JaxMARL: Multi-Agent RL Environments in JAX

Anonymous Author(s)
Affiliation
Address
email

Abstract

Benchmarks play an important role in the development of machine learning algorithms. Reinforcement learning environments are traditionally run on the CPU, limiting their scalability with typical academic compute. However, recent advancements in JAX have enabled the wider use of hardware acceleration to overcome these computational hurdles by producing massively parallel RL training pipelines and environments. This is particularly useful for multi-agent reinforcement learning (MARL) research where not only multiple agents must be considered at each environment step, adding additional computational burden, but also the sample complexity is increased due to non-stationarity, decentralised partial observability, or other MARL challenges. In this paper, we present JaxMARL, the first open-source code base that combines ease-of-use with GPU enabled efficiency, and supports a large number of commonly used MARL environments as well as popular baseline algorithms. Our experiments show that our JAX-based implementations are up to 1400x faster than existing single-threaded baselines. This enables efficient and thorough evaluations, with the potential to alleviate the *evaluation crisis* of the field. We also introduce and benchmark SMAX, a vectorised, simplified version of the StarCraft Multi-Agent Challenge, which removes the need to run the StarCraft II game engine. This not only enables GPU acceleration, but also provides a more flexible MARL environment, unlocking the potential for self-play, meta-learning, and other future applications in MARL.

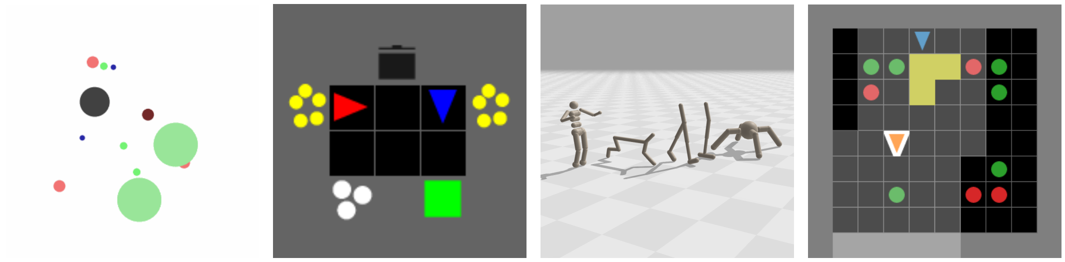
1 Introduction

Benchmarks play a pivotal role in the development of new single and multi-agent reinforcement learning (MARL) algorithms. They allow the community to define problems, facilitate comparison, and concentrate effort. In recent years, Go and Chess inspired the development of MuZero [42] while the StarCraft Multi-Agent Challenge [SMAC, 41] resulted in the development of QMIX [39], a popular MARL technique.

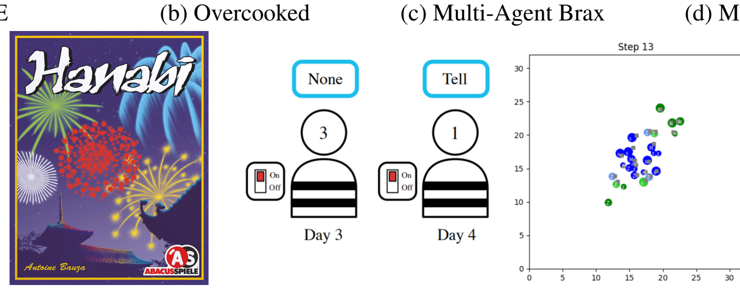
For reinforcement learning (RL) research, a large number of sequential environment interactions are typically required, often making simulation speed a significant bottleneck. This problem is even worse in MARL, where non-stationarity and decentralised partial observability greatly worsen the sample complexity. Hardware acceleration and parallelisation are crucial to alleviating this, but current acceleration and parallelisation methods are typically not implemented in Python, reducing their accessibility for most machine learning researchers. However, recent advances in JAX [7] have opened up new possibilities for using Python code directly with hardware accelerators, enabling the wider use of massively parallel RL training pipelines and environments.

The JAX library provides composable function transformations, allowing for automatic vectorisation, device parallelisation, automatic differentiation and just-in-time (JIT) compilation with XLA, for device-agnostic optimisation. Using JAX, both the environment and model training can be conducted on a hardware accelerator, removing the cost of any data transfer between devices and allowing

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.



(a) MPE (b) Overcooked (c) Multi-Agent Brax (d) Matrix Games



(e) Hanabi (f) Switch Riddle (g) SMAX

Figure 1: JaxMARL environments.

for significant parallelisation. Recently, PureJaxRL [28, 29] has demonstrated the power of this end-to-end JAX-based approach; running both the environment and the model training on a GPU yields a 4000x speedup over a traditional pipeline with a GPU-trained model but a CPU-based environment.

These speedups have significant potential impacts on RL and MARL research. Ideas can be tested and iterated on at a much greater rate, allowing the field to advance faster. Computational barriers for conducting Deep MARL research are lowered, allowing academic labs to utilise billions of frames in their research. Independent researchers are also able to extract significantly more performance from single GPUs.

Alongside the current computational issues faced by MARL researchers, recent research also highlights issues with the evaluation standards and use of benchmarks in the MARL community. In particular, there is a frequent lack of evaluation across a wide array of domains. Of the 75 recent MARL papers analysed by [18], 50% used only one evaluation environment and a further 30% used only two. While the two most used environments, SMAC and MPE, contain multiple different tasks or maps, there is no standard set, allowing authors to carefully select results. This leads to environment overfitting and unclear progress markers.

Instead, novel MARL methods should be tested on a wide range of domains to accurately evaluate their limits and enable better comparisons. The likely issue preventing this is the lack of a unified codebase and the computational burden of further evaluation.

This paper presents JaxMARL, a Python library that for the first time brings together JAX implementations of eight common MARL environments under one API. We additionally provide JAX implementations for four state-of-the-art algorithms, allowing for end-to-end JAX-based training pipelines in a similar fashion to PureJaxRL. This, for the first time, creates a library with end-to-end hardware-accelerated training, simple python implementations, and a broad range of MARL environments. By alleviating computational constraints, JaxMARL allows rapid evaluation of novel methods across a broad set of domains, and hence has the potential to be a powerful tool to address MARL's evaluation crisis.

We also create SMAX, a JAX-based simplification of the centralised training with decentralised execution (CTDE) benchmarks SMAC and SMACv2. SMAX features simplified dynamics, greater flexibility and a more sophisticated but fully-decentralised heuristic AI, while retaining the high-dimensional observation space, complex unit type interactions and procedural scenario generation that lend SMAC and SMACv2 much of their difficulty.

As shown in Figure 1, in addition to SMAX, our library includes the most popular environments from several MARL settings. For centralised training with decentralised execution (CTDE), we include the Multi-Agent Particle Environments (MPE) [27], and Multi-Agent Brax (MABrax). Meanwhile, for zero-shot coordination (ZSC), we include Hanabi and Overcooked. Lastly, from the general sum literature, we include the CoinGame and Spatial-Temporal Representations of Matrix Games (STORM), an extension of simple matrix games into grid-world scenarios. JaxMARL provides the first JAX implementation of these environments and is the first time they have existed within one codebase.

We additionally provide JAX implementations of Independent PPO (IPPO) [43], QMIX, VDN [46] and Independent Q-Learning (IQL) [34], four of the most common MARL algorithms, allowing new techniques to be easily benchmarked against existing practices.

2 Background

2.1 Hardware Accelerated Environments

Hardware acceleration allows for parallel execution, often granting significant speedups. Within the RL community, JAX has gained recent popularity as it enables the use of Python code with any hardware accelerator, increasing accessibility for researchers. Several libraries now provide JAX implementations of RL environments, alleviating the bottleneck of environment simulation steps. These libraries include: Gymnax [24], a library of popular single-agent RL environments; PGX [23], a collection of board games; and Brax [16], a differentiable physics engine. Only PGX provides MARL environments, and these are limited to board games.

2.2 SMAC

StarCraft has been a popular environment in which to test RL algorithms. Frequently this features a centralised controller issuing commands to balance *micromanagement*, the low-level control of individual units, and *macromanagement*, the high level plans for economy and resource management. Torchcraft[48] and TorchcraftAI[1] allow control of a player in StarCraft: Brood War, while the StarCraft II learning environment[54] provides a Python interface for communicating with StarCraft II. This latter environment was used to train AlphaStar[53], a centralised controller which attained grandmaster-level performance in StarCraft II and successfully beat professional human players.

SMAC[41], however, focuses on decentralised unit micromanagement across a range of scenarios divided into three broad categories: *symmetric*, where each side has the same units, *asymmetric*, where the enemy team has more units, and *micro-trick*, which are scenarios designed specifically to feature a particular StarCraft micromanagement strategy. SMACv2[13] demonstrates that open-loop policies can be effective on SMAC and adds additional randomly generated scenarios to rectify SMAC's lack of stochasticity. However, both of these environments rely on running the full game of StarCraft II, which severely limits their performance. SMAClite[32] attempts to alleviate this computational burden by recreating the SMAC environment primarily in NumPy, with some core components written in C++. While this is much more lightweight than SMAC, it cannot be run on a GPU and therefore cannot be parallelised that effectively with typical academic hardware.

3 JaxMARL

We present JaxMARL, a library containing JAX implementations of popular MARL environments and algorithms. Using JAX grants significant acceleration and parallelisation over existing implementations, and we utilise a simple interface to maintain accessibility. This represents the first library that provides python- and JAX-based implementations of a wide range of environments and baselines, therefore creating for the first time a library that is easy-to-use, enables evaluation on many MARL environments, and allows hardware-acceleration.

3.1 API

The interface of JaxMARL is inspired by PettingZoo [50] and Gymnax, ensuring it is simple and can represent a wide range of MARL problems. A simple example of instantiating an environment from

```python
import jax
from jaxmarl import make

key = jax.random.PRNGKey(0)
key, key_reset, key_act, key_step = jax.random.split(key, 4)

# Initialise and reset the environment.
env = make('MPE_simple_world_comm_v3')
obs, state = env.reset(key_reset)

# Sample random actions.
key_act = jax.random.split(key_act, env.num_agents)
actions = {agent: env.action_space(agent).sample(key_act[i]) \
            for i, agent in enumerate(env.agents)}

# Perform the step transition.
obs, state, reward, done, infos = env.step(key_step, state, actions)
```

Figure 2: An example of JaxMARL's API

JaxMARL's registry and executing one transition is presented in Figure 2. As JAX's JIT compilation requires pure functions, our `step` method has two additional inputs compared to PettingZoo's. The `state` object stores the environment's internal state and is updated with each call to `step`, before being passed to subsequent calls. Meanwhile, `key_step` is a pseudo-random key, consumed by JAX functions that require stochasticity. This key is separated from the internal state for clarity.

Similar to PettingZoo, the remaining inputs and outputs are dictionaries keyed by agent names, allowing for differing action and observation spaces. However, as JAX's JIT compilation requires array sizes to have static shapes, the total number of agents in an environment cannot vary during an episode. Thus, we do not use PettingZoo's *agent iterator*. Instead, the maximum number of agents is set upon environment instantiation and any agents that terminate before the end of an episode pass dummy actions thereafter. As asynchronous termination is possible, we signal the end of an episode using a special `"__all__"` key within `done`. The same dummy action approach is taken for environments where agents act asynchronously.

To ensure clarity and reproducibility, we keep strict registration of environments with a suffixed version number. Where implementations match existing ones the version numbers match.

3.2 Environments

JaxMARL contains a diverse range of environments, all implemented in JAX which can achieve speedups of up to 1400x over the CPU implementations of these environments. We also introduce SMAX, a SMAC-like environment implemented entirely in JAX. We introduce these environments and give details on their implementations in this section.

SMAX The StarCraft Multi-Agent Challenge (SMAC) is a popular benchmark, but has a number of shortcomings. First, as noted and addressed in prior work [13], it is not sufficiently stochastic to require complex closed-loop policies. Additionally, SMAC relies on StarCraft II as a simulator. This allows SMAC to use the wide range of units, objects and terrain available in StarCraft II. However, running an entire instance of StarCraft II is slow[32]. StarCraft II runs on the CPU and therefore SMAC's parallelisation is severely limited with typical academic compute.

Another important downside of StarCraft II is the constraints it places on environment features. For example, StarCraft II does not allow units of different races on the same team, limiting the variety of scenarios that can be generated. Secondly, SMAC does not support a competitive self-play setting without significant engineering work. The purpose of SMAX is to address these limitations. It provides access to a SMAC-like, hardware-accelerated, customisable environment that supports self-play and custom unit types and interactions.

Units in SMAX are modelled as circles in a two-dimensional continuous space. SMAX makes a number of additional simplifications to the dynamics of StarCraft II. Details about these are given in the Appendix.

SMAX also features a different, and more sophisticated, heuristic AI. The heuristic in SMAC simply attack-moves to a fixed location [32], and the heuristic in SMACv2 globally pursues the nearest agent. Thus the SMAC AI often does not aggressively pursue enemies that run away, and cannot generalise

Table 1: SMAX scenarios and their unit types

Scenario	Ally Units	Enemy Units	Start Positions
`2s3z`	2 stalkers and 3 zealots	2 stalkers and 3 zealots	Fixed
`3s5z`	3 stalkers and 5 zealots	3 stalkers and 5 zealots	Fixed
`5m_vs_6m`	5 marines	6 marines	Fixed
`10m_vs_11m`	10 marines	11 marines	Fixed
`27m_vs_30m`	27 marines	30 marines	Fixed
`3s5z_vs_3s6z`	3 stalkers and 5 zealots	3 stalkers and 6 zealots	Fixed
`3s_vs_5z`	3 stalkers	5 zealots	Fixed
`6h_vs_8z`	6 hydralisks	8 zealots	Fixed
`smacv2_5_units`	5 uniformly randomly chosen	5 uniformly randomly chosen	SMACv2-style
`smacv2_10_units`	10 uniformly randomly chosen	10 uniformly randomly chosen	SMACv2-style
`smacv2_20_units`	20 uniformly randomly chosen	20 uniformly randomly chosen	SMACv2-style

to the SMACv2 start positions, whereas the SMACv2 heuristic AI conditions on global information and is exploitable because of its tendency to flip-flop between two similarly close enemies. One of the constraints of SMAC is that the heuristic AI must be coded in the map editor, which does not provide a simple coding interface.

SMAX however features a decentralised heuristic AI that can effectively find enemies without the shortsighted targeting of the SMACv2 AI. This guarantees that a 50% win rate is always achievable by copying the heuristic policy exactly. This means any win-rate below 50% represents a concrete failure to learn.

SMAX scenarios incorporate both a number of the original scenarios from SMAC, and scenarios similar to those found in SMACv2. This allows researchers to choose to evaluate on the environments most suitable for their project. The SMACv2 scenarios sample units uniformly across all SMAX unit types (stalker, zealot, hydralisk, zergling, marine, marauder) and ensure fairness by having the enemy and ally teams be the same. The start positions are generated as in SMACv2, with the small difference that the 'surrounded' start positions position the allies and enemies on the outside or inside symmetrically. We provide more details on SMAX in Appendix A.1.

Overcooked Inspired by the popular videogame of the same name, Overcooked is commonly used for assessing fully cooperative and fully observable Human-AI task performance. The goal is to deliver soup as fast as possible, with each soup requiring 3 onions to be placed into a pot, time for the soup to cook, and delivery into bowls. Two agents, or *cooks*, must coordinate to effectively divide the tasks in order to maximise their common reward signal. Our implementation mimics the original from Overcooked-AI [9], including all five original layouts and a simple method for creating additional ones. For a discussion on the limitations of the Overcooked-AI environment, see [25].

Hanabi A fully cooperative partially observable multiplayer card game, where players are aware of other players' cards but not their own. To win, the team must play a series of cards in a specific order while sharing only a limited amount of information between players. As reasoning about the beliefs and intentions of other agents is central to performance, it is a common benchmark for ZSC and theory of mind research. Our implementation is inspired by the Hanabi Learning Environment [3] and includes custom configurations for varying game settings, such as the number of colours/ranks, number of players, and number of hint tokens. Compared to the Hanabi Learning Environment, which is written in C++, our implementation is a single-file and written in Python, making interfacing and starting experiments with it much easier.

Multi-Agent Particle Environments (MPE) The multi-agent particle environments feature a 2D world with simple physics where particle agents can move, communicate, and interact with fixed landmarks. Each specific environment varies the format of the world and the agents' abilities, creating a diverse set of tasks that include both competitive and cooperative settings. We implement all the MPE scenarios featured in the PettingZoo library and the transitions of our implementation map exactly to theirs. We additionally include a fully cooperative predator-prey variant of *simple tag*, presented in [37]. The code is structured to allow for straightforward extensions, enabling further tasks to be added.

Multi-Agent Brax (MABrax) A derivative from Multi-Agent MuJoCo [37], an extension of the MuJoCo Gym environment [51] that is commonly used for benchmarking Continuous Multi-Agent Robotic Control. Our implementation utilises Brax[16] as the underlying physics engine and includes five of Multi-Agent MuJoCo's multi-agent factorisation tasks, where each agent controls a subset of the joints and only observes the local state. The included tasks, illustrated in Figure 1, are: `ant_4x2`, `halfcheetah_6x1`, `hopper_3x1`, `humanoid_9|8`, and `walker2d_2x3`. The task descriptions mirror those from Gymnasium-Robotics [12].

Coin Game A two-player grid-world environment which emulates social dilemmas such as the iterated prisoner's dilemma. Used as a benchmark for the general sum setting, it expands on simpler social dilemmas by mandating learning from a high-dimensional state. Two players, 'red' and 'blue' move in a grid world and are each awarded 1 point for collecting any coin. However, 'red' loses 2 points if 'blue' collects a red coin and vice versa. Thus, if both agents ignore colour when collecting coins their expected reward is 0. Further details are provided in Appendix A.2.

Spatial-Temporal Representations of Matrix Games (STORM) Inspired by Melting Pot 2.0 [2], STORM [22] environment expands on simple matrix games by integrating them into grid-world scenarios. Agents collect resources which define their strategy during interactions and are rewarded based on the specific matrix game payoff matrix. This environment is useful because it allows to embed fully-cooperative, -competitive or general-sum games, such as the prisoner's dilemma [44], which makes it a suitable playground for studying paradigms such as opponent shaping, where agents act with the intent to change other agents' learning dynamics, which has been empirically shown to lead to more prosocial outcomes [15, 55, 30, 58]. Compared to the Coin Game or simple matrix games, the grid-world setting presents a variety of new challenges such as limited visibility, multi-step agent interactions, temporally-extended actions, and longer time horizons. Unlike Melting Pot, our environment features stochasticity, increasing the difficulty [13]. A further environment specification is provided in Appendix A.3.

Switch Riddle Originally used to illustrate the Differentiable Inter-Agent Learning algorithm [14], Switch Riddle is a simple cooperative communication environment that we include as a debugging tool. n prisoners held by a warden can secure their release by collectively ensuring that each has passed through a room with a light bulb and a switch. Each day, a prisoner is chosen at random to enter this room. They have three choices: do nothing, signal to the next prisoner by toggling the light, or inform the warden they think all prisoners have been in the room. The game ends when a prisoner informs the warden or the maximum time steps are reached. The rewards are +1 if the prisoner informs the warden, and all prisoners have been in the room, -1 if the prisoner informs the warden before all prisoners have taken their turn, and 0 otherwise, including when the maximum time steps are reached. We benchmark using the implementation from **(author?)** [57].

3.3 Algorithms

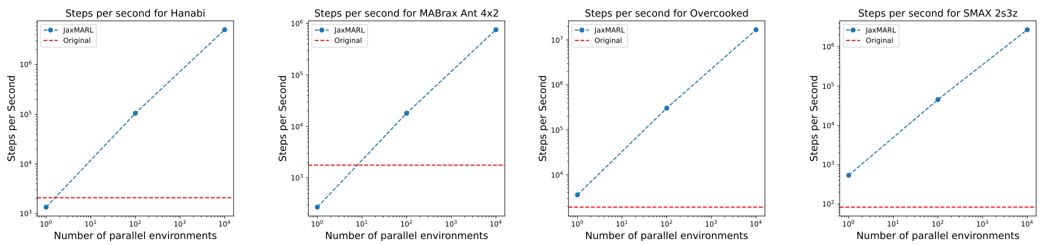
In this section, we present our re-implementation of several well known MARL baseline algorithms using JAX. The primary objective of these baselines is to provide a structured framework for developing MARL algorithms leveraging the advantages of the JaxMARL environments. All the training pipelines are fully compatible with JAX's JIT and VMAP functions, resulting in a significant acceleration of both training and metric evaluation processes. This enables parallelization of training across various seeds and hyperparameters on a single machine. We use the CleanRL philosophy of providing clear, single-file implementations [20].

IPPO Our Independent PPO (IPPO) [43] implementation is based on PureJaxRL [28], with parameter sharing across homogeneous agents. We provide both feed-forward and RNN versions.

Q-learning Methods Our Q-Learning baselines, including Independent Q-Learning (IQL) [49], Value Decomposition Networks (VDN) [47], and QMIX [40], have been implemented in accordance with the PyMARL codebase [40] to ensure consistency with published results and enable direct

Table 2: Benchmark results for JAX-based RL environments (steps-per-second)

Environment	Original, 1 Env	Jax, 1 Env	Jax, 100 Envs	Jax, 10k Envs
MPE Simple Spread	8.34×10^4	5.48×10^3	5.24×10^5	3.99×10^7
MPE Simple Reference	1.46×10^5	5.24×10^3	4.85×10^5	3.35×10^7
Switch Riddle	2.69×10^4	6.24×10^3	7.92×10^5	6.68×10^7
Hanabi	2.10×10^3	1.36×10^3	1.05×10^5	5.02×10^6
Overcooked	1.91×10^3	3.59×10^3	3.04×10^5	1.69×10^7
MABrax Ant 4x2	1.77×10^3	2.70×10^2	1.81×10^4	7.62×10^5
Starcraft 2s3z	8.31×10^1	5.37×10^2	4.53×10^4	2.71×10^6
Starcraft 27m vs 30m	2.73×10^1	1.45×10^2	1.12×10^4	1.90×10^5
Matrix Game in the Grid	–	2.48×10^3	1.75×10^5	1.46×10^7
Coin Game	1.97×10^4	4.67×10^3	4.06×10^5	4.03×10^7



(a) Hanabi (b) MABrax Ant (c) Overcooked (d) Starcraft 2s3z

Figure 3: Speed in four environments JaxMARL frameworks.

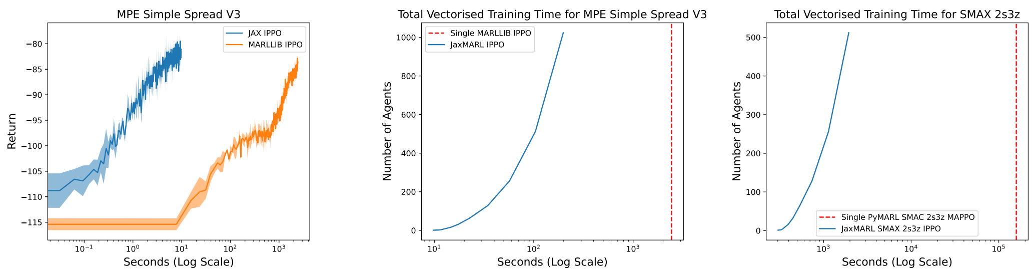
comparisons with PyTorch. Our baselines natively support aggregating trajectories from batched environments, simplifying parallelization. This approach is more convenient than managing environments on distinct threads and subsequently aggregating results, as done in PyMARL. We provide a brief overview of the implemented baselines in the Appendix.

4 Results

In our results, we aim to demonstrate four important properties of our library: the speed of our environments and algorithms compared with more traditional frameworks, and the correctness of our environment and algorithm implementations.

4.1 Environment Speed

We measure the performance in steps per second of the environments when using random actions compared to the original environments in Table 2. All results were collected on a single NVIDIA A100 GPU and AMD EPYC 7763 64-core processor. Environments were rolled out for 1000 sequential steps. Many environments have comparable performance to JaxMARL when comparing single environments, but the ease of parallelisation with Jax allows for much more efficient scaling compared to CPU-based environments. For example, MPE Simple Spread's JAX implementation is only 6.5% of the speed of the original when comparing a single environment, but even when only running 100 environments in parallel, the JAX environment is already over 6x faster. Figure 3 shows the performance against the number of environments for a selection of environments. When considering 10000 environments, the JAX versions are much faster, achieving speedups of over 8500x over the single-threaded environment in the case of Overcooked. Running this many environments in parallel using CPU environments would require a large CPU cluster and sophisticated communication mechanisms. This engineering is typically beyond the resources of academic labs, and therefore JaxMARL can unlock new research for such institutions.



(a) MPE Simple Spread Returns (b) MPE Training Speed (c) SMAX Training Speed

Figure 4: IPPO Speed and Performance in JaxMARL compared to MARLLIB and PyMARL in SMAX and MPE. Return results were averaged across 3 seeds. Performance results show 1 seed collected on the hardware described in Section 4.1.

4.2 Environment Correctness

To confirm the validity of our implementations, we verify that policies trained using our pipeline transfer to the original environment implementations. For MPE, we use an IQL policy, with training curves illustrated in Figure 5, and test correspondence on 1000 environment rollouts. For each rollout, we initialise the internal state of the JAX implementation using the output of PettingZoo's `reset` method, ensuring both episodes begin from identical points. We then execute the rollout with distinct calls to the policy from each implementation and compare the reward signals to confirm correspondence. Using this methodology, we validate correspondence for both Simple Speaker Listener V4 and Simple Spread V3.

4.3 Algorithm Correctness

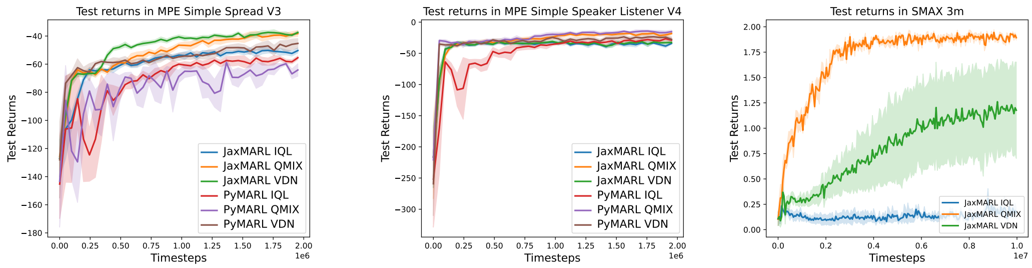
We verify the correctness of our algorithm implementations by comparing to baselines from other libraries on the MPE Simple Spread and Simple Speaker Listener environments. For IPPO we report the mean return across 3 seeds in Figure 4a. Results were collected on the same hardware as listed in Section 4.1. Our IPPO implementation attains the same performance as MARLLIB and runs 250x quicker, taking only ten seconds to train.

For the Q-learning algorithms, we verify the correctness by comparing with PyMARL implementations of the same algorithms on the MPE Simple Spread and Simple Speaker Listener environments. IQL, VDN and QMIX all attain the same or better results than their PyMARL counterparts. The returns are from greedy policies and averaged across 8 runs. The hyperparameters used were the same as for PyMARL. We also demonstrate the performance of the Q-learning algorithms on the SMAX 3m environment, where only QMIX is able to solve it reliably.

4.4 Algorithm Speed

We also demonstrate the improved speed of our IPPO implementation in Figure 4. By vectorising over agents, it is possible to train a vast number of agents in a fraction of the time it takes to train a single agent without hardware-acceleration. For MPE, it is possible to train 1024 agents in 198.4 seconds, which is less than 0.2 seconds per agent. A single run of MARLLIB's IPPO implementation on the same hardware takes around 2435.7 seconds on average. This represents an over 12500x speedup.

For SMAX, we compare the vectorised IPPO baseline to the MAPPO implementation provided in [45]. This implementation features an RNN, compared to the feed-forward baseline in JaxMARL. This was also run on a machine with a 64-core CPU and NVIDIA 2080Ti GPU. Additionally, as discussed in Section 3.2, SMAC and SMAX are different environments. These caveats aside, the differences in performance are so striking that we believe this clearly demonstrates the advantages of our approach. It's possible to train 512 SMAX agents on `2s3z` in under 33 minutes, whereas a single training run of PyTorch IPPO implementation takes 44 hours on average. This is 40000x speedup, with the significant caveats of the differences between the two runs.



(a) MPE Simple Spread (b) MPE Simple Speaker Listener (c) SMAX 3m

Figure 5: Performances of Q-Learning baselines in MPE cooperative scenarios, PyMARL and JaxMARL.

5 Related Work

Several open-source libraries exist for both MARL algorithms and environments. PyMARL [41] provides PyTorch implementations of QMIX, VDN and IQL and integrates easily with SMAC. E-PyMARL [36] extends this by adding the actor-critic algorithms MADDPG [27], MAA2C [33], IA2C [33], and MAPPO, and supports the SMAC, Gym [8], Robot Warehouse [10], Level-Based Foraging [10], and MPE environments. MARLLib [19], based on the open-source RL library RLLib [26], is a recent addition that combines a wide range of competitive, cooperative and mixed environments with a broad set of baseline algorithms. Meanwhile, MALib [59] focuses on population-based MARL across a wide range of environments. However, none of these frameworks feature hardware-accelerated environments and so do not give the associated performance benefits.

There has also been a recent proliferation of hardware-accelerated and JAX-based RL environments. Isaac gym [31] provides a GPU-accelerated simulator for a range of robotics platforms and CuLE [11] is a CUDA reimplementation of the Atari Learning Environment [4]. Both of these environments are GPU-specific and cannot be extended to other hardware accelerators. Jumanji [6] features implementations of mostly single-agent environments with a strong focus on combinatorial problems. These are written in JAX and the authors also provide an actor-critic baseline in addition to random actions. PGX [23] includes several board-game environments written in JAX. Gymnax [24] provides JAX implementations of the BSuite [35], classic continuous control, MinAtar [56] and other assorted environments and comes with a sister-library, gymnax-baselines, which provides PPO and ES baselines. Brax [16] reimplements the MuJoCo simulator in JAX and also provides a PPO implementation as a baseline. VMAS [5] provides a vectorized 2D physics engine written in PyTorch and a set of challenging multi-robot scenarios. Jax-LOB [17] implements a vectorized limit order book as an RL environment that runs on the accelerator. Perhaps the most similar to our work is Mava[38], which provides a MAPPO baseline, as well as integration with the Robot Warehouse environment. However, none of these libraries provides access to a wide range of JAX-based MARL environments as well as both value-based and actor-critic baselines.

Broadly, no other work provides implementations of a similar range of hardware-accelerated cooperative, competitive and mixed environments, while also implementing value-based and actor-critic baselines. Secondly, no other JAX simplification of SMAC exists. All other versions are either tied to the StarCraft II simulator or not hardware accelerated.

6 Conclusion

Hardware acceleration offers important opportunities for MARL research by lowering computational barriers and increasing the speed at which ideas can be iterated. We present JaxMARL, an open-source library of popular MARL environments and baseline algorithms implemented in JAX. We combine ease of use with hardware accelerator enabled efficiency to give significant speed-ups compared to traditional CPU-based implementations. Furthermore, by bringing together a wide range of MARL environments under one codebase, we have the potential to help alleviate issues with MARL's evaluation standards. We hope that JaxMARL will help advance MARL by improving the ability of academic labs to conduct research with thorough and effective evaluations.

References

[1] Torchcraftai: A bot platform for machine learning research on starcraft: Brood war. `https://github.com/TorchCraft/TorchCraftAI`, 2018. GitHub repository.

[2] John P Agapiou, Alexander Sasha Vezhnevets, Edgar A Duéñez-Guzmán, Jayd Matyas, Yiran Mao, Peter Sunehag, Raphael Köster, Udari Madhushani, Kavya Kopparapu, Ramona Comanescu, et al. Melting pot 2.0. *arXiv preprint arXiv:2211.13746*, 2022.

[3] Nolan Bard, Jakob N Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, et al. The hanabi challenge: A new frontier for ai research. *Artificial Intelligence*, 280:103216, 2020.

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

[5] Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. Vmas: A vectorized multi-agent simulator for collective robot learning. *The 16th International Symposium on Distributed Autonomous Robotic Systems*, 2022.

[6] Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Vincent Coyette, Paul Duckworth, Laurence I. Midgley, Tristan Kalloniatis, Sasha Abramowitz, Cemlyn N. Waters, Andries P. Smit, Nathan Grinsztajn, Ulrich A. Mbou Sob, Omayma Mahjoub, Elshadai Tegegn, Mohamed A. Mimouni, Raphael Boige, Ruan de Kock, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. Jumanji: a diverse suite of scalable reinforcement learning environments in jax, 2023.

[7] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[9] Micah Carroll, Rohin Shah, Mark K Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. On the utility of learning about humans for human-ai coordination. *Advances in neural information processing systems*, 32, 2019.

[10] Filippos Christianos, Lukas Schäfer, and Stefano V Albrecht. Shared experience actor-critic for multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[11] Steven Dalton and iuri frosio. Accelerating reinforcement learning through gpu atari emulation. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 19773–19782. Curran Associates, Inc., 2020.

[12] Rodrigo de Lazcano, Kallinteris Andreas, Jun Jet Tai, Seungjae Ryan Lee, and Jordan Terry. Gymnasium robotics, 2023.

[13] Benjamin Ellis, Skander Moalla, Mikayel Samvelyan, Mingfei Sun, Anuj Mahajan, Jakob N Foerster, and Shimon Whiteson. Smacv2: An improved benchmark for cooperative multi-agent reinforcement learning. *arXiv preprint arXiv:2212.07489*, 2022.

[14] 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.

[15] Jakob Foerster, Richard Y Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 122–130, 2018.

[16] 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.

[17] Sascha Frey, Kang Li, Peer Nagy, Silvia Sapora, Chris Lu, Stefan Zohren, Jakob Foerster, and Anisoara Calinescu. Jax-lob: A gpu-accelerated limit order book simulator to unlock large scale reinforcement learning for trading. *arXiv preprint arXiv:2308.13289*, 2023.

[18] Rihab Gorsane, Omayma Mahjoub, Ruan de Kock, Roland Dubb, Siddarth Singh, and Arnu Pretorius. Towards a standardised performance evaluation protocol for cooperative marl. *arXiv preprint arXiv:2209.10485*, 2022.

[19] Siyi Hu, Yifan Zhong, Minquan Gao, Weixun Wang, Hao Dong, Zhihui Li, Xiaodan Liang, Xiaojun Chang, and Yaodong Yang. Marllib: Extending rllib for multi-agent reinforcement learning. *arXiv preprint arXiv:2210.13708*, 2022.

[20] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[21] Roberto Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[22] Akbir Khan, Newton Kwan, Timon Willi, Chris Lu, Andrea Tacchetti, and Jakob Nicolaus Foerster. Context and history aware other-shaping. 2022.

[23] Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-accelerated parallel game simulators for reinforcement learning. *arXiv preprint arXiv:2303.17503*, 2023.

[24] Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022.

[25] Niklas Lauffer, Ameesh Shah, Micah Carroll, Michael D Dennis, and Stuart Russell. Who needs to know? minimal knowledge for optimal coordination. In *International Conference on Machine Learning*, pages 18599–18613. PMLR, 2023.

[26] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International conference on machine learning*, pages 3053–3062. PMLR, 2018.

[27] 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.

[28] Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered policy optimisation. *Advances in Neural Information Processing Systems*, 35:16455–16468, 2022.

[29] Chris Lu, Timon Willi, Alistair Letcher, and Jakob Nicolaus Foerster. Adversarial cheap talk. In *International Conference on Machine Learning*, pages 22917–22941. PMLR, 2023.

[30] Christopher Lu, Timon Willi, Christian A Schroeder De Witt, and Jakob Foerster. Model-free opponent shaping. In *International Conference on Machine Learning*, pages 14398–14411. PMLR, 2022.

[31] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance gpu-based physics simulation for robot learning, 2021.

[32] Adam Michalski, Filippos Christianos, and Stefano V Albrecht. Smaclite: A lightweight environment for multi-agent reinforcement learning. *arXiv preprint arXiv:2305.05566*, 2023.

[33] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[34] 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.

[35] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvari, Satinder Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

[36] Georgios Papoudakis, Filippos Christianos, Lukas Schäfer, and Stefano V. Albrecht. Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS)*, 2021.

[37] Bei Peng, Tabish Rashid, Christian Schroeder de Witt, Pierre-Alexandre Kamienny, Philip Torr, Wendelin Böhmer, and Shimon Whiteson. Facmac: Factored multi-agent centralised policy gradients. *Advances in Neural Information Processing Systems*, 34:12208–12221, 2021.

[38] Arnu Pretorius, Kale ab Tessera, Andries P. Smit, Kevin Eloff, Claude Formanek, St John Grimbly, Siphelele Danisa, Lawrence Francis, Jonathan Shock, Herman Kamper, Willie Brink, Herman Engelbrecht, Alexandre Laterre, and Karim Beguir. Mava: A research framework for distributed multi-agent reinforcement learning. *arXiv preprint arXiv:2107.01460*, 2021.

[39] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *The Journal of Machine Learning Research*, 21(1):7234–7284, 2020.

[40] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 4295–4304. PMLR, 2018.

[41] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The starcraft multi-agent challenge. *arXiv preprint arXiv:1902.04043*, 2019.

[42] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

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

[44] Glenn H Snyder. " prisoner's dilemma" and" chicken" models in international politics. *International Studies Quarterly*, 15(1):66–103, 1971.

[45] Mingfei Sun, Sam Devlin, Jacob Beck, Katja Hofmann, and Shimon Whiteson. Trust region bounds for decentralized ppo under non-stationarity. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 5–13, 2023.

[46] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*, 2017.

[47] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, (AAMAS 2018)*, volume 3, pages 2085–2087, 2018.

[48] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. Torchcraft: a library for machine learning research on real-time strategy games, 2016.

[49] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395, 2017.

[50] 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.

[51] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[52] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[53] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019.

[54] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft ii: A new challenge for reinforcement learning, 2017.

[55] Timon Willi, Alistair Letcher, Johannes Treutlein, and Jakob N. Foerster. COLA: consistent learning with opponent-learning awareness. In *International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 23804–23831, 2022.

[56] Kenny Young and Tian Tian. Minatar: An atari-inspired testbed for thorough and reproducible reinforcement learning experiments. *arXiv preprint arXiv:1903.03176*, 2019.

[57] Qizhen Zhang, Chris Lu, Animesh Garg, and Jakob Foerster. Centralized model and exploration policy for multi-agent rl. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, pages 1500–1508, 2022.

[58] Stephen Zhao, Chris Lu, Roger B Grosse, and Jakob Foerster. Proximal learning with opponent-learning awareness. *Advances in Neural Information Processing Systems*, 35:26324–26336, 2022.

[59] Ming Zhou, Ziyu Wan, Hanjing Wang, Muning Wen, Runzhe Wu, Ying Wen, Yaodong Yang, Yong Yu, Jun Wang, and Weinan Zhang. Malib: A parallel framework for population-based multi-agent reinforcement learning. *Journal of Machine Learning Research*, 24(150):1–12, 2023.

A Further Details on Environments

A.1 SMAX

Observations in SMAX are structured similarly to SMAC. Each agent observes the health, previous action, position, weapon cooldown and unit type of all allies and enemies in its sight range. Like SMACv2[13], we use the sight and attack ranges as prescribed by StarCraft II rather than the fixed values used in SMAC.

SMAX's reward function is similar to the dense reward from SMAC, but rescaled. Agents get 1 point for depleting all the enemies' health and 1 point for winning. When trained in self-play, the reward is simply 1 for winning the episode and -1 for losing. Unlike StarCraft II, where all actions happen in a randomised order in the game loop, some actions in SMAX are simultaneous, meaning draws are possible. In this case both teams get 0 reward.

Like SMAC, each environment step in SMAX consists of eight individual time ticks. SMAX uses a discrete action space, consisting of movement in the four cardinal directions, a stop action, and a shoot action per enemy.

SMAX makes three notable simplifications of the StarCraft II dynamics to reduce complexity. First, zerg units do not regenerate health. This health regeneration is slow at 0.38 health per second, and so likely has little impact on the game. Protoss units also do not have shields. Shields only recharge after 10 seconds out of combat, and therefore are unlikely to recharge during a single micromanagement task. Protoss units have additional health to compensate for their lost shields. Finally, the available unit types are reduced compared to SMAC. SMAX has no medivac, colossus or baneling units. Each of these unit types has special mechanics that were left out for the sake of simplicity.

Collisions are handled by moving agents to their desired location first and then pushing them out from one another.

A.2 Coin Game

Two agents, 'red' and 'blue', move in a wrap-around grid and collect red and blue coloured coins. When an agent collects any coin, the agent receives a reward of 1. However, when 'red' collects a blue coin, 'blue' receives a reward of -2 and vice versa. Once a coin is collected, a new coin of the same colour appears at a random location within the grid. If a coin is collected by both agents simultaneously, the coin is duplicated and both agents collect it. Episodes are of a set length.

A.3 Spatial-Temporal Representations of Matrix Games (STORM)

This environment features directional agents within an 8x8 grid-world with a restricted field of view. Agents cannot move backwards or share the same location. Collisions are resolved by either giving priority to the stationary agent or randomly if both are moving. Agents collect two unique resources: *cooperate* and *defect* coins. Once an agent picks up any coin, the agent's colour shifts, indicating its readiness to interact. The agents can then release an *interact* beam directly ahead; when this beam intersects with another ready agent, both are rewarded based on the specific matrix game payoff matrix. The agents' coin collections determine their strategies. For instance, if an agent has 1 *cooperate* coin and 3 *defect* coins, there's a 25% likelihood of the agent choosing to cooperate. After an interaction, the two agents involved are frozen for five steps, revealing their coin collections to surrounding agents. After five steps, they respawn in a new location, with their coin count set back to zero. Once an episode concludes, the coin placements are shuffled. This grid-based approach to matrix games can be adapted for n-player versions. While STORM is inspired by MeltingPot 2.0, there are noteworthy differences:

- Meltingpot uses pixel-based observations while we allow for direct grid access.
- Meltingpot's grid size is typically 23x15, while ours is 8x8.
- Meltingpot features walls within its layout, ours does not.
- Our environment introduces stochasticity by shuffling the coin placements, which remain static in Meltingpot.
- Our agents begin with an empty coin inventory, making it easier for them to adopt pure cooperate or defect tactics, unlike in Meltingpot where they start with one of each coin.
- MeltingPot is implemented in Lua[21] where as ours is a vectorized implementation in Jax.

We deem the coin shuffling especially crucial because even large environments representing POMDPs, such as SMAC, can be solved without the need for memory if they lack sufficient randomness [13].

B Value-Based MARL Methods and Implementation details

Key features of our framework include parameter sharing, a recurrent neural network (RNN) for agents, an epsilon-greedy exploration strategy with linear decay, a uniform experience replay buffer, and the incorporation of Double Deep Q-Learning (DDQN) [52] techniques to enhance training stability.

Unlike PyMARL, we use the Adam optimizer as the default optimization algorithm. Below is an introduction to common value-based MARL methods.

IQL (Independent Q-Learners) is a straightforward adaptation of Deep Q-Learning to multi-agent scenarios. It features multiple Q-Learner agents that operate independently, optimizing their individual returns. This approach follows a decentralized learning and decentralized execution pipeline.

VDN (Value Decomposition Networks) extends Q-Learning to multi-agent scenarios with a centralized-learning-decentralized-execution framework. Individual agents approximate their own action's Q-Value, which is then summed during training to compute a jointed Q_{tot} for the global state-action pair. Back-propagation of the global DDQN loss in respect to a global team reward optimizes the factorization of the jointed Q-Value.

QMIX improves upon VDN by relaxing the full factorization requirement. It ensures that a global $argmax$ operation on the total Q-Value (Q_{tot}) is equivalent to individual $argmax$ operations on each agent's Q-Value. This is achieved using a feed-forward neural network as the mixing network, which combines agent network outputs to produce Q_{tot} values. The global DDQN loss is computed using a single shared reward function and is back-propagated through the mixer network to the agents' parameters. Hypernetworks generate the mixing network's weights and biases, ensuring non-negativity using an absolute activation function. These hypernetworks are two-layered multi-layer perceptrons with ReLU non-linearity.

C Hyperparameters

Hyperparameter	**Value**
LR	0.0005
NUM_ENVS	25
NUM_STEPS	128
TOTAL_TIMESTEPS	1×10^6
UPDATE_EPOCHS	5
NUM_MINIBATCHES	2
GAMMA	0.99
GAE_LAMBDA	1.0
CLIP_EPS	0.3
ENT_COEF	0.01
VF_COEF	1.0
MAX_GRAD_NORM	0.5
ACTIVATION	tanh
ANNEAL_LR	True

Table 3: Hyperparameters for IPPO MPE

Hyperparameter	**Value**
LR	0.004
NUM_ENVS	64
NUM_STEPS	128
TOTAL_TIMESTEPS	$10,000,000.0$
UPDATE_EPOCHS	2
NUM_MINIBATCHES	2
GAMMA	0.99
GAE_LAMBDA	0.95
CLIP_EPS	0.2
SCALE_CLIP_EPS	False
ENT_COEF	0.0
VF_COEF	0.5
MAX_GRAD_NORM	0.5
ACTIVATION	relu

Table 4: Hyperparameters for SMAX IPPO