ABSTRACT

Title of Dissertation: MULTI-AGENT REINFORCEMENT LEARNING: SYSTEMS FOR EVALUATION AND APPLICATIONS TO COMPLEX SYSTEMS

Jordan Terry
Doctor of Philosophy, 2023

Dissertation Directed by: John Dickerson
Department of Computer Science

Reinforcement learning is a field of artificial intelligence that studies methods for agents to learn by trial and error to take actions in a given system. Famous examples of it have included learning to control real robots, or achieving superhuman performance in most of the most popular and challenging games for humans.

In order to conduct research in this space, researchers use standardized "environments", such as robotics simulations or video games, to evaluate the performance of learning methods. This thesis covers PettingZoo, a library that offers a standardized API and set of reference environments for multi-agent reinforcement learning that's become widely used, SuperSuit, a library that offers a easy-to-use standardized preprocessing wrappers for interfacing with learning libraries, and extensions to the Arcade Learning Environment (a popular tool which reinforcement learning researchers use to interact with Atari 2600 games) that allows for supporting multiplayer game modes.

Using these tools, this thesis also uses multi-agent reinforcement learning to develop a new tool for natural science research. Emergent behaviors refer to the coordinated behaviors of groups of agents such as pedestrians in a crosswalk, birds in flocking formations, cars in traffic or traders in the stock market, and represent some of the most important things that we generally don't understand across many fields of science. In this work, we introduce the first mathematical formalism for the systematic search of all possible good ("mature") emergent behaviors within a multi-agent system through multi-agent reinforcement learning (MARL), and create a naive implementation of this search via deep reinforcement learning that can be applied in arbitrary environments. We show that in 12 multi-agent systems, this naive method is able to find over a hundred total emergent behaviors, the majority of which were previously unknown to the environment authors. Such methods could allow for answering various types of open scientific questions, such as "What behaviors are possible in this system", "What specific conditions in this system allow for this kind of emergent behavior", or "How can I change this system to prevent this emergent behavior."

MULTI-AGENT REINFORCEMENT LEARNING: SYSTEMS FOR EVALUATION AND APPLICATIONS TO COMPLEX SYSTEMS

by

Jordan Terry

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor John Dickerson, Chair/Advisor
Professor Dinesh Manocha
Professor Furong Huang
Professor Irina Rish
Professor Daniel Lathrop

Acknowledgments

I would like to express my gratitude to my advisor, John Dickerson, for his guidance and support throughout my PhD journey, without which this would not have been possible.

I would also like to thank Dan Lathrop for his advise and support over a very long period of time – I truly could not have done this without you.

I would also like to thank my collaborators on the works included in my thesis, most notably Jun Jet Tai, Benjamin Black, Nathan Grammel and Ariel Kwiatkowski, for the massive effort they've collectively put into all these projects.

Much of this work has become a key portion of those in the Farama Foundation, and I will be forever grateful to the massive number of community members in it who have contributed to these tools since their creation, as well as those who use my projects and who have promoted their adoption.

I would also like to thank QinetiQ for supporting a portion of my PhD with the QinetiQ Fundamental Machine Learning Fellowship.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AEC	Agent Environment Cycle
ALE	Arcade Learning Environment
API	Application Programming Interface
DRL	Deep Reinforcement Learning
EFG	Extensive Form Game
MADRL	Mult-Agent Deep Reinforcement Learning
MARL	Multi-Agent Reinforcement Learning
MCMES	Monte Carlo Mature Emergence Search
MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
POSG	Partially Observable Stochastic Game
RL	Reinforcement Learning

Chapter 1: Overview and Background

1.1 Reinforcement Learning

The field of Reinforcement Learning ("RL") studies using machine learning to optimally control a system, largely originating from classical control tasks such as elevator management (e.g. Crites and Barto [1998]) or wing flap control on an airplane (e.g. Monaco et al. [1997]). It is premised on the model that an agent exist in a state s of the universe or a simulation (generally referred to as an *environment*). Observation o is then taken from the state, which may not include all information present. Then, a *policy function* π outputs an action a to be executed in the environment. Following this the state of the environment updates, emits a dimensionless *reward*, and a new observation is taken. The cycle then repeats. This process is formalized in the Partially Observable Markov Decision Process, illustrated in Figure 1.1 and explained in depth to a broad scientific audience in Littman [2009].

The policy function is vaguely analogous to a "brain" as it chooses how to act in an environment, and it is usually represented by a neural network (though this is not required). Various methods exist to iteratively update a policy learned is by iteratively updating itself to maximize to the equation $E_\pi[r_0 + \gamma r_1 + \gamma^2 r_2 + ...]$, where γ is a discount factor very close to 1 that incentivizes taking actions early. Examples of reward functions are 1, 0, or -1 for winning, drawing or loosing a game of chess, or the amount of money made or lost but an agent trading stocks, though

in cases like robotic control reward functions can become very complex. For those interested, the standard introductory materials to the field of reinforcement learning are Silver [2015], Sutton and Barto [2018], Achiam [2018].

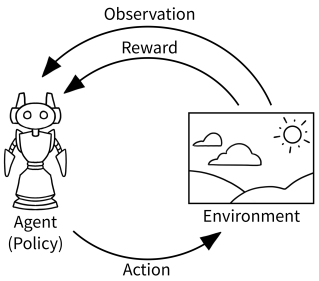


Figure 1.1: An illustration of the process by which an agent interacts with an environment in single-agent reinforcement learning.

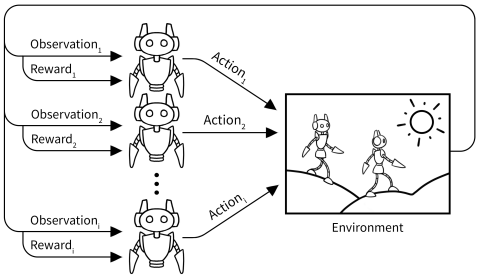


Figure 1.2: An illustration of the process by which multiple agents interact with an environment in multi–agent reinforcement learning.

1.2 Multi-Agent Reinforcement Learning

Multi-agent reinforcement learning studies optimally controlling individual agents in environments where there are multiple agents. These range from cooperative real world tasks like coordinating robots in warehouses to abstract competitive tasks like learning to play Chess or Go, with a spectrum in between. In the most general case, multi-agent reinforcement learning functions the same as single-agent reinforcement learning, except that the evolution of the environment depends on the actions of all agents. In other words each agent may have entirely separate policy functions, reward functions and observations. This allows for fully cooperative and competitive scenarios and everything in between. It is also worth noting that there two variants of multi-agent reinforcement learning: centralized policies, where one single instructs all agents to act at once, and decentralized policies, where each agent has it's own policies. The field

of MARL generally focuses on decentralized learning, and for the purposes of this paper we will exclusively focus on that form. Similar to the POMDP model, the problem of MARL is most often formalized as a Partially Observable Stochastic Games, explained to a general audience in [Hansen et al., 2004, Littman, 1994], and illustrated in Figure 1.1. For those looking to learn more about this, we recommend [Filar and Vrieze, 2012].

1.3 Emergence and Complex Systems

Many of the most fundamental and unsolved questions across different fields of science center around understanding group behaviors that arise from interactions between its constituents. Examples include ”how did human language arise?”, ”what will the economy do next year?”, “how do swarms of animals function, and how was the swarming behavior learned?”, "Why does the El Niño–Southern Oscillation occur and how does it influence hurricane formation in the West Atlantic?" and “why do specific fashion trends spread amongst certain social groups?” Countless far less profound problems in this space exist too, such as under human movements (especially during disasters) for better city planning, understanding why specific strategies in multiplayer computer games spread to popular use, or understanding why birds across North America have been replacing an old song with a new one over the past decade. Progress towards understanding these problems has generally been challenging due to inherent difficulties, such as how system complexity limits the capacity of conventional formal modeling and the inability to isolate and study individual system components.

Scientists who study these systems generally refer to them as *complex systems*- systems with large numbers of components, generally agents or physical bodies, that interact with each

other. Even when the components of these systems follow simple rules, the systems generally have the remarkable ability to produce complex and previously unknown behaviors with no apparent centralized planning. *Emergence* refers to this sort of collective phenomena in general, whereas *emergent behavior* typically refers to emergence that occurs in groups of agents with some level of cognitive capacity, e.g. ants. Another notable hallmark is that their evolution from one state to another is generally nonlinear, e.g. in $f(s_t) \rightarrow s_{t+1}$, where s is the state of a system at a given time step t, f is generally a nonlinear function. This nonlinearity greatly increases the number of potential different behaviors of the system, and generally makes predicting the exact progressions over time practically impossible.

A simple example of emergence most are unknowingly well acquainted with is how pedestrians will walk on one side of the escalator and stand on the other in most major cities across the world. This scenario illustrates many of the key properties of emergence—it is a reproducible behavior that does not necessarily occur (e.g. this has not emerged in all cities with escalators), it is a purely group behavior that cannot be easily inferred simply by watching the actions of the individual, it's purely decentralized and has no controller, the precise reason it does or does not occur in a given city is likely very intricate, it's seemingly an orderly behavior and the precise behavior can change depending on the number of agents (e.g. no elevator splitting occurs with very few pedestrians, at a certain point "escalator spliting" occurs, and with an even greater number of pedestrians a complete gridlock ensues). Surprisingly, this escalator splitting a rather well studied emergent behavior because it is greatly frowned upon by escalator manufacturers and owners, as the uneven load greatly increases wear and the lower density of the walking half the escalator reduces capacity during peak load. Because of all those properties of the system, it's obvious that studying why this occurs and how to change it are fairly nontrivial, which is why this is still a

common behavior we all experience despite all concerned parties working to discourage it.

For those who wish to read more about this space at an introductory level, a review of the general field of complex systems science is found in Siegenfeld and Bar-Yam [2020], an brief overview the role of emergent behavior throughout science is found in Pines [2014] while a longer one can be found in Kauffman [1996], and a historical introduction to chaos theory and nonlinear dynamics (the study of any system with nonlinear updates) is found in Gleick [2008].

Chapter 2: PettingZoo: A Standard API for Multi-Agent Reinforcement Learning

2.1 Introduction

Multi-Agent Reinforcement Learning (MARL) has been behind many of the most publicized achievements of modern machine learning — AlphaGo Zero [Silver et al., 2017], OpenAI Five [OpenAI, 2018], AlphaStar [Vinyals et al., 2019]. These achievements motivated a boom in MARL research, with Google Scholar indexing 9,480 new papers discussing multi-agent reinforcement learning in 2020 alone. Despite this boom, conducting research in MARL remains a significant engineering challenge. A large part of this is because, unlike single agent reinforcement learning which has OpenAI's Gym, no de facto standard API exists in MARL for how agents interface with environments. This makes the reuse of existing learning code for new purposes require substantial effort, consuming researchers' time and preventing more thorough comparisons in research. This lack of a standardized API has also prevented the proliferation of learning libraries in MARL. While a massive number of Gym-based single-agent reinforcement learning libraries or code bases exist (as a rough measure 669 pip-installable packages depend on it at the time of writing GitHub [2021]), only 5 MARL libraries with large user bases exist [Lanctot et al., 2019, Weng et al., 2020, Liang et al., 2018, Samvelyan et al., 2019, Nota, 2020].

The proliferation of these Gym based learning libraries has proved essential to the adoption of applied RL in fields like robotics or finance and without them the growth of applied MARL is a significantly greater challenge. Motivated by this, this paper introduces the PettingZoo library and API, which was created with the goal of making research in MARL more accessible and serving as a multi-agent version of Gym.

Prior to PettingZoo, the numerous single-use MARL APIs almost exclusively inherited their design from the two most prominent mathematical models of games in the MARL literature—Partially Observable Stochastic Games ("POSGs") and Extensive Form Games ("EFGs"). During our development, we discovered that these common models of games are not conceptually clear for multi-agent games implemented in code and cannot form the basis of APIs that cleanly handle all types of multi-agent environments.

To solve this, we introduce a new formal model of games, Agent Environment Cycle ("AEC") games that serves as the basis of the PettingZoo API. We argue that this model is a better conceptual fit for games implemented in code. and is uniquely suitable for general MARL APIs. We then prove that any AEC game can be represented by the standard POSG model, and that any POSG can be represented by an AEC game. To illustrate the importance of the AEC games model, this paper further covers two case studies of meaningful bugs in popular MARL implementations. In both cases, these bugs went unnoticed for a long time. Both stemmed from using confusing models of games, and would have been made impossible by using an AEC games based API.

The PettingZoo library can be installed via `pip install pettingzoo`, the documentation is available at `https://www.pettingzoo.ml`, and the repository is available at `https://github.com/Farama-Foundation/PettingZoo`. This section is based on

work originally published in Terry et al. [2021].

2.2 Background and Related Works

Here we briefly survey the state of modeling and APIs in MARL, beginning by briefly looking at Gym's API (Figure 2.1). This API is the de facto standard in single agent reinforcement learning, has largely served as the basis for subsequent multi-agent APIs, and will be compared to later.

```
import gym
env = gym.make('CartPole-v0')
observation = env.reset()
for _ in range(1000):
    action = policy(observation)
    observation, reward, done, info = env.step(action)
```

Figure 2.1: An example of the basic usage of Gym

```
from ray.rllib.examples.env.multi_agent
    import MultiAgentCartPole
env = MultiAgentCartPole()
observation = env.reset()
for _ in range(1000):
    actions = policies(agents, observation)
    observation, rewards, dones,
        infos = env.step(actions)
```

Figure 2.2: An example of the basic usage of RLlib

The Gym API is a fairly straightforward Python API that borrows from the POMDP conceptualization of RL. The API's simplicity and conceptual clarity has made it highly influential, and it naturally accompanying the pervasive POMDP model that's used as the pervasive mental and mathematical model of reinforcement learning [Brockman et al., 2016]. This makes it easier for anyone with an understanding of the RL framework to understand Gym's API in full.

2.2.1 Partially Observable Stochastic Games and RLlib

Multi-agent reinforcement learning does not have a universal mental and mathematical model like the POMDP model in single-agent reinforcement learning. One of the most popular models is the partially observable stochastic game ("POSG"). This model is very similar to, and strictly more general than, multi-agent MDPs [Boutilier, 1996], Dec-POMDPs [Bernstein

et al., 2002], and Stochastic ("Markov") games [Shapley, 1953]). In a POSG, all agents step together, observe together, and are rewarded together. The full formal definition is presented in Appendix A.1

This model of simultaneous stepping naturally translates into Gym-like APIs, where the actions, observations, rewards, and so on are lists or dictionaries of individual values for agents. This design choice has become the standard for MARL outside of strictly turn-based games like poker, where simultaneous stepping would be a poor conceptual fit [Lowe et al., 2017, Zheng et al., 2018, Gupta et al., 2017, Liu et al., 2019, Liang et al., 2018, Weng et al., 2020]. One example of this is shown in Figure 2.2 with the multi-agent API in RLlib [Liang et al., 2018], where agent-keyed dictionaries of actions, observations and rewards are passed in a simple extension of the Gym API.

This model has made it much easier to apply single agent RL methods to multi-agent settings. However, there are two immediate problems with this model:

1. Supporting strictly turn-based games like chess requires constantly passing dummy actions for non-acting agents (or using similar tricks).

2. Changing the number of agents for agent death or creation is very awkward, as learning code has to cope with lists suddenly changing sizes.

2.2.2 OpenSpiel and Extensive Form Games

In the cases of strictly turn based games where POSG models are poorly suited (e.g. Chess), MARL researchers generally mathematically model the games as Extensive Form Games ("EFG"). The EFG represents games as a tree, *explicitly* representing every possible sequence of

actions as a root to leaf path in the tree. Stochastic aspects of a game (or MARL environment) are captured by adding a "Nature" player (sometimes also called "Chance") which takes actions according to some given probability distribution. For a full definition of EFGs, we refer the reader to Osborne and Rubinstein [1994] or section A.2. OpenSpiel [Lanctot et al., 2019], a major library with a large collection of classical board and card games for MARL bases their API off of the EFG paradigm, the API of which is shown in Figure 2.3.

```
import pyspiel
import numpy as np

game = pyspiel.load_game("kuhn_poker")
state = game.new_initial_state()
while not state.is_terminal():
  if state.is_chance_node():
    # Step the stochastic environment.
    action_list, prob_list = zip(*state.chance_outcomes())
    state.apply_action(np.random.choice(action_list, p=prob_list))
  else:
    # sample an action for the agent
    legal_actions = state.legal_actions()
    observations = state.observation_tensor()
    action = policies(state.current_agent(), legal_actions, observations)
    state.apply_action(action)
    rewards = state.rewards()
```

Figure 2.3: An example of the basic usage of OpenSpiel

The EFG model has been successfully used for solving problems involving theory of mind with methods like game theoretic analysis and tree search. However, for application in general MARL problems, three immediate concerns arise with the EFG model:

1. The model, and the corresponding API, is very complex compared to that of POSGs, and isn't suitable for beginners the way Gym is—this environment API is much more complicated than Gym's API or RLLib's POSG API for example. Furthermore, due to the complexity of the EFG model, reinforcement learning researchers don't ubiquitously use it as a mental model of games in the same way that they use the POSG or POMDP model.

2. The formal definition only includes rewards at the end of games, while reinforcement learning often requires frequent rewards. While this is possible to work around in the API implementation, it is not ideal.

3. The OpenSpiel API does not handle continuous actions (a common and important case in RL), though this was a choice that is not inherent to the EFG model.

It's also worth briefly noting that some simple strictly turn based games are modeled with the single-agent Gym API, with the environment alternating which agent is controlled, [Ha, 2020]. This approach is unable to reasonably scale beyond two agents due to the difficulties of handling changes in agent order (e.g. Uno), agent death, and agent creation.

2.3 PettingZoo Design Goals

Our development of PettingZoo both as a general library and an API centered around the following goals.

2.3.1 Be like Gym

In PettingZoo, we wanted to leverage Gym's ubiquity, simplicity and universality. This created two concrete goals for us:

- Make the API look and feel like Gym, and relatedly make the API pythonic and simple
- Include numerous reference implementations of games with the main package

Reusing as many design metaphors from Gym as possible will help its massive existing user base to almost instantly understand PettingZoo's API. Similarly, for an API to become standard-

ized, it must support a large collection of useful environments to attract users and for adoption to begin, similar to what Gym did.

2.3.2 Be a Universal API

If there is to be a Gym-like API for MARL, it has to be able to support all use cases and types of environments. Accordingly, several technically difficult cases exist that have to be carefully considered:

- Environments with large numbers of agents
- Environments with agent death and creation
- Environments where different agents can be chosen to participate in each episode
- Learning methods that require access to specialty low level features

Two related softer design goals for universal design are ensuring the API is simple enough for beginners to easily use, and making the API easily changeable if the direction of research in the field dramatically changes.

2.4 Case Studies of Problems With The POSG Model in MARL

To supplement the description of the problems with the POSG models described in subsection 2.2.1, we overview problems with basing APIs around these models that could theoretically occur in software games, and then examine real cases of those problems occurring in popular MARL environments. We specifically focus on POSGs here because EFG based APIs

are extraordinarily rare (OpenSpiel is the only major one), while POSG based ones are almost universal.

2.4.1 POSGs Don't Allow Access To Information You Should Have

Another problem with modeling environments using simultaneous actions in the POSG model is that all of an agent's rewards (from all sources) are summed and returned all at once. In a multi-agent game though, this combined reward is often the composite reward from the actions of other agentss and the environment. Similarly, you might want to be able to attribute the source of this reward for various learning reasons, or for debugging purposes to find out the origin of your rewards. However, in thinking about reward origins, having all rewards emitted at once proves to be very confusing because rewards from different sources are all combined. Accessing this information via an API modeled after a POSG requires deviating from the model. This would come in the form of returning a 2D array of rewards instead of a list, which would be difficult to standardize and inconvenient for learning code to parse.

A notable case where this caused an issue in practice is in the popular pursuit gridworld environment from Gupta et al. [2017], shown in Figure 2.4. In it, 8 red controllable pursuer must work together to surround and capture 30 randomly moving blue evaders. The action space of each pursuer is discrete (cardinal directions or do nothing), and the observation space is a 7×7 box centered around a pursuer (depicted by the orange box). When an evader is surrounded on all sides by pursuers or the game boundaries, each contributing pursuer gets a reward of 5.

In pursuit, pursuers move first, and then evaders move randomly, before it's determined if an evader is captured and rewards are emitted. Thus an evader that "should have" been captured

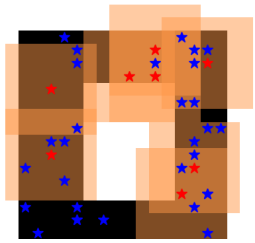
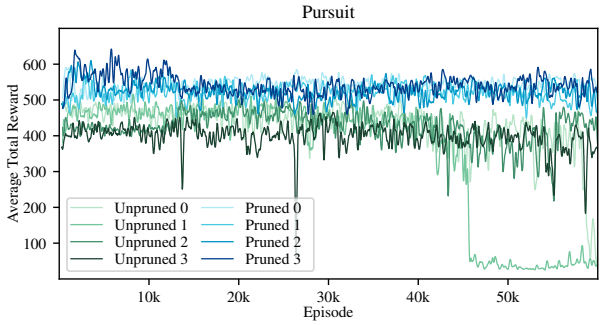


Figure 2.4: The *pursuit* environment from Gupta et al. [2017].

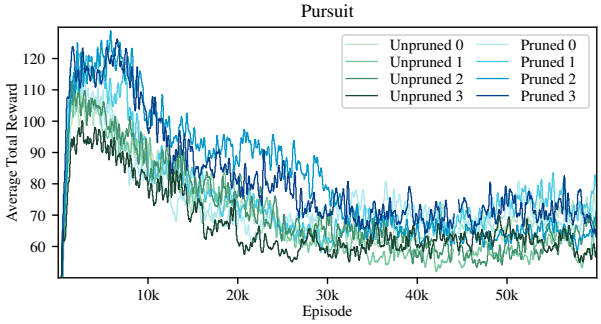
is not actually captured. Having the evaders move second isn't a bug, it's just way of adding complexity to the classic genre of pursuer/evader multi-agent environments [Vidal et al., 2002], and is representative of real problems. When *pursuit* is viewed as an AEC game, we're forced to attribute rewards to individual steps, and the breakdown becomes pursuers receiving deterministic rewards from surrounding the evader, and then random reward due to the evader moving after. Removing this random component of the reward (the part caused by the evaders action after the pursuers had already moved), should then lead to superior performance. In this case the problem was so innocuous that fixing it required switching two lines of code where their order made no obvious difference.

We validated the impact of reward pruning experimentally showing that this change resulted in up to a 22% performance improvement by training parameter shared Ape-X DQN [Horgan et al., 2018] (the best performing model on pursuit [Terry et al., 2020a]) four times using RLLib [Liang et al., 2018] with and without reward pruning, achieving better results with reward pruning every time and 22.03% more total reward on average Figure 2.5a, while PPO [Schulman et al., 2017] learned 16.12% more reward on average with this Figure 2.5b. Saved training logs and all code needed to reproduce the experiments and plots is available in the supplemental materials.

Bugs of this family could easily happen in almost any MARL environment, and analyzing



(a) Learning on the *pursuit* environment with and without pruned rewards, using parameter sharing based on Ape-X DQN. This shows an average of an 22.03% improvement by using this method.



(b) Learning on the *pursuit* environment with and without reward pruning, using parameter sharing based on PPO. Reward pruning increased the total reward by 16.12% on average.

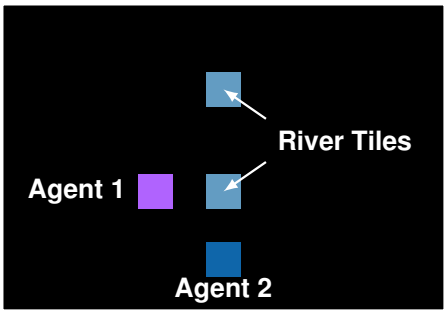
and preventing them is made much easier when using the POSG model. Because every agent's rewards are summed together in the POSG model, this specific problem when looking at the code was extraordinarily non-obvious, whereas when forced to attribute the reward of individual agents this becomes clear. Moreover if an existing environment had this problem, by exposing the actual sources of rewards to learning code researchers are able to remove differing sources of reward to more easily find and remove bugs like this, and in principle learning algorithms could be developed that automatically differently weighted different sources of reward.

2.4.2 POSGs Based APIs Are Not Conceptually Clear For Games Implemented In Code

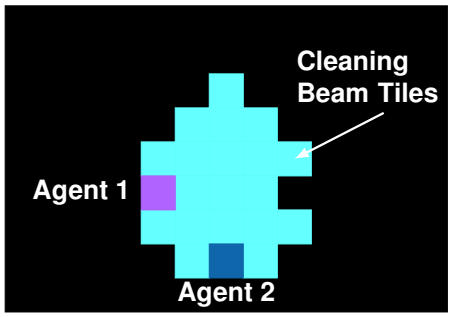
Introducing race conditions is a very easy mistake to make in MARL code in practice, and this occurs because simultaneous models of multi-agent games are not representative of how game code normally executes. This stems from a very common scenario in multi-agent environments where two agents are able to take conflicting actions (i.e. moving into the same space). This discrepancy has to be resolved by the environment (i.e. collision handling); which we call "tie-breaking."

Consider an environment with two agents, Alice and Bob, in which Alice steps first and tie-breaking is biased in Alice's favor. If such an environment were assumed to have simultaneous actions, then observations for both agents would be taken before either acted, causing the observation Bob acts on to no longer be an accurate representation of the environment if a conflict with biased tie-breaking occurs. For example, if both agents tried to step into the same square and Alice got the square because she was first in the list, Bob's observation before acting was effectively inaccurate and the environment was not truly parallel. This behavior is a true race condition—the result of stepping through the environment can inadvertently differ depending on the internal resolution order of agent actions.

In any environment that's even slightly complex, a tremendous number of instances where tie-breaking must be handled will typically occur. In any cases where a single one is missed, the environment will have race conditions that your code will attempt to learn. While finding these will always be important, a valuable tool to mitigate these possibilities is to use an API that treats each agent as acting sequentially, returning new observations afterwards. This entirely prevents



(a) The initial setup with two agents and two river tiles. When the river tiles become dirty, they are shown as a brownish color instead.

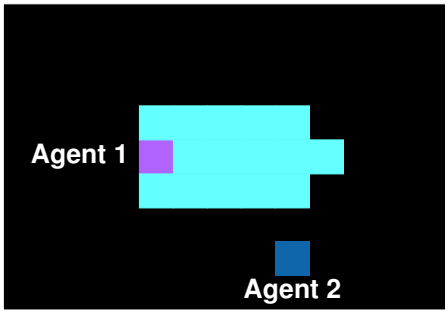


(b) The result of both agents perform the "clean" action. Both river tiles can be are cleaned since Agent 1's action is resolved first.

Figure 2.6: Cleanup, a Sequential Social Dilemma Game from Vinitsky et al. [2019].

the opportunity for introducing race conditions. Moreover, this entire problem stems from the fact that using APIs that model agents as updating sequentially for software MARL environments generally makes more conceptual sense than modeling the updates as simultaneous—unless the authors of environments use very complex parallelization, the environments will *actually* be updated one agent at a time. It is worth mentioning that this race condition cannot occur in an environment simulated in the physical world with continuous time or a simulated environment with a sufficient amount of observation delay (though most actively researched environment in MARL do not currently have any observation delay).

The Sequential Social Dilemma Games, introduced in Leibo et al. [2017], are a populer set of MARL environment where good short-term strategies for single agents lead to bad long-term results for all of the agents. New SSD environments, including the *Cleanup* environment, were introduced in Hughes et al. [2018]. All of these have open source implementations in [Vinitsky et al., 2019]. The states of these games are represented by a grid of tiles, where each tile represents either an agent or a piece of the environment. In the *Cleanup* environment, the environment tiles can be empty tiles, river tiles, and apple tiles. Collecting apple tiles results in a reward for the



(a) If there are no dirty river tiles in the path of the cleaning beams, the beams will extend to the full length of five tiles.

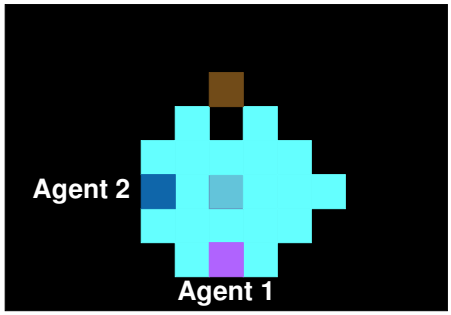
(b) If there is a dirty river tile in the path of a beam, the beam will stop at the tile, changing it to a "clean" river tile.

Figure 2.7: An example of Agent 1 using the "clean" action while facing East. The beams extend to a length of up to five tiles. The "main" beam extends directly in front of the agent, while two auxiliary beams start at the tiles directly next to the agent (one to the left and one to the right) and also extend up to five tiles. A beam stops when it hits a dirty river tile.

agent and the agents must clean the river tiles with a "cleaning beam" for apple tiles to spawn. The cleaning beam extends in front of agents, one tile at a time, until it hits a dirty river tile ("waste") or extends to its maximum length of 5 tiles. Additionally, two more beams extend in front of the agent—one starting in the tile directly to the agent's left, and one from the tile on the right—until each hits a "waste" tile or reaches a length of 5 tiles. The cleaning beam is shown in Figure 2.7a. Note that while beams stop at "waste" tiles, they will continue to extend past clean river tiles.

The agents act sequentially in the same order every turn, including the firing of their beams. In the case of two agents trying to occupy the same space, one is chosen randomly, however the tie breaking with regards to the beams is biased, due to a bug. Consider the setup in Figure 2.6 where each agent chooses the "clean" action for the next step. This results in Agent 1 firing their cleaning beam first, clearing the close river tile. Next, Agent 2 fires their cleaning beam and they are able to clean the far river tile because the close tile has already been cleared by Agent 1. However, if we keep the same placement and actions but switch the labels of the agents, we get a

(a) The same setup as in Figure 2.6, but with the agent labels reversed.



(b) The result of both agents performing the "clean" action, with this agent assignment.

Figure 2.8: The impact of switching the internal agent order on how the environment evolves. When both agents clean, agent 1's action is resolved first, and the main beam stops when it hits the near dirty river tile, so the far river tile is not cleaned. In Figure 2.6, Agent 2's beam was able to reach the far beam because Agent 1's beam cleaned the near tile first.

different result, seen in Figure 2.8. Now, Agent 1 fires first and hits the close river tile and can no longer reach the far river tile. In situations like these, the observation the second agent's policy is using to act on is going to be inherently wrong, and if it had the true environment state before acting it would very likely wish to make a different choice.

This is a serious class of bug that's very easy to introduce when using parallel action-based APIs, while using AEC games-based APIs prevents the class entirely. In this specific instance, the bug had gone unnoticed for years.

2.5 The Agent Environment Cycle Games Model

Motivated by the problems with applying the POSG and EFG models to MARL APIs, we developed the Agent Environment Cycle ("AEC") Game. In this model, agents sequentially see their observation, agents take actions, rewards are emitted from the other agents, and the next agent to act is chosen. This is effectively a sequentially stepping form of the POSG model.

Modeling multi-agent environments sequentially for APIs has numerous benefits:

- It allows for clearer attribution of rewards to different origins, allowing for various learning improvements, as described in subsection 2.4.1.
- It prevents developers adding confusing and easy-to-introduce race conditions, as described in subsection 2.4.2.
- It more closely models how computer games are executed in code, as described in subsection 2.4.2.
- It formally allows for rewards after every step as is required in RL, but is not generally a part of the EFG model, as discussed in subsection 2.2.2.
- It is simple enough to serve as a mental model, especially for beginners, unlike the EFG model as discussed in subsection 2.2.2 and illustrated in the definition in section A.2.
- Changing the number of agents for agent death or creation is less awkward, as learning code does not have to account for lists constantly changing sizes, as discussed in subsection 2.2.1.
- It is the least bad option for a universal API, compared to simultaneous stepping, as alluded to in subsection 2.2.1. Simultaneous stepping requires the use of no-op actions if not all agents can act which are very difficult to deal with, whereas sequentially stepping agents that could all act simultaneously and queuing up their actions is not especially inconvenient.

One additional conceptual feature of the AEC games model exists that we have not previously discussed because it does not usually play a role in APIs (see subsection 2.6.4). In the AEC games model, we deviate from the POSG model by introducing the "environment" agent,

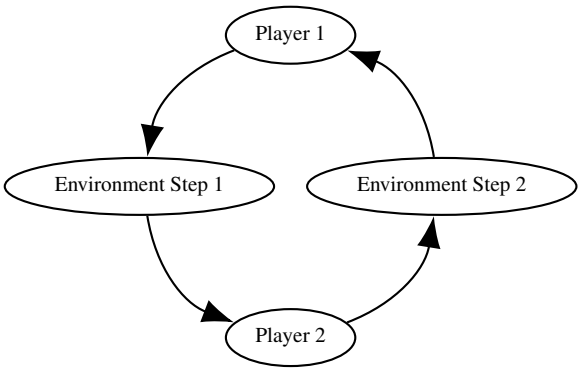


Figure 2.9: The AEC diagram of Chess

which is analogous to the Nature agent from EFGs. When this agent acts in the model it indicates the updating of the environment itself, realizing and reacting to submitting agent actions. This allows for a more comprehensive attribution of rewards, causes of agent death, and discussion of games with strange updating rules and race conditions. An example of the transitions for Chess is shown in Figure 2.9, which serves as the inspiration for the name "agent environment cycle".

2.5.1 Formalizing The AEC Games Model

As mentioned in Section 2.5, the stochastic nature of the state transitions is modeled as an "environment" agent, which does not take an action but rather transitions randomly from the current state to a new state according to some given probability distribution. With the stochasticity of state transitions separated out as a distinct "environment" agent, we can then model the transitions of the actual agents deterministically. To this end, each (non-environment) agent i has a deterministic transition function T_i which depends only on the current state and the action taken, while the environment has a stochastic transition function P which transitions to a new state randomly depending on the current state (it may depend on the actions taken previously by the agents, since the current state is determined by these actions).

Definition 1. Formally, an *Agent-Environment Cycle Game* (AEC Game) is defined by a tuple $\langle \mathcal{S}, s_0, N, (\mathcal{A}_i)_{i\in[N]}, (T_i)_{i\in[N]}, P, (\mathcal{R}_i)_{i\in[N]}, (R_i)_{i\in[N]}, (\Omega_i)_{i\in[N]}, (O_i)_{i\in[N]}, \nu \rangle$, where:

- $\mathcal{S}$ is the set of possible *states*.
- s_0 is the *initial state*.
- N is the *number of agents*. The agents are numbered 1 through N. There is also an additional "environment" agent, denoted as agent 0. We denote the set of agents along with the environment by $\mathcal{N} := [N] \cup \{0\}$.
- $\mathcal{A}_i$ is the set of possible *actions* for agent i. For convenience, we further define $\mathcal{A}_0 = \{\varnothing\}$ (i.e., a single "null action" for environment steps) and $\mathcal{A} := \bigcup_{i\in\mathcal{N}} \mathcal{A}_i$.
- $T_i \colon \mathcal{S} \times \mathcal{A}_i \to \mathcal{S}$ is the *transition function for agents*. State transitions for agent actions are deterministic.
- $P \colon \mathcal{S} \times \mathcal{S} \to [0, 1]$ is the *transition function for the environment*. State transitions for environment steps are stochastic: $P(s, s')$ is the probability that the environment transitions into state s' from state s.
- $\mathcal{R}_i \subseteq \mathbb{R}$ is the set of possible rewards for agent i. We assume this is *finite*.
- $R_i \colon \mathcal{S} \times \mathcal{N} \times \mathcal{A} \times \mathcal{S} \times \mathcal{R}_i \to [0, 1]$ is the *reward function* for agent i. $\mathcal{R}_i \subseteq \mathbb{R}$ denotes the set of all possible rewards for agent i (which we assume to be finite).

  R_i is the *reward function* for agent i. The set of all possible rewards for each agent is assumed to be finite, which we denote $\mathcal{R}_i \subseteq \mathbb{R}$. It is *stochastic*: $R_i(s, j, a, s', r)$ is the

probability of agent i receiving reward r when agent j takes action a while in state s, and the game transitions to state s'. We also define $\mathcal{R} := \bigcup_{i \in [N]} \mathcal{R}_i$.

- Ω_i is the set of possible *observations* for agent i.
- $O_i \colon \mathcal{S} \times \Omega_i \to [0, 1]$ is the *observation function* for agent i. $O_i(s, \omega)$ is the probability of agent i observing ω while in state s.
- $\nu \colon \mathcal{S} \times \mathcal{N} \times \mathcal{A} \times \mathcal{N} \to [0, 1]$ is the *next agent* function. This means that $\nu(s, i, a, j)$ is the probability that agent j will be the next agent permitted to act given that agent i has just taken action a in state s. This should attribute a non-zero probability only when $a \in \mathcal{A}_i$.

In this definition, the game starts in state s_0 and the environment agent acts first. Having the environment agent act first allows the first actual agent to act to be determined randomly if desired (choosing the first agent deterministically can be done easily by having the environment simply do nothing in this first step). The game then evolves in "turns" where in each turn an agent i receives an observation $\omega_i \in \Omega_i$ (any given observation ω is seen with probability $O_i(s, \omega)$) and, based on this observation, chooses an action $a_i \in \mathcal{A}_i$. The game then transitions from the current state s to a new state s' according to the transition function. If $i \in [N]$, the state transition is deterministically $T_i(s, a_i)$. If $i = 0$, the new state is stochastic, so state s' occurs with probability $P(s, s')$. Then, a new agent i' is determined according to the "next agent" function, so that i' is next to act with probability $\nu(s, i, a_i, i')$. The observation ω_i that is received is random, occurring with probability $O_i(s, \omega_i)$. Note that we can allow for the state to transition randomly in response to an agent's action by simply inserting an "environment step" immediately following an agent's action, by setting $\nu(s, i, a_i, 0) = 1$ and allowing the following environment step to transition

the state randomly. At every step, every agent j receives the partial reward r' with probability $R_j(s, i, a_i, s', r')$.

2.5.2 Proof that POSGs are Equivalent to AEC Games

The inclusion of the stochastic ν (next-agent) function in the definition of AEC games allows for capturing many turn-based games with complex turn orders (consider Uno, for instance, where players may be skipped or the order reversed). It is not immediately obvious that this allows for representing games in which agents act simultaneously. However, we show here that in fact AEC games can be used to theoretically model games with simultaneous actions.

To see this, imagine simulating a POSG by way of a "black box" which takes the actions of all agents simultaneously, and then — one by one — feeds them to a purpose-built AEC game whose states are designed to "encode" each agent's action, "queueing" them up over the course of N steps (one for each agent). Once all of the actions have been fed to the AEC game, a single environment step resolves these "queued up" actions all at once. If we design the AEC game in the right way, this total of $N + 1$ steps (N for queueing the actions, and one for the environment to resolve the joint action) produces an outcome that is identical to the result of a single step in the original POSG. This is formalized below.

Theorem 1. *For every POSG, there is an equivalent AEC Game.*

Proof of Theorem 1. Let $G = \langle \mathcal{S}, N, \{\mathcal{A}_i\}, P, \{R_i\}, \{\Omega_i\}, \{O_i\} \rangle$ be a POSG. To prove this, it will be necessary to show precisely what is meant by "equivalent." We will construct a new AEC Game G_{AEC} in such a way that for every $N + 1$ steps of G_{AEC} the probability distribution over possible states is identical to the state distribution for G after a single step, the distributions over

observations received by each agent is identical in G and in G_{AEC}, and the reward obtained by each agent is the same.

We define G_{AEC} as follows:

$$G_{\mathrm{AEC}} = \langle \mathcal{S}', N, \{\mathcal{A}_i\}, \{T_i\}, P', \{R'_i\}, \{\Omega_i\}, \{O'_i\}, \nu \rangle$$

where

- $\mathcal{S}' = \mathcal{S} \times \mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_N$. That is, an element of $\mathcal{S}'$ is a tuple $(s, a_1, a_2, \ldots, a_N)$ where $s \in \mathcal{S}$ and for each $i \in [N]$, $a_i \in \mathcal{A}_i$.
- $T_i((s, a_1, a_2, \ldots, a_i, \ldots, a_N), a'_i) = (s, a_1, a_2, \ldots, a'_i, \ldots, a_N)$.
- Define $P'(\mathbf{s}, \mathbf{s}') = P(s, a_1, a_2, \ldots, a_N, s')$ for each $\mathbf{s} = (s, a_1, a_2, \ldots, a_N)$ and $\mathbf{s}' = (s', a_1, a_2, \ldots, a_N)$. If $\mathbf{s}$ and $\mathbf{s}'$ are such that $a_i \neq a'_i$ for any $i \in [N]$, then $P'(\mathbf{s}, \mathbf{s}') = 0$.
- For $\mathbf{s} = (s, a_1, a_2, \ldots, a_N)$, $\mathbf{s}' = (s', a_1, a_2, \ldots, a_N)$, and $\mathbf{r} = R_i(s, a_1, a_2, \ldots, a_N, s')$, we let $R'_i(\mathbf{s}, 0, \varnothing, \mathbf{s}', \mathbf{r}) = 1$. We define $R'_i = 0$ for all other cases.
- $O'_i(s, a_1, a_2, \ldots, a_N) = O_i(s)$
- $\nu((s, a_1, a_2, \ldots, a_N), i, a'_i, j) = 1$ if $j \equiv i + 1 \pmod{N+1}$ (and equals 0 otherwise).

The AEC game G_{AEC} begins with agent 1. If the initial state of the POSG G was s_0, then the initial state of G_{AEC} is $(s_0, \cdot, \cdot, \ldots, \cdot)$, where all but the first element of the tuple are chosen arbitrarily.

Let $P_{t,s}$ be the probability that the POSG G is in state s after t steps. For an action vector $\mathbf{a} = (a_1, \ldots, a_N) \in \mathcal{A}_1 \times \cdots \times \mathcal{A}_N$, let $P'_{t,s,\mathbf{a}}$ be the probability that G_{AEC} is in state

$(s, a_1, \ldots, a_N)$ after t steps. Finally, let $P'_{t,s} = \sum_{\mathbf{a} \in \mathcal{A}_1 \times \cdots \times \mathcal{A}_N} P'_{t,s,\mathbf{a}}$.

Trivially, $P_{0,s} = P'_{0,s}$ for all $s \in \mathcal{S}$. Now, suppose that after t steps of G, $P_{t,s} = P'_{t(N+1),s}$ for all $s \in \mathcal{S}$ (our inductive hypothesis). For any joint action $\mathbf{a} = (a_1, \ldots, a_N)$, the state distribution of G at step $t+1$ if the joint action $\mathbf{a}$ is taken is given by $P_{t+1,s'} = P_{t,s} \cdot P(s, a_1, \ldots, a_N, s')$. Further, the reward obtained by agent i for this joint action, if the new state is s', is $R_i(s, a_1, \ldots, a_N, s')$. Let $\mathbf{s} = (s, a_1, \ldots, a_N)$ and $\mathbf{s}' = (s', a_1, \ldots, a_N)$. Then, in G_{AEC}, if the agents take actions $a_1, a_2, \ldots, a_N$ respectively on their turns, the state distribution of G_{AEC} at step $(t+1)(N+1)$ is given by $P'_{(t+1)(N+1),s'} = P'_{(t+1)(N+1),s',\mathbf{a}} = P'_{t(N+1),s} P'(\mathbf{s}, \mathbf{s}')$. By the inductive hypothesis, $P'_{t(N+1),s} = P_{t,s}$, and by the definition of $P'(\mathbf{s}, \mathbf{s}')$ in G_{AEC}, it is clear that $P'(\mathbf{s}, \mathbf{s}') = P(s, a_1, \ldots, a_N, s')$. Thus, $P'_{(t+1)(N+1),s'} = P_{t,s} P(s, a_1, \ldots, a_N, s') = P_{t+1,s'}$.

The above establishes a strict equivalence between the state distributions of G at step t and G_{AEC} at step $t(N+1)$ for any t. Between steps $t(N+1)+1$ and $(t+1)(N+1)$ of G_{AEC}, each agent in turn receives an observation and then chooses its action. Specifically, agent i acts at step $t(N)+i$ immediately after receiving an observation ω_i with probability $O'_i(s, a_1, \ldots, a_N) = O_i(s)$. Thus, the marginal probability distribution (when conditioned on transitioning into state s) of the observation received by agent i immediately after acting at time t in G is identical to the marginal distribution of the observation received by i immediately before acting at time $t(N+1)+i$ in G_{AEC}, i.e. $\Pr_{G,t}(\omega_i = \omega \mid s_t = s) = \Pr_{G_{\text{AEC}},t(N+1)+i}(\omega_i = \omega \mid s_{t(N+1),0} = s)$.

The second part of the equivalence is observing that the reward received by an agent i in G after the joint action $\mathbf{a}$ is taken is equivalent to the total reward received by agent i in G_{AEC} across all steps from $t(N+1)+1$ through $(t+1)(N+1)$ when the agents take actions $a_1, \ldots, a_N$ respectively. We can see that this is indeed the case, since the rewards received by agent i in G_{AEC} from step $t(N+1)+1$ through step $(t+1)(N+1)$ is 0 at every step but the environment step

$(t+1)(N+1)$. By definition of R' in G_{AEC}, $R'_i(\mathbf{s}, 0, \varnothing, \mathbf{s}', R_i(s, a_1, \ldots, a_N, s')) = 1$, so the total reward received by any agent i in G_{AEC} is $R_i(s, a_1, \ldots, a_N, s')$. This establishes the second part of our equivalence (that the reward at step $t(N+1)$ in G_{AEC} is identical to the reward at step t of G, if the actions are the same). □

One way to think of this construction is that the actions are still resolved simultaneously via the *environment step* (which is responsible for the stochastic state transition and the production of rewards); we simply break down the production of the joint action into smaller units whereby each agent chooses and "locks in" their actions one step at a time. A toy example to see this equivalence is to imagine a multiplayer card game in which each player has a hand of cards and each turn consists of all players choosing one card from their hand which is revealed simultaneously with all other players. An equivalent game has each player in sequence choosing a card and placing it face down on their turn, followed by a final action (the "environment step" in which all players simultaneously reveal their selected card.

At first, it may appear as though the AEC game is in fact *more* powerful than the POSG, since in addition to being able to handle simultaneous-action games as shown above, it can represent sequential games, including sequential games with complex and dynamic turn orders such as Uno (another aspect of our AEC definition that seems more general than in POSGs is the fact that the reward function in an AEC game is stochastic, allowing rewards to be randomly determined). However, it turns out that a POSG can be used to model a sequential Handling the stochastic rewards and stochastic next-agent function is non-obvious and is omitted here due to space constraints; the construction and proof can be found in subsection 2.5.2.

We next show how to convert an AEC game to a POSG for the case of deterministic re-

wards.

Definition 2. An AEC Game

$$G = \langle \mathcal{S}, N, \{\mathcal{A}_i\}, \{T_i\}, P, \{R_i\}, \{\Omega_i\}, \{O_i\}, \nu \rangle$$

is said to have *deterministic rewards* if for all $i, j \in \mathcal{N}$, all $a \in \mathcal{A}_j$, and all $s, s' \in \mathcal{S}$, there exists a $R_i^*(s, j, a, s')$ such that $R_i(s, j, a, s', r) = 1$ for $r = R_i^*(s, j, a, s')$ (and 0 for all other r).

Notice that an AEC Game with deterministic rewards may still depend on the new state s' which can itself be stochastic in the case of the environment ($j = 0$).

Theorem 2. *Every AEC Game with deterministic rewards has an equivalent POSG.*

Proof. Suppose we have an AEC game

$$G = \langle \mathcal{S}, N, \{\mathcal{A}_i\}, \{T_i\}, P, \{R_i\}, \{\Omega_i\}, \{O_i\}, \nu \rangle$$

with deterministic rewards. We define $G_{\text{POSG}} = \langle \mathcal{S}', N, \{\mathcal{A}_i\}, P', \{R_i'\}, \{\Omega_i\}, \{O_i\} \rangle$ as follows.

- $\mathcal{S}' = \mathcal{S} \times \mathcal{N}$
- $P'((s, i), a_1, \ldots, a_N, (s', i')) = \nu(s, i, a_i, s', i') \cdot \Pr(s' \mid s, i, a_i)$, where

$$\Pr(s' \mid s, i, a_i) = \begin{cases} 1 & \text{if } i > 0, T(s, a_i) = s' \\ P(s, s') & \text{if } i = 0 \\ 0 & \text{o/w} \end{cases}$$

- $R'_i((s, j), a, (s', j')) = R^*_i(s, j, a, s')$

In this construction, the new state in the POSG encodes information about which agent is meant to act. State transitions in the POSG therefore encode both the state transition of the original AEC game and the transition for determining the next agent to act. In each step, the state transition depends only on the agent who's turn it is to act (which is included as part of the state).

This construction adapts POSGs to be strictly turn-based so that it is able to represent AEC Games. □

We now present the full proof.

Theorem 3. *Every AEC Game has an equivalent POSG.*

Proof. Suppose we have an AEC game $G = \langle \mathcal{S}, N, \{\mathcal{A}_i\}, \{T_i\}, P, \{R_i\}, \{\Omega_i\}, \{O_i\}, \nu \rangle$, and $\mathcal{R}$ is the (finite) set of all possible rewards. We define $G_{\text{POSG}} = \langle \mathcal{S}', N, \{\mathcal{A}_i\}, P', \{R'_i\}, \{\Omega_i\}, \{O_i\} \rangle$ as follows.

The state set is $\mathcal{S}' = \mathcal{S} \times \mathcal{N} \times \mathcal{R}^N$. An element of $\mathcal{S}'$ is a tuple $(s, i, \mathbf{r})$, where $\mathbf{r} = (r_1, r_2, \ldots, r_N)$ is a vector of rewards for each agent.

The transition function is given by

$$P'((s, i, \mathbf{r}), a_1, a_2, \ldots, a_N, (s', i', \mathbf{r}')) =$$

$$\nu(s, i, a_i, s', i') \Pr(s' \mid s, i, a_i) \prod_{j \in [N]} R_j(s, i, a_i, s', \mathbf{r}'_i)$$

where

$$\Pr(s' \mid s, i, a_i) = \begin{cases} 1 & \text{if } i > 0 \text{ and } T(s, a_i) = s' \\ P(s, s') & \text{if } i = 0 \\ 0 & \text{o/w} \end{cases}$$

The reward function is given by $R'_i((s, j, \mathbf{r}), a, (s', j', \mathbf{r}')) = \mathbf{r}'_i$ □

2.6 API Design

2.6.1 Basic API

The PettingZoo API is shown in Figure 2.10, and the strong similarities to the Gym API (Figure 2.1) should be obvious — each agent provides an `action` to a `step` function and receives `observation, reward, done, info` as the return values. The observation and state spaces also use the the exact same space objects as Gym. The `render` and `close` methods also function identically to Gym's, showing a current visual frame representing the environment to the screen whenever called. The `reset` method similarly has identical function to Gym — it resets the environment to a starting configuration after being played through. PettingZoo really only has two deviations from the regular Gym API — the `last` and `agent_iter` methods and the corresponding iteration logic.

```
from pettingzoo.butterfly import pistonball_v0
env = pistonball_v0.env()
env.reset()
for agent in env.agent_iter(1000):
    env.render()
    observation, reward, done, info = env.last()
    action = policy(observation, agent)
    env.step(action)
env.close()
```

Figure 2.10: An example of the basic usage of Pettingzoo

2.6.2 The `agent_iter` Method

The `agent_iter` method is a generator method of an environment that returns the next agent that the environment will be acting upon. Because the environment is providing the next agent to act, this cleanly abstracts away any issues surrounding changing agent orders, agent generation, and agent death. This generation also parallels the functionality of the next agent function from the AEC games model. This method, combined with one agent acting at once, allows for the support of every conceivable variation of the set of agents changing.

2.6.3 The `last` Method

An odd aspect of multi-agent environments is that from the perspective of one agent, the other agents are part of the environment. Whereas in the single agent case the observation and rewards can be given immediately, in the multi-agent case an agent has to wait for all other agents to act before it's `observation`, `reward`, `done` and `info` can be fully determined. For this reason, these values are given by the `last` method, and they can then be passed into a policy to choose an action. Less robust implementations would not allow for features like changing agent orders (like the reverse card in Uno).

2.6.4 Additional API Features

The `agents` attribute is a list of all agents in the environment, as strings. The `rewards`, `dones`, `infos` attributes are agent-keyed dictionaries for each attribute (note that the rewards are the instantaneous ones resulting from the most recent action). These allow access to agent properties at all points on a trajectory, regardless of which is selected. The `action_space(agent)`

and `observation_space(agent)` functions return the static action and observation spaces respectively for the agent given as an argument. The `observe(agent)` method provides the observation for a single agent by passing its name as an argument, which can be useful if you need to observe an agent in an unusual context. The `state` method is an optional method returns the global state of an environment, as is required for centralized critic methods. The `agent_selection` method returns the agent that can currently be acted upon per `agent_iter`.

The motivation for allowing access to all these lower level pieces of information is to let researchers to attempt novel, unusual experiments. The space of multi-agent RL has not yet been comprehensively explored, and there are many perfectly plausible reasons you might want access to other agents rewards, observations, and so on. For an API to be universal in an emerging field, it inherently has to allow access to all the information researchers could plausibly want. For this reason we allow access to a fairly straightforward set of lower level attributes and methods in addition to the standard higher level API. As we outline in subsection 2.6.5, we've structured PettingZoo in a way such that including these low-level features doesn't introduce engineering overhead in creating environments, as discussed further in the documentation website.

To handle environments where different agents can be present on each reset of an environment, PettingZoo has an optional `possible_agents` attribute which lists all the agents that might exist in an environment at any point. Environments which generate arbitrary numbers or types of agents will not define a `possible_agents` list, requiring the user to check for new agents being instantiated as the environment runs. After resetting the environment, the `agents` attribute becomes accessible and lists all agents that are currently active. For similar reasons, `num_agents`, `rewards`, `dones`, `infos`, and `agent_selection` are not available until

after a reset.

To handle cases where environments need to have environment agents as per the formal AEC Games model, the standard is to put it into the `agents` with the name `env` and have it take `None` as it's action. We do not require this for all environments by default as it's rarely used and makes the API more cumbersome, but this is an important feature for certain edge cases in research. This connects to the formal model in that, when this feature is not used, the environment actor from the formal model and the agent actor that acted before it are merged together.

2.6.5 Environment Creation and the Parallel API

PettingZoo environments actually only expose the `reset`, `seed`, `step`, `observe`, `render`, and `close` base methods and the `agents`, `rewards`, `dones`, `infos`, `state` and `agent_iter` base attributes. These are then wrapped to add the `last` method. Only having environments implement primitive methods makes creating new environments simpler, and reduces code duplication. This has the useful side effect of allowing all PettingZoo environments to be easily changed to an alternative API by simply writing a new wrapper. We've actually already done this for the default environments and added an additional "parallel API" to them that's almost identical to the RLlib POSG-based API via a wrapper. We added this secondary API because in environments with very large numbers of agents, this can improve runtime by reducing the number of Python function calls.

2.7 Default Environments

Similar to Gym's default environments, PettingZoo includes 63 environments. Half of the included environment classes (MPE, MAgent, and SISL), despite their popularity, existed as unmaintained "research grade" code, have not been available for installation via pip, and have required large amounts of maintenance to run at all before our cleanup and maintainership. We additionally included multiplayer Atari games from Terry and Black [2020], Butterfly environments which are original and of our own creation, and popular classic board and card game environments.

Atari

Atari games represent the single most popular and iconic class of benchmarks in reinforcement learning. Recently, a multi-agent fork of the Arcade Learning Environment was created that allows programmatic control and reward collection of Atari's iconic multi-player games [Terry and Black, 2020]. As in the single player Atari environments, the observation is the rendered frame of the game, which is shared between all agents, so there is no partial observability. Most of these games have competitive or mixed reward structures, making them suitable for general study of adversarial and mixed reinforcement learning. In particular, Terry and Black [2020] categorizes the games into 7 different types: 1v1 tournament games, mixed sum survival games (*Space Invaders*, shown in Figure 2.11a. is an example of this), competitive racing games, long term strategy games, 2v2 tournament games, a four-player free-for-all game and a cooperative game.

Butterfly

Of all the default environments included, the majority of them are competitive. We wanted

to supplement this with a set of interesting graphical cooperative environments. *Pistonball*, depicted in Figure 2.11b, is an environment where pistons need to coordinate to move a ball to the left, while only being able to observe a local part of the screen. It requires learning nontrivial emergent behavior and indirect communication to perform well. *Knights Archers Zombies* is a game in which agents work together to defeat approaching zombies before they can reach the agents. It is designed to be a fast paced, graphically interesting combat game with partial observability and heterogeneous agents, where achieving good performance requires extraordinarily high levels of agent coordination. In *Cooperative pong* two dissimilar paddles work together to keep a ball in play as long as possible. It was intended to be a be very simple cooperative continuous control-type task, with heterogeneous agents. *Prison* was designed to be the simplest possible game in MARL, and to be used as a debugging tool. In this environment, no agent has any interaction with the others, and each agent simply receives a reward of 1 when it paces from one end of its prison cell to the other. *Prospector* was created to be a very challenging game for conventional methods—it has two classes of agents with different goals, action spaces, and observation spaces (something many current cooperative MARL algorithms struggle with), and has very sparse rewards (something all RL algorithms struggle with). It is intended to be a very difficult benchmark for MARL, in the same vein as Montezuma's Revenge.

Classic

Classical board and card games have long been some of the most popular environments in reinforcement learning [Tesauro, 1995, Silver et al., 2016, Bard et al., 2019]. We include all of the standard multiplayer games in RLCard [Zha et al., 2019]: *Dou Dizhu*, *Gin Rummy*, *Leduc Hold'em*, *Limit Texas Hold'em*, *Mahjong*, *No-limit Texas Hold'em*, and *Uno*. We additionally include all AlphaZero games, using the same observation and action spaces—*Chess* and *Go*.

We finally included *Backgammon*, *Connect Four*, *Checkers*, *Rock Paper Scissors*, *Rock Paper Scissors Lizard Spock*, and *Tic Tac Toe* to add a diverse set of simple, popular games to allow for more robust benchmarking of RL methods.

MAgent

The MAgent library, from Zheng et al. [2018] was introduced as a configurable and scalable environment that could support thousands of interactive agents. These environments have mostly been studied as a setting for emergent behavior [AshwiniPokle, 2018], heterogeneous agents [Subramanian et al., 2020], and efficient learning methods with many agents [Chen et al., 2019]. We include a number of preset configurations, for example the *Adversarial Pursuit* environment shown in Figure 2.11d. We make a few changes to the preset configurations used in the original MAgent paper. The global "minimap" observations in the battle environment are turned off by default, requiring implicit communication between the agents for complex emergent behavior to occur. The rewards in *Gather* and *Tiger-Deer* are also slightly changed to prevent emergent behavior from being a direct result of the reward structure.

MPE

The Multi-Agent Particle Environments (MPE) were introduced as part of Mordatch and Abbeel [2017] and first released as part of Lowe et al. [2017]. These are 9 communication oriented environments where particle agents can (sometimes) move, communicate, see each other, push each other around, and interact with fixed landmarks. Environments are cooperative, competitive, or require team play. They have been popular in research for general MARL methods Lowe et al. [2017], emergent communication [Mordatch and Abbeel, 2017], team play [Palmer, 2020], and much more. As part of their inclusion in PettingZoo, we converted the action spaces to a discrete space which is the Cartesian product of the movement and communication action pos-

sibilities. We also added comprehensive documentation, parameterized any local reward shaping (with the default setting being the same as in Lowe et al. [2017]), and made a single render window which captures all the activities of all agents (including communication), making it easier to visualize.

SISL

We finally included the three cooperative environments introduced in Gupta et al. [2017]: *Pursuit*, *Waterworld*, and *Multiwalker*. *Pursuit* is a standard pursuit-evasion game Vidal et al. [2002] where pursuers are controlled in a randomly generated map. Pursuer agents are rewarded for capturing randomly generated evaders by surrounding them on all sides. *Waterworld* is a continuous control game where the pursuing agents cooperatively hunt down food targets while trying to avoid poison targets. *Multiwalker* (Figure 2.11f) is a more challenging continuous control task that is based on Gym's *BipedalWalker* environment. In *Multiwalker*, a package is placed on three independently controlled robot legs. Each robot is given a small positive reward for every unit of forward horizontal movement of the package, while they receive a large penalty for dropping the package.

2.7.1 Butterfly Environment Baselines

When environments are introduced to the literature, it is customary for them to include baselines to provide a general sense of the difficulty of the environment and to provide something to compare against. We do this here for the Butterfly environments that this library introduces for the first time; similar baselines exist in the papers introducing all other environments. For our baseline learning method we used used fully parameter shared PPO [Schulman et al., 2017]

from Stable-Baselines3 (SB3) [Raffin et al., 2019]. We use the SuperSuit wrapper library [Terry et al., 2020b] for preprocessing similar to that in Mnih et al. [2015], convert the observations to grayscale, resize them to 96x96 images, and use frame-stacking to combine the last four observations. Furthermore, for cooperative_pong_v3 and knights_archers_zombies_v7, we invert the color of alternating agent's observations by subtracting it from the maximum observable value to improve learning and differentiate which agent type an observation came from for the parameter shared neural network, per Terry et al. [2020a]. On the prospector_v4 environment, we add an extra channel to the observations which is set to the maximum possible value if the agent belongs to the opposite agent type, else zero. Both these modifications allow us to use parameter-shared PPO across non-homogeneous agents. On prospector_v4 we also pad observation and agent spaces as described in Terry et al. [2020a] to allow for learning with a single fully parameter shared neural network.

After tuning hyperparameters with RL Baselines3 Zoo [Raffin, 2020], our baselines learns an optimal policy in the Pistonball environment and Cooperative Pong environments and learns reasonably in the Knights Archers Zombies and Prospector environments without achieving optimal policies. Plots showing results of 10 training runs of the best hyperparameters are shown in Figure 2.12. All code and hyperparameters for these runs is available at `https://github.com/jkterry1/Butterfly-Baselines`.

2.8 Adoption

In it's relatively short lifespan, PettingZoo has already achieved a meaningful amount of adoption. It is supported by the following learning libraries: The Autonomous Learning Library

[Nota, 2020], AI-Traineree [Laszuk, 2020], PyMARL (ongoing) [Samvelyan et al., 2019], RLlib [Liang et al., 2018], Stable Baselines 2 [Hill et al., 2018] and Stable Baselines 3 [Raffin et al., 2019], similar libraries such as CleanRL [Huang et al., 2020] (through SuperSuit [Terry et al., 2020b]), and Tianshou (ongoing) [Weng et al., 2020]. Perhaps more significantly than any of this, PettingZoo is already being used to teach in both graduate and undergraduate reinforcement learning classes all over the world.

2.9 Conclusion

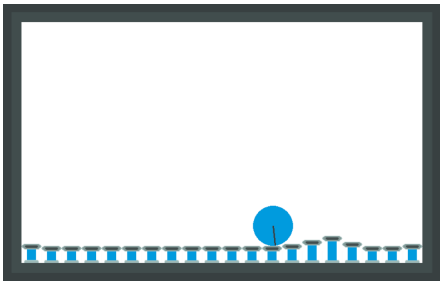
This paper introduces PettingZoo, a Python library of many diverse multi-agent reinforcement learning environments under one simple API, akin to a multi-agent version of OpenAI's Gym library, and introduces the agent environment cycle game model of multi-agent games.

Given the importance of multi-agent reinforcement learning, we believe that PettingZoo is capable of democratizing the field similar to what Gym previously did for single agent reinforcement learning, making it accessible to university scale research and to non-experts. As evidenced by it's early adoption into numerous MARL libraries and courses, PettingZoo is moving in the direction of accomplishing this goal.

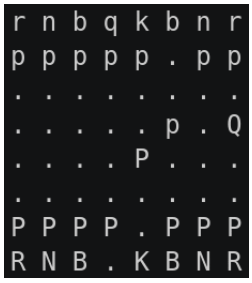
We're aware of one notable limitation of the PettingZoo API. Games with significantly more than 10,000 agents (or potential agents) will have meaningful performance issues because you have to step each agent at once. Efficiently updating environments like this, and inferencing with the associated policies, requires true parallel support which almost certainly should be done in a language other than Python. Because of this, we view this as a practically acceptable limitation.

We see three directions for future work. The first is additions of more interesting environments under our API (possibly from the community, as has happened with Gym). The second direction we envision is a service to allow different researchers' agents to play against each other in competitive games, leveraging the standardized API and environment set. Finally, we envision the development of procedurally generated multi-agent environments to test how well methods generalize, akin to the Gym procgen environments [Cobbe et al., 2019].

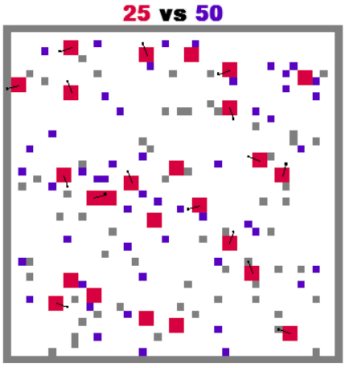
(a) Atari: Space Invaders



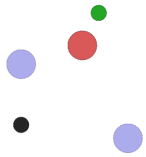
(b) Butterfly: Pistonball



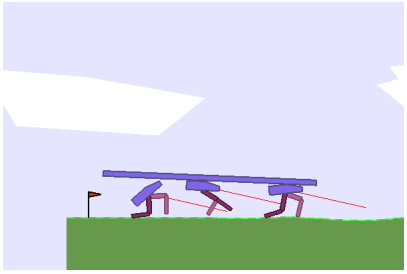
(c) Classic: Chess



(d) MAgent: Adversarial Pursuit

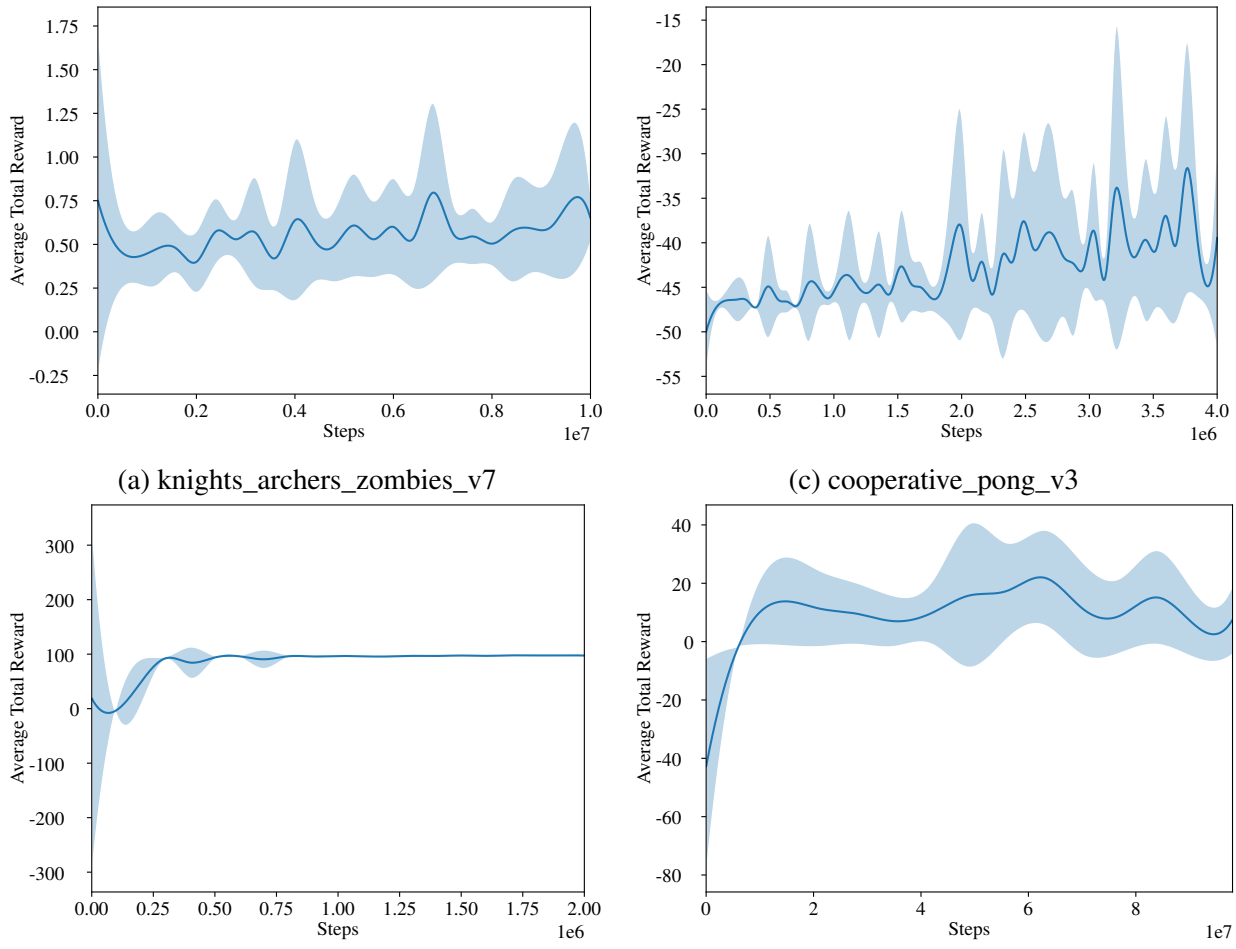


(e) MPE: Simple Adversary



(f) SISL: Multiwalker

Figure 2.11: Example environments from each class.



(a) knights_archers_zombies_v7

(c) cooperative_pong_v3

(b) pistonball_v4

(d) prospector_v4

Figure 2.12: Total reward when each Butterfly environment is learned via parameter-shared PPO.

Chapter 3: Multiplayer Support for the Arcade Learning Environment

3.1 Introduction

In order to add multi-agent Atari Games PettingZoo, we first had to add support for multiplayer game modes to the Arcade Learning Environment ("ALE"), an incredibly influential library that allows for reinforcement learning code to interface with Atari 2600 games via the Stella emulator.

The ALE was first introduced in Bellemare et al. [2013], with further improvements introduced in Machado et al. [2018] (for single player games only). It was a fork of the Stella Atari 2600 emulator, that allowed for a simple C interface to take actions in a supported Atari game, and receive the resulting screen state and game reward.

Reinforcement Learning ("RL") considers learning a policy — a function that takes in an observation from an environment and emits an action — that achieves the maximum expected discounted reward when playing in an environment. Atari 2600 games were viewed as a useful challenge to pursue for reinforcement learning as they were interesting and difficult to humans, and contained a wide array of premade games to test an AI against to prevent researcher bias and to allow for comprehensive benchmarking of a technique [Machado et al., 2018]. They were also special for having visual observation spaces, and were practically useful because they had small graphical observation spaces, ran fast, had generally simple game mechanics that were still

challenging to humans, and had a frequently updating well defined reward.

The ALE served as the benchmark suite in Mnih et al. [2013] and Mnih et al. [2015], which introduced and showed the power of the Deep Q Network method of learning games, essentially creating the field of Deep Reinforcement Learning in the process. These games have remained a very popular benchmark in deep reinforcement learning, as developing very effective methods for all games is still a difficult challenge years later [Hessel et al., 2018, Ecoffet et al., 2019]. These ALE games were later packaged with a much simpler and more general Python interface into Gym [Brockman et al., 2016], which is how the ALE environments are most frequently used today.

Following the boom in single agent Deep Reinforcement Learning research, work has been ongoing to learn optimal policies for agents in environments where multiple agents interact, be the interactions strictly competitive, strictly cooperative, or mixed sum [Silver et al., 2017, Lowe et al., 2017, Gupta et al., 2017, Terry et al., 2020a]. For the same reasons that single player Atari 2600 games became very popular suite of benchmark environments, we feel that the multiplayer Atari 2600 games are uniquely well suited benchmark environments for multi-agent reinforcement learning, and have extended the ALE to support them, allowing easy programmatic interfacing with multiplayer Atari 2600 games for the first time, and a comprehensive suite of differing multi-agent RL benchmarks for the first time.

The environments introduced by this work are available at `https://github.com/Farama-Foundation/Multi-Agent-ALE`, and the learning code used as baselines for these environments is available at `https://github.com/jkterry1/MA-ALE2`. This section is based on work originally published in Terry and Black [2020].

3.2 Related Works

There are a number of popular multi-player game collections adapted to support reinforcement learning (RL). OpenSpiel offers a large collection of discrete classic games and methods to solve those games [Lanctot et al., 2019]. However, none have the sort of physical dynamics you see in many Atari games. The Multi-Particle Environments (MPE) have also been a popular benchmark for general sum RL [Mordatch and Abbeel, 2017, Lowe et al., 2017], however, these environments are not terribly diverse and are mostly solved. There is also a great deal of work in environments for cooperative RL, such as the Starcraft Multi-Agent challenge [Samvelyan et al., 2019], however the cooperative reward structure simplifies the multi-agent dynamics considerably more than in the competitive or mixed case. Competitive mass battle environments also exist in works like MAgent [Zheng et al., 2018], however, these tend to be based on grids, and their absolute performance is hard to evaluate. There are also a number of one-off RL environments designed to show a point in a particular paper, for example, OpenAI's Emergent Tool Use environment [Baker et al., 2019a]. While interesting in their scope, these environments tend to not be broadly useful.

However, no set of multi-agent RL benchmarks exist that simultaneously satisfy the following useful criteria that Atari meets.

1. Graphical observations
2. Easy comparison to human performance
3. Diverse incentives (Atari games simultaneously encompass zero sum, mixed sum and cooperative environments)

4. Long episodes and reward horizons

It is also of note that the Multi-Agent ALE environments introduced in this work have since been included in a related work, PettingZoo [Terry et al., 2021], which compiles many RL environments into a single unified API.

3.3 API

In order to support multiplayer games, a few changes to the existing ALE API, outlined in Bellemare et al. [2013], had to be made. The `getAvailableModes` method was overloaded to receive number of players as an optional argument and return game modes with that number of players. The `step` method was overloaded to accept a list of actions, and return a list of rewards (one reward and action for each player). The `allLives` method was added to return the number of lives for each player as a list. We used the standard of 0 lives meaning the agent is alive, and the game will end after one more life. All these changes are backwards compatible and will not impact the single player modes of a game, except for one: we changed game modes to be more aligned with those listed in their manuals to make adding multiplayer modes feasible.

An example of using ALE API for the multiplayer mode of *space invaders* is shown below:

```
ale = multi_agent_ale_py.ALEInterface()
ale.loadROM("ROMS/space_invaders.bin")
minimal_actions = ale.getMinimalActionSet()
modes = ale.getAvailableModes(num_players=2)
ale.setMode(modes[0])
```

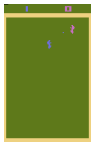
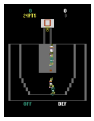
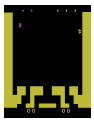
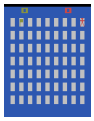
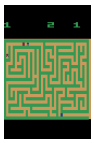
```
ale.reset_game()
while not ale.game_over():
    observation = ale.getScreenRGB()
    a1 = policy1(observation)
    a2 = policy2(observation)
    r1,r2 = ale.act([a1,a2])
```

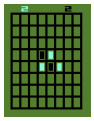
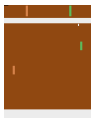
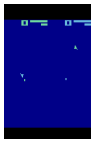
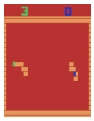
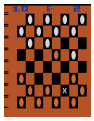
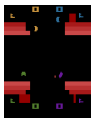
3.4 Games Included

We added support for 18 multiplayer games, 14 of which were extended from existing ALE games, and 4 are brand new. All added games are shown in Table 3.1. A few games have alternate game modes which offer completely different game play, effectively adding 6 more games for a total of 24. These modes are listed as parenthesized names in Table 3.1. These modes as well as other noteworthy game modes are described in Appendix B.

Having a comprehensive and "unbiased" suite of games is considered key to the success of the ALE [Bellemare et al., 2013, Machado et al., 2018]. Of the games that were already in the ALE, we tried to add support as indiscriminately as possible, choosing 14/19 possible multiplayer games. The 4 we did not add multiplayer support to were Hangman (an asymmetry in the game play lets one player be guaranteed to win), Lost Luggage (the two different players are heterogeneous in ways that make adding this game beyond the scope of this work), as well as Freeway and Blackjack (they offer no interactions between the two players). Out of the four new games, we selected Combat, Joust and Warlords for their fame amongst human players and Maze Craze because of the very unique planning challenges it offers. All but Joust are multiplayer only

Table 3.1: All Multiplayer ROMs and Game Types Supported

ROM (Modes)	Image	Game theory	Players
Boxing		Competitive	2
Combat (Tank, Plane)		Competitive	2
Double Dunk		Competitive	2
Entombed (Cooperative, Competitive)		Competitive and Cooperative	2
Flag Capture		Competitive	2
Ice Hockey		Competitive	2
Joust		Mixed	2
Mario Bros		Mixed	2
Maze Craze		Competitive	2

ROM (Modes)	Image	Game theory	Players
Othello		Competitive	2
Video Olympics (Pong, Basketball, Foozpong, Quadrapong, Volleyball)		Competitive	2/4
Space Invaders		Mixed	2
Space War		Competitive	2
Surround		Competitive	2
Tennis		Competitive	2
Video Checkers		Competitive	2
Warlords		Competitive	4
Wizard of Wor		Mixed	2

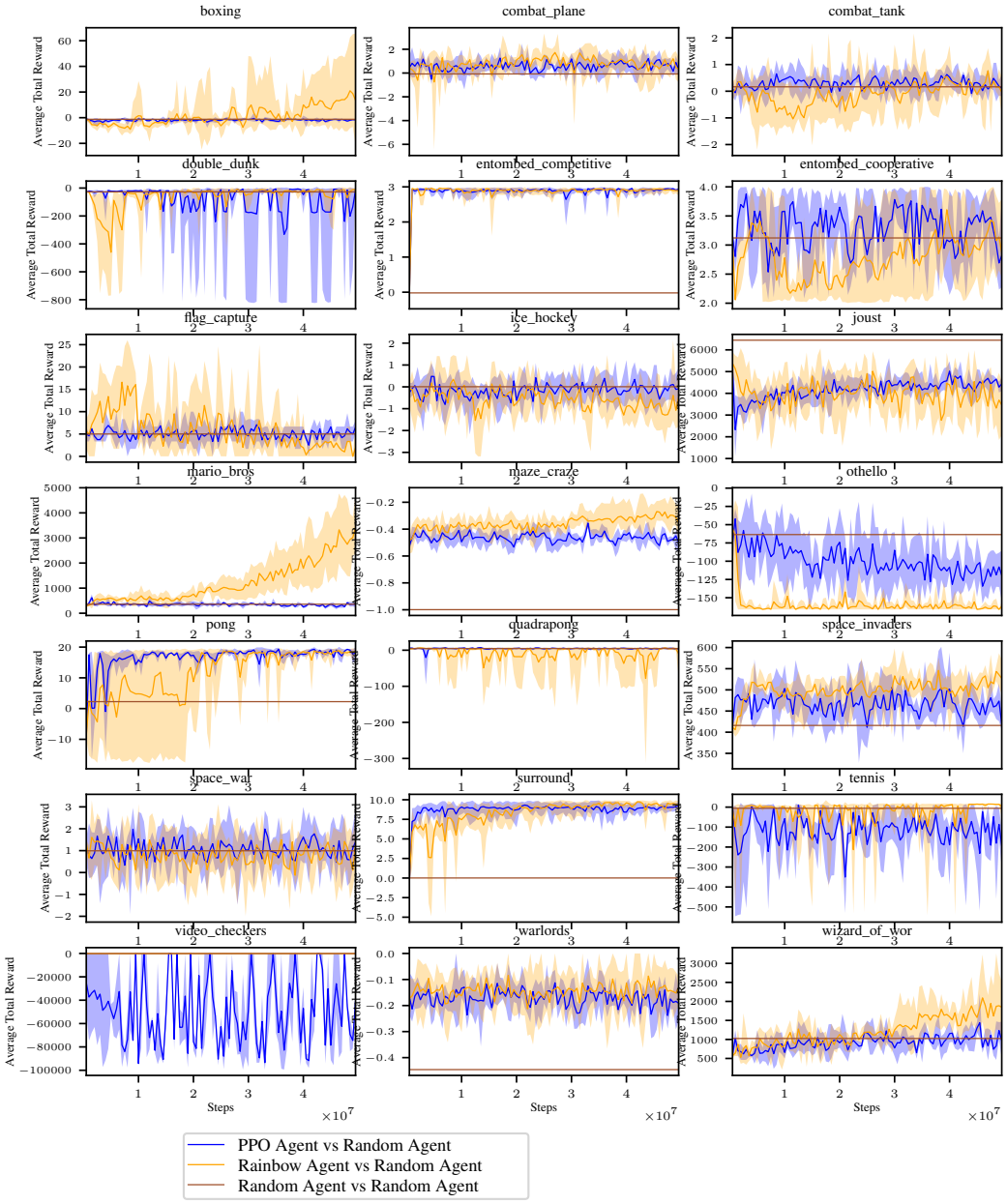


Figure 3.1: Average total reward per step for policy vs random agent on all Atari environments.

games (and therefore couldn't have been included in the ALE before).

This selection process gave rise to an interesting mix of games that can be roughly categorized in 6 groups:

1v1 Tournament: Boxing, Combat, Ice Hockey, Video Olympics (two player), Double

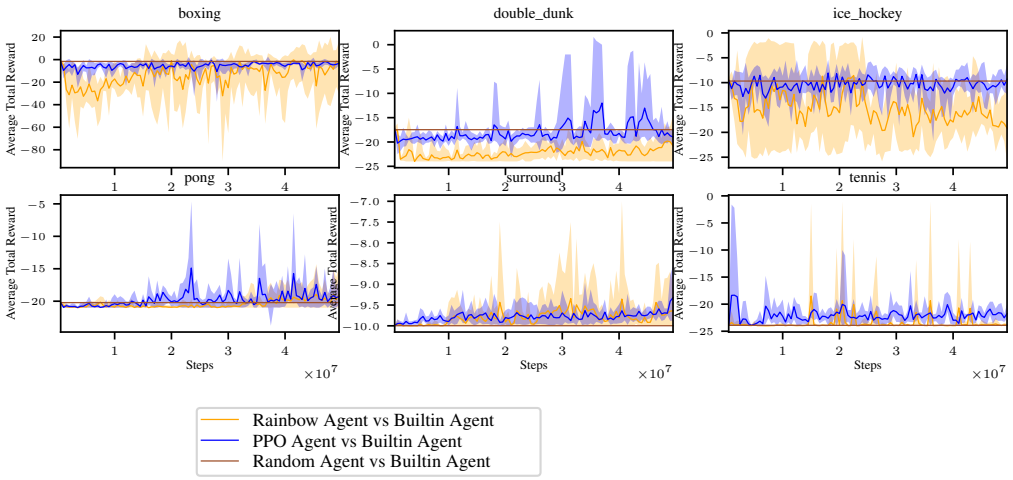


Figure 3.2: Average total reward per step for policy vs builtin agent.

Dunk, Tennis and Space War. These games are zero-sum competitions, so a match or tournament needs to be run in order to compare the quality of different policies. They are also fast paced, with non-sparse reward in random play, and relatively short dependencies between actions and rewards. But to master these games, your agent requires careful analysis of your opponent's position, prediction of their future movements, and very precise movements with quick reaction time.

Mixed-sum survival: Joust, Mario Bros, Wizard of Wor and Space Invaders. These are 2 player games characterized by waves of dangerous computer-controlled opponents which players are rewarded for destroying. The computer-controlled opponents are the main source of risk and reward (allowing players to work together), but your opponent can still benefit by doing you harm. In Joust and Wizard of Wor, you can score points by directly destroying your opponent, in Space Invaders you score points when the Aliens destroy your opponent, and in Mario Bros your opponent can steal the reward you worked for. In all of these games, a robust agent has to be able to handle both aggressive competitive strategies and cooperative strategies from an opponent to

maximize their score.

Competitive racing: Flag Capture, Entombed (Competitive Mode) and Maze Craze. These are fully competitive games more focused on understanding the environment than on the opponent to complete the race first. However as the agents develop, competitive strategies are possible. In Flag Capture, the information one player reveals about the map can be exploited by the other player. In Entombed, players have their ability to block their opponent using power-ups. In Maze Craze, we support numerous multiplayer modes (detailed in Appendix B.3). These typically have little player interaction, but in three modes players can add fake walls that can be moved through, but can be confusing to the opponent.

Long term strategy games: Video checkers and Othello. These games are classic zero-sum 1v1 board games of long term strategy with an Atari interface. Due to this inefficient interface, a single turn in the board game will require many steps in the arcade game. In addition, like in the classic games, strong players must consider the consequences of moves many turns later, so very long term planning and credit assignment is required to create strong policies.

Four-player free-for-all: Warlords. Each player's objective is to be the last one standing out of the initial four players. Because of this, cooperation between players is possible until there are only 2 players left. For example, players can coordinate attacks against the strongest or best positioned player so that player will not have a large advantage at the end of the game when it is just 1v1.

2v2 Tournament: Video Olympics. The Video Olympics ROM includes 2v2 team versions of every 1v1 game type (specific modes are detailed in Appendix B.5). The agents on one team will need to cooperate in order to both defend themselves and effectively attack the opposing team.

Cooperative games: Entombed (Cooperative Mode). In Entombed, the players are rewarded based on how long they can both stay alive in the environment. They have the ability to help each other out by using their power-ups to clear obstacles in the maze. This mode is detailed in Appendix B.2.

Finally, we had to change the reward structure for games where one player can indefinitely stall the game by not acting (Double Dunk, Othello, Tennis and Video Checkers). We accomplished this by making a player lose the game with the associated negative reward if a player stalls the game for several hundred frames.

3.5 Baselines

In order to characterize the difficulty of these environments, and to provide a baseline for future work, we trained agents in every environment using self play/parameter sharing with Rainbow DQN [Hessel et al., 2018] and PPO [Schulman et al., 2017], using the hyperparameters specified in subsection 3.5.2. This choice was motivated by the known robust performance of Rainbow DQN and PPO on Atari and other difficult environments. Training was done using the Autonomous Learning Library [Nota, 2020], based on the PettingZoo versions of the environments in Terry et al. [2021], using standard Atari preprocessing wrappers from Terry et al. [2020b] described in subsection 3.5.2.

To evaluate the quality of the trained agent as training progressed, we evaluate the total average reward the first (trained) agent received when playing against a random agent. The baseline is the reward of a random agent playing against a random agent. Choosing to evaluate the trained policy against a fixed policy which the agent was not trained against (the random

policy) was an intentional choice to capture how well the self-play training generalized to a novel agent. The results of this evaluation are presented in Figure 3.1.

Several environment policies learned interesting emergent behavior. For example, in Boxing the agent learns to back up when its opponent punches and it learns to push its opponent into the wall so it cannot dodge as effectively. In other environments, the agent learns a very simple, but effective strategy. In Tennis, for example, the agent quickly moves into the optimal spot to receive a basic serve, and does not move from there. Surprisingly, in Joust and Wizard of Wor, the trained agent performed significantly worse than a random agent. One likely explanation is that this pathological training is the result of the mixed game theoretic structure of these two environments (discussed in section 3.4). A policy may end up focusing on combating the opponent, making both agents choose conservative policies, lowering the overall accumulation of reward. It is likely that a league system, similar to Vinyals et al. [2019], is needed to overcome many of the challenges faced in learning these environments.

Animated gifs of the trained agent playing a random agent are included for all environments in the supplementary materials.

3.5.1 Preprocessing

Preprocessing was performed with SuperSuit wrappers [Terry et al., 2020b]. Observations are generated by ALE as grayscale images, then resized to 84x84 with area interpolation. There is a max accumulation of 2 timesteps, a frame skip of 4, followed by a frame stack of 4. Reward clipping between -1 and 1 was used during training only. Finally, a fairly complex agent indicator is applied. This agent indicator takes in each input channel and outputs 4 concatenated channels.

For two player games these channels are: `[original, original if agent_idx == 1 else 255 - original, 255 * (agent_idx % 2), 255 * ((agent_idx + 1) // 2) % 2)]`. For four player games, these channels are: `[original, (255 * agent_idx) // 4 + original, 255 * (agent_idx % 2), 255 * ((agent_idx+1) // 2) % 2)]`.

3.5.2 Hyperparameters

Table 3.2 and Table 3.3 shows the hyperparameters used for training the baseline with selfplay.

Hyperparameter	Value
discount_factor	0.99
lr	6.25e-5
eps	1.5e-4
minibatch_size	128
update_frequency	32
target_update_frequency	1000
replay_start_size	20000
replay_buffer_size	1000000
initial_exploration	0.02
final_exploration	0.
test_exploration	0.001
alpha	0.5
beta	0.5
n_steps	3
atoms	51
v_min	-10
v_max	10
sigma	0.5

Table 3.2: Hyperparameters for Rainbow DQN on each Atari environment.

Hyperparameter	Value
discount_factor	0.99
lr	2.5e-4
eps	1.5e-4
minibatches	4
epochs	4
n_envs	8
n_steps	128
lam	0.95
clip_grad	0.5
entropy_loss_scaling	0.01
value_loss_scaling	0.5
clip_initial	0.1
clip_final	0.01

Table 3.3: Hyperparameters for PPO on each Atari environment.

3.6 Conclusion

This work introduces multiplayer game support for the Arcade Learning Environment (ALE) for 18 ROMs, enabling 24 diverse multiplayer games. This builds off both the ubiquity and utility of Atari games as benchmarking environments for reinforcement learning, and the recent rise in research in multi-agent reinforcement learning.

We hope that this framework will enable accurate benchmarking for more general multi-agent reinforcement learning methods that can handle graphical observations, highly diverse game-play, and diverse multi-agent reward structures. We believe that multi-agent algorithms with these characteristics will generalize better to real world scenarios, and that no existing set of benchmarks has satisfied these needs particularly well.

The Multi-Agent ALE package is publicly available on PyPI and can be installed via `pip install multi-agent-ale-py`. AutoROM, a separate PyPI package, can be used to easily install the needed Atari ROMs in an automated manner. The Atari games here are additionally

included with a simpler Python API in PettingZoo, which is akin to a multi-agent version of OpenAI's Gym library.

In order to better characterize the utility of each environment, we additionally have shown experimental self-play baselines for all new environments. These baselines indicate which environments have sparse reward, which ones can be trained easily with self-play, and which ones have pathological training under self-play. Future work can use these results to guide the direction of their research. These results will also allow the general community know what sort of results are trivial, and what sort of results indicate an interesting new method. It is likely that the usage of league, similar to Vinyals et al. [2019] will be required.

Chapter 4: SuperSuit: Simple Microwrappers for Reinforcement Learning Environments

4.1 Introduction

SuperSuit was created as a stand alone package to be able to provide robust wrappers for both PettingZoo and Gym (which lacked them at the time). This has since been largely upstreamed into Gym/Gymnasium.

Applying transformations to information passing between a model and an environment in reinforcement learning is an integral part of every major experimental work in the field [Mnih et al., 2013, Vinyals et al., 2019, Silver et al., 2017, Berner et al., 2019]. Techniques popular on Atari environments include scaling down observations with image processing methods or making the observation greyscale to reduce processing time with neural networks, "stacking" frames together to help establish velocity, or skipping frames to increase training speed [Mnih et al., 2013].

These "wrappers" are very useful, but using them in practice has pain points. For code modularity, ease of debugging, and ease of hyper-parameter tuning, it's generally preferable to define the wrapper function(s) outside the environment. Ideally these very commonly used functions would be distributed in a library, so that the implementation used is as fast as possible. This

is fairly important considering how many times it would be called in large research projects.

Gym [Brockman et al., 2016] has become the standard API and set of benchmark environments for single-agent reinforcement learning. PettingZoo [Terry et al., 2021] has recently been released, achieving similar goals for multi-agent reinforcement learning environments. The only existing library with wrappers for reinforcement learning are those included inside Gym, but those are primarily the initially popular wrappers for Atari preprocessing [Mnih et al., 2013]. Newer preprocessing methods for Atari [Machado et al., 2018], other types of environments, or multi-agent environments are omitted. Many Gym wrappers are also missing "quality of life" features, like outputting arrays in a shape compatible with CNNs by default. Accordingly, people typically write their own wrappers themselves. This leads to lower code quality and performance throughout the field for such key functions, and makes the possibility of bugs greater. Accordingly, we've released the SuperSuit Python library to include all widely used wrappers for both Gym and PettingZoo environments. Each wrapper is a function that takes an environment object and returns one, and for clarity and modularity, only includes a single function, hence our terming them "microwrappers". This section is based on work originally published in Terry et al. [2020b].

4.2 Wrapper Methods

The observation wrappers we include are:

- Agent Indication (Multi-Agent Only) [Gupta et al., 2017] for showing which agent is which in a multi-agent game in the observations
- Color Reduction (Greyscaling, etc.) for reducing the amount of colors in an environment

observation to save compute

- Flatten Observation to turn a multidimensional observation into a 1D vector for compatibility with certain neural network architectures
- Frame Skipping [Mnih et al., 2013] for skipping certain observations in order to introduce a human like delay before acting
- Frame Stacking [Mnih et al., 2013] for returning an observation that includes multiple previous states to artificially create a sort of short term memory about past states of the environment
- Max Observation [Machado et al., 2018] for removing certain common visual artifacts with certain Atari 2600 games
- NaN Random to choose a random action if a neural network malfunctions and outputs an NaN
- NaN Noop to choose the designated "do nothing" action if a neural network malfunctions and outputs an NaN
- Observation Delay to delay observations by a certain number of steps to artificially add real-life-esque delays
- Observation Normalization to normalize the values of an observation to certain bounds (usually 0 to 1) for better compatibility with most deep neural networks
- Observation Padding (Multi-Agent Only) [Terry et al., 2020a] for making multi-agent environments with heterogeneous observation sizes compatible with a single neural network

size

- Recast Observation Type to change the datatype of an observation
- Reshape Observation to change the shape of an observation for compatibility with different neural network architectures

The action wrappers we include are:

- Action Clipping [Fujita and Maeda, 2018] for clipping actions to an acceptable range, usually to restrict the output range of the neural network to that expected by the environment
- Action Space Padding (Multi-Agent Only) [Terry et al., 2020a] for making multi-agent environments with heterogeneous action sizes compatible with a single neural network size
- Black Death (Multi-Agent Only) for preventing removed agents in multi-agent environments from being being removed from the environment, and returning black observations and ignoring observations instead. This allows for better compatibility with our
- Scale Actions for rescaling actions output by a neural network (usually 0 to 1) to the size expected by the environment
- Sticky Actions [Machado et al., 2018] for repeating the agents actions instead of allowing it to take a new one, in order to add "human like" limitations on control

The only reward wrapper we include is:

- Reward Clipping [Mnih et al., 2013] for clipping rewards to certain defined bounds to prevent the neural network from taking steps too large

Additionally, we introduce *lambda wrappers* that take an environment and a lambda function as an argument and the lambda function to the environment it, allowing people to easily create custom wrappers. Separate lambda wrappers exist to apply functions to actions, observations, or rewards.

4.3 Conclusion

We introduce *SuperSuit*, a Python library that includes reasonable implementations of all popular RL wrappers, for environments of both the Gym and PettingZoo API specification. This will allow researchers to conduct more computationally efficient experiments, to try new RL wrappers much more easily, and to reduce the likelihood of bugs due to one-off implementations. The library is available at `https://github.com/Farama-Foundation/SuperSuit`, and can be installed via pip.

Chapter 5: MCMES: A Very Simple Method For Discovering Novel Emergent Behaviors in Multiagent Systems Through Reinforcement Learning

5.1 Introduction

Many of the most fundamental and unsolved questions across different fields of science center around understanding group behaviors that arise from interactions between its constituents. Examples include “How did human language arise?”, “What will the economy do next year?”, “How do swarms of animals function, and why is the swarming behavior incentivized?”, “Why do specific fashion trends spread amongst certain social groups?”, “How will a pandemic progress” or ‘Why have birds across North America have been replacing old songs with specific new ones over the past decade?” Countless other problems in this space exist too, such as understanding human movements for better city planning, especially during disasters; understanding why specific strategies in multiplayer computer games spread to common use. A simple example of emergence that most people are unknowingly well acquainted with is that in many major cities around the world, pedestrians will walk on one side of the escalator, and stand on the other. These questions are inherently difficult to answer, because of the inherent complex nature of the systems and difficulties in measuring agents.

The field of reinforcement learning (“RL”) studies using machine learning to optimally

control an agent in a system. These tasks span simple control tasks such as elevator management [Crites and Barto, 1998] or wing flap control on an airplane [Monaco et al., 1997] to robotic grasping Kalashnikov et al. [2018], controlling nuclear reactors Degrave et al. [2022] or learning superhuman performance in many challenging games for the first time Berner et al. [2019], Vinyals et al. [2019], Silver et al. [2016], Wurman et al. [2022]. RL is based on a model in which an *agent* acts in an *environment*, observing its state and collecting rewards. When the environment is in a state s, the agent receives a numeric observation o, which may not have all information about the state of the environment. Based on that, the agent uses its *policy* π to select an *action* a which is then executed in the environment, modifying its state, and giving the agent a scalar *reward* r. The cycle then repeats, with the objective being maximizing the total reward obtained over the agent's lifetime. This looping process is formalized as a Partially Observable Markov Decision Process (POMDP), described in Littman [2009].

Multiagent reinforcement learning studies optimally controlling multiple agents acting in a shared environment. The tasks it can solve range from cooperative real world problems like coordinating robots in warehouses, to abstract competitive ones like playing Chess or Go, with a spectrum in between. In the most general case, MARL functions the same way as single-agent RL, but the evolution of the environment depends on the actions of all agents. Each agent may have entirely separate policy functions, reward functions and observations, which allows for fully cooperative and competitive scenarios, and everything in between.

If you want to understand what a group of agents in a system is doing, it is fairly natural to begin by describing each agent's behavior rules as a function, similarly to agent based modeling. This has been done since the early days of RL research Wu and Sun [2001], Sun et al. [2001], Castelfranchi [2001], Jong and Steels [2003], Nowak et al. [2000, 2001]. More recently,

more advanced forms of emergence have been learned. For example Foerster et al. [2016] first showed it is possible to observe the emergence of language that resembles natural languages, using MARL-trained agents. This has been reproduced and improved many more times, notably in Lazaridou et al. [2016] and Havrylov and Titov [2017]. Furthermore, Baker et al. [2019b] and Team et al. [2021] show complex emergent behaviors arising in tag-like games, AlphaGo Silver et al. [2016] was able to learn novel emergent behaviors in Go that were previously unknown to players (famously with "move 37"), and AlphaStar Vinyals et al. [2019] learned unexpected emergent behaviors in StarCraft 2 such as the overproduction of worker units.

There are also recent high profile instances where multiagent reinforcement learning has been leveraged to study problems in science. The AI Economist used MARL to find optimal tax policies in a simulation that includes citizens participating in a market-like environment, as well as a government agent which decides the tax policy Zheng et al. [2020]. Multiagent reinforcement learning has also been used to learn policies for fish swimming in a close formation in a realistic simulation, finding evidence for how swimming in schools may result in better energy efficiency Verma et al. [2018]. A recent work also researches the use of reinforcement learning to explore the different strategies that would emerge as a result of changing the rules of chess Tomašev et al. [2020]. Further examples in the literature with similar methodologies are Gazzola et al. [2016], Reddy et al. [2016], Herbert-Read et al. [2011], Karpas et al. [2017], Hahn et al. [2019], Hill et al. [2021].

5.2 Methodology

5.2.1 The Intuition Behind MCMES

The behavior of any embodied agent taking actions in a system can be described by a sufficiently large, and potentially stochastic, function that maps the agents internal state and observation of the world to the actions it takes, like a reinforcement learning policy. Neural networks are universal function approximators Hornik et al. [1989], i.e. for any well behaved function, there exists a neural network that approximates it with arbitrary accuracy. Given this, in any multiagent system, every possible collective behavior should be arbitrarily well represented by a sufficiently large policy network for each agent. By performing a Monte Carlo search of all policies as parameterized by neural networks, one could find sets of policies encoding all possible distinct emergent behaviors in a system. This, however, would not be very practical, as the vast majority of behaviors would be various random movements or would otherwise fail to meaningfully perform useful or interesting actions in the environment. Instead, it is better to sample from the subset of the policies whose expected performance exceeds a certain threshold – either the RL reward, or another metric where applicable. This way, we select only policies which are "mature" and perform their task relatively well. We term this approach to searching for emergent behaviors Monte Carlo Mature Emergence Search, or "MCMES."

This intuition can be mathematically formalized as follows: let each agent in a system with n agents be identified by $i \in \{1, 2, ..., n\}$. At each time step t, each agent has the numeric observation $\mathbf{o}_{i,t} \in \mathbf{O}$ and takes an action $\mathbf{a}_{i,t} \in \mathbf{A}$, with a neural network policy π with parameters $\theta_i \in \Theta_i$ (note that we treat architecture as a parameter here, which is uncommon in the literature).

We then define $\xi \in \Xi$ as all possible combination of all policies from all agents. ξ (which we call "collective policies") therefore inherently represents different emergent behaviors within a system. Given an evaluation function $L : \Xi \to \mathbb{R}$, with a maturity threshold $m \in \mathbb{R}$, the approach of MCMES is to repeatedly sample ξ from regions of Ξ where $\mathbb{E}[L(\xi)] > m$, see if the inequality holds empirically, and then explore what interesting behaviors are encoded in the set of mature ξs that are found.

5.2.2 Implementation Details

By training a group of agents with deep RL, if the agents learn a sufficiently effective behavior as determined by the rewards (or other metrics), the set of individual policies is a sample from the space of mature policies. If we retrain the agents with identical hyperparameters, but a different random initialization of the neural network, the resulting policy will likely be slightly different. Thus, training over and over again is essentially a crude form of MCMES. To increase the variation of found behaviors, we can also train with sets of different hyperparameters for the chosen RL algorithm. We name this approach "MCMES through hyperparameter variation". While it is a very simple and biased approach, we evaluate its performance and find that it already achieves quite interesting results.

To validate this approach, we choose 12 diverse and scientifically interesting multiagent environments from different domains and differing levels of complexity. We then chose a reward threshold for a "mature" policy for each that indicate reasonably advanced behaviors, conducted a hyperparameter search using Optuna Akiba et al. [2019] for 150-400 iterations, picked the 16 best performing sets of hyperparameters, retrained with them for 10 trials each, and then rendered the

behavior of the learned policies (which performed above our maturity threshold) 5 times to make sure we properly capture the behaviors. For initial demonstration purposes, we classified the videos manually. We primarily conducted these experiments using the PPO learning algorithm Schulman et al. [2017], but performed similar experiments with DQN Mnih et al. [2015] to show that our approach applied to different methods.

We implemented our search by using a lightly modified version of the RL Baselines Zoo Raffin [2020] library, an open source library made to conduct hyperparameter tuning searches on Stable Baselines 3 (SB3) Raffin et al. [2019] using Optuna Akiba et al. [2019] for the search procedure. All environments are implemented with a standard PettingZoo (PZ) Terry et al. [2021], which is a standardized API for MARL environments. Our goal in doing this was to use the most widely used, well documented, and easy to use Python libraries to enable this code to be more easily reuseable by practitioners from other fields of science who are not highly experienced in RL. The code is available online.[1]. Note that the Crowd Circle environment uses a separate custom learning stack due to implementation differences, which also validates the efficacy of our method beyond one learning library. A motivated programmer familiar with Python should be able to apply our codebase to new problems of interest with a modest level of engineering effort.

5.2.3 Test Environment Selection

To see if our simple method for finding new emergent behaviors works, we wanted to apply it to existing multiagent RL environments reflective of many domains where searching for emergent behaviors is useful. To this end, we choose the following environments (8 unique, with 4 additional variations). Note that images of the environments we chose, their detailed mechanics,

[1] `https://github.com/jkterry1/MCMES`

and the maturity thresholds we chose are included in the supplementary materials C.1.

1. Video Games

    - Cooperative Pong Terry et al. [2021]. A variation of the classic *Pong* arcade game where two agents work together to control two asymmetric paddles to keep the ball from going off screen for as long as possible; this game is very simple.
    - Knights Archers Zombies from Terry et al. [2021]. Two archer units and two knight units work together to keep a procedurally generated mob of zombies from crossing from one side of the screen to the other or killing any of the agents.

2. Physics-based Decision Making Problems

    - Pistonball (with continuous or discrete actions and 5 or 20 pistons) Terry et al. [2021]. A ball is dropped on top of pistons on one side of the screen, and can only be observed by nearby pistons. The pistons must work together to roll the other side of the screen as fast as possible. The actions can be a continuous number representing how much the pistons should move, or a discrete value representing if it should move up, down, or stay still.

3. Robotics

    - Multiwalker Terry et al. [2021]. Three bipedal robots with lidar sensors must work together to balance and carry a large bar as far forward as possible.

4. Economics

- Harvest Leibo et al. [2021], Hughes et al. [2018]. A gridworld environment where agents must learn to compete or work together with each other for to harvest resources on a map with a finite resource generation rate.

5. Traffic Control

   - The Ingolstadt7 and Ingolstadt21 environments Ault and Sharon [2021] for the SUMO traffic simulator Krajzewicz [2010], as bundled in Alegre [2019]. Both environments contain maps of roads from Ingolstadt, Germany, where the agents agents work together to cooperatively control traffic lights so that the vehicles can move as efficiently as possible. Ingolstadt7 has 7 intersections and hence 7 agents, while Ingolstadt21 has 21 intersections and 21 agents. The Ingolstadt21 map also fully contains the Ingolstadt7 map and lights.

6. Crowd Simulation

   - Crowd Circle Kwiatkowski et al. [2022]. 12 agents are placed on the perimeter of a circle, and each has to navigate to the opposite point on the circle as fast as possible while walking at a comfortable speed and avoiding collisions.

7. Population Biology / Classic Multiagent

   - Pursuit Gupta et al. [2017]. A gridworld with multiple hunting agents that must cooperate to hunt randomly moving prey.

The environment variations we chose allow us to answer a few interesting questions. The expansion to more agents and more complexity in the case of Pistonball with 5 or 20 pistons and

Ingolstadt7 vs ingolstadt21 (the larger number has more smart traffic lights under control and a large map) allows us to see if MCMES can find more, new, or more complex emergent behaviors in more complex variations of tested environments, like you would expect to be the case. If more behaviors are found, this suggests that MCMES can be used to study how different environments configurations allow for different emergent behaviors.

5.3 Discussion

5.3.1 Found Behaviors

The full list of emergent behaviors we found in each method, a detailed description of them and the number of times we found it for each set of hyperparameters are included in the supplementary materials. All videos of found behaviors are available for public download [2]. Table C.10 to C.18 in the supplemental materials surveys the emergent behaviors discovered in great detail.

Table 5.1: The number of behaviors discovered for each environment and configuration tested

Environment	Number of Found Behaviors
Cooperative Pong	1
Crowd Circle	2
Harvest	5
Ingolstadt7	1
Ingolstadt21	2
Knights Archers Zombies	6
Multiwalker	4
Pistonball (Continuous, 5 pistons, PPO)	9
Pistonball (Continuous, 20 pistons, PPO)	18
Pistonball (Discrete, 20 pistons, PPO)	22
Pistonball (Discrete, 20 pistons, DQN)	14
Pursuit	2

[2] https://archive.org/details/MCMESVideos.tar

To get a sense of the scale of diversity of behaviors discovered, we list the description of the behaviors found for "Pistonball (Continuous, 5 pistons, PPO)". For context, Pistonball is a simple environment where pistons that can see the space above them directly, and to the left and right, work together to roll a ball to the left as fast as possible, shown in Figure 5.1.

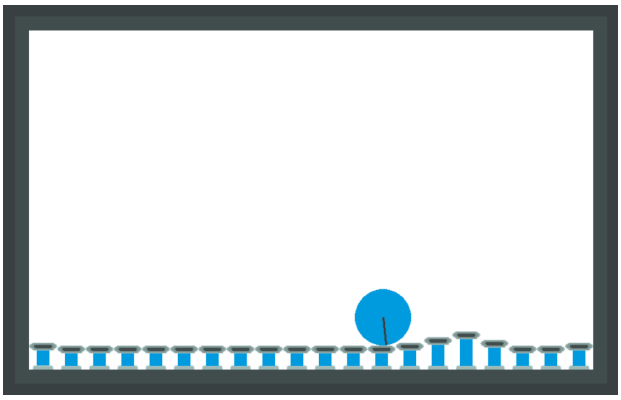


Figure 5.1: A single frame of the Pistonball environment

Crowdsurfing Pistons move in such a way that the ball moves along a moving cavity.

Raise and Drop Pistons initially raise the ball up high, before briefly causing the ball to bounce such that it falls and rolls a far distance.

Ramp Pistons build a ramp at the area where the ball stops.

Tapping Pistons perform a quick and short up and down motion when they're slightly behind the ball and the ball is moving relatively slowly.

Wave The pistons move in such a way that there is a constant hill behind the ball.

Ramp then Tapping A ramp is built at the beginning to get the ball moving, before it is tapped the rest of the way.

Ramp then Wave The pistons build a ramp at the beginning before they wave it the rest of the way.

Rear Push The pistons gently extend if they are behind the ball to make the ball move.

Rear Push then Ramp The pistons form a slow ramp that appears as though a transition between rear push then ramp.

5.3.1.1 Discussion Of Found Behaviors

While Pistonball had the most diverse behaviors of every environment tested, in virtually every environment searched we found a breadth of very qualitatively different emergent behaviors. This seemingly common level of highly diverse emergent behaviors across many popular multiagent RL environments is a phenomenon that has not been previously documented within the community.

Furthermore, while an anecdotal result, the vast majority of the behaviors found were not previously known to the environment authors as possible good strategies. This supports that our naive method for MCMES appears to be capable of discovering emergent behaviors previously unknown to authors across a diverse set of environments. This claim is also supported by the fact that our MCMES methods initially discovered previously unknown bugs in most of the environments tested on, requiring correspondence with environment authors in order to resolve them. It is also worth mentioning that it has not previously been shown that assorted common multiagent RL environments are even capable of

Additionally, our results show three important expected properties for an effective method

to find emergent behaviors. First, our method works well with both DQN and PPO. Secondly, in the two simplest and most constrained environments, only a single emergent behavior is found. Finally, in the cases of Pistonball and Ingolstadt, increasing the complexity of the environment allows for fundamentally new kinds of emergent behaviors to be discovered. Aside from being an expected behavior, this serves as initial validation that this method can be used to probe the consequences on emergent behaviors when changing environment designs for systems, which up until now is not generally possible in science. This approach could also allow for determining "why" an emergent behavior occurs, by changing various environment properties and seeing when the emergent behavior allows it to be possible.

Finally, our experiments show that repeatedly conducting training with a single set of hyperparameters is typically able to only find a small number of the behaviors that our full naive MCMES search can find. Out of every single set of hyperparameters tested on an environment that we found more than two emergent behaviors in, only a single set hyperparameters for a single environment was able to find all emergent behaviors we discovered (hyperparameter 8 in Harvest). This would suggest that conducting some sort of search for emergent behaviors beyond repeatedly training is required to make scientific claims about emergent behaviors found in a system using multiagent deep reinforcement learning. The hyperparameter origins of all found emergent behaviors are shown in C.3 in the supplementary materials.

5.3.2 Potential Impacts on Science and Engineering

Understanding the origins of emergent behaviors in populations of agents is frequently elusive to science. If we imagine trying to answer even simple questions like "Why do birds flock

the way they do", one might initially try to examine an individual bird, – but the flocking behavior only emerges as a group behavior, with many birds flying together. This is illustrative of the general study of emergent behaviors. While possible to learn through careful analysis and work such as monitoring the vitals of flocking birds through mobile sensors, this requires extraordinary effort to study a comparatively very small and simple system Sankey and Portugal [2019]. Such an undertaking would be a tremendous endeavour for a larger and more complex systems, like the movement of a humans during a pandemic or a natural disaster. Given a simulation of bird flight and a good method to explore emergent behaviors, you can see what the possible behaviors are, as well as change the configuration of the simulation to see when learned emergent behaviors become possible or not. This sort of A/B testing, when done carefully, allows for determining the origins or causes of different emergent behaviors of interest. While agent based modeling is currently widely used to do this Drogoul and Ferber [2018], it employs simple learning algorithms for their agents, whereas reinforcement learning is consistently able to learn better performance than humans in incredibly complex tasks that embody physical intelligenceSilver et al. [2016], Vinyals et al. [2019], Berner et al. [2019].

Said differently, methods such as this essentially enable doing mechanism design for complex systems. *Mechanism design* Nisan and Ronen [1999] is an interdisciplinary field of research that uses computational game theory to derive incentive structures that result in good outcomes for a system, such as the best way to for taxi-esque companies like Uber to implement surge pricing Gilley [2022], or for the government auction parts of the electromagnetic spectrum to different broadcasting companies to most efficiently use it. Given a sufficiently complex system with enough agents, these symbolic methods inherently break down. The ability to search for emergent behaviors could allow researchers doing this work to conduct empirical studies of

the impacts of different environment features emergent behaviors where these methods break down, to test the incentive structures before they're tested in the real world, and to search for possible new perverse incentives leading to undesired behaviors to be discovered. Beyond economics or allocation, methods like this could enable the design of mechanisms for public health applications, like shaping the movement of humans during with infectious diseases to reduce contangiancy or to civil engineering applications like road and sidewalk design to improve traffic, travel times and traveler safety. Methods like MCMES also allow for testing incentives in scenarios are impossible, such as human behaviors in unprecedented disasters like nuclear weapon deployments or public health crises like novel pandemics.

Another interesting consequence is that the learned policies are represented digitally, policies that produce known emergent behaviors could be put into new situations or isolated scenarios to better probe how the behavior works. Interpretability methods Alharin et al. [2020] could be applied to the neural networks to understand what exactly they are doing, and the policies of the behaviors can be directly analyzed in the worst case scenario. While trying to interpret deep neural networks is challenging, it is still much easier than trying to read the brain of an animal.

MCMES and descendant methods could also be used to discover fundamentally new kinds of emergent behaviors in systems that were unknown to science in completely blue sky directions, like possible new forms of macroeconomic, political or sociological trends given sufficiently advanced simulations. It could be used to discover previously unknown "black swan" events, for which there is virtually no other way to discover, like new methods to conduct financial fraud against banks or new group trading strategies in the stock market.

5.3.3 Implications For Prior Works

The incredibly wide diversity of behaviors we find in our simple test environments casts doubt about existing scientific claims made about complex systems via multiagent reinforcement learning that do not use at least some method of finding multiple emergent behaviors.

The AI Economist Zheng et al. [2020] is a framework which uses MARL to simulate the convergence of tax policies in a resource management environment, specifically attempting to gain scientific insight on how tax policies can be used to incentivize optimal emergent behaviors. This has become influential in macroeconomics as a new potential method for many of the same reasons we bring up in this work. However, this work only uses one set of hyperparameters (repeatedly trained) and did not specifically explore the broader space of possible emergent behaviors. Our work suggests it is very likely that alternative emergent behaviors exist, which may mean that the tax policy discovered is not optimal, which is of even greater concern if such a policy were ever attempted to be applied in the real world.

Similarly, Verma et al. [2018] learned policies for schooling fish in an attempt to learn how physical forces result in schooling behaviors. However, they also only tested one set of hyperparameters and used no method (MCMES or otherwise) to check that no other behaviors could emerge from the same physical conditions. In light of our experiments, it is similarly plausible that additional emergent behaviors are present, and therefore it should be explored before arriving at concrete scientific claims about the origins of the behaviors of fish.

Numerous other papers have attempted to do this and suffer similar flaws, but two additional high profile examples with similar concerns are Tomašev et al. [2020], which used AlphaGo to see likely convergent strategies with different chess rules, and Köster et al. [2022],

which used MARL to try to understand the origins of social behaviors by seeing what emergent properties resulted from different social conditions. Both suffer from this same doubt about potential unknown emergent behaviors making the broader scientific conclusions uncertain.

It is important to keep in mind that applying alternative or new methods to search for emergent behaviors, if effective, could remediate these issues. This is a concern that arises from experiments that do not search for different behaviors, not a requirement to use our method or this work specifically.

5.3.4 Future Directions

The approach we take to implementing MCMES work is an undeniably simple proof of concept. Many obvious engineering steps could be taken to improve its practical use, including:

- Ensuring the black box hyperparameter tuning method used is optimally chosen to save compute during initial training.
- Creating a flexible and purpose made software library implementing these methods with different configurations.
- Using highly performance optimized software for the many training and evaluation runs conducted.
- Creating hardware accelerated environments (simulations that run entirely on accelerator hardware to GPUS and therefore are often 100x-1000x faster than those that run on CPUs) for testing purposes Freeman et al. [2021], Lan et al. [2021].

Additionally, there are many methodological improvements that could be made to our work,

such as:

- Developing an effective system for automatically classifying behaviors, to save human time and make this approach more scalable to larger problems with more behaviors, and to make the behavior classifications more rigorous and less subject to individual human biases. This can be done using discriminator networks as is done in DIAYN Eysenbach et al. [2018].
- Developing methods to have higher likelihoods of either sampling the full space of possible good emergent behaviors, or of finding especially weird behaviors.
- Creating more rigorous protocols for when to define a behavior as mature, as opposed to using an arbitrarily chosen expected reward

There are further many obvious avenues for applying this or descendant methods in new and interesting ways:

- We hope that this method is used to solve the science and engineering problems we previously described, and given initial simulations doing so likely should not be particularly difficult. Applying interpretability methods to learned behaviors to understand their underlying logic would also be highly scientifically relevant.
- Our work only tests environments with up to 20 agents (Pistonball, which unsurprisingly has the most behaviors found), however searching in environments with hundreds or thousands or more agents would likely produce profoundly interesting results, and multiagent deep reinforcement learning is already able to do so Zheng et al. [2018], Yang et al. [2018].
- While not explored in this work, our method should work perfectly well when applied to searching for new behaviors in single agent Simulations

One final point of discussion is "does this uncommon approach to searching for novel behaviors make sense when compared to other curiosity or exploration driven models, such as self-supervised skill search Eysenbach et al. [2018] and policy search Bagnell and Schneider [2001]." We believe it does – unlike the alternatives, our approach of random sampling has the unique advantage of being able to scale to arbitrarily harder and more complex problems with a greater diversity of behaviors given arbitrarily more compute, simply by sampling more and more. While less satisfying, historically in AI (and famously summarized in Sutton's Bitter Lesson Sutton [2019]), the simpler methods which have been able to harness more compute have always won out in terms of performance and real world utility. This, more than any other, is our driving insight into why we hope this method and descendant ones can be useful in real world problems.

Appendix A: Multi-Agent Game Model Definitions

A.1 Defining Partially Observable Stochastic Games

The formal definition of a POSG is shown in Definition 3. This definition can be viewed as the typical Stochastic Games model [Shapley, 1953] with the addition of POMDP-style partial observability.

Definition 3. Formally, a *Partially-Observable Stochastic Game* (POSG) is defined by a tuple $\langle \mathcal{S}, s_0, N, (\mathcal{A}_i)_{i\in[N]}, P, (R_i)_{i\in[N]}, , (\Omega_i)_{i\in[N]}, , (O_i)_{i\in[N]} \rangle$, where:

- $\mathcal{S}$ is the set of possible *states*.
- s_0 is the *initial state*.
- N is the *number of agents*. The *set of agents* is $[N]$.
- $\mathcal{A}_i$ is the set of possible *actions* for agent i.
- $P \colon \mathcal{S} \times \prod_{i\in[N]} \mathcal{A}_i \times \mathcal{S} \to [0,1]$ is the *transition function*. It has the property that for all $s \in \mathcal{S}$, for all $(a_1, a_2, \ldots, a_N) \in \prod_{i\in[N]} \mathcal{A}_i$, $\sum_{s'\in\mathcal{S}} P(s, a_1, a_2, \ldots, a_N, s') = 1$.
- $R_i \colon \mathcal{S} \times \prod_{i\in[N]} \mathcal{A}_i \times \mathcal{S} \to \mathbb{R}$ is the *reward function* for agent i.
- Ω_i is the set of possible *observations* for agent i.

- $O_i \colon \mathcal{A}_i \times \mathcal{S} \times \Omega_i \to [0, 1]$ is the *observation function*. $O_i(a, s, \cdot)$ is a probability distribution: that is, $\sum_{\omega \in \Omega_i} O_i(a, s, \omega) = 1$ for all $a \in \mathcal{A}_i$ and $s \in \mathcal{S}$.

A.2 Defining Extensive Form Games

The definition given here follows closely that of Osborne and Rubinstein [1994], to which we refer the reader for a more in-depth discussion of Extensive Form Games and their formal definition.

Definition 4. An Extensive Form Game is defined by:

- A set of agents $[N] = \{1, 2, \ldots, N\}$.
- A "Nature" player denoted as "agent" 0. For convenience, we define $\mathcal{N} := [N] \cup \{0\}$. The Nature agent is responsible for describing the random, stochastic, or luck-based elements of the game, as described below.
- A set $\tilde{\mathcal{A}}$ of *action sequences*. An action sequence is a tuple $\vec{a} = (a_1, a_2, \ldots, a_k)$ where each element indicates an action taken by an agent. In infinite games, action sequences need not be finite. The set $\tilde{\mathcal{A}}$ indicates all possible sequences of actions that may be taken in the game (i.e., "histories" of players' moves or agents' actions). It satisfies the following properties:
  - The empty sequence is in the set: $\varnothing \in \tilde{\mathcal{A}}$.
  - If $(a_1, \ldots, a_k) \in \tilde{\mathcal{A}}$, then for $l < k$ we also have $(a_1, \ldots, a_l) \in \tilde{\mathcal{A}}$.
  - In infinite games, if an infinite sequence $(a_1, a_2, \ldots)$ satisfies the property that for all k, $(a_1, a_2, \ldots, a_k) \in \tilde{\mathcal{A}}$, then $(a_1, a_2, \ldots) \in \tilde{\mathcal{A}}$.

For a finite sequence $\vec{a} = (a_1, \ldots, a_k)$, denote by $(\vec{a}, a)$ the sequence $(a_1, \ldots, a_k, a)$. Then the set of actions available in the next turn following a sequence $\vec{a}$ is given by $\mathcal{A}(\vec{a}) := \{a \mid (\vec{a}, a) \in \tilde{\mathcal{A}}\}$ (for convenience, we define $\mathcal{A}(\vec{a}) = \varnothing$ if $\vec{a}$ is infinite). We say a sequence of actions $\vec{a}$ is *terminal* if it is either infinite or if it is a maximal finite sequence, i.e. $\vec{a}$ is terminal if and only if $\mathcal{A}(\vec{a}) = \varnothing$. We denote the set of terminal sequences by $T := \{\vec{a} \mid \mathcal{A}(\vec{a}) = \varnothing\}$.

- A function $\tau \colon (\tilde{\mathcal{A}} \setminus T) \to \mathcal{N}$, which specifies the agent whose turn it is to act next after a given sequence of actions. Note that this is not stochastic, but random player order can be captured by inserting a Nature turn.

- A probability distribution $P(\vec{a}, \cdot)$ for Nature's actions. It is defined only for action sequences for which Nature acts next, i.e. sequences $\vec{a} \in \tilde{\mathcal{A}}$ for which $\tau(\vec{a}) = 0$. Specifically, $P(\vec{a}, a)$ is the probability that Nature takes action a after the sequence of actions $\vec{a}$ has occurred.

- For each agent $i \in [N]$, a *partition* H_i of the sequences of actions $\tilde{\mathcal{A}}_i := \{\vec{a} \mid \tau(\vec{a}) = i\}$. The partition H_i is called the *information partition* of agent i, and elements of H_i are called *information sets*. For convenience, define $H := \bigcup_{i \in [N]} H_i$. The information sets must obey the additional property that for any information set $h \in H$ and any two action sequences $\vec{a}, \vec{a}' \in H$, we have $\tau(\vec{a}) = \tau(\vec{a}')$ and $\mathcal{A}(\vec{a}) = \mathcal{A}(\vec{a}')$.

- For each agent $i \in [N]$, a *reward function* $R_i \colon T \to \mathbb{R}$.

Appendix B: Complete Description of Multiplayer Game Modes in the Arcade Learning Environment

Many games have non-default modes that fundamentally change the nature of the game. We elaborate on one's we find notable, or treat as independent games, below.

B.1 Combat

Combat has two main style of play. Tank mode, where the player crawls around a field (potentially with obstacles), and plane mode, where the player cannot control their speed, only their direction. Within the tank category, there are a few options of interest. Table B.1 lists the game modes of interest

- Maze: Map has obstacles which block movement and bullets
- Billiards: If this option is off, bullets are guided (turn when your tank turns). If it is on, bullets can and must (as in billiards) bounce off walls, allowing you to hit around corners. Note that you cannot hit yourself with a ricocheting bullet.
- Invisible: Tanks are not visible except when firing and when running into an obstacle.

In plane mode, there are two types of planes (Jet or Bi-Plane), which can either have strait

Table B.1: Combat Tank modes

Mode	Maze	Billiards	Invisible
1	F	F	F
2	T	F	F
8	F	T	F
9	T	T	F
10	F	F	T
11	T	F	T
13	F	T	T
14	T	T	T

or guided missiles. Jets are faster, and guided missile's direction can be changed by the palyer changing theier direction. Table B.2 lists the game mode numbers.

Table B.2: Plane Tank modes

Mode	Guided Missiles	Jet plane
15	F	F
16	T	F
21	F	T
22	T	T

B.2 Entombed

The original Entombed Atari game does not have an official scoring method in two player (unlike in one player). However, the official manual lists two types of gameplay, competitive play and cooperative play. We implement and distinguish between these two scoring methods using modes:

- Competitive play (Mode 2, default): A player is rewarded, and their opponent penalized when the opponent loses a life.
- Cooperative play (Mode 3): Similar (but not identically) to single player mode, both players

are are rewarded 1 point after passing each of the 5 invisible sections of a particular stage or restarting a stage. Since restarting a stage occurs after losing a life, the players do receive a reward after dying.

B.3 Maze Craze

Maze Craze offers numerous game types, and a visibility setting for each game type. Representing the mode as $4n + k$, the game mode is n and the visibility mode is k. If $k = 0$ the maze is fully visible, if $k = 1$ or $k = 2$ only part of the map is visible, and if $k = 3$. The modes below are the fully visible versions of their game type; only ones we believe are of interest are included for brevity:

- Race (mode 0): First to the end of the maze wins.
- Robbers (mode 4): Randomly moving robbers will kill you if you run into them. Avoid the robbers and complete the maze.
- Capture (mode 44): You need to capture the randomly moving robbers before you can complete the maze. The players can also place green squares that appear identical to the maze walls, confusing their opponents pathing.

B.4 Space invaders

Space invaders modes offer a set of 5 possible options, all combinations of which are possible.

- Moving Shields: Shields move back and forth. Using them for protection is therefore more difficult.
- Zigzagging Bombs: Alien's bombs randomly move horizontally, making them harder to avoid.
- Fast Bombs: Alien's bombs move much faster, making them much harder to avoid.
- Invisible Invaders: Aliens are invisible, making them much harder to hit.
- Alternating Turns: Ability to fire alternates between the players. The switch occurs either when you fire, or after a set period of time.

A particular combination of options can be set by using the following formula (where variables are encoded as 0 or 1):

$$\begin{aligned} 33 + & \text{ MOVING SHIELDS} \\ & + 2 \text{ ZIGZAGGING BOMBS} \\ & + 4 \text{ FAST BOMBS} \\ & + 8 \text{ INVISIBLE INVADERS} \\ & + 16 \text{ ALTERNATING TURNS} \end{aligned}$$

B.5 Video Olympics

Video Olympics is best known for containing Pong, the classic game. However, as its name suggests, Video Olympics contains a wide variety of multiplayer games. Below is a list of the

games we've found to be of note for reinforcement learning research. Table B.3 contains the mode numbers.

- Classic Pong: The classic game.
- Quadrapong: All 4 sides of the map are protected by one player. Score is tracked by team.
- Volleyball: Keep the ball from hitting the ground on your side of the court. The players can "jump" vertically as well as move horizontally.
- Foozpong: There are multiple intertwined rows of defense and offense between the goals.
- Basketball: Put the ball in your opponent's hool.

Table B.3: Video Olympics Modes

Game	Two player	Four player
Classic Pong	4	6
Foozpong	19	21
Quadrapong	N/A	33
Volleyball	39	41
Basketball	45	49

Appendix C: MCMES Data And Additional Figures

C.1 Detailed Explanations of Environment Dynamics

C.1.1 Cooperative Pong

The Cooperative Pong environment (shown in Fig. C.1) is a twist on the popular Pong video game by Atari, first seen in in reinforcement learning in Mnih et al. [2015]. In contrast to the original competitive Atari version of Pong, this environment instead requires cooperative behavior. The objective is to keep the ball from exiting either the left or right side of the screen for as long as possible. Two agents play in this environment, where one agent controls a flat paddle on the left side of the screen, and the second agent controls a triangle-shaped paddle on the right side of the screen. The paddles move in a vertical motion, being able to only block approximately 10% of the whole vertical edge. The environment is assumed to be frictionless and ball collisions only occur with the top and bottom edge of the screen as well as with the paddles. The total kinetic energy of the ball is constant. All agents get a reward of 0.11 at each timestep that the ball is still in play, and a reward of -10 when the environment ends when the ball goes out of the screen. Each agent observes the entire screen, which is a $280 \times 480 \times 3$ RGB image, and can take two discrete actions - moving the paddle up or down. Our experiments use the v4 version of this environment from the PettingZoo library Terry et al. [2021]. The maturity threshold chosen

was a mean reward of more than 90.

C.1.2 Crowd Circle

The Crowd Circle environment (shown in Fig. C.2) is based on a scenario known in Crowd Simulation literature, in which a number of people (or simulated agents) start on the perimeter of a circle, and are tasked to go to the antipodal point, while avoiding collisions. In this work, we use a layout based on Wolinski et al. [2014], with 12 agents on a circle with a radius of approximately 4 meters. Each agent's observation consists of its normalized position, goal position, rotation and current velocity in the global frame. At each decision step, an agent can arbitrarily set its velocity vector, bounded by a maximum magnitude of 2 m/s. Decisions are sampled at a frequency of 12 Hz, with 10 physics update steps performed between decisions, where the agents' previous decisions are repeated.

The reward function is a linear combination of terms corresponding to navigation, urgency, collision avoidance, and maintaining a comfortable speed close to a preferred value of 1.33 m/s. The coefficients of the reward function are chosen manually so that they all contribute to the final behavior, and are equal to the values used in Kwiatkowski et al. [2022].

It is worth noting that the reward function is designed somewhat arbitrarily, and is only a proxy for the real goal of believable or energy-efficient behavior. For this reason, in order to evaluate a behavior independently of the arbitrary reward, we also measure the energy consumption using the model proposed by Whittle [2008], and use it as the primary metric. This choice is inspired by prior work suggesting that humans tend to choose trajectories that minimize total energy usage when moving within crowds Bruneau et al. [2015]. The maturity threshold chosen

was a mean energy usage of at most 33.

C.1.3 Harvest

The Harvest multi-agent environment (shown in Fig. C.3) was first introduced in Pérolat et al. [2017]. It is a gridworld environment where the goal of the agents is to collect apples; each apple gives the agent a reward of 1. Once harvested, the apple on a patch regrows at a certain rate proportional to the number of apples in close proximity. If the local number of apples is zero, then the apples there will never regrow. The goal is for agents to learn a cooperative behavior despite this tragedy of the commons dilemma which ensure that they harvest as many apples as possible without over-harvesting them.

We use the implementation from Leibo et al. [2021], with 16 agents in a 248x456x3 RGB grid. Episodes last for 1000 timesteps, observations are 88x88x3 RGB agent-centric images with 4 frames stacked together, with 8 discrete actions – four for movement directions, two for rotation, one to fire a zapping beam, and one no-op. Note that the zapping beam here disables agents as in Pérolat et al. [2017], instead of giving them a negative reward as in Hughes et al. [2018]. The zapping beam is a means to protect a tract of apples from other agents or penalize agents which over-harvest. The maturity threshold chosen was a mean reward score of at least 40.

C.1.4 Ingolstadt (7 and 21)

The Ingolstadt 7 and Ingolstadt 21 environments (shown in Fig. C.5) are part of the SUMO-RL package Ault and Sharon [2021] which itself is derived from the SUMO simulator Krajzewicz [2010]. The environment is a traffic network, where each agent controls a set of traffic lights at an

intersection. Each agent can take one of n discrete actions, where each action indicates a specific, pre-determined traffic light configuration for a particular intersection, such as the ones shown in Fig. C.4. For Ingolstadt 7, there are 7 intersections corresponding to 7 agents; for Ingolstadt 21, there are 21 intersections corresponding to 21 agents. The Ingolstadt 7 environment is actually a subset of the Ingolstadt 21 environment, being a part of the traffic network that is the bottom right of Ingolstadt 21. The environment is a traffic network with a set of road intersections, each intersection is controlled by a traffic signal agent, and each traffic signal agent can take one of n traffic light configurations. Each agent observes a vector of state variables which include a one-hot indicator of the current light configuration, an indicator of whether a minimum amount of time has passed since a configuration change has occurred, measurements of the number of vehicles present for each lane, and measurements of how many cars are queued for each lane. Every time an agent changes the traffic light configuration, a yellow light phase occurs for a predetermined amount of time. Each vehicle in the environment can enter a waiting configuration, where it has a velocity of less than 0.1 m/s, and is effectively considered to be waiting at a traffic light. The accumulated wait time at a particular timestep for a particular agent is simply the total number of waiting vehicles in all incoming lanes for an agent, multiplied by the time delta for each timestep. The reward for each agent is then equal to the difference in accumulated wait time between the current timestep and the previous timestep. In this manner, each agent is therefore encouraged to get large volumes of vehicles moving at once, without incurring long wait times. A more detailed description of the environment configuration is available in Krajzewicz [2010]. The maturity threshold chosen was a mean reward of at least -0.1 for both Ingolstadt7 and Ingolstadt21.

C.1.5 Knights Archers Zombies

The Knights Archers Zombies environment (shown in Fig. C.6) involves four cooperative agents. The environment falls under the class of multiplayer cooperative games, where agents are incentivized to cooperate toward the common goal of preventing the game from ending. The agents consist of two archers with bows that fire projectiles travelling in straight lines, and knights that have swords which can be swung only a short distance away from their positions. Zombies spawn from the top of the screen and walk down in unpredictable paths. When a zombie collides with an arrow, the zombie dies and the arrow disappears; when it collides with a sword, the zombie dies; when it collides with either an archer or a knight, the archer or knight dies. An agent that kills a zombie gets a reward of 1, there are no penalties for dying to a zombie. The observation space of this environment is a flat vector consisting of a concatenation of vectors depicting properties of each entity in the environment, where an entity can either be an arrow, sword, zombie, knight or archer. The properties of each entity are the heading, global position, and the type of the entity. The agents can take one of 6 discrete actions – forward movement, backward movement, left rotation, right rotation, weapon use or do nothing. Our experiments use the v10 version of this environment from the PettingZoo library Terry et al. [2021]. The maturity threshold chosen is a mean reward of at least 3.

C.1.6 Multiwalker

The Multiwalker environment (shown in Fig. C.7) is a sidescroller environment with three bipedal robots attempting to carry a large rectangular package as far right as possible. This environment is the multiagent variant of the original bipedal walking environment available in

Brockman et al. [2016], which is itself a physics-based robotics environment aiming at allowing agents to learn bipedal walking with a very simplified simulated robot. Each agent gets the same positive reward which depends on the distance the package travels over each time step. If any walker's torso touches the ground, it will be considered to have fallen, and it receives a reward of -110, while every other agent gets a reward of -100. Any agent falling causes the environment to come to an end. The environment also ends when the package touches the ground, in which case each agent gets a reward of -100. When the package crosses 75 meters to the right, the environment ends and no penalty is incurred on any agent. For reference, the hull length of a single walker is 6.4 meters. For every meter that the package moves to the right, each agent gets a reward of 4. The observation for each agent is a 31-dimensional vector indicating the agent's joint positions, ground contact indicators, relative positions of other agents, relative position and angle of the package, and 10 lidar sensor readings aimed at various points of the ground around the agent. Each agent has a continuous action space consisting of four actions, each being the normalized torque applied at each joint. Our experiments use the v9 version of this environment from the PettingZoo library Terry et al. [2021]. The maturity threshold chosen is a mean reward of at least 70.

C.1.7 Pistonball

The Pistonball environment (shown in Fig. C.8) is a physics-based 2D environment where 5 to 20 vertically moving pistons line the bottom of the screen, each controlled by its own agent, resulting in a total of 5 to 20 agents. Each piston can be actuated individually at any given time, but has a limited actuation range. A ball the size of 3 piston widths starts on the right side of the

screen. The goal of the environment is for the pistons to cooperatively get the ball to roll to the left edge of the screen by actuating the pistons at the right time. All agents get the same reward, which depends on the distance the ball rolls to the left in the given timestep. Each agent observes a $457 \times 120 \times 3$ RGB image which is centered around the agent, as well as the two pistons (or wall) next to the agent, and the space immediately above the agent. The agents have a single continuous action to take, which is the normalized distance that the piston should be extended. Our experiments use the v6 version of this environment from the PettingZoo library Terry et al. [2021]. The maturity threshold chosen is a mean reward of at least 90.

C.1.8 Pursuit

The Pursuit environment (shown in Fig. C.9) is a 16×16 grid world with a large immovable obstacle in the center, first introduced in Zheng et al. [2018]. Pursuit is an environment aimed at discovering collective intelligence and swarm behaviors for environments with many independent agents that must all depend on each other. It has 8 controlled agents known as pursuers and 30 uncontrolled agents known as evaders. The evaders move around randomly. Each pursuer only observes a 7×7 grid around itself. Each pursuer gets a uniform reward of -1 at every time step, a reward of 0.01 for being 1 unit next to an evader, and a reward of 5 when being part of a group of evaders that succesfully catch a pursuer. Each agent can take one of 5 discrete actions, which consist of taking a step in either one of the four cardinal directions or to do nothing. For an evader to be considered captured, it must be surrounded in all four of its immediate cardinal positions by either evaders or a wall. When an evader is captured, it disappears from the environment and each evader that was responsible for its capture gets the reward of 5. Our experiments use

Figure C.1: The Cooperative Pong environment.

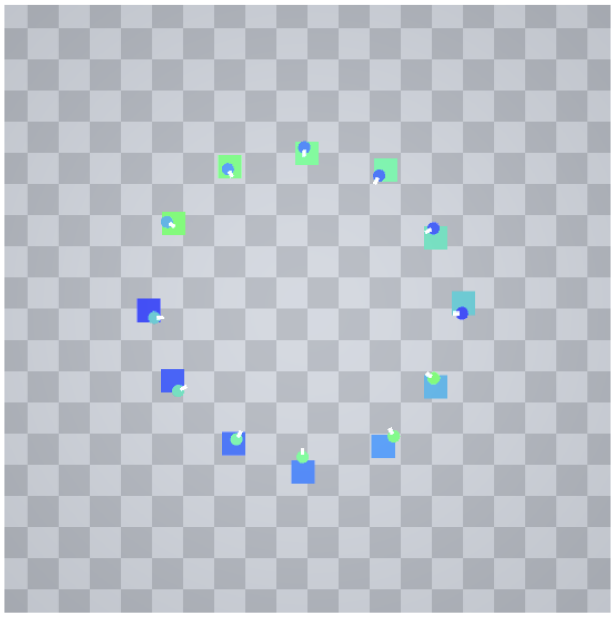


Figure C.2: The Crowd Circle environment. 12 agents are placed on the perimeter of a circle, and must reach the antipodal points.

the v6 version of this environment from the PettingZoo library Terry et al. [2021]. The maturity threshold chosen is a mean reward of at least 50.

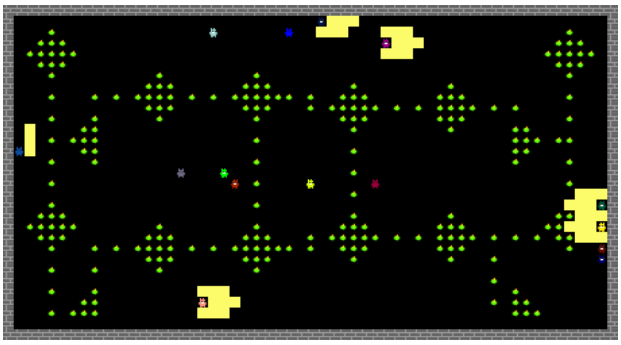


Figure C.3: The Harvest environment, the apples are represented as green objects, while the agents are represented by the various coloured entities scattered around the map. An agent's zapping beam can be viewed as the yellow flash in from of some of the agents.

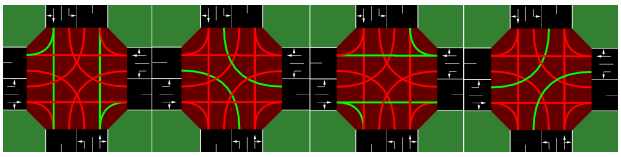


Figure C.4: A set of traffic light configurations for a 2 way intersection. Green lines indicate lanes that are free to move and actuated with a green light. Red lanes are correspondingly actuated by a red light.

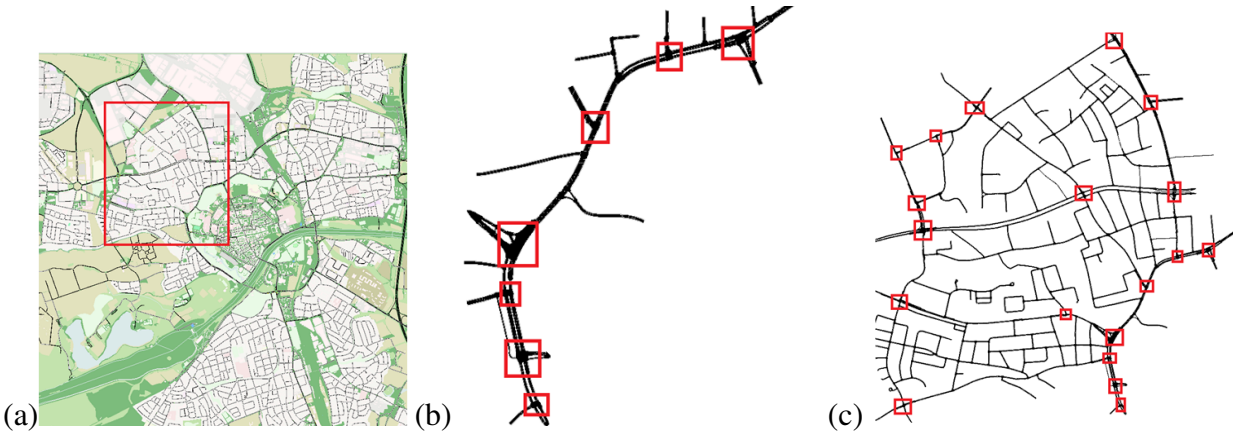


Figure C.5: Breakdown of where the Ingolstadt 7 and Ingolstadt 21 environments come from. (a) The part in Ingolstadt, a city in Bavaria, Germany where the road networks are derived from. (b) Ingolstadt 7 map, the red boxes indicate intersections where the agents can control traffic light configurations. (c) Ingolstadt 21 map, the red boxes indicate intersections where the agents can control traffic lights. Note that the Ingolstadt 7 map is major road in the bottom right of the Ingolstadt 21 map. Krajzewicz [2010]

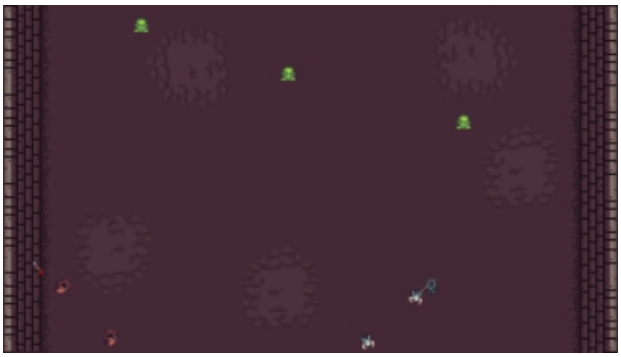


Figure C.6: The Knights Archer Zombies environment. Two knights and two archers kill zombies to prevent them from reaching the bottom of the screen. The knights are on the bottom right of the screen, while the archers are on the bottom left. The zombies are shown in green. The rightmost knight is swinging its sword while the left one is not. The left most archer has fired an arrow that is going off the screen.

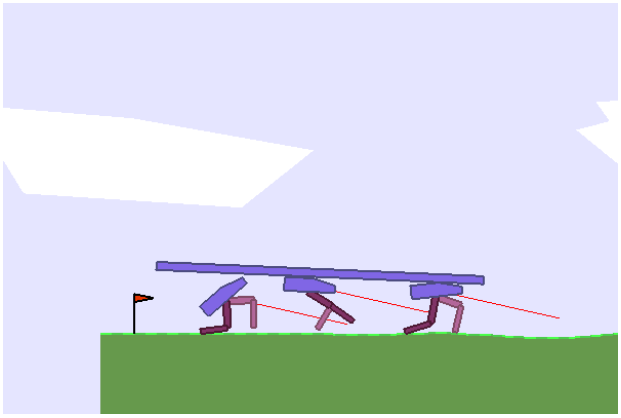


Figure C.7: The Multiwalker environment. Three bipedal robots start off with a rectangular package on their heads. The goal is to get the package as far right as possible.

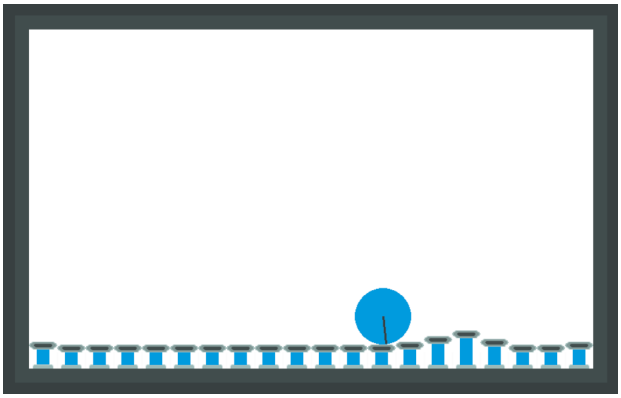


Figure C.8: The Pistonball environment. A line of pistons lie the bottom of the screen, each controlled by a separate agent. The goal is to get the ball to the right wall as fast as possible.

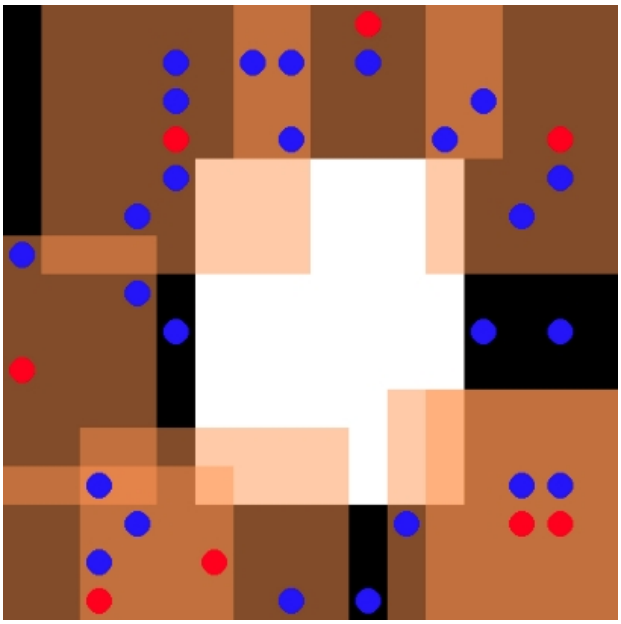


Figure C.9: The Pursuit environment. The evaders are shown as red dots, while evaders are shown as blue dots. The large white obstacle at the center of the environment is immovable, and orange boxes centered around the pursuers define the observations that the pursuers see.

C.2 Tables of Emergent Behaviors Found

C.2.1 Cooperative Pong

There is only one emergent from the Cooperative Pong environment, where the agents simply work to keep the ball in play.

C.2.2 Crowd Circle

Efficient navigation This is the desired behavior, where all agents move towards their destination at a stable speed of 1.33 m/s. This results in a lower reward contribution from the urgency term, but a higher contribution from maintaining a comfortable speed.

Fast navigation In this behavior, agents move towards their goals at the maximum allowed speed of 2 m/s. This behavior is, in principle, suboptimal according to the intended design. However, the distribution of the achieved energy usages of trained agents significantly overlaps with the values achieved by efficient navigation agents, and due to the multiobjective nature of the task at hand, it is also Pareto-optimal in the various components of the reward function.

C.2.3 Harvest

Table C.1: List of behaviors discovered in the Harvest environment with their number of occurrence among all the training runs.

Behavior	Number of occurrences
Protective	16
Aggressive	11
Mutual Protection	7
Over-Harvesting	12
Running Through Arena	10

Protective An agent protects and farms resources from a certain patch, tagging other agents that intrude on the patch, but otherwise staying in its own area.

Aggressive Aggressive is similar to Protective, with the difference being that a single agent protects a significantly larger area - more than two patches - at the same time. This usually occurs in the corners, where the agent spends a lot of time firing on nearby agents and restricting access.

Mutual Protection Multiple agents protect and harvest a certain patch, farming the patch with fewer shots fired on average.

Over Harvesting Agents over-harvest a large region of patches in the arena. Since the individual agent's field of view is limited, they cannot find any more apples to collect, and essentially wander around randomly until they find more apples. Generally, this behavior also causes agents to have have lower average rewards.

Running Through Arena Agents run through multiple patches in the arena constantly; the opposite of Protective. Note that this is the optimal strategy for a single agent in a map - see Pérolat et al. [2017].

C.2.4 Ingolstadt7

In Ingolstadt 7, all agents eventually learn the same emergent behavior. The top intersection was mostly non-congested and clear due to the traffic light configuration appropriately changing to prevent too much buildup. The bottom loop (off to the right, not the very bottom) was constantly congested even when cross traffic was light since the traffic light configuration never changed. Allowing right turns on red lights from the rightmost lane may reduce this congestion, but this behavior is not learnt in any of the experiments.

C.2.5 Ingolstadt21

Table C.2: List of behaviors discovered in the Ingolstadt 21 environment with their number of occurrence amongst all the training runs.

Behavior	Number of occurrences
Congested	42
Stop and Go	35

Interestingly, the main part of this map where different behaviors can occur is the traffic section where the Ingolstadt 7 map was derived; all other parts of the map developed smooth traffic.

Congested In this behavior, intersections on the bottom right corner of the map tend to be backed up with many more cars. The agents in this environment favour building up long car queues before releasing the queue, and then letting the queue pass through with minimal resistance. This results in more congested behavior throughout the street on the bottom right corner.

Stop and Go In this behavior, the entire map shows less-congested behavior, where the traffic light configuration changes appropriately to prevent buildup. The agents in this environment favour releasing cars quickly, preventing much buildup. The immediate downside to this is that cars often encounter more red light intersections, causing more stop and go traffic, although there seem to be less buildup in general.

C.2.6 Knights Archers Zombies

Table C.3: List of behaviors discovered in the Knights Archers Zombies environment with their number of occurrence amongst all the training runs.

Behavior	Number of occurrences
Aggressive	6
Suboptimal Knights	26
Archer Dominant	28
Knights Hide	2
Arrow Corridor	1
Safe	9

Aggressive All agents actively seek out and kill zombies by moving up to the zombies as soon as they spawn. This includes archers going up to zombies and shooting them, and knights going up to zombies and killing them.

Suboptimal Knights Knights move up to the top edge of the screen and continuously swing their swords. This usually ends in them being killed by zombies early on as there is a small delay in between sword swings that leaves them vulnerable. The likely cause of this behavior is because the knights' sword range cannot compete with the archers' use of arrows. In an attempt to get zombies killed before they are killed by the archers, they choose to take out zombies the moment they spawn which entails them going to the top edge and blindly swinging their swords.

Archer Dominant All agents hardly change positions, with the knights performing constant sword swings while the archers fire arrows in the direction of zombies. With this behavior, the knights almost always end the environment with no reward.

Knights Hide The knights hide in the bottom left corner of the map, leaving all the killing of zombies to the archers.

Arrow Corridor The archers position themselves and constantly fire arrows in a way that makes it impossible for any zombies to kill one archer without first getting killed by arrows fired by another archer. This effectively splits the map into a 'safe zone' and a 'danger zone'. The knights hide in the 'safe zone' throughout the entire episode. A visual depiction of this behavior is shown in Fig. C.10.

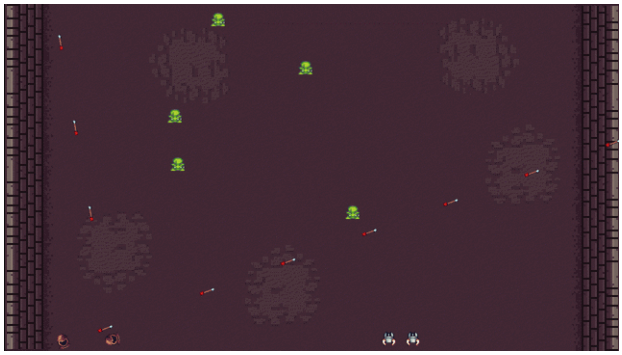


Figure C.10: The arrow corridor behavior in the Knights Archers Zombies environment. The archers constantly fire arrows in the directions shown in the image, which makes it impossible for a zombie to get to any of the agents without first getting killed by an arrow.

Safe All agents retreat to the bottom edge of the map, with the archers firing arrows in the direction of any zombies that spawn. The knights don't do anything unless a zombies comes close enough for it to be killed by a sword swing.

C.2.7 Multiwalker

Table C.4: List of behaviors discovered in the Multiwalker environment with their number of occurrence amongst all the training runs.

Behavior	Number of occurrences
Rocking	9
Rear Bouncing	8
Sliding	21
Carrying	8

Rocking

All agents walk in the same manner in that their hulls are rocked back and forth, acting as a conveyor for the package on the top.

Rear Bouncing

The rightmost agent walks with its hull as stable as possible while the leftmost agent walks by bouncing its hull up and down. Over time, the package gradually bounces and slides more toward the right.

Sliding

The rightmost agent walks with its hull held at a lower height while the leftmost agent walks with its hull held at a higher height. Through the vibrations induced from walking, the package progressively bounces up and down, and coupled with the height difference of the top and bottom agents' hulls, the package gradually slides towards the right.

Carrying

All agents keep their upper hulls relatively stable and immobile, such that the package has enough friction to stay put on the hulls. All agents then walk in the same manner to progressively

move the hull forward.

C.2.8 Pistonball (Continuous)

Table C.5: List of behaviors discovered in the the Pistonball environment using 20 pistons trained with PPO with their number of occurrence amongst all the training runs.

Behavior	Number of occurrences
Bumping	1
Crowdsurfing	50
Ramp	78
Tapping	36
Wave	108
Sporadic Boosting	9
Multiramp	210
Crowdsurfing then Ramp	13
Crowdsurfing then Tapping	13
Crowdsurfing then Wave	4
Ramp then Bumping	3
Ramp then Crowdsurfing	1
Ramp then Rollover	2
Ramp then Tapping	28
Ramp then Wave	114
Tapping then Rollover	3
Wave then Ramp	7
Wave then Rollover	3

Bumping

The pistons bump the location directly behind the ball using 2 or more pistons at a time. This differs from taping which only uses one piston at a time.

Crowdsurfing

Pistons move in such a way that the ball moves along a moving cavity.

Ramp

Pistons build a ramp starting from the right end of the screen to the left.

Tapping

The pistons move the ball by continuously tapping the location right behind the ball when it slows down.

Wave

The pistons form a singular wave behind the ball that pushes the ball to the end.

Sporadic Boosting

The pistons randomly boost the ball by bumping the back of it when the ball slows down or comes to a stop.

Multiramp

The pistons form multiple ramps throughout the path of the ball. This causes the ball to have a stop-go motion due to continuously encountering the highside of a ramp after rolling off the one prior.

Crowdsurfing then Ramp

The pistons perform crowdsurfing to get the ball up to speed before forming a ramp sending the ball the rest of the way.

Crowdsurfing then Tapping

The ball crowdsurfs for a moment to gather speed, then the pistons utilize tapping to keep its speed up.

Crowdsurfing then Wave

The pistons start off by crowdsurfing the ball, then proceed to wave it the rest of the way.

Ramp then Bumping

The pistons build a ramp initially, and bump the ball whenever it loses speed.

Ramp then Crowdsurfing

The pistons start off by building a ramp, then proceed to crowdsurf it the rest of the way once the ball slows down.

Ramp then Rollover

The pistons form a ramp, then right before the ball reaches the end, they form a wave rolling the ball the rest of the way.

Ramp then Tapping

The pistons form a ramp then tap the ball whenever it slows down.

Ramp then Wave

The pistons form a ramp which gets the ball moving, before waving it the rest of the way.

Tapping then Rollover

The ball is tapped most of the way before a small wave is used to force the ball to the end.

Wave then Ramp

The ball starts moving with a wave before the pistons transition into a ramp that sends the ball the rest of the way.

Wave then Rollover

The pistons start with a wave before slowing the ball down at the end and rolling the ball over with a small wave.

C.2.9 Pistonball (Continuous, 5 Pistons)

Table C.6: List of behaviors discovered in the the Pistonball environment using 5 pistons with their number of occurrence amongst all the training runs.

Behavior	Number of Occurance
Crowdsurfing	39
Raise and Drop	34
Ramp	340
Tapping	100
Wave	125
Ramp then Tapping	6
Ramp then Wave	2
Rear Push	73
Rear Push then Ramp	3

Crowdsurfing

Pistons move in such a way that the ball moves along a moving cavity.

Raise and Drop

Pistons initially raise the ball up high, before briefly causing the ball to bounce such that it falls and rolls a far distance.

Ramp

Pistons build a ramp at the area where the ball stops.

Tapping

Pistons perform a quick and short up and down motion when they're slightly behind the ball and the ball is moving relatively slowly.

Wave

The pistons move in such a way that there is a constant hill behind the ball.

Ramp then Tapping

A ramp is built at the beginning to get the ball moving, before it is tapped the rest of the way.

Ramp then Wave

The pistons build a ramp at the beginning before they wave it the rest of the way.

Rear Push

The pistons gently extend if they are behind the ball to make the ball move.

Rear Push then Ramp

The pistons form a slow ramp that appears as though a transition between rear push then ramp.

C.2.10 Pistonball (Discrete, DQN)

Table C.7: List of behaviors discovered in the discrete-action Pistonball environment using 20 pistons trained with DQN with their number of occurrence amongst all the training runs.

Behavior	Number of occurrences
Crowdsurfing	126
Crowdsurfing then Ramp	3
Crowdsurfing then Sporadic Boosting	2
High Wave	6
Tight Wave	203
Raise and Drop	1
Ramp	4
Sporadic Boosting	28
Reverse Wave	2
Small Ramp	2
Tapping	42
Wave	231
Tight Wave then Sporadic Boosting	4
Raise and Drop then Tapping	5
Ramp then Rollover	2
Ramp then Tapping	14
Ramp then Wave	37
Sporadic Boosting then Wave	2
Tapping then Rollover	5
Tapping then Wave	4
Wave then Ramp	3
Wave then Rollover	7
Wave then Tapping	2

Crowdsurfing

Pistons move in such a way that the ball moves along a moving cavity.

High Wave

The pistons move in such a way that a wave forms behind the ball, but do so in such a manner that the pistons are in a higher vertical position than the normal wave.

Tight Wave

The pistons form a wave that consist of two or less raised pistons behind the ball.

Raise and Drop

Pistons initially raise the ball up high, before briefly causing the ball to bounce such that it falls and rolls a far distance.

Ramp

Pistons build a ramp starting from the right end of the screen to the left.

Sporadic Boosting

The pistons boost the ball by bumping the back of it when the ball slows down or comes to a stop.

Reverse Wave

The pistons form a double wave – a larger one followed by a smaller one, and the ball rides the divot between these two waves, giving the impression of the ball following a large wave.

Small ramp

Similar to ramp, but on a much smaller scale.

Tapping

The pistons move the ball by continuously tapping the location right behind the ball when it slows down.

Wave

The pistons form a singular wave behind the ball that pushes the ball to the end.

Crowdsurfing then Ramp

The pistons perform crowdsurfing to get the ball up to speed before forming a ramp sending the ball the rest of the way.

Tight Wave then Sporadic Boosting

The pistons form a tight wave and then performs sporadic boosting at the end.

Raise and drop then tapping

The pistons perform a raise and drop, and then taps the ball whenever it slows down.

Ramp then Rollover

The pistons form a ramp, then right before the ball reaches the end, they form a wave rolling the ball the rest of the way.

Ramp then Tapping

The pistons form a ramp then tap the ball whenever it slows down.

Ramp then Wave

The pistons form a ramp which gets the ball moving, before waving it the rest of the way.

Sporadic Boosting then Wave

The ball is at first randomly boosted, and then waved when it gathers speed.

Tapping then Rollover

The ball is tapped most of the way before a small wave is used to force the ball to the end.

Tapping then Wave

The ball is initially tapped, before a wave is used to send it the rest of the way to the end.

Wave then Ramp

The ball starts moving with a wave before the pistons transition into a ramp that sends the ball the rest of the way.

Wave then Rollover

The pistons start with a wave before slowing the ball down at the end and rolling the ball over with a small wave.

Wave then Tapping

The pistons start with a wave to get the ball moving before transitioning to tapping to keep the ball's speed up.

C.2.11 Pistonball (Discrete, PPO)

Table C.8: List of behaviors discovered in the the discrete-action Pistonball environment using 20 pistons trained with PPO with their number of occurrence amongst all the training runs.

Behavior	Number of occurrences
Crowdsurfing	11
Raise and Drop	5
Raise and Drop then Tapping	1
Raise and Drop then Wave	1
Ramp	24
Ramp then Raise and Drop	3
Ramp then Tapping	10
Ramp then Wave	5
Tapping	5
Tapping then Ramp	1
Wave	27
Wave then Raise and Drop	2
Wave then Ramp	5
Wave then Tapping	3

Crowdsurfing

Pistons move in such a way that the ball moves along a moving cavity.

Raise and Drop

Pistons initially raise the ball up high, before briefly causing the ball to bounce such that it falls and rolls to the goal.

Raise and drop then Tapping

The pistons perform a raise and drop, and then tap the ball whenever it slows down.

Raise and Drop then Wave

The pistons perform a raise and drop and wave it when it slows down.

Ramp

Pistons build a ramp starting from the right end of the screen to the left.

Ramp then Raise and Drop

Pistons build a ramp initially, then perform a raise and drop at the end.

Ramp then Tapping

The pistons ramp the ball, then tap it constantly to keep its speed up.

Ramp then Wave

The pistons start with a ramp which transitions into a wave.

Tapping

The pistons tamp the ball at the right moments when it slows down to speed it up again.

Tapping then Ramp

The pistons start by tapping then ball then transition to a ramp when the ball is at a sufficient speed.

Wave

The pistons form a singular wave behind the ball that pushes the ball to the end.

Wave then Raise and Drop

The pistons start with a wave which increases in amplitude until it eventually becomes a raise and drop.

Wave then Ramp

The pistons start with a wave then die down into a ramp.

Wave then Tapping

The pistons start with a wave until the ball is fast enough, then start tapping to keep its

speed up.

C.2.12 Pursuit

Table C.9: List of behaviors discovered in the Pursuit environment with their number of occurrence amongst all the training runs.

Behavior	Number of occurrences
Pack Hunting	42
Multi Group	38

Pack Hunting

The pack hunting behavior includes all 8 agents moving as a single pack and sweeping through the environment in either a clockwise or counter clockwise rotation until all evaders are caught. The benefit of this behavior is that the evaders tend to be caught faster, but not all agents in the pack get rewards.

Scattered Hunting

In this behavior, there exists two or more groups that move around the map. Whenever a group finds an evader, they stick to it until enough groups come together to form enough pursuers to capture the evader. The agents tend to not cluster as close to each other in an attempt to have more exploration over the whole map at once.

C.3 Hyperparameter Behavior Breakdown

C.3.1 Cooperative Pong

No different behaviors found.

C.3.2 Crowd Circle

Hyperparameter origins not logged with the stack.

C.3.3 Harvest

The breakdown of hyperparameters to distict behaviors is shown in Table C.10.

Table C.10: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Protective	0(8), 2(2), 3(2), 4(1), 8(3), 15(1)	16
Aggressive	0(4), 6(1), 8(2), 9(1), 14(1), 15(2)	11
Mutual Protection	0(4), 8(1), 10(1), 14(1)	7
Over-Harvesting	2(3), 3(1), 6(1), 8(2), 9(1), 11(1), 15(3)	12
Running Through the Arena	0(3), 3(1), 8(2), 11(1), 14(2), 15(1)	10

C.3.4 Ingolstadt7

No different behaviors found.

C.3.5 Ingolstadt21

The breakdown of hyperparameters to distict behaviors is shown in Table C.11.

Table C.11: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Congested	0(3), 3(13), 4(4), 7(4), 8(3), 9(2), 11(2), 12(1), 13(3), 15(7)	42
Stop and Go	0(7), 3(6), 4(1), 7(4), 8(3), 9(10), 14(2), 15(2)	35

C.3.6 Knights Archers Zombies

The breakdown of hyperparameters to distict behaviors is shown in Table C.12.

Table C.12: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Aggressive	6(4), 8(2)	6
Suboptimal Knights	0(3), 2(8), 8(6), 11(5), 14(2), 15(2)	26
Archer Dominant	0(13), 2(1), 3(2), 4(3), 8(3), 9(1), 14(2), 15(2)	28
Knights Hide	8(2)	2
Arrow Corridor	4(1)	1
Safe	3(1), 4(2), 5(2), 12(4)	9

C.3.7 Multiwalker

The breakdown of hyperparameters to distict behaviors is shown in Table C.13.

Table C.13: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Rocking	6(5), 7(6), 8(4), 9(5), 10(1)	21
Rear Bouncing	0(5), 10(3)	8
Sliding	2(3), 7(4), 14(1)	8
Carrying	8(2), 10(5), 14(2)	9

C.3.8 Pistonball (Continuous)

The breakdown of hyperparameters to distict behaviors is shown in Table C.14.

Table C.14: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Bumping	0(1)	1
Crowdsurfing	4(16), 5(1), 7(1), 9(7), 10(1), 11(5), 13(2), 14(4), 15(13)	50
Ramp	0(5), 1(1), 2(2), 4(1), 5(5), 6(10), 7(11), 8(5), 9(9), 10(6), 11(4), 12(5), 13(11), 15(1)	78
Tapping	0(1), 1(1), 4(3), 5(7), 6(1), 7(3), 8(4), 9(8), 10(3), 12(1), 13(2), 14(1)	36
Wave	0(13), 1(4), 2(5), 4(5), 5(15), 6(3), 7(5), 8(10), 9(10), 10(9), 11(12), 12(2), 13(11), 14(3), 15(1),	108
Sporadic Boosting	0(2), 2(2), 9(1), 10(1), 14(1), 15(2),	9
Multiramp	0(16), 1(24), 2(10), 4(9), 5(5), 6(18), 7(22), 8(12), 9(13), 10(17), 11(8), 12(4), 13(10), 14(28), 15(14),	210
Crowdsurfing then Ramp	1(2), 2(2), 4(2), 5(3), 8(1), 14(1), 15(2),	13
Crowdsurfing then Tapping	4(1)	1
Crowdsurfing then Wave	0(1), 1(2), 5(1)	4
Ramp then Bumping	0(1), 4(1), 5(1)	3
Ramp then Crowdsurfing	1(1)	1
Ramp then Rollover	8(1), 14(1)	2
Ramp then Tapping	11(2), 13(4), 14(2), 1(3), 2(4), 4(1), 5(3), 6(2), 7(2), 8(4), 9(1)	28
Ramp then Wave	0(6), 10(13), 11(6), 12(3), 13(11), 14(5), 15(7), 1(9), 2(22), 5(8), 6(5), 7(6), 8(12), 9(1)	114
Tapping then Rollover	14(1), 2(1), 6(1)	3
Wave then Ramp	0(4), 13(1), 1(1), 4(1)	7
Wave then Rollover	1(1), 2(1), 8(1)	3

C.3.9 Pistonball (Continuous, 5 Pistons)

The breakdown of hyperparameters to distict behaviors is shown in Table C.15.

Table C.15: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Crowdsurfing	10(1), 11(3), 12(3), 13(4), 14(1), 15(3), 1(3), 2(4), 3(4), 4(3), 5(7), 7(1), 8(2)	39
Raise and Drop	0(5), 10(2), 11(4), 12(2), 13(1), 1(4), 2(1), 3(1), 4(1), 5(6), 6(4), 8(2), 9(1)	34
Ramp	0(13), 10(27), 11(19), 12(11), 13(20), 14(20), 15(36), 1(29), 2(23), 3(12), 4(20), 5(23), 6(14), 7(20), 8(6), 9(47)	340
Tapping	0(1), 1(1), 4(3), 5(7), 6(1), 7(3), 8(4), 9(8), 10(3), 12(1), 13(2), 14(1)	100
Wave	0(8), 1(9), 2(6), 3(9), 4(10), 5(7), 6(7), 7(7), 8(3), 9(2), 10(10), 11(7), 12(11), 13(9), 14(17), 15(3)	125
Ramp then Tapping	2(1), 7(1), 8(2), 12(1), 13(1),	6
Ramp then Wave	3(1), 15(1)	2
Rear Push	0(8), 1(4), 2(7), 3(6), 4(6), 5(6), 6(6), 7(5), 8(5), 10(3), 11(7), 12(2), 13(3), 14(4), 15(1),	73
Rear Push then Ramp	0(1), 11(1), 12(1)	3

C.3.10 Pistonball (Discrete, DQN)

The breakdown of hyperparameters to distict behaviors is shown in Table C.16.

Table C.16: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Crowdsurfing	0(1), 1(7), 2(5), 3(5), 4(11), 5(10), 6(17), 7(10), 8(10), 9(6) 10(11), 11(3), 12(5), 13(5), 14(5), 15(15)	126
Crowdsurfing then Ramp	10(1), 15(2)	3
Crowdsurfing then Sporadic Boosting	5(1), 10(1),	2
High Wave	0(4), 5(1), 13(1)	6
Tight Wave	1(17), 2(25), 3(23), 4(27), 5(23), 6(28), 7(7), 8(19), 9(23) 10(1), 11(6), 12(1), 13(1), 14(2)	203
Raise and Drop	12(1)	1
Ramp	3(1), 10(1), 11(1), 12(1)	4
Sporadic Boosting	0(1), 1(3), 2(1), 4(6), 5(5), 8(1) 10(4), 11(3), 13(1), 15(3)	28
Reverse Wave	11(2)	2
Small Ramp	12(2)	2
Tapping	0(4), 1(5), 2(12), 3(2), 5(1), 7(7) 10(4), 12(1), 13(1), 14(5)	42
Wave	0(23), 1(9), 2(6), 3(2), 4(2), 6(3), 7(9), 8(16), 9(17), 10(15), 11(31), 12(19), 13(36), 14(21), 15(22)	231
Tight Wave then Sporadic Boosting	5(1), 8(2), 9(1)	4
Raise and Drop then Tapping	0(1), 7(1), 8(1), 10(1), 12(1)	5
Ramp then Rollover	14(2)	2
Ramp then Tapping	0(1), 10(3), 11(2), 12(5), 14(3)	14
Ramp then Wave	0(2), 1(1), 2(1), 9(1), 10(6), 11(2), 12(8), 13(1), 14(10), 15(5),	37
Sporadic Boosting then Wave	0(1), 5(1)	2
Tapping then Rollover	12(5)	5
Tapping then Wave	14(1), 4(3)	4
Wave then Ramp	1(3)	3
Wave then Rollover	13(3), 14(1), 15(3)	7
Wave then Tapping	8(1), 9(1)	2

C.3.11 Pistonball (Discrete, PPO)

The breakdown of hyperparameters to distict behaviors is shown in Table C.17.

Table C.17: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Crowdsurfing	0(1), 1(1), 2(3), 3(2), 7(2), 11(1), 12(1),	11
Raise and Drop	0(1), 1(2), 2(1), 12(1)	5
Raise and Drop then Tapping	1(1)	1
Raise and Drop then Wave	12(1)	1
Ramp	0(6), 1(3), 2(3), 3(8), 12(2), 14(2)	24
Ramp then Raise and Drop	1(2), 2(1)	3
Ramp then Tapping	0(5), 1(4), 2(1)	10
Ramp then Wave	1(2), 3(1), 9(1), 12(1)	5
Tapping	0(3), 1(1), 12(1),	5
Tapping then Ramp	0(1)	1
Wave	0(7), 1(4), 2(2), 3(7), 14(7)	27
Wave then Raise and Drop	0(1), 12(1)	2
Wave then Ramp	0(1), 2(1), 3(3)	5
Wave then Tapping	2(2), 3(1)	3

C.3.12 Pursuit

The breakdown of hyperparameters to distict behaviors is shown in Table C.18.

Table C.18: List of discovered behaviors and their hyperparameter sets with number of occurrences in brackets.

Behavior	Hyperparameter Sets	Total Occurance
Pack Hunting	0(1), 1(4), 4(4), 5(5), 7(4), 8(10), 10(8), 11(3), 12(3)	42
Multi Group	0(1), 1(5), 4(11), 5(3), 7(2), 8(4), 9(3), 10(4), 12(3), 14(2)	38

Bibliography

Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 66–83. Springer, 2017.

Robert H Crites and Andrew G Barto. Elevator group control using multiple reinforcement learning agents. *Machine learning*, 33(2):235–262, 1998.

Jeffrey Monaco, David Ward, and Andrew Barto. Automated aircraft recovery via reinforcement learning: Initial experiments. *Advances in neural information processing systems*, 10:1022–1028, 1997.

Michael L Littman. A tutorial on partially observable markov decision processes. *Journal of Mathematical Psychology*, 53(3):119–125, 2009.

David Silver. Introduction to reinforcement learning. Lecture series, 2015. Available at the time of writing at https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ.

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

Joshua Achiam. OpenAI spinning up. Online Textbook, 2018. Available at the time of writing at https://spinningup.openai.com/en/latest/.

Eric A Hansen, Daniel S Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In *AAAI*, volume 4, pages 709–715, 2004.

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

Jerzy Filar and Koos Vrieze. *Competitive Markov decision processes*. Springer Science & Business Media, 2012.

Alexander F Siegenfeld and Yaneer Bar-Yam. An introduction to complex systems science and its applications. *Complexity*, 2020, 2020.

David Pines. Emergence: A unifying theme for 21st century science. *Santa Fe Institute Bulletin*, 28(2), Nov 2014. URL `https://medium.com/sfi-30-foundations-frontiers/emergence-a-unifying-theme-for-21st-century-science-4324ac0f951e`.

Stuart Kauffman. *At home in the universe: The search for the laws of self-organization and complexity*. Oxford university press, 1996.

James Gleick. *Chaos: Making a new science*. Penguin, 2008.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

GitHub. openai/gym dependents, 2021. URL https://web.archive.org/web/20210527224052/https://github.com/openai/gym/network/dependents?dependent_type=PACKAGE.

Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinícius Flores Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas W. Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. Openspiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019. URL http://arxiv.org/abs/1908.09453.

Jiayi Weng, Minghao Zhang, Alexis Duburcq, Kaichao You, Dong Yan, Hang Su, and Jun Zhu. Tianshou. https://github.com/thu-ml/tianshou, 2020.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

Mikayel Samvelyan, Tabish Rashid, Christian Schröder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philip H. S. Torr, Jakob N. Foerster, and Shimon Whiteson. The starcraft multi-agent challenge. *CoRR*, abs/1902.04043, 2019. URL http://arxiv.org/abs/1902.04043.

Chris Nota. The autonomous learning library. https://github.com/cpnota/autonomous-learning-library, 2020.

J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, pages 195–210. Morgan Kaufmann Publishers Inc., 1996.

Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of Operations Research*, 27(4):819–840, 2002. doi: 10.1287/moor.27.4.819.297. URL https://doi.org/10.1287/moor.27.4.819.297.

L. S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10): 1095–1100, 1953. ISSN 0027-8424. doi: 10.1073/pnas.39.10.1095.

Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Neural Information Processing Systems (NIPS)*, 2017.

Lianmin Zheng, Jiacheng Yang, Han Cai, Ming Zhou, Weinan Zhang, Jun Wang, and Yong Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Siqi Liu, Guy Lever, Josh Merel, Saran Tunyasuvunakool, Nicolas Heess, and Thore Graepel. Emergent coordination through competition. *CoRR*, abs/1902.07151, 2019. URL http://arxiv.org/abs/1902.07151.

Martin J Osborne and Ariel Rubinstein. *A course in game theory*. MIT press, 1994.

David Ha. Slime volleyball gym environment. https://github.com/hardmaru/slimevolleygym, 2020.

Rene Vidal, Omid Shakernia, H Jin Kim, David Hyunchul Shim, and Shankar Sastry. Probabilistic pursuit-evasion games: theory, implementation, and experimental evaluation. *IEEE transactions on robotics and automation*, 18(5):662–669, 2002.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

J K Terry, Nathaniel Grammel, Ananth Hari, Luis Santos, and Benjamin Black. Revisiting parameter sharing in multi-agent deep reinforcement learning. *arXiv preprint arXiv:2005.13625*, 2020a.

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

Eugene Vinitsky, Natasha Jaques, Joel Leibo, Antonio Castenada, and Edward Hughes. An open source implementation of sequential social dilemma games. https://github.com/eugenevinitsky/sequential_social_dilemma_games/, 2019. GitHub repository.

Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. *arXiv preprint arXiv:1702.03037*, 2017.

Edward Hughes, Joel Z Leibo, Matthew Phillips, Karl Tuyls, Edgar Dueñez-Guzman, Antonio García Castañeda, Iain Dunning, Tina Zhu, Kevin McKee, Raphael Koster, et al. Inequity aversion improves cooperation in intertemporal social dilemmas. In *Advances in neural information processing systems*, pages 3326–3336, 2018.

J K Terry and Benjamin Black. Multiplayer support for the arcade learning environment. *arXiv preprint arXiv:2009.09341*, 2020.

Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995. ISSN 0001-0782. doi: 10.1145/203330.203343. URL https://doi.org/10.1145/203330.203343.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587): 484–489, 2016.

Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. The hanabi challenge: A new frontier for AI research. *CoRR*, abs/1902.00506, 2019. URL http://arxiv.org/abs/1902.00506.

Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*, 2019.

StefanieAnnaBaby LingLi AshwiniPokle. Analysis of emergent behavior in multi agent environments using deep reinforcement learning. 2018.

Sriram Ganapathi Subramanian, P. Poupart, Matthew E. Taylor, and N. Hegde. Multi type mean field reinforcement learning. In *AAMAS*, 2020.

Y. Chen, M. Zhou, Ying Wen, Y. Yang, Y. Su, W. Zhang, Dell Zhang, J. Wang, and Han Liu. Factorized q-learning for large-scale multi-agent systems. In *DAI '19*, 2019.

Igor Mordatch and Pieter Abbeel. Emergence of grounded compositional language in multi-agent populations. *arXiv preprint arXiv:1703.04908*, 2017.

Gregory Palmer. *Independent learning approaches: Overcoming multi-agent learning pathologies in team-games*. The University of Liverpool (United Kingdom), 2020.

Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

J K Terry, Benjamin Black, and Ananth Hari. Supersuit: Simple microwrappers for reinforcement learning environments. *arXiv preprint arXiv:2008.08932*, 2020b.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Antonin Raffin. Rl baselines3 zoo. `https://github.com/DLR-RM/rl-baselines3-zoo`, 2020.

Dawid Laszuk. Ai-traineree. `https://github.com/laszukdawid/ai-traineree`, 2020.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

Shengyi Huang, Rousslan Dossa, and Chang Ye. Cleanrl: High-quality single-file implementation of deep reinforcement learning algorithms. `https://github.com/vwxyzjn/cleanrl/`, 2020.

Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588*, 2019.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

Bowen Baker, Ingmar Kanitscheider, Todor M. Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *CoRR*, abs/1909.07528, 2019a. URL `http://arxiv.org/abs/1909.07528`.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Yasuhiro Fujita and Shin-ichi Maeda. Clipped action policy gradient. *arXiv preprint arXiv:1802.07564*, 2018.

Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on Robot Learning*, pages 651–673. PMLR, 2018.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897): 414–419, 2022.

Peter R Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, et al. Outracing champion gran turismo drivers with deep reinforcement learning. *Nature*, 602 (7896):223–228, 2022.

DJ Wu and Yanjun Sun. The emergence of trust in multi-agent bidding: a computational approach. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, pages 8–pp. IEEE, 2001.

Ruoying Sun, Shoji Tatsumi, and Gang Zhao. Multiagent reinforcement learning method with an improved ant colony system. In *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat. No. 01CH37236)*, volume 3, pages 1612–1617. IEEE, 2001.

Cristiano Castelfranchi. The theory of social functions: Challenges for multi-agent-based social simulation and multi-agent learning. *Journal of Cognitive Systems Research*, 2:5–38, 2001.

Edwin D de Jong and Luc Steels. A distributed learning algorithm for communication development. *Complex Systems*, 2003.

Martin A Nowak, Joshua B Plotkin, and Vincent AA Jansen. The evolution of syntactic communication. *Nature*, 404(6777):495–498, 2000.

Martin A Nowak, Natalia L Komarova, and Partha Niyogi. Evolution of universal grammar. *Science*, 291(5501):114–118, 2001.

Jakob Foerster, Ioannis Alexandros Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/file/c7635bfd99248a2cdef8249ef7bfbef4-Paper.pdf.

Angeliki Lazaridou, Alexander Peysakhovich, and Marco Baroni. Multi-agent cooperation and the emergence of (natural) language. *arXiv preprint arXiv:1612.07182*, 2016.

Serhii Havrylov and Ivan Titov. Emergence of language with multi-agent games: Learning to communicate with sequences of symbols. *arXiv preprint arXiv:1705.11192*, 2017.

Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*, 2019b.

Ended Learning Team, Adam Stooke, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, et al. Open-ended learning leads to generally capable agents. *arXiv preprint arXiv:2107.12808*, 2021.

Stephan Zheng, Alexander Trott, Sunil Srinivasa, Nikhil Naik, Melvin Gruesbeck, David C Parkes, and Richard Socher. The ai economist: Improving equality and productivity with ai-driven tax policies. *arXiv preprint arXiv:2004.13332*, 2020.

Siddhartha Verma, Guido Novati, and Petros Koumoutsakos. Efficient collective swimming by harnessing vortices through deep reinforcement learning. *Proceedings of the National Academy of Sciences*, 115(23):5849–5854, 2018.

Nenad Tomašev, Ulrich Paquet, Demis Hassabis, and Vladimir Kramnik. Assessing game balance with alphazero: Exploring alternative rule sets in chess. *arXiv preprint arXiv:2009.04374*, 2020.

Mattia Gazzola, Andrew A Tchieu, Dmitry Alexeev, Alexia de Brauer, and Petros Koumoutsakos. Learning to school in the presence of hydrodynamic interactions. *Journal of Fluid Mechanics*, 789:726–749, 2016.

Gautam Reddy, Antonio Celani, Terrence J Sejnowski, and Massimo Vergassola. Learning to soar in turbulent environments. *Proceedings of the National Academy of Sciences*, 113(33): E4877–E4884, 2016.

James E Herbert-Read, Andrea Perna, Richard P Mann, Timothy M Schaerf, David JT Sumpter, and Ashley JW Ward. Inferring the rules of interaction of shoaling fish. *Proceedings of the National Academy of Sciences*, 108(46):18726–18731, 2011.

Ehud D Karpas, Adi Shklarsh, and Elad Schneidman. Information socialtaxis and efficient collective behavior emerging in groups of information-seeking agents. *Proceedings of the National Academy of Sciences*, 114(22):5589–5594, 2017.

Carsten Hahn, Thomy Phan, Thomas Gabor, Lenz Belzner, and Claudia Linnhoff-Popien. Emergent escape-based flocking behavior using multi-agent reinforcement learning. *arXiv preprint arXiv:1905.04077*, 2019.

Edward Hill, Marco Bardoscia, and Arthur Turrell. Solving heterogeneous general equilibrium economic models with deep reinforcement learning. *arXiv preprint arXiv:2103.16977*, 2021.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

Joel Z Leibo, Edgar A Dueñez-Guzman, Alexander Vezhnevets, John P Agapiou, Peter Sunehag, Raphael Koster, Jayd Matyas, Charlie Beattie, Igor Mordatch, and Thore Graepel. Scalable evaluation of multi-agent reinforcement learning with melting pot. In *International Conference on Machine Learning*, pages 6187–6199. PMLR, 2021.

James Ault and Guni Sharon. Reinforcement learning benchmarks for traffic signal control. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

Daniel Krajzewicz. Traffic simulation with sumo–simulation of urban mobility. In *Fundamentals of traffic simulation*, pages 269–293. Springer, 2010.

Lucas N. Alegre. SUMO-RL. `https://github.com/LucasAlegre/sumo-rl`, 2019.

Ariel Kwiatkowski, Vicky Kalogeiton, Julien Pettré, and Marie-Paule Cani. Understanding reinforcement learned crowds, 2022. URL `https://arxiv.org/abs/2209.09344`.

Daniel WE Sankey and Steven J Portugal. When flocking is costly: reduced cluster-flock density over long-duration flight in pigeons. *The Science of Nature*, 106(7):1–5, 2019.

Alexis Drogoul and Jacques Ferber. Multi-agent simulation as a tool for studying emergent processes in societies. In *Simulating societies*, pages 127–142. Routledge, 2018.

Noam Nisan and Amir Ronen. Algorithmic mechanism design. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 129–140, 1999.

Joseph Gilley. H3: Tiling the earth with hexagons. Recorded lecture, 2022. Available at the time of writing at https://www.youtube.com/watch?v=ay2uwtRO3QE.

Alnour Alharin, Thanh-Nam Doan, and Mina Sartipi. Reinforcement learning interpretation methods: A survey. *IEEE Access*, 8:171058–171077, 2020.

Raphael Köster, Dylan Hadfield-Menell, Richard Everett, Laura Weidinger, Gillian K Hadfield, and Joel Z Leibo. Spurious normativity enhances learning of compliance and enforcement behavior in artificial agents. *Proceedings of the National Academy of Sciences*, 119(3), 2022.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL `http://github.com/google/brax`.

Tian Lan, Sunil Srinivasa, and Stephan Zheng. Warpdrive: Extremely fast end-to-end deep multi-agent reinforcement learning on a gpu, 2021.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

Yaodong Yang, Rui Luo, Minne Li, Ming Zhou, Weinan Zhang, and Jun Wang. Mean field multi-agent reinforcement learning. *arXiv preprint arXiv:1802.05438*, 2018.

J Andrew Bagnell and Jeff G Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 2, pages 1615–1620. IEEE, 2001.

Richard Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13:12, 2019.

D. Wolinski, S. J. Guy, A.-H. Olivier, M. Lin, D. Manocha, and J. Pettré. Parameter estimation and comparative evaluation of crowd simulations. *Computer Graphics Forum*, 33(2):303–312, May 2014. doi: 10.1111/cgf.12328. URL `https://doi.org/10.1111/cgf.12328`.

Michael W. Whittle. *Gait analysis: an introduction*. Butterworth-Heinemann, Elsevier, Edinburgh, 4th ed., reprinted edition, 2008. ISBN 978-0-7506-8883-3.

Julien Bruneau, Anne-Hélène Olivier, and Julien Pettré. Going Through, Going Around: A Study on Individual Avoidance of Groups. *IEEE Transactions on Visualization and Computer Graphics*, 21(4):9, April 2015. doi: 10.1109/TVCG.2015.2391862.

Julien Pérolat, Joel Z. Leibo, Vinícius Flores Zambaldi, Charlie Beattie, Karl Tuyls, and Thore Graepel. A multi-agent reinforcement learning model of common-pool resource appropriation. In *NIPS*, 2017.