PettingZoo: A Standard API for Multi-Agent Reinforcement Learning

J. K. Terry[*][†]
j.k.terry@swarmlabs.com

Benjamin Black[*][†]
benjamin.black@swarmlabs.com

Nathaniel Grammel[†]
ngrammel@umd.edu

Mario Jayakumar[†]
mariojay@umd.edu

Ananth Hari[‡]
ahari1@umd.edu

Ryan Sullivan[*][†]
ryan.sullivan@swarmlabs.com

Luis Santos[§]
lss@umd.edu

Rodrigo Perez [¶]
rlazcano@umd.edu

Caroline Horsch[*][†]
caroline.horsch@swarmlabs.com

Clemens Dieffendahl [‖]
dieffendahl@campus.tu-berlin.de

Niall L. Williams[†]
niallw@umd.edu

Yashas Lokesh[†]
yashloke@umd.edu

Praveen Ravi[†]
pravi@umd.edu

Abstract

This paper introduces the PettingZoo library and the accompanying Agent Environment Cycle ("AEC") games model. PettingZoo is a library of diverse sets of multi-agent environments with a universal, elegant Python API. PettingZoo was developed with the goal of accelerating research in Multi-Agent Reinforcement Learning ("MARL"), by making work more interchangeable, accessible and reproducible akin to what OpenAI's Gym library did for single-agent reinforcement learning. PettingZoo's API, while inheriting many features of Gym, is unique amongst MARL APIs in that it's based around the novel AEC games model. We argue, in part through case studies on major problems in popular MARL environments, that the popular game models are poor conceptual models of games commonly used in MARL and accordingly can promote confusing bugs that are hard to detect, and that the AEC games model addresses these problems.

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

[*] Swarm Labs
[†] Department of Computer Science | University of Maryland, College Park
[‡] Department of Electrical and Computer Engineering | University of Maryland, College Park
[§] Department of Mechanical Engineering | University of Maryland, College Park
[¶] Maryland Robotics Center | University of Maryland, College Park
[‖] Faculty of Electrical Engineering and Computer Science | Technical University of Berlin

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

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 repository is available at `https://www.pettingzoo.ml`.

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 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 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: 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.

2.1 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 C.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., 2017, 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 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 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 Appendix C.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 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 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.

3 PettingZoo Design Goals

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

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

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.

4 Case Studies of Problems With The POSG Model in MARL

To supplement the description of the problems with the POSG models described in Section 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.

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 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.

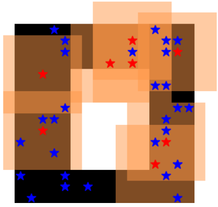


Figure 4: The *pursuit* environment from Gupta et al. [2017].

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 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 experimentally validate this performance improvement in Appendix A.1, showing that on average this change resulted in up to a 22% performance in the expected reward of a learned policy.

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

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 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).

In Appendix A.1 we go through a case study of a race condition like this happening in the open source implementation of the social sequential dilemma game environments [Vinitsky et al., 2019]. These are popular multi-agent grid world environments intended to study emergent behaviors for various forms of resource management, and has imperfect tie-breaking in a case where two agents try to act on resources in the same grid while using a simultaneous API. This bug in particular illustrates how extraordinarily difficult making all tie-breaking truly unbiased is in practice even for fairly simple environments. We defer this to the appendix as explaining the specific origin requires a large amount of exposition and diagrams about the rules of the environment.

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 Section 4.1.
- It prevents developers adding confusing and easy-to-introduce race conditions, as described in Section 4.2.
- It more closely models how computer games are executed in code, as described in Section 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 Section 2.2.
- It is simple enough to serve as a mental model, especially for beginners, unlike the EFG model as discussed in Section 2.2 and illustrated in the definition in Appendix C.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 Section 2.1.
- It is the least bad option for a universal API, compared to simultaneous stepping, as alluded to in Section 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.

In Appendix C.3 we mathematically formalize the AEC games model, however understanding the formalism in full is not essential to understanding the paper. In Appendix D we further prove that for every AEC game an equivalent POSG exists and that for every POSG an equivalent AEC game exists. This shows that the AEC games model is as powerful a model as the most common current model of multi-agent environments.

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 Section 6.4). In the AEC games model, we deviate from the POSG model by introducing the "environment" agent, 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 5, which serves as the inspiration for the name "agent environment cycle".

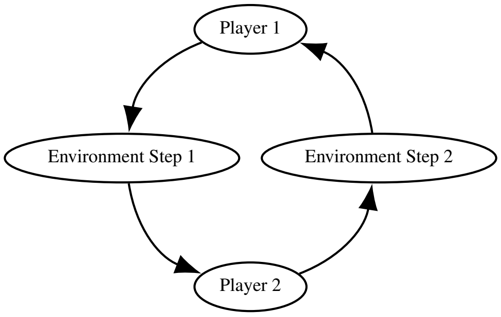


Figure 5: The AEC diagram of Chess

6 API Design

6.1 Basic API

The PettingZoo API is shown in Figure 6, and the strong similarities to the Gym API (Figure 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 6: An example of the basic usage of Pettingzoo

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.

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).

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 Section 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.

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.

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. All default environments included are surveyed in depth in Appendix B.

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.

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].

Acknowledgements

Justin Terry was supported during part of this work by the QinetiQ Fundamental Machine Learning Fellowship. Thank you to Kyle Sang for their contributions to the documentation website. Thank you Rohan Potdar and Sang Hyun Son for their contributions to the Butterfly benchmarks. Thank you to Deepthi Raghunandan and Christian Clauss for their contributions to testing and continuous integration. Thank you to the PettingZoo community for the numerous bug reports and contributions to the package, especially Ross Allen and their group.

References

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`.

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`.

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.

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

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.

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

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

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.

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

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.

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

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.

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`.

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

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.

Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

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.

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`.

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.

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.

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

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

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

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

G. Palmer. Independent learning approaches: Overcoming multi-agent learning pathologies in team-games. 2020.

Stefanie Anna Baby Ling Li Ashwini Pokle. Analysis of emergent behavior in multi agent environments using deep reinforcement learning. 2018.

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

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

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`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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.

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.

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.

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

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.

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

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

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

Justin 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*, 2020d.

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.

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.

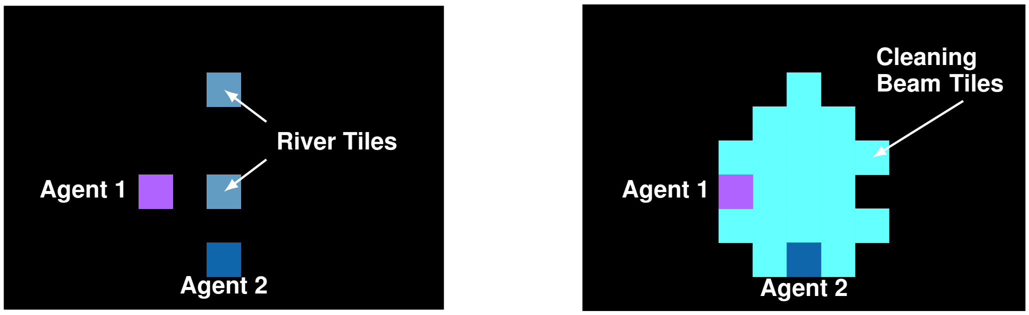
Eugene Vinitsky, Natasha Jaques, Joel Leibo, Antonio Castenada, and Edward Hughes. An open source implementation of sequential social dilemma games. https://github.com/eugenevinitsky/sequential_social_dilemma_games/, 2019. GitHub repository.

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.

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

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.

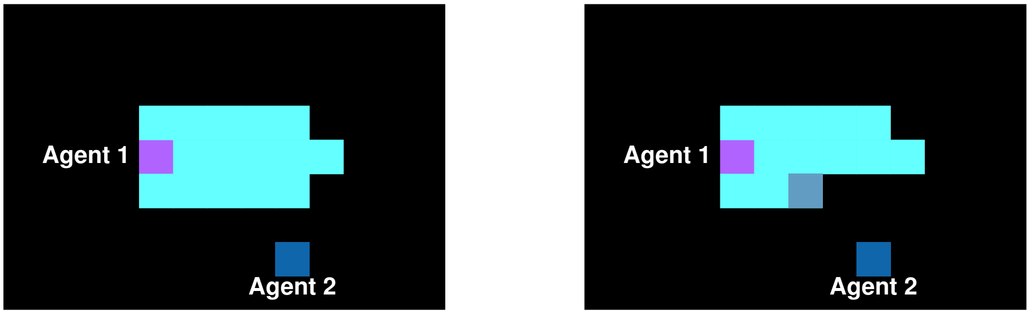
Lianmin Zheng, Jiacheng Yang, Han Cai, Weinan Zhang, Jun Wang, and Yong Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. *arXiv preprint arXiv:1712.00600*, 2017.



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



(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.

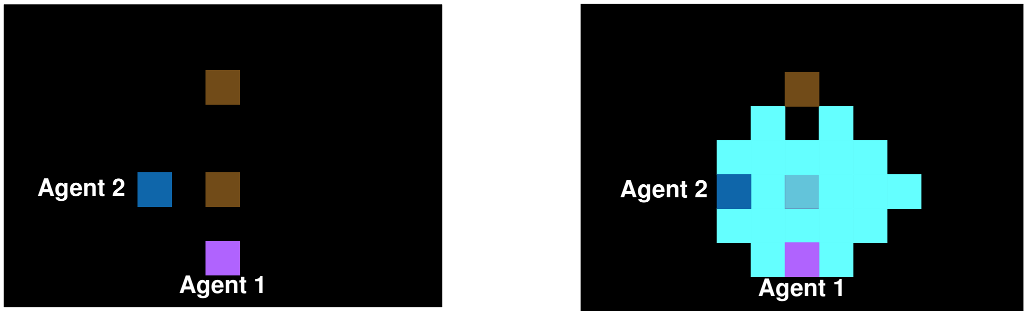
Figure 8: An example of Agent 1 using the "clean" action while facing East. The beams extend to a length of up to five tiles. The "main" beam extends directly in front of the agent, while two auxiliary beams start at the tiles directly next to the agent (one to the left and one to the right) and also extend up to five tiles. A beam stops when it hits a dirty river tile.

A Additional Case Study Information

A.1 Race Conditions in Sequential Social Dilemma Games

The Sequential Social Dilemma Games, introduced in Leibo et al. [2017], are a kind 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 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 8a. 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 7 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



(a) The same setup as in Figure 7, but with the agent labels reversed.

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

Figure 9: 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 7, Agent 2's beam was able to reach the far beam because Agent 1's beam cleaned the near tile first.

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 different result, seen in Figure 9. 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.

A.2 Reward Defects in Pursuit

We validated the impact of reward pruning experimentally by training parameter shared Ape-X DQN [Horgan et al., 2018] (the best performing model on pursuit [Terry et al., 2020d]) four times using RLLib [Liang et al., 2017] with and without reward pruning, achieving better results with reward pruning every time and 22.03% more total reward on average Figure 10a, while PPO [Schulman et al., 2017] learned 16.12% more reward on average with this Figure 10b. Saved training logs and all code needed to reproduce the experiments and plots is available in the supplemental materials.

B Default Environments

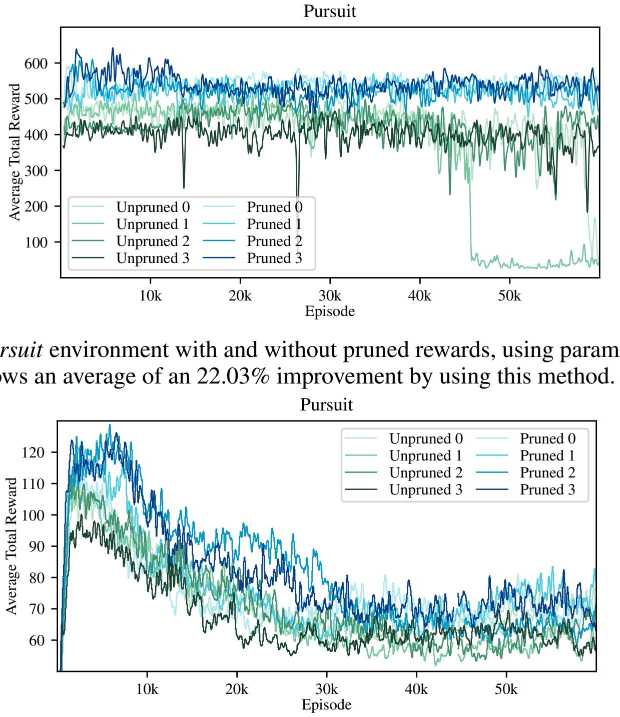
This section surveys all the environments that are included in PettingZoo by default.

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 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



(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.

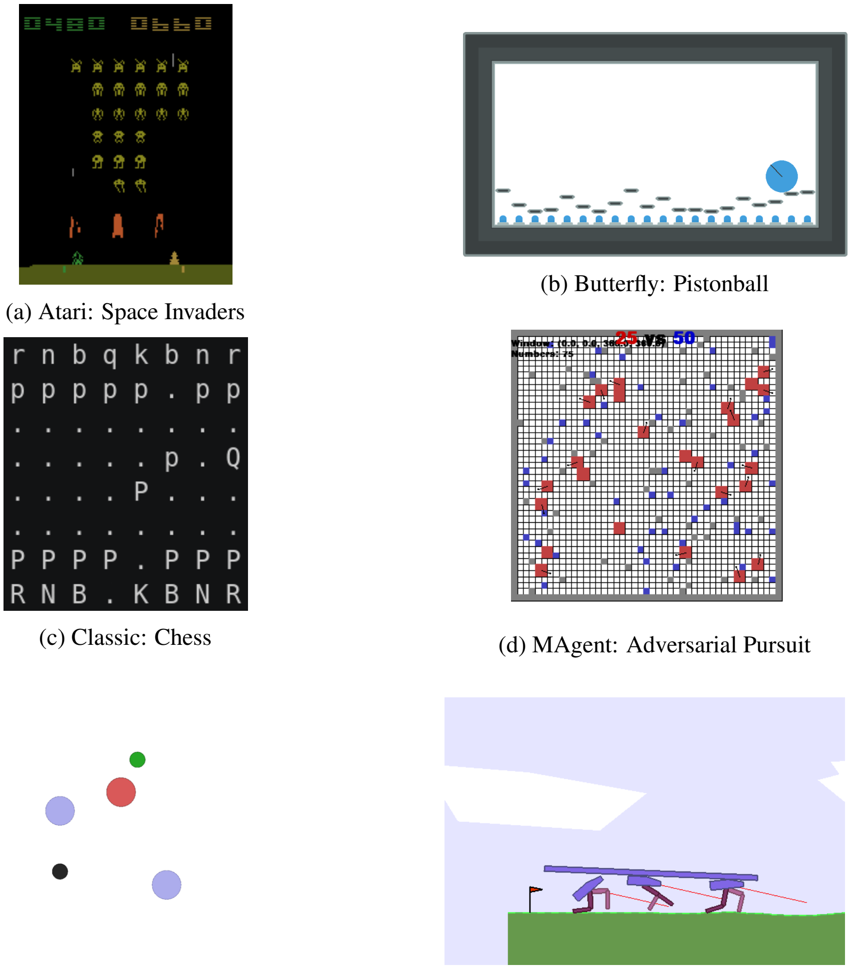
in Figure 11b, is an environment where pistons need to coordinate to move a ball to the left, while only being able to observe a local part of the screen. It requires learning nontrivial emergent behavior and indirect communication to perform well. *Knights Archers Zombies* is a game in which agents 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. [2017] 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 [Pokle, 2018], heterogeneous agents [Subramanian et al., 2020], and efficient learning methods with many agents [Chen et al., 2019]. We include a



(a) Atari: Space Invaders

(b) Butterfly: Pistonball

(c) Classic: Chess

(d) MAgent: Adversarial Pursuit

(f) SISL: Multiwalker

(e) MPE: Simple Adversary

Figure 11: Example environments from each class.

number of preset configurations, for example the *Adversarial Pursuit* environment shown in Figure 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 possibilities. 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 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.

B.1 Butterfly Baselines

Whne 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., 2020c] 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, learns in the Cooperative Pong and Prospector environments without achieving optimal policies, and does not learn almost at all in the Knights Archers Zombies environment. Plots showing results of 10 training runs of the best hyperparameters are shown in Figure 12. All code and hyperparameters for these runs is available at `https://github.com/jkterry1/Butterfly-Baselines`.

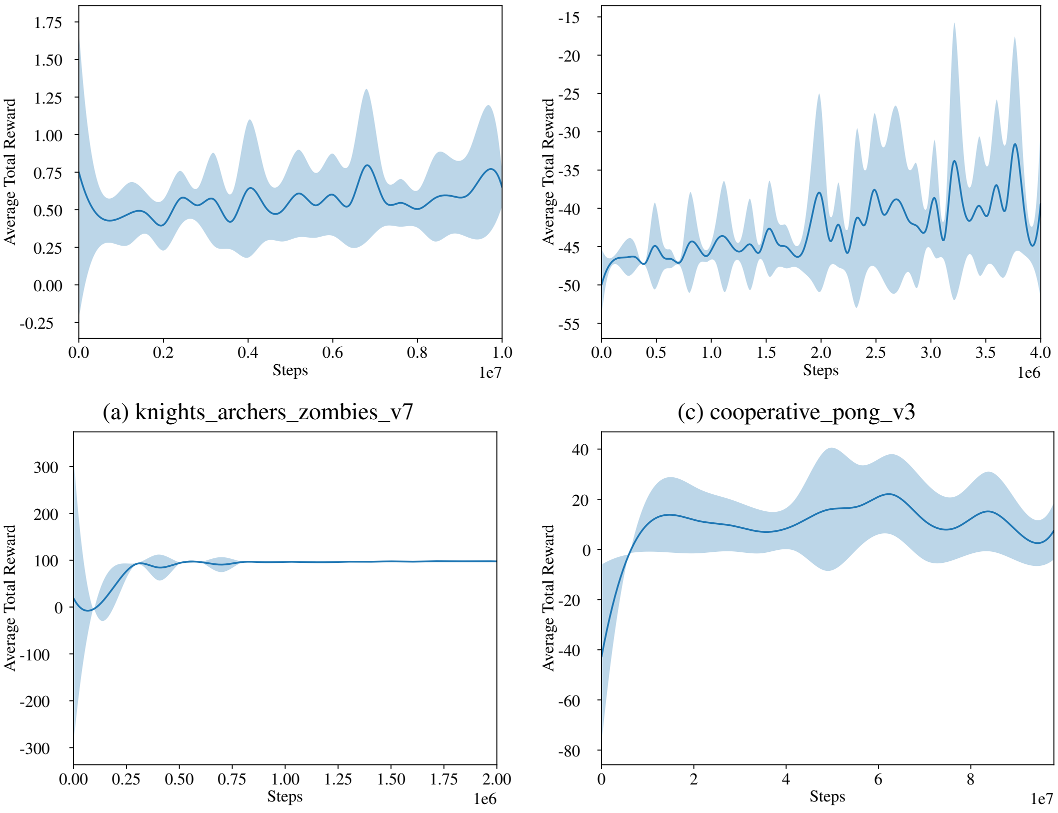
C Formal Definitions

C.1 Partially Observable Stochastic Games

The formal definition of a POSG is shown in Definition 1. This definition can be viewed as the typical Stochastic Games model [Shapley, 1953] with the addition of POMDP-style partial observability.

Definition 1. A *Partially-Observable Stochastic Game* (POSG) is 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.



(a) knights_archers_zombies_v7

(c) cooperative_pong_v3

(b) pistonball_v4

(d) prospector_v4

Figure 12: Total reward when learning on each Butterfly environment via parameter-shared PPO.

- 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*. It has the property that $\sum_{\omega \in \Omega_i} O_i(a, s, \omega) = 1$ for all $a \in \mathcal{A}_i$ and $s \in \mathcal{S}$.

C.2 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 2. 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* $R_i \colon T \to \mathbb{R}$.

C.3 Agent Environment Cycle Games

As mentioned in Section 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 3. An *Agent-Environment Cycle Game* (AEC Game) is 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.
- $O_i : \mathcal{S} \times \Omega_i \to [0,1]$ is the *observation function* for agent i. $O_i(s,\omega)$ is the probability of agent i observing ω while in state s.
- $\nu : \mathcal{S} \times \mathcal{N} \times \mathcal{A} \times \mathcal{N} \to [0,1]$ 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 $R_j(s,i,a_i,s',r')$.

D Omitted Proofs

D.1 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:

$$G_{\text{AEC}} = \langle \mathcal{S}', N, \{\mathcal{A}_i\}, \{T_i\}, P', \{R_i'\}, \{\Omega_i\}, \{O_i'\}, \nu \rangle$$

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 $i \in [N]$, $a_i \in \mathcal{A}_i$.

- $T_i((s, a_1, a_2, \ldots, a_i, \ldots, a_N), a'_i) = (s, a_1, a_2, \ldots, a'_i, \ldots, a_N)$.
- For $\mathbf{s} = (s, a_1, a_2, \ldots, a_N)$ and $\mathbf{s}' = (s', a_1, a_2, \ldots, a_N)$, we define $P'(\mathbf{s}, \mathbf{s}') = P(s, a_1, a_2, \ldots, a_N, s')$. 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. $\Pr_{\text{G},t}(\omega_i = \omega \mid s_t = s) = \Pr_{G_{\text{AEC}},t(N+1)+i}(\omega_i = \omega \mid s_{t(N+1),0} = s)$.

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 Appendix D.1.

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

Definition 4. 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 $R'_i((s, j, \mathbf{r}), a, (s', j', \mathbf{r}')) = \mathbf{r}'_i$ □