episode. We need to multiply this number by n, as we need n episodes to approximate the true winrate for a single budget value b, obtaining $O(n\frac{l}{2}o_{mcts}b)$. Finally, we are sweeping in the budget range of b_{min}, b_{max}, which means that we need to multiply the single budget estimation by $\sum_{b_{min}}^{b_{max}} b = \frac{(b_{min}+b_{max})}{2}(b_{max} - b_{min} + 1)$, for the worst case scenario where we need to finish the sweep over the entire range. Thus, the simplified time complexity is $O(n\frac{l}{2}o_{MCTS}(\frac{(b_{min}+b_{max})}{2}(b_{max} - b_{min} + 1)))$.

4.5 Experiments

We now describe an experiment carried out to show the sample efficiency superiority of BRExIt versus ExIt, where we test the following two hypotheses:

1. Using a full distribution available only under centralized training schemes as a target for opponent model learning is more beneficial than the literature standard of one hot action encodings. This is explored in Section 4.6.1.

2. BRExIt is more sample efficient than ExIt at distilling into its apprentice π a policy which acts as best response against a fixed agent π', due to the richer learning targets for the apprentice. This is explored in Section 4.6.2.

We carry our experiment in the fully observable, sequential game of Connect4, chosen because of the same reasons as in the last chapter, due to it being computational amenable and its high degree of skill transitivity [Czarnecki et al., 2020], as explored in earlier chapters.

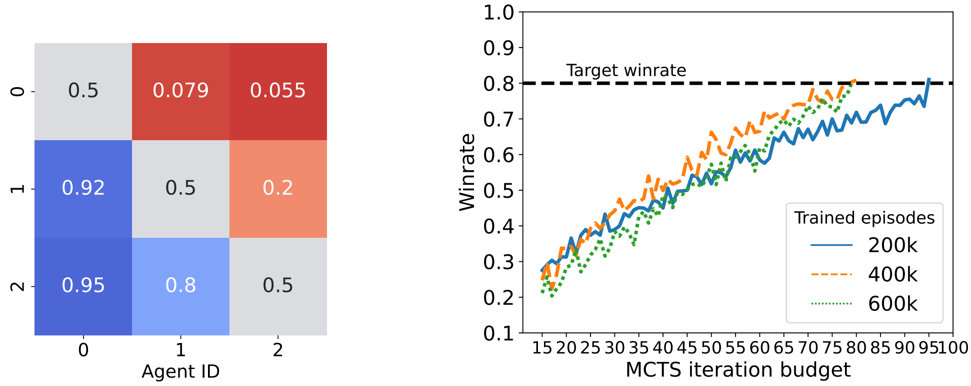
4.5.1 Test Agents: Training & Benchmarking

We generated three monotonically stronger agents to be used as test against against whom to benchmark sample efficiency: $[\pi_{weak}, \pi_{medium}, \pi_{strong}]$. These were created by freezing copies of a PPO agent trained under $\delta = 0$-Uniform self-play after 200k, 400k and 600k episodes. We evaluate an additional opponent policy π_{mixed}, representing a mixture over all previous agents, which uniformly samples one of the three policies to follow at the start of each episode. Table 4.1 shows the hyperparameters used for this training. We now give a detailed description of how these agents were trained and how the strength labels of *weak, medium & strong* were chosen.

Hyperparameter name	Value
Horizon (T)	2048
Adam stepsize	10^{-5}
Num. epochs	10
Minibatch size	256
Discount (γ)	0.99
GAE parameter (λ)	0.95
Entropy coeff.	0.01
Clipping parameter (ϵ)	0.2
Gradient norm normalization	1

Table 4.1: PPO hyperparameters used to generate test agents. No formal hyperparameter sweep was conducted, and the final values were chosen after a few manual trials.

To evaluate the relative strength of the test agents we computed their corresponding win-rate evaluation matrix to study pair-wise agent performances, as shown in Figure 4.3a. We see that older agents (age is represented by agent ID) can consistently beat previous versions of themselves, with the Nash support over the three agents putting all weight on the final agent. This shows a transitive improvement within the population of agents that emerges from our single training run. However, the nature of PPO agents is reactive, as there is no planning involved during their training. The results from Figure 4.3a might come as a result of the PPO agents learning how to exploit specific weaknesses of previous agents, weaknesses that



(a) Winrates of every test agent against each other, where each entry was computed by playing 1000 head-to-head matches in the game of Connect4. The position of each agent (first player or second player) was chosen randomly at the beginning of each match to enforce symmetry.

(b) Evolution of winrate by MCTS against each of the individual 3 test agents (excluding π_{mixed}). As the MCTS computational budget increases on the horizontal axis, so does its playing strength. The black horizontal dashed line at the 80% winrate mark is shown to compare what is the required budget to reach such winrate against all test agents. The 80% MCTS equivalent strengths for the 200K, 400K and 600K agents are 96, 81 and 81 respectively.

Figure 4.3: Quantitative metrics meassuring the performance of the test agents.

might not be present in other agents generated via planning based methods. Because BRExIt during training uses planning to choose its actions, we ultimately want to give a ranking to our test agents against planning based methods. In order to label our test agents with *weak*, *medium* and *strong* we use *MCTS equivalent strength*.

Figure 4.3b shows a sweep of MCTS equivalent strength up to 80% winrate. Surprisingly, we see that the test agent with only 200K training episodes has the highest MCTS equivalent strength for a 80% winrate, even though it is the weakest when matched uniquely against other agents from the test population as seen in Figure 4.3a. We hypothesize that in the relatively low computational budgets swept over in Figure 4.3b, the heavily stochastic behaviour of the 200K agent can thwart shallow plans made by low budget MCTS agents. For the remaining two agents, the agent trained the longest at 600K episodes consistently stays below the 400K agent for the majority of budget values in Figure 4.3b, even though they both feature the same MCTS equivalent strength at 80% winrate. We tilt this draw in favour of the 600K agent as it beats the 400k agent. From the strength analysis conducted for Figure 4.3a and Figure 4.3b, we label the 400K, 600K and 200K agents as *weak*, *medium* and *strong* respectively.

Finally, the neural network architecture used to represent the actor-critic architecture of the test agents is the same as for the PPO agents trained in Chapter 3. In order to level the playing field to compare all algorithms, we have adjusted the actor-critic architecture used by the test agents and vanilla ExIt, and the opponent modelling enhanced actor-critic architecture used by BRExIt described in Section 4.4.2 to feature a similar number of parameters, approximately 27.000 trainable parameters.

4.5.2 BRExIt Ablation Training Against Fixed Agents

We train 7 types of agents, performing an additive construction from ExIt to BRExIt.

1. Expert Iteration (**ExIt**): The original expert iteration algorithm as presented in [Anthony et al., 2017] and [Silver et al., 2016], without a rollout phase.

2. Expert iteration with Opponent Modelling for Feature Shaping (**ExIt-OMFS**): Vanilla ExIt but featuring BRExIt's neural network architecture and using opponent modelling as an auxiliary task. These opponents models are not queried at any point, serving only as feature shaping mechanisms.

3. Best Response Expert Iteration with learnt Opponent Models within Search (**BRExIt-OMS**): BRExIt which uses its own opponent models $\hat{\pi}^{\psi_j} \subset \hat{\pi}^{\psi}$ for MCTS' expansion phase. It does not require access to the true opponent models during search.

4. Best Response Expert Iteration (**BRExIt**): As presented in Section 4.4, requiring access to the ground truth opponent policies to be used to bias MCTS search. The search procedure here is akin to the one presented in [Timbers et al., 2020]. It also learns opponent models purely for feature shaping mechanisms, as the ground truth models are used within search.

The opponent models used in the last three algorithms above exploit centralized training assumptions and use full distributions as training targets. We also train separate versions using one-hot encoded actions as targets. Each algorithm was independently trained 20 times against each of the 4 test agents, yielding a total of 560 training runs. All algorithms use an MCTS budget of 50 iterations due to computational constraints, with policy updates occurring every 800 games[5]. See Table 4.2 for a full set of hyperparameters. Following state-of-the-art statistical practices [Agarwal et al., 2021] we use Inter Quartile Metrics (IQM) for all results (graphs and statistics), discarding the worst and best performing 25% runs to obtain performance metrics less susceptible to outliers.

4.6 Results

Figure 4.4 shows the number of episodes required by ExIt and algorithms that used full distribution opponent model targets to train their apprentices to reach a 50% winrate against each test opponent. Figure 4.5 shows the episodes required by all algorithms using opponent models, contrasting the performance between one-hot encoded targets and full distribution targets. Figure 4.6 shows the probability of improvement (PoI) that one ablation has over another [Agarwal et al., 2021] based on results from Figure 4.4, defined as the probability that algorithm X takes less time to converge to a 50% winrate than algorithm Y for an arbitrary training run against an arbitrary test opponent.

[5] The avid reader will realize that the length of a game can vary due to the stochasticity of the test and training policies, and thus the number of datapoints collected after 800 games can vary too. On preliminary experiments, we did not observe that agents that had obtained larger datasets gained any advantage.

Hyperparameter name	Value
MCTS budget	50
MCTS rollout budget	0
MCTS exploration factor	2
Dirichlet noise	False
Batch size	512
Train every n episodes	800
Epochs per iteration	5
Reduce temperature after n moves	10
Initial generations in memory	5
Final generations in memory	200
Increase memory every n generations	5
Learning rate	1.5e-3
Activation function	*ReLu*
Actor activation function	*Softmax*
Opponent models activation function	*Softmax*
Critic activation function	*tanh*

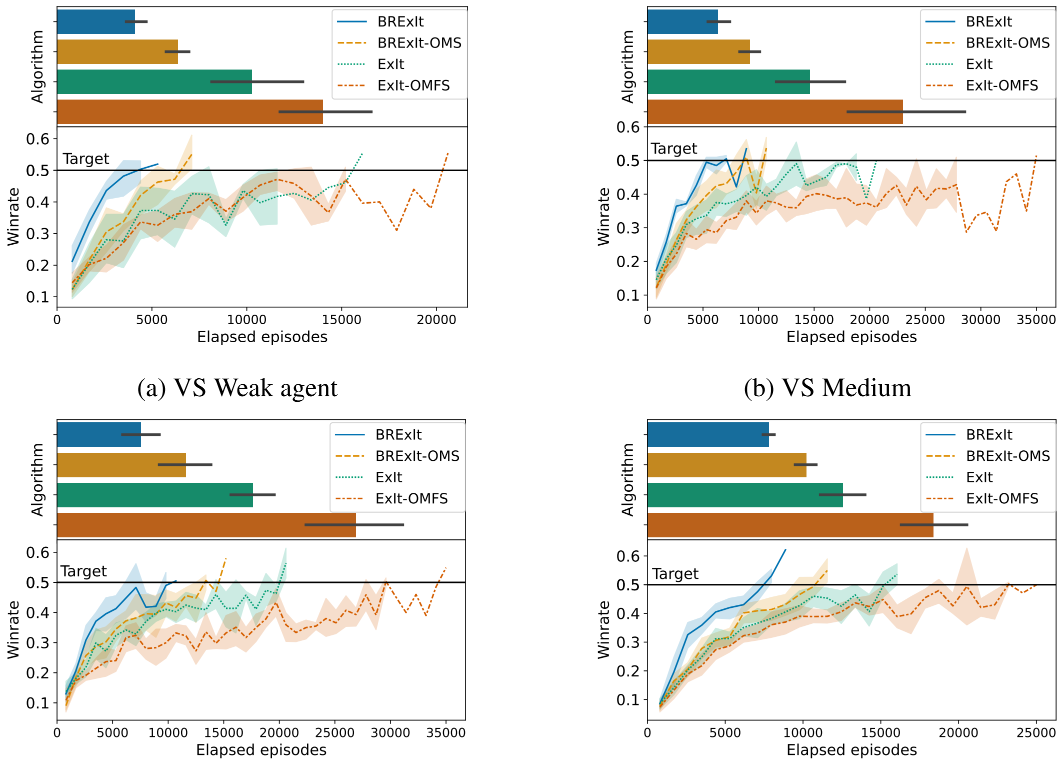
Table 4.2: BRExIt, BRExIt-OMS, ExIt and ExIt-OMFS hyperparameters.

4.6.1 Full Distribution VS One-Hot Opponent Model Targets

We observe no statistical difference between agents trained with algorithms using one-hot encoded opponent model targets when compared to using full action distributions. To assess this, we independently compare each pair containing an algorithm from Figure 4.5 and its respective one-hot target counterpart by running a two-sample Kolmogorov-Smirnov test. The samples to compare are the sets of elapsed episodes (one datapoint per training run) required to reach the experiment's target winrate. We obtain $p \gg 0.05$ for each algorithmic combination, so we cannot reject the null hypotheses that both data samples come from the same distribution; the only exception being ExIt-OMFS, which shows a statistically significant decrease in performance when using one-hot encoded targets.

This runs against the intuition that richer targets for opponent modelss will in turn improve the quality of the apprentice. However, we observed that full distributional targets do yield opponent models with better prediction capabilities, as indicated by lower loss of the opponent model during training. This hints at the possibility that opponent model's usefulness increases only up to a certain degree of accuracy, echoing recent findings [Goodman and Lucas, 2020]. Hence, while BRExIt does require access to ground truth policies during search, the search in BRExIt-OMS only needs opponent models trained on action observations, which is promising for using BRExIt-OMS in settings where stronger assumptions about access to opponent policies might not be available to the training scheme.

Table 4.3 shows the p-values corresponding to the different two-sided Kolmogorov-Smirnov tests, which measures if there is a significant statistical difference between 2 distributions $f(x)$ and $g(x)$. In our case, f, g correspond to an algorithmic ablation, x to a test agent policy and $f(x), g(x)$ to the stochastic functions that determine how many episodes



(a) VS Weak agent

(b) VS Medium

(c) VS Strong

(d) VS Mixed

Figure 4.4: Comparison of algorithm variants training versus fixed opponents (a-d) until reaching the target threshold of 50% winrate. **Top-halfs** show the number of episodes required to reach the threshold (lower is better). **Bottom-halfs** show progression of mean winrates during training (higher is better). Whiskers (top) and shaded areas (bottom) show 95% bootstrap confidence intervals. A datapoint is computed every policy update (i.e every 800 episodes) by evaluating the winrate of the apprentice policy against the opponent over 100 episodes. Note that runs terminate when crossing the target, and thus mean winrates cover fewer training runs for high episode numbers.

are required to achieve the target winrate of 50% against policy x. Each of the 10 training runs used for each algorithm f gives us one data point for $f(x)$. Thus, our comparisons are between sets of 10 datapoints $f(x) = [f_i(x)]_{i=1}^{10}$ and similarly $g(x) = [g_i(x)]_{i=1}^{10}$.

4.6.2 On BRExIt's Sample Efficiency

Figure 4.4 shows that BRExIt consistently requires the least number samples to reach the target winrate across all test agents. BRExIt regularly outperforms BRExIt-OMS (80% PoI from Figure 4.6), successfully exploiting the centralized assumption of opponent policies being available during search for extra performance. If opponent policies can only be sampled to generate game trajectories during training but not during search, using learnt opponent models during search is still beneficial, as we see that BRExIt-OMS consistently outperforms ExIt (79 % PoI). Surprisingly, ExIt-OMFS performs worse than ExIt by a significant margin (ExIt has a 80% PoI against ExIt-OMFS), providing empirical evidence that opponent model with static opponents is *detrimental* for ExIt if opponent models are not exploited within MCTS.

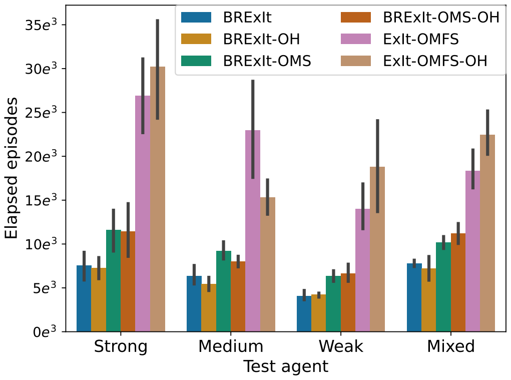


Figure 4.5: To compare one-hot (OH) and full policy distribution targets, each algorithmic ablation is shown next to its counterpart. Each bar shows the number of episodes required for each apprentice policy to reach a winrate of 50% winrate against the opponent test agent. Lower is better; black bars indicate 95% confidence intervals.

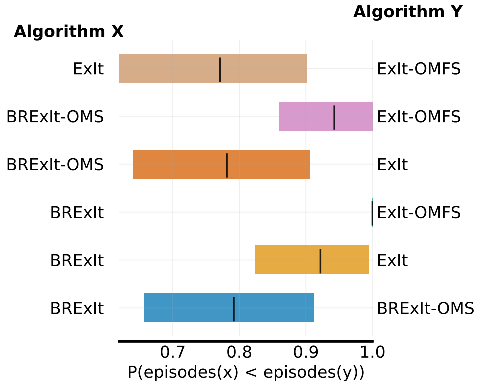


Figure 4.6: Each row shows the probability (vertical marker) that the algorithm X (left) trains its apprentice's policy to reach a winrate of 50 % with fewer episodes than the algorithm Y (right). Higher is better for algorithm X; colored bars indicate 95% bootstrap confidence intervals. Note that because every BRExIt run finished earlier than ExIt-OMFS, the former features a 100% PoI.

Algorithm	Test agent	p-value	Significant
BRExIt	Weak	0.930	No
BRExIt-OMS	Weak	0.931	No
ExIt-OMFS	Weak	0.930	No
BRExIt	Medium	0.929	No
BRExIt-OMS	Medium	0.474	No
ExIt-OMFS	Medium	0.474	No
BRExIt	Strong	0.930	No
BRExIt-OMS	Strong	0.999	No
ExIt-OMFS	Strong	0.930	No
BRExIt	Multiple	0.142	No
BRExIt-OMS	Multiple	0.930	No
ExIt-OMFS	Multiple	0.025	Yes

Table 4.3: P-values corresponding to the different two-sided Kolmogorov-Smirnov tests comparing full distribution targets vs one-hot encoded action targets.

This goes against previous results [Wu, 2019], which explored opponent model within ExIt merely as a feature shaping mechanism and claimed modest improvements when predicting the opponent's policy on the next state. At first sight, differences may be attributed to the fact that we instead model the opponent policy on the current state because as it is needed to compute node priors during search. However this might be due to any other number of reasons, such as a difference in environment (Connect4 vs. Go), difference in number of MCTS iterations (we use 50 vs. they use 600), amongst other factors.

These empirical results show both BRExIt and the less constraining BRExIt-OMS featured a $> 93\%$ and $> 85\%$ probability of improvement against vanilla ExIt in terms of reaching a 50% winrate against against the test agents.. This warrants their usage in scenarios where traditionally ExIt style algorithms have been used and builds on the recent literature trend of discovering that opponent model can is a useful auxiliary task for MARL.

The overarching focus of this chapter is to increase the sample efficiency of an oracle as a best response operator within a centralized training scheme. With this in mind, and having empirically established the better sample efficiency of BRExIt within the Connect4 environment, we turn our attention to exploring the training schemes under which its predecessors ExIt and AlphaGo were used in the literature and contrast the potential outcomes of having instead used BRExIt as an oracle.

4.6.3 A Reflection on the Evolution of AlphaGo's Training Scheme

ExIt was popularized over three publications which presented increasingly stronger playing agents: AlphaGo [Silver et al., 2016], AlphaGo Zero [Silver et al., 2017b] and AlphaZero [Silver et al., 2018]. Each publication kept ExIt's internal workings mostly unchanged except for hyperparameter choices and the progressive removal of environment specific enhancements. Yet, every publication meaningfully changed the higher level training scheme that governed ExIt's training, which we discuss here along with the implication of these changes for training ExIt or BRExIt agents.

The initial AlphaGo used $\delta = 0$ self-play, where a population kept previous versions of the policy network from which opponents were randomly sampled at the beginning of each episode. The latest version of the policy network was added to the population after a set number of model updates. The authors claimed that this opponent randomization kept the apprentice from overfitting to its own policy. Recall that the underlying ExIt agent biases its expert search towards a best response against the apprentice's own policy, by using the apprentice π_θ to compute action priors on every node in the search tree. This means that the expert generated policies targets will not try to exploit the opponents it trained against, choosing instead more conservative searches which regress against a response to a Nash equilibrium. Recent studies show that this conservativeness can be detrimental in terms of finding diverse sets of policies throughout training for population-based training schemes [Muller et al., 2019, Liu et al., 2021]. Instead, they advocate for a more aggressive computation of best responses against iteratively generated known opponents. Through this practice, a wider area of the policy space is explored by the training agents, allowing the higher level training scheme to better decide on which opponents to use as targets to guide future exploration [Balduzzi et al., 2019]. We argue that BRExIt has this exploitative property built-in at the agent level by actively biasing its search towards a best response against ground truth opponent policies. Future work could focus on empirically verifying this claim.

AlphaGo Zero and AlphaZero differ in similar ways. The former opts for a restricted version of naive self-play in which the learning ExIt agent plays against an identical copy of itself; but if after some amount of training episodes it fails to beat its previous checkpoint by a set winrate margin, its network parameters are rolled back to that of the checkpoint's, enforcing monotonic improvement on saved checkpoints. Finally, AlphaZero dropped this last constraint, and simply used naive self-play, as they claimed improved performance. We note that under naive self-play as a training scheme, BRExIt using ground truth opponent models and ExIt are equivalent (ignoring the feature shaping effects that opponent models have on the apprentice network). This is because in naive self-play, the opponent acts using the same apprentice policy network as the learning agent. Hence, by having the ground truth opponent models being the same as the apprentice policy network, we have $\pi_{-i} = \hat{\pi}^{\psi}$, meaning that the computation of $P(s,a)$ will be identical for BRExIt and ExIt. In this case the learning agent is approximating a best response against its opponents, even in the case of ExIt. We hypothesise that this might be one of the reasons why naive self-play performs better than $\delta = 0$ uniform for the training of ExIt agents.

4.7 Conclusion

This chapter investigated an approach of granting opponent awareness at the algorithmic level, spurred by the positive findings from the last chapter for doing so at the training scheme level. This opponent awareness took the shape of opponent models via policy reconstruction. An existing trend of policy reconstruction methods within DRL was extended to the ExIt framework, introducing the BRExIt algorithm. BRExIt augments ExIt by introducing opponent models in two sections: within the expert planning phase to bias its search towards a best response against models; and within the apprentice's model, to use opponent modelling as an auxiliary task. In the game of Connect4, we demonstrated BRExIt's superior sample efficiency over ExIt at training a good performing policy against fixed agents, which are often the target objectives within the inner loops of modern MARL training schemes explored in the previous chapter.

Our results open multiple avenues for future work, which we briefly discuss starting from the algorithmic level upwards. The usage of opponent modelling could be extended from just policy reconstruction for each opponent $\hat{\pi}^{\psi}$, to also modelling their corresponding state-value functions $V^{\hat{\pi}^{\psi}}$, so that the backpropagation phase of MCTS could further use agent specific value functions, instead of always using the apprentice's value function $V^{\pi_{\theta}}$ to determine the value of unexplored parts of the search tree arrived at after the selection phase. At the level of training schemes, future work can focus on measuring whether the quality of populations generated by different training schemes using BRExIt surpasses that of populations where ExIt is used as a policy improvement operator. In other words, we have shown that (for a single environment) BRExIt is more efficient than ExIt within the inner loop of the double loop posited by modern MARL training schemes. It still remains to be seen whether this improvements translates to the outer loop.

4.7.1 The Significance of Opponent Models for Game Studios

Refactoring the codebase of a video game to make it compliant with the reinforcement learning loop is a significant undertaking, to the extent where major game engines such as Unity keep dedicated teams for this purpose [Juliani et al., 2018]. Once a team has put in the effort to introduce the RL loop and training pipelines into their game, the implementation of opponent models within agents as discussed in this chapter should be a relatively cheap addition in terms of code and engineering effort. This is especially the case if taking into account the reduced cost in terms of money and time that higher sample efficiency brings. Therefore we bring forward the advice that teams that are thinking of using MARL within their games should consider opponent modelling as a viable low hanging fruit enhancement for their RL approaches.

Training opponent models on gameplay data can be of interest to game designers beyond their usage within RL agents. In essence, opponent models try to capture what is knowable about an opponent purely from observed behavioural patterns. The performance of such models trained on large corpus of gameplay data can give designers a proxy indicator of how much information about future agent behaviour can be captured from observed actions. This allows designers to have a quantitative metric for how clearly are player intentions based purely on their behaviour. Intuitively, this metric should be high for cooperative games (for players want to signal their strategy to make it easier for others to cooperate) and low for competitive games (as players would hide their intentations to reduce exploitability). This is especially the case for console games which typically lack vocal communication. We bring this idea to highlight that opponent models need not only be used for improving the performance of RL agents and we expect their use to become more prevalent with time.

4.8 Implementation Details: Ablation Agnostic Performance Improvements

We end this chapter with a series of performance improvements shared across all algorithmic ablations, which are orthogonal to the novel contributions presented in this chapter but were nonetheless required to train our agents within our computational budget. We argue that these improvement methods benefit all ablations equally, and thus do not affect our comparisons. We present them here for completeness and to aid reimplementations efforts. As discussed and exposed numerous times in this thesis, DRL algorithms require a large amount of simulated experiences to train its policies, and more so for deep MARL. This is further exacerbated when planning based algorithms are used as part of RL algorithms, as is the case with MCTS within BRExIt. A single step in the main game requires many games simulated games inside of MCTS. On top of this, RL methods can be sensitive to hyperparameters, requiring a lot of manual tuning and stabilizing implementation details. Because of this, it is common place to add ad-hoc or environment specific methods to reduce the amount of compute required to train policies and increase their robustness.

4.8.1 Less Exploration: Removing Dirichlet Noise on MCTS Root Priors

The algorithm AlphaGo [Silver et al., 2017a] introduced the notion of adding noise to the prior probabilities in the root note. By adding noise, we modify the priors over actions used in the selection phase, given by the apprentice's state-value function. This is used as an exploration mechanism, by redistributing some probability weight given by the priors over all valid actions. The Dirichlet distribution is used to sample that noise: $P(s,a) = 0.25 * Dirichlet(\alpha) + 0.75 * P(s,a)$, where $\alpha = \sqrt{2}$. This ensures that moves that the apprentice's policy would ignore are visited at least a few times by MCTS. During an MCTS search with thousands of iterations (as was the case with AlphaZero), the effects of this Dirichlet noise on the root node's priors would be washed out by state evaluations propagated upwards from further down the tree. However, in our use case, due to our relatively low MCTS budget of 50, we cannot afford to use Dirichlet noise, as this exploratory noise would randomize too much the search's output policy π_{MCTS}.

4.8.2 Data Augmentation via Exploiting State Space Symmetries

The board of Connect4, a 6×7 matrix, is symmetrical over the vertical axis. We can exploit this to reduce the complexity of the environment. Let's define a function over states $\sigma : \mathcal{S} \rightarrow \mathcal{S}$ which swaps columns with indexes $\{1,2,3\}$ with those with index $\{5,6,7\}$; and also over policies $\sigma(\Pi) \rightarrow \Pi$ by also swapping the probability weight over those column indices. Strategically speaking, those two situations are identical, because of the symmetry on the state space. AlphaZero was shown to be able to naturally learn this symmetry over enough training episodes [Silver et al., 2017a]. We add this symmetry prior into our algorithm to reduce computational costs. This is done by augmenting every datapoint at the time it is going to be appended to the replay buffer by adding a copy with a corresponding symmetric transformation, on both the state and the policy target (and the opponent policy target in case of BRExIt). The value target remains the same. More formally, for every datapoint we obtain a tuple:

$$\begin{aligned} &\{s_t, \pi_{MCTS}(\cdot|s_t), G_t\} \rightarrow \\ &(\{s_t, \pi_{MCTS}(\cdot|s_t), G_t\}, \{\sigma(s_t), \sigma(\pi)_{MCTS}(\cdot|s_t), G_t\}) \end{aligned}$$

4.8.3 Improved Value Targets via Averaging with MCTS Values

In the original ExIt specification, as described in Equation 4.5, at every timestep of each simulated episode, the following information is stored in a replay buffer $\{s_t, \pi_{MCTS}(\cdot|s_t), G_t^i\}$, where G_t^i denotes player i's total reward observed at the end of the episode. What this means for our state value function $V(s_t)$ is that we are regressing all observed states in an episode

$[s_0, s_1, \ldots]$ to the same value G^i_t. This target features a lot of variance, as we are mapping potentially dozens of states to the same target value. We can reduce the variance, with the downside of introducing bias, by re-using state-dependant computation from each state-dependant MCTS call, meaning that we won't use the same value target for each state. For every datapoint we substitute G^i_t by another variable z^i_t, for which we try three different alternatives:

$$z^i_t = \frac{1}{2} * (\underbrace{\sum_{a \in \mathcal{A}} \pi_{MCTS}(a|s) * Q(s,a)}_{\text{Definition of } V^{\pi_{MCTS}}(s)} + G^i_t) \tag{4.8}$$

First, averaging the original original target with MCTS's root node value. Due to the Monte Carlo nature of MCTS, this includes the value of sub-optimal actions that were discarded during simulations as part of its exploration. This in turn induced pessimism in $V(s)$ estimations by our learnt critics $V_\phi(s)$, where they would consistently estimate lower win-rates than those observed in practice.

$$z^i_t = \frac{1}{2} * (\underbrace{\max_{a \in \mathcal{A}} * Q(s,a)}_{\text{Greedy w.r.t } Q(s,\cdot)} + G^i_t) \tag{4.9}$$

Instead, we tried using the value of the root node's most valued child. This also introduced a bias, although this time in a positive direction, because the policy that will be followed corresponds to MCTS's normalized child visitations, Equation 4.4, which will put some probability weight on actions which are not the most valuable.

$$z^i_t = \frac{1}{2} * (\underbrace{Q(s, \arg\max_{a \in \mathcal{A}} N(s,a))}_{\text{MCTS action selection}} + G^i_t) \tag{4.10}$$

Ultimately, we decided to average the episode returns with the $Q(s,a)$ value associated with the action used in the real environment, or equivalently, the action selected by MCTS selection phase. We used a standard action selection phase which given the root node, picks the most visited action, represented by the underlined section of Equation 4.10. Theoretically this still yields biased targets, as our actors π_θ imitate an MCTS derived policy π_{MCTS} while our critic V_ϕ will regress against state-values which will not exactly match the actions taken π_{MCTS}. However, not only did this seem to work well in practice, we saw little to no bias in our critic's estimations compared to the real episode returns using this bootstrapped estimations.

4.8.4 Extra Exploitation via Policy Temperature

It is beneficial to have a high degree of exploration on the initial states of an environment, specially during early stages of training. The intuition being that exploratory policies that feature non-zero probability of taking many different actions will cover a larger volume of $\mathcal{S}$ when compared to deterministic policies, the former allowing the discovery of good performing policies. This is specially important early in an episode, where exploration is much more likely to find undiscovered states rather than nearing the end of an episode. Nearing the end of an episode, more exploratory policies can fail to exploit opportunities to perform a coup de grâce to obtain large rewards which would be readily gathered by more exploitative policies.

The policy targets derived from MCTS's normalized child visitations π_{MCTS} feature a high degree of exploration, especially with our low computational budget, as MCTS would discard low quality moves given larger computational budgets. This is convenient early during training for the aforementioned exploration on initial states, but is detrimental when learning exploitative policies against fixed agents, as it slows learning. Thus, ideally we would like to focus initial moves on exploration, and switch to exploitation later into an episode.

The way we achieve this is by having a temperature parameter τ, which we use to exponentiate MCTS' child visitations *before* they are normalized:

$$\pi_{MCTS}(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_{a'} N(s,a')^{1/\tau}} \tag{4.11}$$

We set $\tau = 1$ for the first 10 moves, after which we reduce it to $\tau = 0.01$, which effectively moves all probability weight to the most visited action, generating a highly exploitative policy. Therefore, the first 10 moves will contain MCTS's exploration, and the rest will feature only the most-visited action. We note that there exists research on this area, with a focus on removing exploration elements from MCTS policy targets with the hope of aiding interpretability [Soemers et al., 2019].

4.8.5 Parallelized MCTS with Centralized Apprentice

We simulate multiple games of Connect4 in parallel, With every MCTS procedure running on an individual CPU core. A single process hosts the apprentice's neural network, acting as a server, whose only function is to receive states from all MCTS processes. These are evaluated for (1) node priors $P(s,a)$ and (2) state evaluations $V(s)$. It busy polls all connections to MCTS processes for incoming states, batching all requests into a single neural network forward pass. This allows us to scale horizontally, theoretically linearly with the number of available CPUs. Because of the size of the network, Connect4's small state size and the average batch size, we saw no performance improvement hosting the apprentice in CPU vs GPU. Surprisingly, our bottleneck came from Python's communication of PyTorch tensors among processes.

4.8.6 Gradient norm clipping

Many RL algorithms which involve computing gradients with respect to a loss function suffer from high variance in the estimation of these gradients. If the parameters of a model are updated by applying gradients of a large magnitude, these might move the model's parameters into poor performing sections of the parameter space. A proposed solution is to introduce gradient clipping [Dosovitskiy et al., 2015]. It's most naive implementation involves clipping the value of *each individual* gradient to a hyperparameter threshold c. A more nuanced alternative is gradient norm clipping. It involves concatenating all gradients of all parameters into a single vector g. If the vector's nth norm is greater than a threshold value c, then the vector is normalized according to:

$$g = \begin{cases} g & \text{if} ||g||_n \leq c \\ c\frac{g}{||g||_n} & \text{otherwise} \end{cases} \tag{4.12}$$

For all experiments we used the L_2 norm. We tried thresholds of 5, 3 and 1. Even though we found that higher thresholds yielded a sharper increase in performance early on, a threshold of $c = 1$ ended up achieving better stability and final performance than the other values.

4.8.7 Growing Replay Buffer Size

When observing the rate of improvement of RL agents with respect to conventional metrics, such as reward for single agent scenarios or winrate versus opponents for multiagent RL settings, it is common to see exponential initial improvements which eventually slow down into relative plateaus. We make the observation that, given a sufficiently large experience replay, the initial experiences generated by policies that were exponentially outclassed by later policies will remain in the experience buffer for a long time. This can be of some benefit for learning a critic $V(s)$, as large replay buffers will almost certainly increase their coverage of the state space. However, we argue that the imitation-based nature of the actor in ExIt based algorithms distinctively suffers from the inclusion of datapoints within their replay buffer at early stages of training. This is because early expert policy targets π_{MCTS} will be copied directly. These will bias the apprentice's policy towards early bad play until these early targets are removed from the experience replay.

Taking this effect into account, we implemented a slowly increasing replay buffer, where the size of the dataset would begin small, and then slowly increase as more training generations elapsed. This allowed us to quickly phase out very early data before settling to our fixed window size. We start with a replay buffer which holds only the latest 5 generations of experiences. Every 5 generations, we increase the replay buffer capacity by 5 extra generations, until a seldom reached maximum of 200 generations. This yielded modest performance improvements.

Chapter 5

Metagame Autobalancing for Competitive Multiplayer Games

5.1 Introduction

This chapter looks at the lowest module of the MARL stack, the environment itself, that which defines both the task to solve and the interface by which agents can act to solve it. The focus of both Chapter 3 and Chapter 4 has been on how to improve upon the efficiency of MARL, primarily motivated by the problematic fact that even state-of-the-art MARL algorithms require a prohibitively large number of samples to be economically viable choices for the vast majority of game studios. This chapter shifts its focus, while keeping the thesis' overarching purpose of boosting the usage of MARL within the games industry. Instead of investigating how to approach the definition or manipulation of environments to further our understanding of the field of MARL or to improve upon its efficiency, we shall use MARL as a tool to aid the field of *game design*.

To elicit this shift in focus, we make an analogy with game theory. The game theoretical subfield of mechanism design [Mishra, 2008] asks the question: how do we design a game such that the optimal strategies followed by self-interested rational agents feature certain properties of interest? This question is often asked to create rulesets for scenarios like public work constructions or auctions, with the goal of shaping their corresponding optimal strategy for each agent to be truthful and collaborative, among other desirable properties. Similarly, assuming that we have access to MARL methods that will be able to approximate an optimal policy as a proxy for human play for any given video game environment, we ask the question: how can we automate game design choices regarding the game environments so that the policies resulting from reward-focused agents feature a gameplay style which respects designer intent?

The usage of machine learning for game design is an emerging field that is being used in very diverse settings [Yannakakis and Togelius, 2018]. These go beyond the use or creation of gameplaying agents, such as player retention and estimation of player churn[1] [Lee et al., 2018]. There is a growing interest in using AI in games for e-sports, with works such as [Katona et al., 2019] which provide tools to enhance professional casters with statistics

[1] Player churn: when players are about to quit the game. Knowledge about this can inform game design choices, for example, what game elements capture most player's attention.

about predicted upcoming in-game events. Content generation with modern paradigms like procedural content generation using machine learning [Summerville et al., 2018] has enjoyed numerous publications in recent years, although it still fails to see adoption within the industry. During game design the inherently sequential nature of RL allows this paradigm to create joint experiences between humans and AI designers [Shu et al., 2021]. At each step of the creation of a level, such as deciding what next asset to place in a game map, human actions can be augmented or fully replaced by suggestions from an RL designer-agent [Khalifa et al., 2020]. In this vein, the application of RL agents to automate Quality & Assurance has accrued a growing number of publications in recent years [Gordillo et al., 2021]. The main idea is to automate the playing of games, removing the human in the loop by RL agents. Both during and after training, these agents are deployed in the game to ensure certain quality conditions, like the game not crashing after arbitrary action sequences or ensuring certain areas of the game not being accessible. Because of the programmatic nature of these agents, it is also possible to gather a plethora of logs and other useful metrics to guide further game design [Bergdahl et al., 2020].

In this chapter, we tackle the challenge of *automated game balancing*. The balance of a game is understood as the relative strength of all of the game mechanics [Jaffe, 2013], which in practice is parameterized by a (normally very large) choice of game variables such as character healths, animation durations, attack damages and many other game-specific variables. Traditionally, this balancing act requires extensive periods of expert analysis, human play testing and debates among designers [Pfau et al., 2020]. The process of automating game balancing is to remove the human in the loop, and cast the process as an automated and quantitative optimization procedure.

In this final research chapter we present a tool for balancing multi-player games during game design. Section 5.2 further explores the appeal of automated game balancing. Some preliminary notation is introduced in Section 5.3 to understand the details of some of the concepts in the literature review of video game balancing in Section 5.4 and our proposed autobalancing algorithm in Section 5.5. Our approach requires a designer to construct an intuitive graphical representation of game balancing defined as an instantiation of a target metagame between defined strategies. The term metagame here essentially captures the same idea as in previous chapters of this thesis, it represents the relative scores that high-level strategies should experience upon normal rational play (i.e playing to win). This permits more sophisticated balance targets to be defined beyond the traditional requirement of equal win chances between all possible strategies (in the case of fully competitive games). Our proposed algorithm is a simulation-based optimization algorithm which automates game balancing by finding an appropriate game parameterization by casting this task as a graph distance minimization problem. An in depth explanation of the underlying workings of this tool is given in Section 5.6. We introduce a more complex fighting game in Section 5.7, used to showcase the capabilities of our algorithm, presenting experiments and their corresponding results in Sections 5.8 through 5.9. Finally, Section 5.10 concludes the chapter.

5.2 The Need for Automated Videogame Balancing

Achieving game balance is a primary concern for game designers yet, as previously mentioned, balancing games is a largely manual process of trial and error. This is especially problematic in asymmetric multiplayer games where perceived fairness has a drastic impact on the player experience. Changes to individual game elements or rules can have an impact on the balance between high-level strategies that depend on these. Unfortunately this impact is unknown before changes are made and can only be guessed at by designers through experience and intuition. In other words, for any interesting game, there are non trivial relationships between the parameters of a game and the ensuing game balance. As an example, imagine a shooter game where a weapon requires 3 successful shots to kill a target. We can modify the damage of the weapon, but as long as the number of successful shots required to kill that target is still 3, then any changes within this range will have little to no impact in the game's balancing, as long as such balancing takes this weapon's killing capabilities into consideration. However, moving the damage over a threshold that permits killing a target after just 2 shots can cause tremendous changes in the game's balance.

We term this balance between emergent high-level strategies the *metagame balance*. While in-house tools can be built for the adjustment and authoring of individual game elements, there are no general high level tools for videogame balancing in a game agnostic fashion. That is to say there are no tools which can programmatically adjust game elements to reach a certain, well defined, notion of game balance that are not purpose-built for a single game. There are publications which slowly lead up to this ideal, and there exist tools which can evaluate a metagame's balance [Jaffe et al., 2012a, Tomašev et al., 2020], a crucial yet incomplete part of a solution to this problem. An alternative approach to the discovery of metagame changes that arise from game changes is through data analytics. Large scale multiplayer titles that have access to large quantities of player data can use a variety of techniques to make judgements about the state of the metagame and provide designers with insight into future adjustments [Lee and Ramler, 2015]. There are, however, several problems with this approach. Analytics can only discover balance issues in content that is live, and by that point balance issues may have already negatively impacted the player experience: this is a reactive approach and not a preventive one. Worse, games which do not have access to large volumes of player data, such as less popular games, cannot use this technique at all. Furthermore, the process of data analytics itself is not typically within the skill-set of game designers. It is common for studios that run multiplayer games to hire data scientists to fill this need. This, in combination with the trial and error nature of the balance process, results in increased costs, becoming as a bottleneck for the development of new content [Jacob et al., 2020]. With current trends indicating a systematic increase in the cost of game development [Shaker et al., 2016], the complexity of game balancing tasks is also expected to increase. The current bottleneck in this process is human play testing, as they are slow and many testers are required for long play-sessions, which only grow longer with more complex games. Human play testing does not scale.

5.3 Preliminary Notation

Here we introduce chapter specific notation. Some of these concepts were not covered in Chapter 2, while others we re-introduce through a different lens, to aid the understanding of our proposed method in Section 5.5 and to contrast it with the existing literature.

5.3.1 Game Parameterization

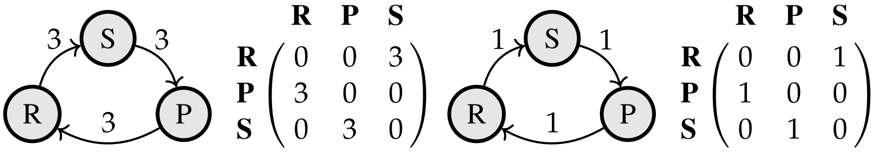
Every video game presents a (potentially very large) number of values that characterize the game experience, which we shall refer to as *game parameters*. These values can be numerical (such as gravitational strength, movement speed, health) or categorical (whether friendly fire is activated, to which team a character belongs). As a designer, choosing a good set of parameters can be the difference between an excellent game and an unplayable one. We let $E_{\boldsymbol{\theta}}$ denote a game environment parameterized by an n-sized parameter vector $\boldsymbol{\theta} \in \{\Pi_{i \leq n} \Theta_i\}$, where $\{\Pi_{i \leq n} \Theta_i\}$ represents the joint parameter space, and Θ_i the individual space of possible values for the ith parameter $\theta_i \in \boldsymbol{\theta}$.

5.3.2 Metagames

What a *metagame* is can mean different things to different players, with definitions often being game specific. For example in deck-building games such as Hearthstone, the "meta" is usually interpreted to indicate which decks are currently popular or especially strong; while in EVE Online an important part of the "meta" is player diplomatic alliances, as well as which ship types are good against which others. Regardless of the videogame title, they all share a comparative notion that contrasts an idea of value between elements in the game. See [Carter et al., 2012] for a good discussion of this notation.

We shall keep using the meaning we have been using until now, defining a metagame as the interplay between a set of high-level strategies that are abstracted from the atomic game actions [Omidshafiei et al., 2019, Balduzzi et al., 2018]. Reasoning about a game involves thinking about how each individual action will affect the outcome of the game. In contrast, a metagame considers more general terms, such as how an aggressive strategy will fare against a defensive one. In metagames, high level strategies are considered instead of primitive game actions. For example, consider a card game like Poker. Reasoning about a Poker metagame can mean, as a non-exhasutive example, reasoning about how bluff-oriented strategies will deal against risk-averse strategies.

The level of abstraction represented in a metagame is defined by the metagame designer, and the same game can allow for a multitude of different levels of abstraction. For instance, in the digital card game of Hearthstone, meta-strategies may correspond to playing different deck types, or whether to play more offensively or defensively within the same deck. A game designer may want to ensure that no one deck type dominates, but be happy that a particular deck can only win if played offensively.



$$\begin{array}{c|ccc} & \mathbf{R} & \mathbf{P} & \mathbf{S} \\ \hline \mathbf{R} & 0 & 0 & 3 \\ \mathbf{P} & 3 & 0 & 0 \\ \mathbf{S} & 0 & 3 & 0 \end{array} \qquad \begin{array}{c|ccc} & \mathbf{R} & \mathbf{P} & \mathbf{S} \\ \hline \mathbf{R} & 0 & 0 & 1 \\ \mathbf{P} & 1 & 0 & 0 \\ \mathbf{S} & 0 & 1 & 0 \end{array}$$

(a) G_1 (b) Matrix for G_1 (c) G_2 (d) Matrix for G_2

Figure 5.1: Two graphs, $\boldsymbol{G_1}$ and $\boldsymbol{G_2}$ represented both as adjacency matrices and in graphical form. The absolute average distance between both graphs of 3 nodes is: $d(\boldsymbol{G_1}, \boldsymbol{G_2}) = \sum_i \sum_j \frac{|g^1_{ij} - g^2_{ij}|}{3} = 2$.

5.3.3 Empirical Response Graphs

A *directed weighted graph* of $v \in \mathbb{N}^+$ nodes can be denoted by an adjacency matrix $\boldsymbol{G} \in \mathbb{R}^{v \times v}$. Each row i in $\boldsymbol{G}$ signifies the weight of all of the directed edges stemming from node i. Thus, $g_{i,j} \in \mathbb{R}^+$ corresponds to the weight of the edge connecting node i to node j, where $1 \leq i, j \leq v$.

Given an evaluation matrix $\boldsymbol{A_\pi} \in \mathbb{R}^{n \times n}$ computed from a set of strategies (or agents) $\boldsymbol{\pi}$, let its *response graph* [Wellman, 2006] represent the dynamics [Omidshafiei et al., 2019, Liu et al., 2021] between agents in $\boldsymbol{\pi}$. That is, a representation of which strategies (or agents) perform favourably against which other strategies in $\boldsymbol{\pi}$. In a competitive scenario, a response graph shows which strategies win against which others. As a graph, each strategy i is represented by a node. An edge connecting node i to node j indicates that i dominates j. The weight of the edge is a quantitative metric of how favourably strategy i performs against j. Figure 5.2a shows a response graph for the game of Rock-Paper-Scissors (RPS).

A response graph can be readily computed from an evaluation matrix. Each row i in an evaluation matrix $\boldsymbol{A}$ denotes which strategies i both wins and loses against, the former being indicated by positive entries and the later by negative ones. Therefore, generating a response graph $\boldsymbol{G}$ from an evaluation matrix $\boldsymbol{A}$ is as simple as setting all negative entries of $\boldsymbol{A}$ to 0 such that, for instance, $\boldsymbol{A} = \left(\begin{smallmatrix} 1 & -2 \\ 2 & -1 \end{smallmatrix}\right)$, becomes $\boldsymbol{G} = \left(\begin{smallmatrix} 1 & 0 \\ 2 & 0 \end{smallmatrix}\right)$.

5.3.4 Graph Distance

There is a rich literature on measuring distance between graphs [Gao et al., ,Robles-Kelly and Hancock, 2003]. We concern ourselves here with a basic case. We are interested in measuring the distance between two graphs which share the same number of nodes, $\boldsymbol{G_1}, \boldsymbol{G_2} \in \mathbb{R}^{v \times v}$, and differ *only* in the weight of the edges connecting the graph nodes. Because graphs can be represented as matrices, we look at differences between matrices. We denote the distance between two graphs $\boldsymbol{G^1}$ and $\boldsymbol{G^2}$ by $d(\boldsymbol{G^1}, \boldsymbol{G^2}) \in \mathbb{R}$. Equation 5.1 represents the average absolute edge difference and Equation 5.2 represents the mean squared difference (MSE). We decided to use the latter because its squaring factor yields larger difference values when there are outliers (i.e large differences in edge values), yet preliminary results showed no empirical difference between both equations. Thus, we report only the results where MSE (Equation 5.2) was used.

$$\frac{\sum_{i,j} |g^1_{ij} - g^2_{ij}|}{n} \tag{5.1}$$

$$\frac{\sum_{i,j} (g^1_{ij} - g^2_{ij})^2}{n} \tag{5.2}$$

5.4 Related work

Quantitative methods for understanding games have been proposed in many forms. [Beyer et al., 2016] presents a similar balancing overview to ours, introducing a generic iterative an automated balancing process. It works by sampling game parameters, using AI players (or real humans) to test if a desirable balancing has been achieved. Our main differentiating contribution are balancing graphs as a designer friendly balancing description. Several strategies for the assessment of games without real player data are described in [Nelson, 2011]. Our research most closely resembles the strategy defined as "Hypothetical player-testing", in which we are "trying to characterize only how a game operates with a particular player model" [Nelson, 2011]. The forms that this type of hypothetical player-testing analysis can take are discussed in depth in [Jaffe, 2013], and specifically our work is concerned with the subcategory of quantitative analysis defined as Automated Analysis, helping designers evaluate (and often modify) their games without human playtests [Jaffe, 2013].

Machine Learning offers tools for automatic metagame analysis. Harnessing existing supervised learning algorithms, [Argue, 2014] used random forests and different neural network architectures to assess metagame balance by predicting the outcomes of individual matches using hand-crafted features that describe the strategies being used. The authors make an assessment of the overall balance of the metagame by measuring “the prevalence of parallel strategies" [Argue, 2014], assuming balance to mean equal prevalence. While this is informative, it is predicated on a definition of balance that may not align with the goals of other game designers working on other projects, which may have definitions for balance that extend beyond prevalence. Additionally, such techniques are limited to assessing the current balance of a game context rather than providing a solution for balance issues that are discovered.

MCTS has been used for this type of simulation based analysis in the past, in [Zook et al., 2015] MCTS agents are used to model players at various skill levels in Scrabble and Cardonomicon to extract metrics that describe game balance. However, this type of analysis is concerned with the discovery of issues and takes no steps towards providing a solution to a balance problem once discovered.

The work by Liu and Marschner [Liu and Marschner, 2017] uses the Sinkhorn-Knopp algorithm to balance a mathematical model, according to game theoretical constructs, representing a simplified version of the popular game Pokemon. In Pokemon, each pokemon type[2] has advantages and disadvantages against various other types. The authors tune these type features to make them all equally viable pokemon types. This approach concerns itself with mathematical comparisons between strategies based on an existing table of matchup statistics, which may not exist for most games, especially those still in development.

[2] An overview of Pokemon types: `https://bulbapedia.bulbagarden.net/wiki/Type`

Leigh et al. [Leigh et al., 2008] used co-evolution to evolve optimal strategies for CaST, a capture the flag game. Populations of agents were evolved in an environment with a set of game parameters. The distribution of the resulting agents across simplex heat maps of different strategies was used to assess whether or not the game was balanced with those game parameters by considering balance to be a situation where any core strategy should beat one of the other core strategies and lose against another, similar to our definition of *cyclic* balancing used throughout the thesis and in [Balduzzi et al., 2018, Balduzzi et al., 2019]. They manually modified play parameters and iterated to find a configuration with a desirable heatmap. Our approach builds upon this work by automating the manual parameter adjustment stage, it also broadens the definition of balance by allowing the designer to specify a target metagame balance, instead of simply a fully cyclic metagame state.

There are multiple balancing studies on the popular game of Hearthstone[3], mostly using evolutionary algorithms. Some try to find the space of possible strategies within the current game balance of the game [Fontaine et al., 2019]. Other studies [de Mesentier Silva et al., 2019] use an evolutionary algorithm to search for a combination of changes to card attributes that create a hearthstone metagame where every deck has a 50% winrate. Our work can be considered to be a generalization of this approach, which is not limited to Hearthstone and does not prescribe the precise conditions under which a game can be considered balanced.

More general methods to evaluate balance and inform designers exist in the form of the restricted play work. The authors of [Jaffe et al., 2012b] use agents with specific gameplay restrictions, such as limitations on the specific moves that can be made by an agent, to provide insight into specific balance questions such as "How important is playing unpredictably?". This work highlights the need for designer input in the analysis process, as many definitions of balance are limited to assessments of similarity between strategies. These methods are well suited to use in conjunction with ours and could allow a designer to specify metrics beyond winrate in the target balance graph, such as the frequency of specific moves.

It seems that the common factor between most current efforts on autobalancing are that they are tied to game specific metrics and that they lack the flexibility to define different descriptions of what it means for a game to be balanced. Our method, described in the following section, solves both these issues.

5.5 Autobalancing

In this section we present our autobalancing algorithm in its most general form, together with its pseudocode in Algorithm 18. It also presents descriptions of its modular components, their interplay and brief suggestions for its use.

[3] https://playhearthstone.com/en-us

5.5.1 Optimization Setup

Let $E_{\boldsymbol{\theta}}$ be a game environment parameterized by vector $\boldsymbol{\theta} \in \mathbb{R}^n$, whose possible values are bound by vectors $\boldsymbol{\theta}^{min}$ and $\boldsymbol{\theta}^{max}$. Let $\boldsymbol{G_t}$ denote the target metagame response graph presented by a game designer for game $E_{\boldsymbol{\theta}}$. Let $\boldsymbol{G_\theta}$ represent the empirical metagame response graph for game $E_{\boldsymbol{\theta}}$ derived from arbitrarily many game simulations, using a set of game-playing agents $\boldsymbol{\pi}$. This set might be given in advance or trained via MARL methods or other approaches. The nodes in $\boldsymbol{G_\theta}$ corresponds to elements in the metagame and the edges correspond to the realized metagame balance between these elements. Finally, let $\mathcal{L}(\cdot,\cdot)$ represent a cost or distance function between two graphs.

The mathematical formulation for finding a parameter vector $\boldsymbol{\theta}$ that yields a metagame for a game environment $E_{\boldsymbol{\theta}}$ respecting designer choice $\boldsymbol{G_t}$ is a non-linear optimization problem:

$$\underset{\theta}{\arg\min} \quad \mathcal{L}(\boldsymbol{G_\theta}, \boldsymbol{G_t}) \tag{5.3}$$

$$s.t \quad \theta_i^{min} \leq \theta_i \leq \theta_i^{max} \; \forall i \in \{|\boldsymbol{\theta}|\} \tag{5.4}$$

Algorithm 18: Automated Balancing Algorithm

```
Input: Target designer metagame response graph: $\boldsymbol{G_t}$
Input: Ranges for each parameter: $\boldsymbol{\theta}^{min}, \boldsymbol{\theta}^{max}$
Input: Convergence threshold: $\epsilon$
Input: Initial game parameterization: $\boldsymbol{\theta_0}$ ;      // If unspecified, randomize
Initialize best estimate $\boldsymbol{\theta}_{best}, \mathcal{L}_{best} = \boldsymbol{\theta_0}, \infty$;
Initialize observed datapoints $D = [\,]$;
repeat
    Train agents $\boldsymbol{\pi}$ inside $E_{\boldsymbol{\theta}_t}$ (if $\boldsymbol{\pi}$ not given);
    Construct evaluation matrix $\boldsymbol{A}_{\boldsymbol{\theta}_t}$ from $\boldsymbol{\pi}$;
    Generate response graph $\boldsymbol{G}_{\boldsymbol{\theta}_t}$;
    Compute graph distance $d_t = \mathcal{L}(\boldsymbol{G}_{\boldsymbol{\theta}_t}, \boldsymbol{G_t})$;
    Add new datapoint $D = D \cup (\boldsymbol{\theta_t}, d_t)$;
    if $d_t < \mathcal{L}_{best}$ then
        Update best estimate $\boldsymbol{\theta}_{best}, \mathcal{L}_{best} = \boldsymbol{\theta_t}, d_t$;
    end
    $\boldsymbol{\theta_{t+1}} = update(\boldsymbol{\theta_t}, D)$;
until $\mathcal{L}(\boldsymbol{G}_{\boldsymbol{\theta}_t}, \boldsymbol{G_t}) < \epsilon$;
return $\boldsymbol{\theta}_{best}$;
```

There are four notes to be made about our algorithm:

1. **It can be dynamically parallelized**: multiple parameter vectors can be evaluated simultaneously. This plays nicely with RL methodologies, which can scale horizontally[4]. Furthermore, because each set of game parameters $\boldsymbol{\theta}$ can independently be evaluated, computational resources can be added or removed from the optimization

[4] To scale horizontally means to scale with the number of CPUs, instead of vertically, which means scaling with respect to the strength of each individual CPU.

procedure at will without disrupting the overall process. It could be useful, for instance, to increase the compute of the optimization process overnight, when workers are not actively using their computers, and de-scale the compute in the morning, when the resources are needed for other kind of tasks. Or if the studio has access to servers used to host player matches, during periods of low player count, some of the free resources could be used for autobalancing purposes for future content.

2. **It allows for initial designer choice**: such that designers can designate an initial parameter vector and a prior over the search space, which can lead to speedup in the convergence of the algorithm. Often time the definition of this prior is constrained by the choice of optimizer. Modern optimizers will often allow to choose different distributions over the parameter space for each parameter, or allow for the definition of the search space to be over a discrete set of values.

3. **An arbitrary subset of the game parameters can be fixed**: $\boldsymbol{\theta}$ can represent a subset of the entire game parameters, allowing some parameters to be completely left out from the optimization algorithm by having fixed values. Furthermore, game designers might want to only explore subset of values within a specific parameter, thus making the optimization problem a constrained one. This is important if there are certain core aspects of a game that the designer does not want to be altered throughout the automated game balancing.

4. **Deterministic results are not guaranteed**. There are three potential sources of stochasticity, the game dynamics $E_{\boldsymbol{\theta}}$, the agent policies $\boldsymbol{\pi}$ and the optimizer's underlying logic for deciding on new parameter candidates (line 12 of Algorithm 18). This hurts reproducibility, as starting the balancing algorithm from the same initial conditions can yield different results. We argue that this is not a drastic problem. Firstly, running the procedure multiple times can lead the optimizer to different local optima, resulting in different parameters that reach a similar balancing, thus elucidating the designers about various ways in which their target balancing can be obtained. Secondly, product oriented practitioners might be more interested in the final set of found parameters rather than on making the procedure deterministic to allow it to be contrasted against other method oriented practitioners.

There are three potential bottlenecks in Algorithm 1 in terms of the computational requirements of (1) the update of the parameter vector by the optimizer (line 12 of Algorithm 18) (2) the construction of the evaluation matrix $A_{\boldsymbol{\theta}_t}$ (line 5 of Algorithm 18) and (3) training game-playing policies (line 4 of Algorithm 18):

1. If the set of parameters is too large, certain optimizers can take a long time to decide on the next game parameterization to evaluate, with the computational complexity potentially increasing with the number of attempted trials, as is the case with Bayesian optimizers.

2. Computing each entry in an evaluation matrix $a_{ij} \in \boldsymbol{A}_{\boldsymbol{\theta}_t}$ requires running many game episodes in which metrics for nodes i, j and their corresponding edges are captured, with the cost of computing $\boldsymbol{A}_{\boldsymbol{\theta}_t}$ growing with respect to the number of nodes in the metagame balancing graph. If the nodes represent different gameplaying strategies and the edges the relative winrates between them, then the computational cost grows exponentially with respect to the number of nodes.

3. Training AI agents for games can be a difficult and very computationally expensive task, as we have seen through the course of this thesis. We acknowledge that this is a major engineering effort and a current bottleneck for any minimally complex game of interest. We will not focus on this problem for the remaining on this chapter and shall assume that we have access to such game-playing agents on demand.

This latter issue is the same issue that has appeared multiple times in this thesis when computing any metagame metric derived from empirical evaluation matrices. Amongst other uses, Chapter 3 used these metagames to decide the cross-population ranking of agents among different populations using relative population performance in Section 3.9. Chapter 4 used them to ensure that the training of test agents to be used against our BRExIt algorithm did showcase a positive correlation between training time and playing strength using Nash averaging in Section 4.5.1. In this final chapter, we use the same principle to quantitatively ascertain the balancing of a given game. We want to highlight both that these relatively recent concepts of empirical evaluation matrices are profoundly useful for a vast variety of tasks, and that further work on reducing their workload will noticeably benefit the fields that use them.

5.5.2 Choosing an Optimizer

We want to emphasize that our algorithm can use any black-box optimization method. We experimented with two Bayesian optimizers to compute updates to our parameter vector $\boldsymbol{\theta}$. Initially we used the algorithm Tree-structured Parzen Estimator [Bergstra et al., 2011], as implemented in the Python framework Optuna [Akiba et al., 2019], later we used Gaussian processes with a Matern kernel [Williams and Rasmussen, 2006] as implemented in Scikit-learn [Pedregosa et al., 2011]. However, these could be replaced with any other optimization method.

Most commonly in the literature of automated game balancing, evolutionary algorithms have been used [Morosan and Poli, 2017], where a population corresponding to a set of different game parameterizations iteratively goes back and forth between evaluating each parameterization according to a game specific fitness function for how balanced the game is, and replacing the least fit members of the population with (hopefully) better ones derived from the best performing parameters. We chose to move away from these algorithms in favour of non-parametric Bayesian methods so as to avoid the potentially issue of hyperparameter tuning.

5.5.3 Choosing a Metagame Abstraction

For most games, there are many possible levels of abstraction available when deciding what the metagame captured by the target response graph represents. Choosing an abstraction level to balance can be a difficult task, but we argue that reasoning about different metagames at varying abstraction levels is a necessary task in balancing any multi-agent game, and one that comes naturally both to game designers and seasoned players alike. On a positive note, the fact that metagames can be represented at many levels of abstraction grants our method the versatility to generalize to various stages of balancing. That is to say, our method can be used at different points of game development to balance different aspects of the game.

In practice, to choose an abstraction level means to choose the meaning of the nodes and the edges in a balancing graph as well as the game parameters that will affect it. Generally each node on the response graph represents a specific strategy, unit or game-style. We shall now explore this idea via a few examples:

In a competitive RPG[5] each node of the response graph might represent a character class: Paladin, Wizard, Rogue... with edges corresponding to the winrate in a 1v1 fight between two nodes and the parameters being class specific values. In a related cooperative setting, each node can correspond to a team composition of the aforementioned classes, with the edges corresponding to the extra time duration a team will take to complete a dungeon compared to a baseline team. In this case the parameters can be the health and damage of all enemies in the dungeon, as the designers might not be interested in tweaking class specific values if they are already balanced for another metagame abstraction like the previous example. At a lower level, in games with large open areas like PUBG[6] or Fortnite[7] each node might represent an individual weapon. The edges can denote the average distance at which a player with one weapon kills a player with another, with the tunable parameters being weapon related: damage, bullet spread and a discrete set of peephole styles. This can ensure that aesthetic choices, like a weapon looking like a sniper rifle or a small pistol, reflect the gameplay style that players expect (i.e the sniper rifle killing players from a longer distance than any other weapon).

It quickly becomes apparent to the reader that there is a vast set of available metagame abstractions even for simple games. Choosing the right set of metagames to balance to ensure a good player experience is crucial. With our algorithm we allow the designer to focus on this higher level question, while the search for the game parameters themselves which will realize such balancings being taken care of by our algorithm.

5.5.4 Generating Gameplaying Agents

As we have seen multiple times throughout the course of this thesis, in order to compute an evaluation matrix, such as A_{θ_t} for a given game E_θ we require a set of gameplaying agents π to simulate play against each other. The algorithmic choice for how to train these agents is orthogonal to the use of our method. However, we acknowledge that the creation

[5] https://en.wikipedia.org/wiki/Role-playing_game

[6] https://www.pubgmobile.com/fr/home.shtml

[7] https://www.epicgames.com/fortnite/en-US/home

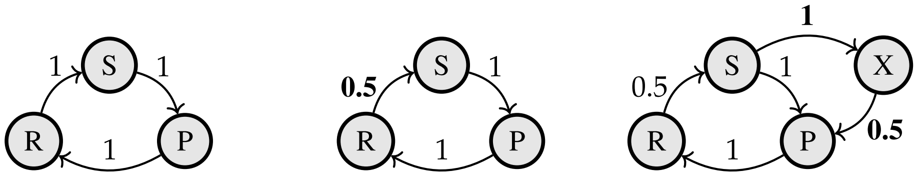
of these agents is a significant engineering and technical effort for any but the most trivial of games. Regarding the generation of these agents, they could be hand-crafted heuristic agents, agents trained via reinforcement learning, agents trained via evolutionary algorithms, planning based agents using MCTS or other alternatives [Yannakakis and Togelius, 2018]. At the time of writing there is no right choice or clear winner amongst these methods. For a game studio the best choice should be decided from a combination of in-house expertise on specific methodologies and technical constraints imposed by the game itself, such as having access to a fast simulator for the game environment to allow for model-based methods.

Another important factor to take into account is that of the *expected behaviour* of the game playing agents. What we mean by this is that different techniques will generate different kinds of agents, even if there are significant local variation amongst algorithms from the same field. As mentioned in Chapter 4 in Section 4.4.4, RL agents tend to behave reactively whereas model-based methods tend to behave proactively with respect to their computed internal plans. The choice of what algorithm to use for the generation of playing agents as part of our autobalancing procedure depends on the desired expected behaviour wanted by the game designers.

5.6 Motivating Examples

In this section we cement the intuition and understanding of our proposed autobalancing algorithm by detailing its workings on a set of trivial examples.

For simplicity, we assume that all parameters in the following examples are bound between $[-1, +1]$. As a graph distance metric we use $\mathcal{L}(\cdot, \cdot) = MSE(\cdot, \cdot)$ from Equation 5.2. Furthermore, we shall remove non-linear relationships from our balancings by treating the metagame balance itself as the game parameters $\boldsymbol{\theta}$ for a given game environment $E_{\boldsymbol{\theta}}$. Thus, once we have decided on a candidate game parameterization $\boldsymbol{\theta}'$, we don't need to (1) train agents (skipping line 4 of Algorithm 18 and (2) simulate trajectories from which to derive any further metrics to compute our metagame (trivializing line 5 of Algorithm 18). This is because our set of parameters already defines the metagame $\boldsymbol{A}_{\boldsymbol{\theta}'}$, from which the graph distance to the target game balance can be computed directly. This greatly simplifies our examples without loss of generality.



(a) RPS (b) Biased RPS (c) Extended RPS

Figure 5.2: Target graphs for the three motivational examples.

5.6.1 Rock Paper Scissors

Imagine we want to create the game of Rock Paper Scissors. As a designer choice, we want to allow for three actions, Rock (R), Paper (P) and Scissors (S). We would like for paper to beat rock, rock to beat scissors and scissors to beat paper, with mirror actions negating each other. We would also like all interactions to be equally strong in terms of some scoring function, meaning that the score obtained when paper beats rock should be as large as when rock beats scissors, etc. Such strategic balancing is captured in Figure 5.2a. These interactions can be represented as a 2-player, symmetric, zero-sum normal form game $\boldsymbol{E}_{\boldsymbol{\theta}}^{RPS}$, parameterized by $\boldsymbol{\theta} = [\theta_{rp}, \theta_{rs}, \theta_{ps}]$. Where θ_{rp} denotes the payoff for player 1 when playing Rock against Paper, θ_{rs} when playing Rock against Scissors and θ_{ps} when playing Paper against Scissors. The normal form parameterized version of RPS is captured in Equation 5.5. We ask the question: Which parameter vector $\boldsymbol{\theta}$ would yield a game $\boldsymbol{E}_{\boldsymbol{\theta}}^{RPS}$ balanced as in Figure 5.2a?.

$$\boldsymbol{E}_{\boldsymbol{\theta}}^{RPS} = \left[\begin{array}{cccc} & \mathbf{R} & \mathbf{P} & \mathbf{S} \\ \mathbf{R} & 0 & \theta_{rp} & \theta_{rs} \\ \mathbf{P} & -\theta_{rp} & 0 & \theta_{ps} \\ \mathbf{S} & -\theta_{rs} & -\theta_{ps} & 0 \end{array}\right] \tag{5.5}$$

We begin by assuming the target balance response graph $\boldsymbol{G_t}$ from Figure 5.2a is given by a game designer without specifying any initial values, as they might lack any informed priors. Thus, we start by sampling a random parameter vector within the permitted space for each parameter $[-1, +1]$, say, $\boldsymbol{\theta_0} = [\theta_{rp} : -1, \theta_{rs} : 1, \theta_{ps} : 0]$. On a normal scenario, we would now proceed to generate game-playing agents for $E_{\boldsymbol{\theta}_0}^{RPS}$ and have them play against each other so that we can derive the resulting metagame $\boldsymbol{A}_{\boldsymbol{\theta}_0}$ from simulated trajectories. Fortunately, our simplifying assumption allows us to observe the metagame directly, yielding $\boldsymbol{A}_{\boldsymbol{\theta}_0} = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}$, whose response graph $\boldsymbol{G}_{\boldsymbol{\theta}_0} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ is depicted in Figure 5.3. We then compute the distance between $\boldsymbol{G}_{\boldsymbol{\theta}_0}$ and $\boldsymbol{G_t}$, $d_{\boldsymbol{\theta}_0} = \mathcal{L}(\boldsymbol{G}_{\boldsymbol{\theta}_t}, \boldsymbol{G_t}) = 0.25$. Using this new datapoint $(\boldsymbol{\theta_0}, d_{\boldsymbol{\theta}_0})$ we update our black box optimization model, which in our case is a Bayesian method, and sample a new candidate parameter vector $\boldsymbol{\theta_1}$. This process is looped until convergence to a predefined threshold value, or when an arbitrary computational budget is spent.

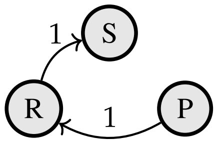
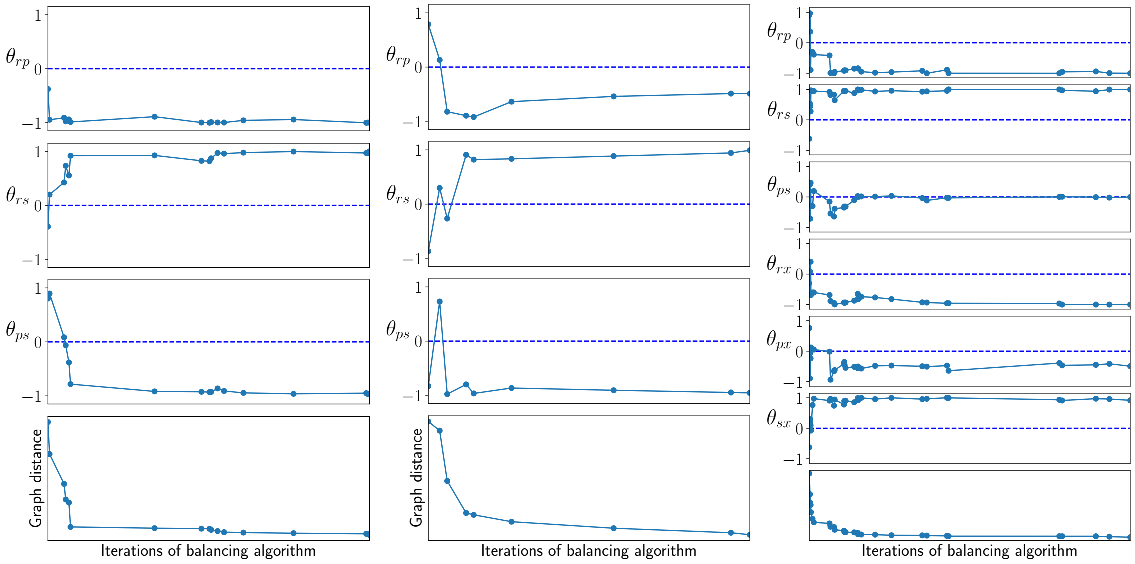


Figure 5.3

Figure 5.4a shows the progression of the values of the parameter vector $\boldsymbol{\theta}$ throughout the optimization process for the target balancing described in Figure 5.2a. As we can see, the parameters found during the optimization process converge towards the true target values: $\boldsymbol{\theta}_{final}^{RPS} = [\theta_{rp} : -1, \theta_{rs} : 1, \theta_{ps} : -1]$.

5.6.2 Biased Rock Paper Scissors

Consider another version of Rock Paper Scissors where we want to *weaken* the strength of playing Rock, as denoted in Figure 5.2b with an edge value of 0.5. This could be of interest for designers in scenarios where they want to discourage the use of a specific strategy at a certain point in the game. For our algorithm, this simply amounts to discovering a lower payoff θ_{rp} obtained by playing Rock against Scissors. The optimization process follows the same steps as described earlier, with Figure 5.4b showing the progression of the optimization process converging towards the target values: $\boldsymbol{\theta}^{bias}_{final} = [\theta_{rp} : -1, \theta_{rs} : 0.5, \theta_{ps} : -1]$.



(a) RPS.

(b) Biased RPS.

(c) Extended RPS.

Figure 5.4: Progression of the best candidate set of parameters given by the optimization algorithm for the three motivational examples (a-c). Only candidates parameter vectors which improve upon the last previous best candidate are plotted.

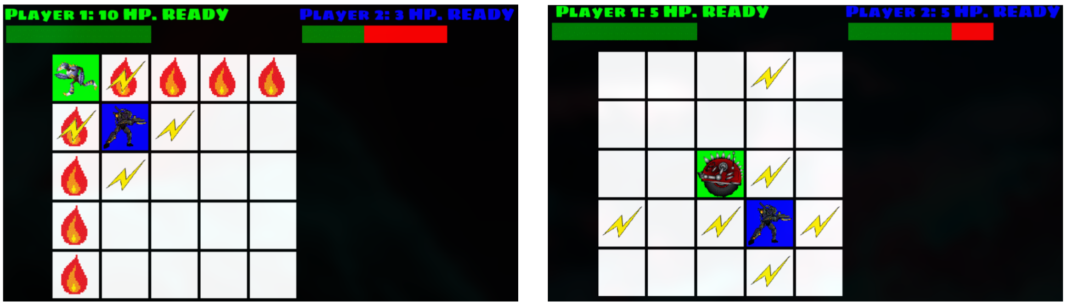
5.6.3 Extended Rock Paper Scissors

Our final toy example showcases an *extended* version of RPS, with an extra **X** action which only interacts by being beaten by scissors and weakly winning (again, an interaction valued by 0.5) against paper, as showed in Figure 5.2c. For this case, we would simply extend the matrix $E^{RPS}_{\boldsymbol{\theta}}$ with an extra row and column to denote the new potential edge interactions with the new node **X**, which entails adding three new parameters: $\theta_{rx}, \theta_{sx}, \theta_{px}$. From this new parameter vector, we can run our algorithm to find the right parameterization. We formulate this example to show that adding new nodes into the balancing graphs, while increasing the necessary compute required by the autobalancing algorithm, does not increase the complexity of its use by practitioners. Figure 5.4c shows the progression of candidate parameters found by the optimization process, which end up converging towards the parameters that yield the target balance: $\boldsymbol{\theta}^{extended}_{final} = [\theta_{rp} : -1, \theta_{rs} : 1, \theta_{ps} : -1, \theta_{rx} : 0, \theta_{sx} : 1, \theta_{px} : -0.5]$.

5.7 Usage on a Real Game

The parameters optimized in the previous section directly influenced the metagame matrix, which is not a realistic scenario for the type of game environments encountered by game designers. The game parameters that designers can directly change impact game mechanics, which only indirectly affect the outcome of a game. Therefore, for the remainder of this section, we consider the perspective of the game designer, to showcase the use of our algorithm in a realistic challenge on a game of our own making.

5.7.1 Workshop Warfare: a more Realistic Game



(a) Torch bot vs Nail bot. Both of them have activated their special action, throwing flames and shooting bolts respectively.

(b) Saw bot vs Nail bot. Saw bot is damaging Nail bot by being adjacent to it, while the latter is using its special ability.

Figure 5.5: Screenshots of the game.

Workshop Warfare[8] is a 2-player, zero-sum, symmetric, turn based, simultaneous action game. The theme of the game is a 1v1 battle between robots on a 5x5 2D grid with the objective of depleting the opponent's health[9]. Each player chooses 1 out of 3 available robots (Figure 5.6) to fight the opponent's robot of choice, with each robot featuring a different style of play. All robots feature the same action space: standing still (S), moving up (U), down (D), left (L), right (R) and a special action (A) for a total of 6 actions. The special action (A) varies per robot and will be explained later.

(a) Nail Bot

(b) Saw Bot

(c) Torch Bot

Figure 5.6: Eligible characters in Workshop Warfare.

Workshop Warfare works on a "tick" basis, which represents the temporal dimension of the game. Each bot has an associated number of ticks shared across all actions, representing how many in-game ticks must elapse before the bot can act again. This property can be thought as a time cost or robot "speed". Actions take place immediately upon being taken. A bot is said to be "sleeping" during the period that it cannot take actions. Standing still (S) has no cost, meaning that if a bot stands still, it remains in the same location while delaying

[8] The game is open source, and follows an OpenAI Gym interface [Brockman et al., 2016]: https://github.com/Danielhp95/GGJ-2020-cool-game

[9] Akin to TV shows like Battle Bots https://www.wikiwand.com/en/BattleBots

its action to the next tick. This allows for a degree of strategic depth, as one can time certain actions to happen at specific points in reaction to the opponent's moves.

There are no time restrictions placed upon the players at the time of selecting an action. This makes it amenable for forward planning methods that use a given computational budget to decide on what action to take. Thus, when autobalancing, this budget can be scaled without affecting the flow of the game, this is further explained in Section 5.8.3.

To clarify the tick based system, we describe an example scenario in Figure 5.7. Imagine a match with two bots, with a speed of 2 and 4 ticks respectively and assuming that both bots know about each other's speeds. On tick 0 they both select the move upwards (U) action, moving upwards by 1 square in the grid. The next tick will elapse without anything happening, as both bots are sleeping. On tick 2, only bot 2 will be able to act again, where it decides to stand still (S), delaying its action until the next tick. Thinking that bot 1 might begin an offense, bot 2 begins moving away to the left (L) on tick 3, knowing that on the tick after bot 1's next action it will be able to react defensively. On tick 4 bot 1 decides to use its special action (A), which is always an offensive action. Bot 2 was right in its prediction about bot 1, and follows through with its evasive plan and continues its movement to the left (L) on tick 5.

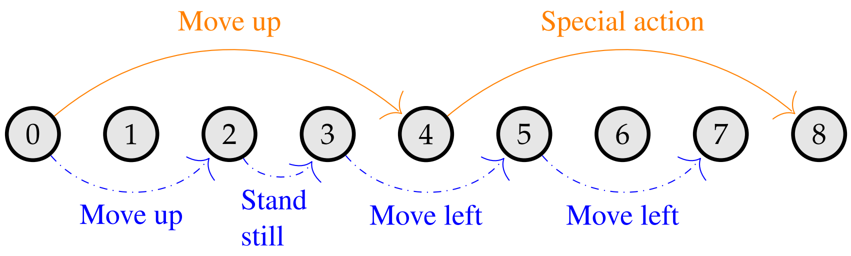


Figure 5.7: Example of the tick based system. Nodes correspond to the in-game tick counter. The edges correspond to actions taken by bots and their corresponding tick cost. Orange edges on top of the nodes denote actions from bot 1, and conversely bottom blue dashed edges represent actions for bot 2.

We now describe all three bots, whose in-game sprites are shown in Figure 5.6. *Torch bot* is equipped with a damaging blow torch, and can shoot a continuous beam of fire of limited range in all four directions for a short amount of time. *Nail bot* has a nail gun, and can shoot nails in all four directions at once. When fired, each nail travels in a fixed direction, at a speed of one grid cell per tick, independently of the bot's later movement and deals damage if they hit the opponent, disappearing upon contact with either a bot or the end of the grid. *Saw bot*'s spikes deal damage by being adjacent to the opponent. Its ability is to temporarily increase its damage output for a given number of in-game ticks.

5.7.2 Game Parameterization

The parameters of the game that we will allow to be modified by our algorithm are those corresponding to the playable characters, the bots. All bots share some common parameters, although their individual values can differ from bot to bot, while other parameters are bot specific and relate to a bot's special action (A). These are:

Common parameters

- **Health**: Damage a bot can sustain before being destroyed.
- **Cooldown**: After the special action (A) is activated, number of ticks that need to elapse before that action can be used again.
- **Damage**: Damage dealt by flames (Torch bot), nails (Nail bot) or spikes (Saw bot).
- **Ticks between moves**: Number of ticks that need to elapse before another action can be taken.

Bot-specific parameters

- **Torch range**: Length of the blow torch flame, in number of grid tiles (Torch bot).
- **Torch duration**: number of ticks the torch flame is active (Torch bot).
- **Saw damage buff**: Temporary change in damage dealt (Saw bot).
- **Saw buff duration**: Duration of buff to damage (Saw bot).

This concludes the explanation of the game we will be using in our following experiments. We now proceed to use our autobalancing method to balance a metagame according to different designer intents purely from their graphical description.

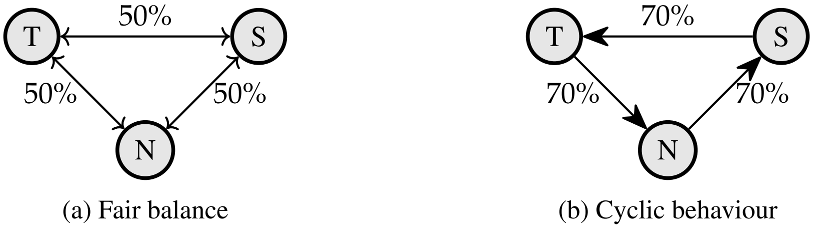
5.8 Experiments on Real Game

We first choose our level of abstraction, and what elements we want to balance as game designers. We choose the metagame to represent the *winrates* between all bot matchups. We want these winrates to represent the winrate between rational competitive players, that is, players who play to win understanding that their opponent also aims for the same goal. As a proxy of rational players we use AI agents controlled by MCTS, as detailed below in section 5.8.3.

We note that the parameters that were optimized in the motivational examples in Section 5.6 were real valued ($\mathbb{R}$), whereas all the parameters in this section are natural numbers ($\mathbb{N}$). Even so, the number of parameter combinations of Workshop Warfare's parameter space would be prohibitively time consuming for any human designers to manually explore thoroughly.

5.8.1 Target Metagame Balance

We will consider two different metagame targets as described in Figure 5.8, each corresponding to a different design goal for the same game and acting as separate experiments. Independently for each target, our balancing algorithm will attempt to find a parameter vector $\boldsymbol{\theta}$ which yields a metagame balance, in terms of bot winrates, that approximates the target balancing graph. The two design goals we target are *fair* balancing and *cyclic* balancing. Fair balancing dictates that all bots should stand an equal chance of winning against all other bots.



(a) Fair balance

(b) Cyclic behaviour

Figure 5.8: Target graphs for the two experiments on Workshop Warfare. Note that graph 5.8a is bidirectional.

That is to say, all bot winrates should be 50%. On the other hand, cyclic balancing dictates that some bots should stand a higher chance at winning against certain bots than against others. Torch bot should have a 70% winrate against Nail bot, with the same applying to Nail bot against Saw bot and Saw bot against Torch bot. This is a relaxed form of Rock-Paper-Scissors, and will benefit players that are able to guess which bot their opponent will choose, much as for deck selection in a deck-building game.

5.8.2 Computing an Evaluation Matrix

In contrast with the motivational examples in Section 5.6, the metagames that we aim to balance in Workshop Warfare experiments require deriving metrics from in-game trajectories.

Workshop Warfare is a 2-player symmetric zero-sum game. This means that we can exploit the fact that the winrate of bot a against bot b is $w_{ab} = 1 - w_{ba}$. Let $\boldsymbol{\theta}$ denote an arbitrary parameter vector for Workshop Warfare. Let $w^{\boldsymbol{\theta}}_{ST}$ denote the winrate of Saw bot vs Torch bot, Saw bot vs Nail bot $w^{\boldsymbol{\theta}}_{SN}$ and Torch bot vs Nail bot $w^{\boldsymbol{\theta}}_{TN}$ obtained by AI agents when playing in the game under parameterization $\boldsymbol{\theta}$. Our algorithm will attempt to find the right set of game parameters $\boldsymbol{\theta}$ that yields either a cyclic or fair balancing in terms of these winrates. To compute these winrates, we simulate many head-to-head matches where each bot is controlled by an agent using MCTS to guide its actions. Each matchup's winrates are computed from the result of 100 game simulations. A higher number of game simulations would result in a more accurate prediction of the true winrate between two bots, at the cost of more computational time.

5.8.3 Gameplaying Agents via Monte Carlo Tree Search

We use MCTS to generate gameplaying agents to populate the bots in the environment. Due to the simultaneous action nature of Workshop Warfare, we use a different version of MCTS compared to what we have been using in previous chapters of this thesis[10], as we have only simulated sequential games so far. We model our simultaneous MCTS implementation after [Auger, 2011]. As a brief summary, multiple trees are searched simultaneously. During the search procedure, a separate tree is kept for each player, and during the selection phase all trees are simultaneously descended, using tree statistics in the tree for the relevant player at each selection step.

[10] Its implementation can be found alongside the sequential version that has been previously showcased: `https://github.com/Danielhp95/Regym/tree/master/regym/rl_algorithms/MCTS`

	Fair balance: θ_{fair}			Cyclic balance: θ_{cyclic}		
Match	Target	Found	Error	Target	Found	Error
w_{ST}	50%	48.33% ± 2.62%	3.00% ± 1.00%	70%	68% ± 6.37%	5.33% ± 4.02%
w_{SN}	50%	50.30% ± 1.69%	1.66% ± 0.47%	30%	28% ± 4.10%	4.00% ± 1.63%
w_{TN}	50%	49.66% ± 4.10%	3.66% ± 1.88%	70%	70% ± 8.83%	8.00% ± 3.74%

Table 5.1: Average winrates obtained by MCTS agents after playing on the game under the best candidate parameterizations found on each run for both experiments. Results are presented alongside their standard deviations computed from 3 different runs. Where Saw Vs. Torch (w_{ST}), Saw Vs. Nail (w_{SN}) and Torch Vs. Nail (w_{TN}) after balancing and their corresponding errors.

All MCTS agents use a computational budget of 625 iterations. A higher computational budget is directly related to a higher skill level [Lanzi, 2019]. Following this idea, our method could be used to balance a game at different levels of play by changing the computational budget.

We use a reward scheme that incentivizes bots to interact with one another by (1) giving negative score to actions that would increase distance between bots (2) giving positive / negative score to damaging the opponent / being damaged and (3) giving a score for winning the game. The magnitude of rewards (1), (2), and (3) vary between 0-10, 10-99, and 1000 respectively so as to represent a hierarchy of goals for the agent to follow. Note how different reward schemes will lead to different behaviours, which links to the notion of expected behaviour touched upon in Section 5.5.4.

5.8.4 Experimental Details

For each of the two target balancings presented in Figure 5.8 we ran 3 experiments in order to ascertain the stochasticity of our proposed algorithm. Each experiment was given 6 parallel workers to try out different parameterizations simultaneously. Each experiment received a computational budget of 48h, yielding an average number of 350 game parameterizations tried per experiment. The total compute across all experiments was 36 days carried out during 48h of wall-clock time. On the one hand, this shows the advantages of automation, as multiple parameterizations can be evaluated simultaneously. On the other hand, reaching good quality parameterizations requires a lot of trial and error, which prevents current methods from being used in complex existing games.

5.9 Results

The final metagames that emerged from the found parameters for each experiment averaged over all training runs are present in Table 5.1, with all the corresponding game parameterizations found for each run displayed in Table 5.2. We first give an analysis of the evolution of the optimization process followed by a game-design oriented analysis of the game parameterizations found.

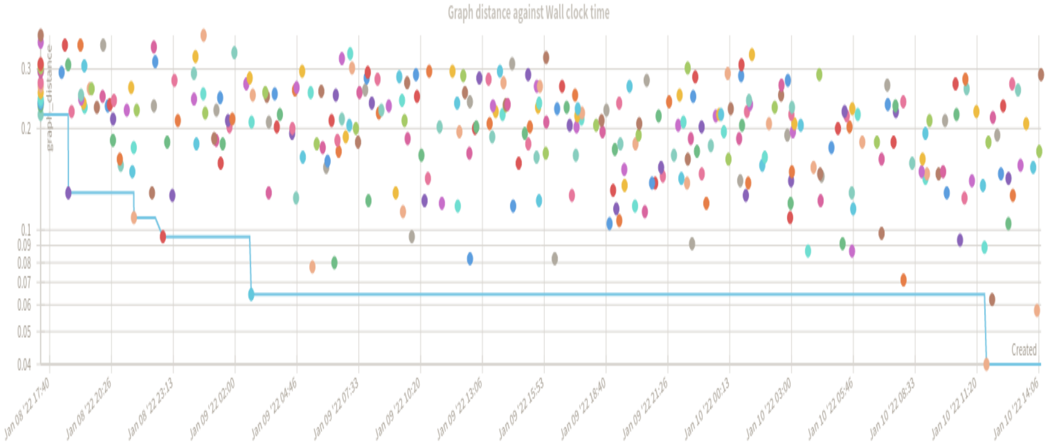


Figure 5.9: Evolution of graph distance for the cyclic balancing experiment C_1, serving as an exemplary run. The vertical axis in log-scale indicates the graph distance to the target metagame balance associated with a given game parameterization and the horizontal axis indicates timestamps for the different parameterizations, spawning over 48h. The best parameters found at any given timestamp are linked with a blue line to show the improvement over time. The plot contains the 349 parameter configurations evaluated during this run.

Let's focus our attention on Table 5.1, which presents quantitative metrics for the success of our experiments, telling us how close the metagames that emerged from the found parameterizations are to the target metagames. The fair and cyclic balancing experiments obtained an average error over all winrates of 2.73% and 5.73% respectively. Even though the cyclic balancing case appears harder to balance as estimated by a larger reported error, we see that our algorithm was successful in finding parameterizations whose emergent metagame balances closely approximate designer's intent. This serves as empirical evidence to the effectiveness of our proposed algorithm.

Figure 5.9 showcases the evolution of the graph distance over time as new game parameterizations were tested for the cyclic balancing experiment C_1, serving as a representative run for the entire set of experiments. On average an algorithmic iteration (choosing a set of game parameters and evaluating them) was completed every 20-25 minutes with most of the computational time being spent on MCTS internal simulations, which speaks to the computational complexity of evaluating a game parameterization. Although a linear speedup could be gained by increasing the number of CPUs, further improvements aimed reducing the computational load of the algorithm would be needed to scale our algorithm to real-world games.

The optimization process of every run followed a very similar pattern to the one shown in Figure 5.9. During the initial stages of optimization we observe a high rate of parameterizations found which improve upon the last best candidate, with the first 5 improvements in Figure 5.9 happening within the initial 9h. These are clear indications that automated game balancing can have a faster turn around of results when compared to slower human search. Yet this might not necessarily be the case for the final fine tuning stages. Note that there is a gap of approximately 20h in wall-clock time where no parameter vector was found which improved upon the best candidate. This is clearly an issue, specially for more computationally intensive games. From the user's perspective, our method does not return during those 20h any progress towards finding a game parameterization that realizes the target metagame balancing due to no new set of better parameter candidates being found. We argue that this computation is not necessarily wasted. If our algorithm is paired with rich data-oriented logging mechanisms, there is a plethora of metrics not directly relevant to the optimization process, which could be extracted from our algorithm's computation that may be of use to the game developers [Bergdahl et al., 2020].

5.9.1 Gameplay Analysis

		Bounds		**Fair: θ_{fair}**			**Cyclic: θ_{cyclic}**		
Bot Type	Parameter	Min	Max	F_1	F_2	F_3	C_1	C_2	C_3
Torch	Health	1	15	9	14	3	7	8	14
	Cooldown	1	15	3	7	9	5	1	3
	Damage	1	10	3	3	7	3	8	7
	Ticks between move	1	15	6	13	15	4	7	7
	Torch duration	1	10	3	3	10	2	5	5
	Torch range	1	4	4	4	4	3	1	1
Nail	Health	1	10	4	8	5	3	3	2
	Cooldown	1	15	1	1	15	1	3	4
	Damage	1	10	7	10	5	9	7	9
	Ticks between move	1	15	2	9	5	2	6	5
Saw	Health	1	10	6	2	1	4	7	1
	Cooldown	1	15	3	6	7	3	2	14
	Damage	1	10	2	5	5	6	1	1
	Damage change	1	10	8	6	9	6	10	7
	Ability duration	1	10	6	1	8	3	4	5
	Ticks between move	1	15	4	15	11	5	14	12

Table 5.2: Game parameters sorted by bot type found after the termination of each of the 3 runs for both balancing experiments. The columns left of the found parameterizations indicate minimum and maximum values for each parameter. F_i and C_i denote the game parameters found for the ith fair and cyclic experiments respectively.

As a game designer, one of the most important question to ask upon the termination of our experiments is: *how do the different bots play under a given parameterization?* To respond to this we briefly analyze one of the parameterizations found from each experiment (F_1 and C_1 from Table 5.2) as a means to understand how emerging balancing arises from game parameterizations.

Fair Balancing

Torch bot, with the most health (9), slowest movement (6) and lowest damage (3), plays like a "tank"[11]. Nail bot is a "glass cannon"; the fastest (2) and most damaging (7) character with the lowest health (4). Its cooldown of 1 tick allows it to quickly react to opponents close by, and to barrage other bots from a distance. Saw bot moves at a medium speed 4 and has to carefully approach opponents, but once it reaches them a victory is always guaranteed. This fair balancing suffered from an aggregate winrate error of 5%.

Cyclic Balancing

Bot behaviours are similar to the previous case, with some differences. Saw bot is slower (5), often using the stand still action (S) to time movement to avoid damage. It exploits Torch bot's shorter range (3) and longer cooldown (5) to wait for an opening from a distance Nail bot, as fast as before but even more damaging (9) is able to position itself for a single nail that kills the slower Saw bot. Because Nail bot now has only 3 health, it dies to a single touch by Torch bot's flame, making it significantly weaker against it. This cyclic balancing suffered from an aggregate winrate error of 18%.

5.9.2 Behavioural Diversity Emerging from Different Parameterizations

Notice how the different game parameterizations present in Table 5.2 vary heavily even within different runs from the same experiment, which ties back to the non-deterministic nature of our balancing algorithm discussed in Section 5.5. This means that (1) with such difference in values there might be few behavioural traits in the agents shared across parameterizations and that (2) even for a simple game such as ours, there is a large diversity of parameter combinations which can satisfy a desired target balance. This parametric and behavioural diversity could be considered harmful insofar as the latter might defy the original intent of the designer, known in the field of AI as value misalignment. In other words, in order for agents to fulfill the target balancing under a given parameterization they might be forced to play in unexpected and unintended ways. To name a few such potential misalignments withing our found balancings: in the F_1 balancing case, all bots can die from either 1 or 2 hits, which makes for short-lived matches. Furthermore, Saw bot's ability lasts for 6 ticks, whilst having a low cooldown of just 3 ticks. This makes its ability an almost permanent effect rather than a special action. We argue that this behavioural misalignment could be given tighter authorial control by introducing regularizers such as more nuanced reward schemes for the AI agents, as we do for MCTS. Note that the playstyle displayed by MCTS tends to be very offensive at the beginning, and very defensive later on. More human-like methods for generating gameplaying agents would greatly benefit the result of our algorithm (line 5 of Algorithm 18), although we emphasize that this problem is orthogonal to our balancing algorithm.

[11] https://en.wikipedia.org/wiki/Tank_(video_games)

5.10 Conclusion

In this chapter we have highlighted the importance and difficulties of the ever growing task of metagame balancing in game design. We have also identified how this task can be posited as a problem within the abstraction level of the environment as defined by the MARL stack. To attempt a solution to this problem, we have presented an algorithm to automate the balancing of a game according to a target designer intent. This algorithm is supported by a combination of ideas from AI for gameplaying, optimization, graph theory and game design. Our procedure algorithm comes alongside a general graph based definition for metagame balance descriptions. Such descriptions allow for the representation of the relative strength of game elements at an arbitrary level of abstraction within a game. We have also demonstrated its empirical convergence in a simple toy domain and showcased its potential in a richer game environment. To our knowledge, our work is one of the first steps in the field of game design towards creating robust tools for automated metagame balancing in multiagent games. The issues of computational time, non human-like AI behaviour and the complexity of generating gameplaying agents remain as obstacles in the path towards accessible adoption of our algorithm by designers.

Our contributions could be transformed into the "backend" of an actual tool. To make it amenable to be used by non-technical individuals, a user-friendly "frontend" should be developed, exposing an interface to (1) parameterize a game and (2) make it easy to specify a level of abstraction and its corresponding balance graph.

Chapter 6

Conclusion

Having presented all of the research contributions to every layer of the MARL stack in the previous research chapters, we conclude the thesis in this chapter. Section 6.1 revisits all the research questions posited in the introduction chapter alongside their assumptions and limitations in Section 6.2. Section 6.3 summarises all of the contributions presented within thesis. Finally, Section 6.4 expands on the future work mentioned at the end of each Chapter and closing remarks.

6.1 Revisiting Original Research Questions

The introduction in Chapter 1 presented the 3-layered MARL stack, and posited a research question centered on each level of the stack. We now proceed to analyze the extent to which each research question has been answered within the body of this thesis.

Training Schemes

Within Chapter 3 the following research question was examined:

> **Research question 1:** Does the choice of opponents that a learning agent plays against have a measurable effect on the learning dynamics of such agent?

We consider that we have positively answered this research question based on the qualitative and quantitative experiments carried out in RirRPS and Connect4 & RirRPS respectively. Our reasoning goes as follows: agent policies and value functions are updated based on the experiences collected during training. In the first qualitative experiment we demonstrated that the evolution of the exploration of the trajectory space within RirRPS would drastically vary depending on the self-play training scheme used. Different explorations of the trajectory space will in turn provide agents with different data to update their models on, which directly affects the dynamics of learning for the learning agent. We gave further evidence with quantitative results in the (mostly cyclic) environment of RirRPS and (mostly transitive) game of Connect4. Once again, empirical evidence suggested that the choice of opponents drastically affects the learning dynamics. Algorithms like Naive self-play or $\delta = 0.5$-Uniform would suffer from cyclic policy evolutions, while PSRO or $\delta = 0$-Uniform would avoid them.

A limitation that must be noted is that all underlying agents used the same algorithm, Proximal Policy Optimization [Schulman et al., 2017], to update their models. That is to say, we tested the effects of various self-play training schemes against a single agent level algorithm. It could be argued that the outcome of the experiments could have differed had we used other agent level algorithms. For instance, had we used off-policy algorithms which make use of a replay buffer, perhaps the effects of opponent selection would not be as pronounced. This is because actor-critic updates on such algorithms sample from a (potentially) large replay buffer, which can contain trajectories from previous opponents that potentially dilutes the effect of the latest opponents being chosen.

Agents

Within Chapter 4 the following research question was examined:

> **Research question 2:** Is the sample efficiency of model-based algorithms improved by learning opponent models during play and then using them inside model simulations?

We conclude that an affirmative answer to this question can be given under certain conditions based on the results from Chapter 4. There we saw that merely equipping ExIt with opponent modelling as an auxiliary task for feature shaping purposes was actually detrimental to its overall performance, an algorithmic ablation which we called ExIt-OMFS. As previously explained, this detrimental effect has not been present on other work using opponent models in sequential games [Wu, 2019], leading us to believe that experimenting with other procedures to generate opponent models in sequential settings could convert these negative results into positive ones. However, the sample efficiency of ExIt did improve once these opponents models (or the ground truth opponent policies) were used as well *within* the expert's search yielding the BRExIt algorithm, leading to a positive answer to our research question if opponent models are deeply integrated within ExIt. This findings were also motivated by game theoretic and RL notions that give weight to the necessity of using opponent awareness as an ingredient to generate policies which act as a best response against given or estimated opponents. A similar limitation to note with the previous research question is that we have only tested research question 2 with the BRExIt algorithm.

Environments

Within Chapter 5 the following research question was examined:

> **Research question 3:** Can agents trained via MARL be used as proxy players as part of the automation of game balancing?

Finally, we also give an affirmative answer to this research question. The experiments conducted in the final research chapter showed that MARL agents can be used to evaluate the balancing of a game under a given parameterization, as every target balancing was obtained with acceptable error margins for each experiment conducted. This puts MARL agents as viable candidates for generating gameplay data from which to evaluate the meta-game balancing of a game under a certain parameterization, traditionally a human-centric task. A clear limitation of the experimental approach taken is that we have only used MCTS as a game-playing agent, which does not exhibit a very human-like behaviour and requires a model of the environment to be executed. It remains to be seen how to easily integrate SP as an alternative to training game-playing agents, as it stands as the most obvious training scheme to train agents for a game under development, instead of MCTS for when it is not good enough or when a model of the game is not available.

6.2 Limitations

We present the main limitations shared across all studies presented throughout this thesis:

Low Number of Test Environments

All of our experiments are run on a small set of environments, which in conjunction with the lack of theoretical proofs reduces the empirical and theoretical generalizability of our results to a broader set of environments. Chapter 3 performs its experiments in RirRPS and Connect4, with Chapter 4 focusing only on Connect4 and Chapter 5 using our own developed game, Workshop Warfare. Thus, the claims made throughout this thesis should be conservatively assumed to be true within the environments in which they were empirically verified, serving as potential indicators of similar outcomes for other environments. Under a more positive lens, all of the used environments have been 2-player zero-sum games which are known to be decomposable into transitive and cyclic components [Balduzzi et al., 2019]. By using the highly cyclic RirRPS and highly transitive Connect4 we have covered both basis for further 2-player zero-sum games, strengthening the generalizability of our results.

Low Computational Budget

Many of the algorithms cited in this thesis have originally used orders of magnitude more computational budget than what we have available [Silver et al., 2016, Silver et al., 2018, Berner et al., 2019, Lanctot et al., 2017]. Other publications [Baker et al., 2019] claim that within complex MARL environments it is only after a large numbers of simulation steps[1] have elapsed that interesting behaviours will emerge. From this insight it could be estimated that the conclusions and insights drawn from our studies might not be indicative of the evolution of agent behaviour within realms of larger compute. As a concrete example: BRExIt initially biases MCTS towards a best response against the opponent policies present in an environment. This biasing effect is slowly washed out by large MCTS budgets, meaning the topology of large search trees constructed by BRExIt will resemble those generated by ExIt. From this, it is not unreasonable to speculate that the performance difference between BRExIt and ExIt might decrease with large amounts of compute. This in itself is not detrimental to

[1] In the order of thousands of millions of environment steps and batch sizes larger than 10^5.

our results, as BRExIt showing higher sample efficiency than ExIt in low compute settings is also useful for potential users that might not have access to large amounts of compute.

6.3 Summary of Contributions

Here we enumerate the main contributions that can be drawn from the presented work:

A Generalized Framework for Defining SP Algorithms

After a thorough study of the self-play literature in Chapter 3, a generalized self-play framework is presented which brings a large number of reviewed algorithms under a unified notation. It defines self-play as a centralized training scheme which trains agents by pitting them against themselves or checkpoints of older versions. By using the concepts of a menagerie, policy sampling distribution and curator, these components have been shown to be descriptive enough to capture a large set of existing algorithms that had previously remained unconnected. Under this unified notation, we present qualitative and quantitative evidence to answer research question 1 and further showcase that the choice of self-play training scheme can greatly impact the quality of the agents trained under them.

BRExIt: An Opponent Modelling Based Extension to the ExIt Framework

Recent advances in opponent modelling within MARL and planning based methods are combined in Chapter 4 to extend Expert Iteration into Best Response Expert Iteration. BRExIt uses opponent modelling as both an auxiliary task and a biasing mechanism within MCTS to generate policy targets which approximate a best response against the other agent policies present in the environment. This method is motivated as a better alternative than its predecessor, Exit, if used as a policy improvement operator within centralized training schemes such as self-play. We also present MCTS relative strength as an objective scalar metric aimed at estimating the transitive skill of a target agent.

A Metagame Balance Description Format & Metagame Autobalancing Algorithm

Drawing from the various uses of the concept of metagames in Chapters 2, 3 and 4, their usage is ported as a game design tool to define the balance of a game at a arbitrary levels of abstraction. A game designer friendly graph based representation of metagame balancing is introduced as a description of balancing intent. Given a game and a target metagame balance by a game designer, we present an algorithm that automates the process of finding the low level environment parameters whose emerging metagame balance by rational AI agents will match the target balance.

6.4 Future Work

The contributions brought by this thesis open up the opportunity for further development in the field. Among all future research possibilities, these are the ones that we would emphasize the most for further exploration:

Meta Self-Play

This thesis has put a heavy emphasis on the conceptual separation between levels of abstraction within elements of the MARL stack, with Chapters 3 and 4 delimiting the independent concerns and challenges of training schemes and learning agents respectively. We hypothesize that there exists a space of agent level RL algorithms which could be made aware not only of the fact that they are being trained on a sequence of opponents, but also on the overarching structure dictating such choice of opponents. By giving a learning agent knowledge of the higher level self-play scheme used to train it we could explore the largely uncharted space of meta self-play algorithms. In doing so, an agent could adapt its behaviour so as to be competent against future opponents to come, instead of single mindedly optimizing against the opponent that it happens to be playing against, while letting the higher training scheme worry about which set of opponents to present to the learning agent. Formally, this refers to the idea of adding meta-learning capabilities to existing MARL algorithms, which would adapt a policy's parameter updates so as to remain fine-tunable to remain good against an estimated distribution over future opponent policies to play against.

Further Studies of Opponent Modelling Inside Planning

As explored deeply in Chapter 4, BRExIt features opponent modelling inside of MCTS by utilizing either opponent models or the true opponent policies to generate priors over which tree nodes to explore during the selection phase. We note that there is still room for further use of opponent models within MCTS. Specifically, on top of policy reconstruction which aims at estimating opponent policies, BRExIt could also learn models of opponents' value functions which could in turn be exploited during the initial step of the backpropagation phase[2]. The presented version of BRExIt uses a learnt value function, V^{π}, to compute an initial value for nodes in the search tree at the beginning of the backpropagation phase. This value function V^{π} is trained to compute the expected return for the apprentice policy π. However, there will be nodes in the tree corresponding to other opponent policies π_o. By using V^{π} to compute the value to propagate up the tree, we bias the valuation of every node towards V^{π}, which might be an arbitrarily different value to the true expected valuation of the other opponent policies present in the environment V^{π_o}. We argue that if we had access to the value function V^{π_o} for all opponent policies we could improve the quality of the MCTS search and increase the sample efficiency of BRExIt. This would require to change BRExIt

[2] The definition of MCTS allows the computation of a value to be backpropagated as being part of either the last step of the rollout phase or the first step of the backpropagation phase. We decide for the latter given that our implementation of BRExIt forgoes the rollout phase.

to be able to approximate a value function for either ground truth or learnt opponent policies. This is a non-trivial task and was thus left out from our previous investigations.

Using BRExIt as an oracle within double loop MARL training schemes

At the level of training schemes, it would be very interesting to measure whether the quality of populations generated by different training schemes using BRExIt surpasses that of populations where ExIt is used as a policy improvement operator. In Chapter 4 we have seen that for the game of Connect4 BRExIt is more performant than ExIt. However, this is not necessarily an indication that BRExIt would outperform ExIt in the long run if used within the inner loop of the double loop posited by modern MARL training schemes such as PSRO. To measure this we could replicate experiments like the ones in Chapter 3 which measured the evolution of relative population performances.

6.5 Closing Remarks

In all MARL scenarios studied within this thesis, paying close attention to the influence of agents on each other's behaviour has been shown to be of paramount importance. We hope that the work presented in this thesis helps to focus future research on opponent awareness at all levels of the MARL stack as a high impact area. On the other hand, we also hope that our findings can be of help to game makers for understanding the various conceptual abstractions that MARL is built upon so as to facilitate its usage within their artistic creations.

Bibliography

[Abramson, 2013] Abramson, B. (2013). A cure for pathological behavior in games that use minimax. *arXiv preprint arXiv:1304.3444*.

[Agarwal et al., 2021] Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. C., and Bellemare, M. (2021). Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 34.

[Akiba et al., 2019] Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. *CoRR*, abs/1907.10902.

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

[Anthony et al., 2017] Anthony, T., Tian, Z., and Barber, D. (2017). Thinking Fast and Slow with Deep Learning and Tree Search. (Il):1–19.

[Argue, 2014] Argue, R. (2014). Supervised learning as a tool for metagame analysis.

[Arslan and Yüksel, 2016] Arslan, G. and Yüksel, S. (2016). Decentralized q-learning for stochastic teams and games. *IEEE Transactions on Automatic Control*, 62(4):1545–1558.

[Auger, 2011] Auger, D. (2011). Multiple tree for partially observable monte-carlo tree search. In *European Conference on the Applications of Evolutionary Computation*, pages 53–62. Springer.

[Aytar et al., 2018] Aytar, Y., Pfaff, T., Budden, D., Paine, T. L., Wang, Z., and de Freitas, N. (2018). Playing hard exploration games by watching youtube. *arXiv preprint arXiv:1805.11592*.

[Badia et al., 2020] Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D., and Blundell, C. (2020). Agent57: Outperforming the atari human benchmark. In *International Conference on Machine Learning*, pages 507–517. PMLR.

[Baker et al., 2019] Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., and Mordatch, I. (2019). Emergent tool use from multi-agent interaction. *Machine Learning, Cornell University*.

[Balduzzi et al., 2019] Balduzzi, D., Garnelo, M., Bachrach, Y., Czarnecki, W. M., Pérolat, J., Jaderberg, M., and Graepel, T. (2019). Open-ended learning in symmetric zero-sum games. corr, abs/1901.08106, 2019.

[Balduzzi et al., 2018] Balduzzi, D., Tuyls, K., Perolat, J., and Graepel, T. (2018). Re-evaluating evaluation. *arXiv preprint arXiv:1806.02643*.

[Bansal et al., 2017] Bansal, T., Pachocki, J., Sidor, S., Sutskever, I., and Mordatch, I. (2017). Emergent Complexity via Multi-Agent Competition. 2:1–12.

[Bassich et al., 2020] Bassich, A., Foglino, F., Leonetti, M., and Kudenko, D. (2020). Curriculum learning with a progression function. *arXiv preprint arXiv:2008.00511*.

[Beck et al., 2019] Beck, J., Ciosek, K., Devlin, S., Tschiatschek, S., Zhang, C., and Hofmann, K. (2019). Amrl: aggregated memory for reinforcement learning. In *International Conference on Learning Representations*.

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

[Bellman, 1957] Bellman, R. (1957). *Dynamic Programming*, volume 70.

[Bergdahl et al., 2020] Bergdahl, J., Gordillo, C., Tollmar, K., and Gisslén, L. (2020). Augmenting automated game testing with deep reinforcement learning. In *2020 IEEE Conference on Games (CoG)*, pages 600–603. IEEE.

[Bergstra et al., 2011] Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc.

[Berner et al., 2019] Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*.

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

[Beyer et al., 2016] Beyer, M., Agureikin, A., Anokhin, A., Laenger, C., Nolte, F., Winterberg, J., Renka, M., Rieger, M., Pflanzl, N., Preuss, M., et al. (2016). An integrated process for game balancing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE.

[Borkar, 1988] Borkar, V. S. (1988). A convex analytic approach to Markov decision processes. *Probability Theory and Related Fields*, 78(4):583–602.

[Boutilier, 1996] Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. *Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, pages 195–210.

[Bouzy, 2007] Bouzy, B. (2007). Old-fashioned computer go vs monte-carlo go, invited tutorial. In *IEEE 2007 Symposium on Computational Intelligence in Games, CIG*, volume 7.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[Brown, 1951] Brown, G. W. (1951). Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13(1):374–376.

[Browne et al., 2012] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of {Monte Carlo Tree Search} methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

[Carroll et al., 2019] Carroll, M., Shah, R., Ho, M. K., Griffiths, T., Seshia, S., Abbeel, P., and Dragan, A. (2019). On the utility of learning about humans for human-ai coordination. In *Advances in Neural Information Processing Systems*, pages 5175–5186.

[Carter et al., 2012] Carter, M., Gibbs, M., and Harrop, M. (2012). Metagames, paragames and orthogames: A new vocabulary. In *Proceedings of the international conference on the foundations of digital games*, pages 11–17.

[Chang et al., 2020] Chang, M., Kaushik, S., Weinberg, S. M., Griffiths, T., and Levine, S. (2020). Decentralized reinforcement learning: Global decision-making via local economic transactions. In *International Conference on Machine Learning*, pages 1437–1447. PMLR.

[Chaslot et al., 2006] Chaslot, G., Saito, J.-T., Bouzy, B., Uiterwijk, J., and Van Den Herik, H. J. (2006). Monte-carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91.

[Chaslot et al., 2008] Chaslot, G. M. J., Winands, M. H., HERIK, H. J. V. D., Uiterwijk, J. W., and Bouzy, B. (2008). Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357.

[Chen et al., 2021] Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., and Mordatch, I. (2021). Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097.

[Coulom, 2006] Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer.

[Czarnecki et al., 2020] Czarnecki, W. M., Gidel, G., Tracey, B., Tuyls, K., Omidshafiei, S., Balduzzi, D., and Jaderberg, M. (2020). Real world games look like spinning tops. *arXiv preprint arXiv:2004.09468*.

[De Asis et al., 2017] De Asis, K., Ca, K., Hernandez-Garcia, J. F., Ca, J., Holland, G. Z., and Sutton, R. S. (2017). Multi-step Reinforcement Learning: A Unifying Algorithm.

[de Mesentier Silva et al., 2019] de Mesentier Silva, F., Canaan, R., Lee, S., Fontaine, M. C., Togelius, J., and Hoover, A. K. (2019). Evolving the hearthstone meta. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE.

[Deisenroth and Rasmussen, 2011] Deisenroth, M. P. and Rasmussen, C. E. (2011). PILCO: A Model-Based and Data-Efficient Approach to Policy Search.

[Denamganaï and Walker, 2020] Denamganaï, K. and Walker, J. A. (2020). Referentialgym: A nomenclature and framework for language emergence & grounding in (visual) referential games. *arXiv preprint arXiv:2012.09486*.

[Devlin and Kudenko, 2016] Devlin, S. and Kudenko, D. (2016). Plan-based reward shaping for multi-agent reinforcement learning. *The Knowledge Engineering Review*, 31(1):44–58.

[Devlin and Kudenko, 2012] Devlin, S. M. and Kudenko, D. (2012). Dynamic potential-based reward shaping. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, pages 433–440. IFAAMAS.

[Dosovitskiy et al., 2015] Dosovitskiy, A., Springenberg, J. T., and Brox, T. (2015). Learning to generate chairs with convolutional neural networks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June:1538–1546.

[Duan et al., 2016] Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. (2016). Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*.

[Eccles et al., 2019] Eccles, T., Bachrach, Y., Lever, G., Lazaridou, A., and Graepel, T. (2019). Biases for emergent communication in multi-agent reinforcement learning. *arXiv preprint arXiv:1912.05676*.

[Elo, 1978] Elo, A. (1978). The rating of chess players, past and present (arco, new york).

[Espeholt et al., 2018] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. (2018). IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures.

[Fan et al., 2020] Fan, T., Long, P., Liu, W., and Pan, J. (2020). Distributed multi-robot collision avoidance via deep reinforcement learning for navigation in complex scenarios. *The International Journal of Robotics Research*, 39(7):856–892.

[Fedus et al., 2020] Fedus, W., Ramachandran, P., Agarwal, R., Bengio, Y., Larochelle, H., Rowland, M., and Dabney, W. (2020). Revisiting fundamentals of experience replay. In *International Conference on Machine Learning*, pages 3061–3071. PMLR.

[Firoiu et al., 2017] Firoiu, V., Whitney, W. F., and Tenenbaum, J. B. (2017). Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning.

[Foerster et al., 2017] Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P. H., Kohli, P., and Whiteson, S. (2017). Stabilising experience replay for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 1146–1155. PMLR.

[Fontaine et al., 2019] Fontaine, M. C., Lee, S., Soros, L. B., de Mesentier Silva, F., Togelius, J., and Hoover, A. K. (2019). Mapping hearthstone deck spaces through map-elites with sliding boundaries. In *Proceedings of The Genetic and Evolutionary Computation Conference*, pages 161–169.

[Gao et al.,] Gao, X., Xiao, B., Tao, D., and Li, X. A survey of graph edit distance.

[Ge, 2016] Ge, K. (2016). Expected value and markov chains.

[Gelly and Silver, 2007] Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280.

[Goodman and Lucas, 2020] Goodman, J. and Lucas, S. (2020). Does it matter how well i know what you're thinking? opponent modelling in an rts game. *arXiv preprint arXiv:2006.08659*.

[Gordillo et al., 2021] Gordillo, C., Bergdahl, J., Tollmar, K., and Gisslén, L. (2021). Improving playtesting coverage via curiosity driven reinforcement learning agents. *arXiv preprint arXiv:2103.13798*.

[Guzdial et al., 2017] Guzdial, M., Li, B., and Riedl, M. O. (2017). Game Engine Learning from Video. *International Conference on Artificial Intelligence (IJCAI)*.

[Hafner et al., 2020] Hafner, D., Lillicrap, T., Norouzi, M., and Ba, J. (2020). Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*.

[Hansen et al., 2004] Hansen, E. A., Bernstein, D. S., and Zilberstein, S. (2004). Dynamic programming for partially observable stochastic games. In *AAAI*, volume 4, pages 709–715.

[Harris et al., 2020] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

[He et al., 2016] He, H., Boyd-Graber, J., Kwok, K., and Daumé III, H. (2016). Opponent modeling in deep reinforcement learning. In *International Conference on Machine Learning*, pages 1804–1813.

[Henderson et al., 2018] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32.

[Herbrich et al., 2007] Herbrich, R., Minka, T., and Graepel, T. (2007). Trueskill™: a bayesian skill rating system. In *Advances in neural information processing systems*, pages 569–576.

[Hernandez et al., 2019] Hernandez, D., Denamganaï, K., Gao, Y., York, P., Devlin, S., Samothrakis, S., and Walker, J. A. (2019). A generalized framework for self-play training. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE.

[Hernandez-Leal et al., 2019] Hernandez-Leal, P., Kartal, B., and Taylor, M. E. (2019). Agent modeling as auxiliary task for deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 31–37.

[Hong et al., 2018] Hong, Z.-W., Su, S.-Y., Shann, T.-Y., Chang, Y.-H., and Lee, C.-Y. (2018). A deep policy inference q-network for multi-agent systems. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1388–1396. International Foundation for Autonomous Agents and Multiagent Systems.

[Horgan et al., 2018] Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. (2018). Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*.

[Howard, 1960] Howard, R. A. (1960). Dynamic programming and markov processes.

[Jacob et al., 2020] Jacob, M., Devlin, S., and Hofmann, K. (2020). "it's unwieldy and it takes a lot of time"—challenges and opportunities for creating agents in commercial games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 88–94.

[Jaderberg et al., 2019] Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castaneda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., et al. (2019). Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865.

[Jaderberg et al., 2017] Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al. (2017). Population based training of neural networks. *arXiv preprint arXiv:1711.09846*.

[Jaderberg et al., 2016] Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2016). Reinforcement Learning with Unsupervised Auxiliary Tasks. pages 1–14.

[Jaffe et al., 2012a] Jaffe, A., Miller, A., Andersen, E., Liu, Y.-E., Karlin, A., and Popovic, Z. (2012a). Evaluating competitive game balance with restricted play. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Jaffe et al., 2012b] Jaffe, A., Miller, A., Andersen, E., Liu, Y.-E., Karlin, A., and Popovic, Z. (2012b). Evaluating competitive game balance with restricted play.

[Jaffe, 2013] Jaffe, A. B. (2013). *Understanding game balance with quantitative methods*. PhD thesis.

[Juliani et al., 2018] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., et al. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

[Kaelbling et al., 1998] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134.

[Kanervisto et al., 2020] Kanervisto, A., Kinnunen, T., and Hautamäki, V. (2020). Policy supervectors: General characterization of agents by their behaviour. *arXiv preprint arXiv:2012.01244*.

[Katona et al., 2019] Katona, A., Spick, R., Hodge, V. J., Demediuk, S., Block, F., Drachen, A., and Walker, J. A. (2019). Time to die: Death prediction in dota 2 using deep learning. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE.

[Khalifa et al., 2020] Khalifa, A., Bontrager, P., Earle, S., and Togelius, J. (2020). Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 95–101.

[Kocsis and Szepesvári, 2006] Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

[Konda and Tsitsiklis, 2000] Konda, V. R. and Tsitsiklis, J. N. (2000). Actor-Critic Algorithms. *Nips*, 42(4):1143–1166.

[Kumar et al., 2020] Kumar, A., Zhou, A., Tucker, G., and Levine, S. (2020). Conservative q-learning for offline reinforcement learning. *arXiv preprint arXiv:2006.04779*.

[Lanctot et al., 2017] Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Perolat, J., Silver, D., and Graepel, T. (2017). A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. (Nips).

[Lanzi, 2019] Lanzi, P. L. (2019). Evaluating the complexity of players' strategies using mcts iterations. In *2019 IEEE Conference on Games (CoG)*, page 1–8. IEEE.

[Laurent et al., 2011] Laurent, G. J., Matignon, L., and Fort-Piat, N. L. (2011). The world of independent learners is not Markovian. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, 15(1):55–64.

[Lee and Ramler, 2015] Lee, C.-S. and Ramler, I. (2015). Investigating the impact of game features on champion usage in league of legends. In *FDG*.

[Lee et al., 2018] Lee, E., Jang, Y., Yoon, D., Jeon, J., Yang, S.-i., Lee, S.-K., Kim, D.-W., Chen, P. P., Guitart, A., Bertens, P., et al. (2018). Game data mining competition on churn prediction and survival analysis using commercial game log data. *arXiv preprint arXiv:1802.02301*.

[Leigh et al., 2008] Leigh, R., Schonfeld, J., and Louis, S. J. (2008). Using coevolution to understand and validate game balance in continuous games. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1563–1570.

[Levine et al., 2020] Levine, S., Kumar, A., Tucker, G., and Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*.

[Lillicrap et al., 2015] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning.

[Lin, 1993] Lin, L.-j. (1993). Reinforcement Learning for Robots Using Neural Networks. *Report, CMU*, pages 1–155.

[Liskowski, 2013] Liskowski, P. (2013). Quantitative analysis of the hall of fame coevolutionary archives. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 1683–1686.

[Littman, 1994a] Littman, M. L. (1994a). Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier.

[Littman, 1994b] Littman, M. L. (1994b). Markov games as a framework for multi-agent reinforcement learning. *Machine Learning Proceedings 1994*, pages 157–163.

[Liu and Marschner, 2017] Liu, A. J. and Marschner, S. (2017). Balancing zero-sum games with one variable per strategy. In *AIIDE*.

[Liu et al., 2021] Liu, X., Jia, H., Wen, Y., Yang, Y., Hu, Y., Chen, Y., Fan, C., and Hu, Z. (2021). Towards unifying behavioral and response diversity for open-ended learning in zero-sum games. *Advances in Neural Information Processing Systems*, 34.

[Maaten and Hinton, 2008] Maaten, L. v. d. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605.

[Machado et al., 2014] Machado, M. C., Srinivasan, S., and Bowling, M. (2014). Domain-independent optimistic initialization for reinforcement learning. *arXiv preprint arXiv:1410.4604*.

[Malysheva et al., 2018] Malysheva, A., Kudenko, D., and Shpilman, A. (2018). Learning to run with potential-based reward shaping and demonstrations from video data. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 286–291. IEEE.

[Mao et al., 2020] Mao, W., Zhang, K., Xie, Q., and Basar, T. (2020). Poly-hoot: Monte-carlo planning in continuous space mdps with non-asymptotic analysis. *Advances in Neural Information Processing Systems*, 33:4549–4559.

[McInnes et al., 2020] McInnes, L., Healy, J., and Melville, J. (2020). Umap: Uniform manifold approximation and projection for dimension reduction.

[McMahan et al., 2003] McMahan, H. B., Gordon, G. J., and Blum, A. (2003). Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 536–543.

[Mirowski et al., 2016] Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., et al. (2016). Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*.

[Mishra, 2008] Mishra, D. (2008). An introduction to mechanism design theory. *The Indian Economic Journal*, 56(2):137–165.

[Mitchell, 1997] Mitchell, T. (1997). Machine learning.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. pages 1–9.

[Mohamed and Rezende, 2015] Mohamed, S. and Rezende, D. J. (2015). Variational information maximisation for intrinsically motivated reinforcement learning. *arXiv preprint arXiv:1509.08731*.

[Morosan and Poli, 2017] Morosan, M. and Poli, R. (2017). Evolving a designer-balanced neural network for ms pacman. In *2017 9th Computer Science and Electronic Engineering (CEEC)*, pages 100–105. IEEE.

[Muller et al., 2019] Muller, P., Omidshafiei, S., Rowland, M., Tuyls, K., Perolat, J., Liu, S., Hennes, D., Marris, L., Lanctot, M., Hughes, E., et al. (2019). A generalized training approach for multiagent learning. In *International Conference on Learning Representations*.

[Narvekar et al., 2020] Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M. E., and Stone, P. (2020). Curriculum learning for reinforcement learning domains: A framework and survey. *arXiv preprint arXiv:2003.04960*.

[Nash et al., 1950] Nash, J. F. et al. (1950). Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49.

[Nashed and Zilberstein, 2022] Nashed, S. and Zilberstein, S. (2022). A survey of opponent modeling in adversarial domains. *Journal of Artificial Intelligence Research*, 73:277–327.

[Nelson, 2011] Nelson, M. J. (2011). Game metrics without players: Strategies for understanding game artifacts. In *Artificial Intelligence in the Game Design Process*.

[Nieves et al., 2021] Nieves, N. P., Yang, Y., Slumbers, O., Mguni, D. H., Wen, Y., and Wang, J. (2021). Modelling behavioural diversity for learning in open-ended games. *arXiv preprint arXiv:2103.07927*.

[Nogueira et al., 2013] Nogueira, M., Cotta, C., and Fernández-Leiva, A. J. (2013). An analysis of hall-of-fame strategies in competitive coevolutionary algorithms for self-learning in rts games. In *International Conference on Learning and Intelligent Optimization*, pages 174–188. Springer.

[Norris, 1998] Norris, J. R. (1998). *Markov chains*. Number 2. Cambridge university press.

[Oliehoek and Amato, 2014] Oliehoek, F. A. and Amato, C. (2014). Best response bayesian reinforcement learning for multiagent systems with state uncertainty. In *Proceedings of the Ninth AAMAS Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains (MSDM)*.

[Oliehoek et al., 2006] Oliehoek, F. A., De Jong, E. D., and Vlassis, N. (2006). The parallel nash memory for asymmetric games. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 337–344.

[Omidshafiei et al., 2019] Omidshafiei, S., Papadimitriou, C., Piliouras, G., Tuyls, K., Rowland, M., Lespiau, J.-B., Czarnecki, W. M., Lanctot, M., Perolat, J., and Munos, R. (2019). α-rank: Multi-agent evaluation by evolution. *Scientific reports*, 9(1):1–29.

[Ortiz et al., 2007] Ortiz, L. E., Schapire, R. E., and Kakade, S. M. (2007). Maximum entropy correlated equilibria. In *Artificial Intelligence and Statistics*, pages 347–354. PMLR.

[Ossenkopf, 2019] Ossenkopf, M. (2019). Enhancing language emergence through empathy.

[Owen and Owen, 1982] Owen, G. and Owen, G. (1982). Game Theory. *Collection*.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[Pathak et al., 2017] Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Pfau et al., 2020] Pfau, J., Liapis, A., Volkmar, G., Yannakakis, G. N., and Malaka, R. (2020). Dungeons amp; replicants: Automated game balancing via deep player behavior modeling. In *2020 IEEE Conference on Games (CoG)*, pages 431–438.

[Preston and Jacobs, 2009] Preston, S. D. and Jacobs, L. F. (2009). Mechanisms of cache decision making in fox squirrels (sciurus niger). *Journal of Mammalogy*, 90(4):787–795.

[Robles-Kelly and Hancock, 2003] Robles-Kelly, A. and Hancock, E. R. (2003). Edit distance from graph spectra. In *Computer Vision, IEEE International Conference on*, volume 2, pages 234–234. IEEE Computer Society.

[Rosin, 2011] Rosin, C. D. (2011). Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230.

[Rowland et al., 2020] Rowland, M., Omidshafiei, S., Tuyls, K., Perolat, J., Valko, M., Piliouras, G., and Munos, R. (2020). Multiagent evaluation under incomplete information.

[Sak et al., 2014] Sak, H., Senior, A. W., and Beaufays, F. (2014). Long short-term memory recurrent neural network architectures for large scale acoustic modeling.

[Samuel, 1959] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229.

[Schaul et al., 2015] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized Experience Replay. pages 1–21.

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

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms. pages 1–12.

[Shaker et al., 2016] Shaker, N., Togelius, J., and Nelson, M. J. (2016). *Procedural content generation in games*. Springer.

[Shapley, 1953] Shapley, L. S. (1953). Stochastic games. *Proceedings of the national academy of sciences*, 39(10):1095–1100.

[Shu et al., 2021] Shu, T., Liu, J., and Yannakakis, G. N. (2021). Experience-driven pcg via reinforcement learning: A super mario bros study. *arXiv preprint arXiv:2106.15877*.

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

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

[Silver et al., 2017a] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2017a). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. pages 1–19.

[Silver et al., 2014] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395.

[Silver et al., 2017b] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Van Den Driessche, G., Graepel, T., and Hassabis, D. (2017b). Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359.

[Silver et al., 2021] Silver, D., Singh, S., Precup, D., and Sutton, R. S. (2021). Reward is enough. *Artificial Intelligence*, 299.

[Soemers, 2014] Soemers, D. (2014). Tactical Planning Using MCTS in the Game of StarCraft. page 12.

[Soemers et al., 2019] Soemers, D. J., Piette, E., Stephenson, M., and Browne, C. (2019). Learning policies from self-play with policy gradients and mcts value estimates. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE.

[Soemers et al., 2020] Soemers, D. J., Piette, É., Stephenson, M., and Browne, C. (2020). Manipulating the distributions of experience used for self-play learning in expert iteration. *arXiv preprint arXiv:2006.00283*.

[Summerville et al., 2018] Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., Nealen, A., and Togelius, J. (2018). Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270.

[Sutton, 1991] Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*.

[Sutton et al., 2007] Sutton, R. S., Koop, A., and Silver, D. (2007). On the role of tracking in stationary environments. In *Proceedings of the 24th international conference on Machine learning*, pages 871–878.

[Tamar et al., 2017] Tamar, A., Wu, Y., Thomas, G., Levine, S., and Abbeel, P. (2017). Value iteration networks. *IJCAI International Joint Conference on Artificial Intelligence*, (Nips):4949–4953.

[Tan et al., 2021] Tan, A. H., Bejarano, F. P., and Nejat, G. (2021). Deep reinforcement learning for decentralized multi-robot exploration with macro actions. *arXiv preprint arXiv:2110.02181*.

[Taylor and Stone, 2009] Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7).

[Tesauro, 1990] Tesauro, G. (1990). Neurogammon: A neural-network backgammon program. In *1990 IJCNN international joint conference on neural networks*, pages 33–39. IEEE.

[Tesauro, 1992] Tesauro, G. (1992). Practical Issues in Temporal Difference Learning. *Machine Learning*, 8(3-4):257–277.

[Tesauro, 1995] Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Commun. ACM*, 38:58–68.

[Timbers et al., 2020] Timbers, F., Lockhart, E., Lanctot, M., Schmid, M., Schrittwieser, J., Hubert, T., and Bowling, M. (2020). Approximate exploitability: Learning a best response in large games. *arXiv preprint arXiv:2004.09677*.

[Tomašev et al., 2020] Tomašev, N., Paquet, U., Hassabis, D., and Kramnik, V. (2020). Assessing game balance with alphazero: Exploring alternative rule sets in chess. *arXiv preprint arXiv:2009.04374*.

[Van Der Ree and Wiering, 2013] Van Der Ree, M. and Wiering, M. (2013). Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play. *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, ADPRL*, pages 108–115.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

[Vodopivec et al., 2017] Vodopivec, T., Samothrakis, S., and Ster, B. (2017). On monte carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, 60:881–936.

[Waskom, 2021] Waskom, M. L. (2021). seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021.

[Watkins, 1989] Watkins, C. J. (1989). *Learning from delayed rewards*. PhD thesis, King's College.

[Wellman, 2006] Wellman, M. P. (2006). Methods for empirical game-theoretic analysis. In *AAAI*, pages 1552–1556.

[Wen et al., 2020] Wen, C., Yao, X., Wang, Y., and Tan, X. (2020). Smix (λ): Enhancing centralized value functions for cooperative multi-agent reinforcement learning. In *AAAI*, pages 7301–7308.

[Wilkins et al., 2020a] Wilkins, B., Watkins, C., and Stathis, K. (2020a). Anomaly detection in video games. *CoRR*, abs/2005.10211.

[Wilkins et al., 2020b] Wilkins, B., Watkins, C., and Stathis, K. (2020b). A metric learning approach to anomaly detection in video games. In *2020 IEEE Conference on Games (CoG)*, pages 604–607. IEEE.

[Willemsen et al., 2020] Willemsen, D., Baier, H., and Kaisers, M. (2020). Value targets in off-policy alphazero: a new greedy backup. In *Adaptive and Learning Agents (ALA) Workshop*.

[Williams and Rasmussen, 2006] Williams, C. K. and Rasmussen, C. E. (2006). *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA.

[Williams, 1992] Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3):229–256.

[Wu, 2019] Wu, D. J. (2019). Accelerating self-play learning in go. *arXiv preprint arXiv:1902.10565*.

[Yannakakis and Togelius, 2018] Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. Springer. `http://gameaibook.org`.

[Ye et al., 2021] Ye, W., Liu, S., Kurutach, T., Abbeel, P., and Gao, Y. (2021). Mastering atari games with limited data. *arXiv preprint arXiv:2111.00210*.

[Yu et al., 2021] Yu, T., Kumar, A., Rafailov, R., Rajeswaran, A., Levine, S., and Finn, C. (2021). Combo: Conservative offline model-based policy optimization. *arXiv preprint arXiv:2102.08363*.

[Zhou et al., 2020a] Zhou, M., Liu, Z., Sui, P., Li, Y., and Chung, Y. Y. (2020a). Learning implicit credit assignment for cooperative multi-agent reinforcement learning. *arXiv preprint arXiv:2007.02529*.

[Zhou et al., 2020b] Zhou, P., Chen, X., Liu, Z., Braud, T., Hui, P., and Kangasharju, J. (2020b). Drle: Decentralized reinforcement learning at the edge for traffic light control in the iov. *IEEE Transactions on Intelligent Transportation Systems*, 22(4):2262–2273.

[Zook et al., 2015] Zook, A., Harrison, B., and Riedl, M. O. (2015). Monte-carlo tree search for simulation-based play strategy analysis. In *FDG*.