JaxMARL: Multi-Agent RL Environments and Algorithms in JAX

Alexander Rutherford[1*†] **Benjamin Ellis**[1*†] **Matteo Gallici**[2*†] **Jonathan Cook**[1†]
Andrei Lupu[1†] **Garðar Ingvarsson**[3†] **Timon Willi**[1†] **Ravi Hammond**[1†]
Akbir Khan[3] **Christian Schroeder de Witt**[1] **Alexandra Souly**[3]
Saptarashmi Bandyopadhyay[4] **Mikayel Samvelyan**[3] **Minqi Jiang**[3] **Robert Lange**[5]
Shimon Whiteson[1] **Bruno Lacerda**[1] **Nick Hawes**[1] **Tim Rocktäschel**[3]
Chris Lu[1*†] **Jakob Foerster**[1]
[1]University of Oxford, [2]Universitat Politècnica de Catalunya, [3]University College London,
[4]University of Maryland, [5]Technical University Berlin

1 Further Background on SMAC

StarCraft is a popular environment for testing RL algorithms. It typically features 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.

SMAC [12], instead, 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 [5] 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 increases their CPU and memory requirements. SMAClite [10] 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 effectively with typical academic hardware, which commonly has very few CPU cores compared to industry clusters.

2 Further Details on Environments

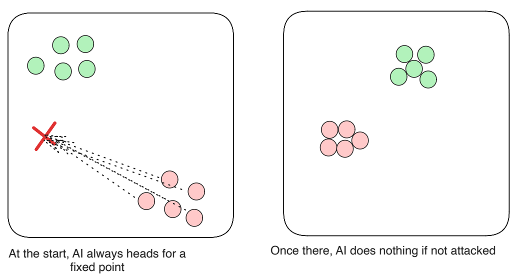
2.1 SMAX

The StarCraft Multi-Agent Challenge (SMAC) is a popular benchmark but has a number of shortcomings. First, as noted and addressed in SMACv2, SMAC is not particularly stochastic. This means that non-trivial win-rates are possible on many SMAC maps by conditioning a policy only on the timestep and agent ID. Additionally, SMAC relies on StarCraft II as a simulator. While this allows SMAC to use the wide range of units, objects and terrain available in StarCraft II, running an entire instance of StarCraft II is slow and memory intensive. StarCraft II runs on the CPU and therefore SMAC's parallelisation is severely limited with typical academic compute.

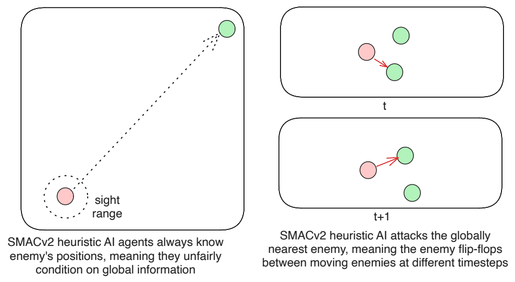
Using the StarCraft II game engine also constrains environment design. For example, StarCraft II groups units into three races and does not allow units of different races on the same team, limiting the

*Equal Contribution
†Core Contributor

Submitted to the 38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks. Do not distribute.


At the start, AI always heads for a fixed point
Once there, AI does nothing if not attacked

(a) SMAC heuristic AI operation


sight range
SMACv2 heuristic AI agents always know enemy's positions, meaning they unfairly condition on global information
t
t+1
SMACv2 heuristic AI attacks the globally nearest enemy, meaning the enemy flip-flops between moving enemies at different timesteps

(b) SMACv2 heuristic AI operation

Figure 1: As shown in Figure 1a, SMAC heuristic AI is decentralised, but does not generalise to new start positions. SMACv2 heuristic AI solves the problem of not being able to locate enemies on the map, but does so via conditioning on the global state, which means that some scenarios might be unwinnable. Additionally, the SMACv2 heuristic AI targets the closest enemy, which can lead to flip-flopping between targets. This is shown in Figure 1b

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 simplified SMAC-like, hardware-accelerated, customisable environment that supports self-play and custom unit types. SMAX models units as discs in a continuous 2D space. As listed in Table 1, we include all SMAC(v1) scenarios alongside three inspired by SMAC(v2).

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[5], we use the sight and attack ranges as prescribed by StarCraft II rather than the fixed values used in SMAC.

SMAX and SMAC have different returns. SMAC's reward function, like SMAX's, is split into two parts: one part for depleting enemy health, and another for winning the episode. However, in SMAC, the part which rewards depleting enemy health scales with the number of agents. This is most clearly demonstrated in `27m_vs_30m`, where a random policy gets a return of around 10 out of a maximum of 20 because almost all the reward is for depleting enemy health or killing agents, rather than winning the episode. In SMAX, however, 50% of the total return is always for depleting enemy health, and 50% for winning.

SMAX also features a different, and more sophisticated, heuristic AI. The heuristic in SMAC simply moves to a fixed location, attacking any enemies it encounters along the way, 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 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. SMAC's heuristic AI must be coded in the map editor, which does not provide a simple coding interface. Figure 1 demonstrates these limitations.

In contrast, SMAX features a decentralised heuristic AI that can effectively find enemies without requiring the global information of the SMACv2 heuristic. This guarantees that in principle a 50% win rate is always achievable by copying the decentralised heuristic policy exactly. This means any win-rate below 50% represents a concrete failure to learn. Some of the capabilities of the SMAX heuristic AI are illustrated in the Figure below.

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.

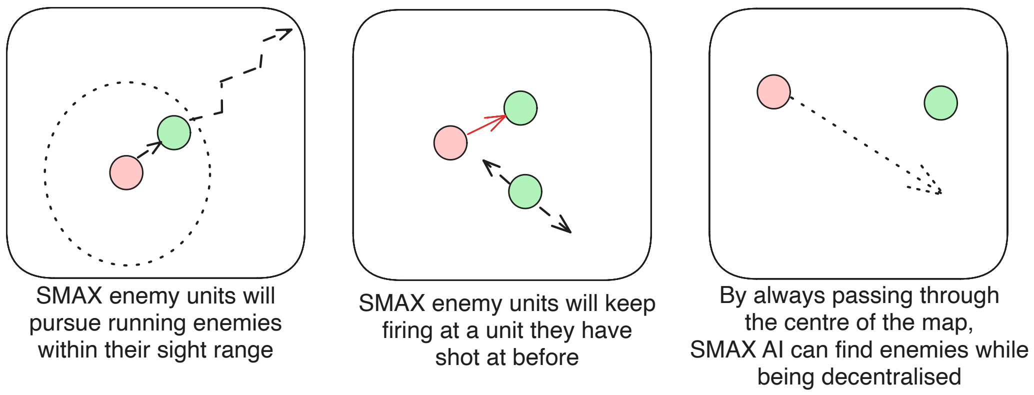

SMAX enemy units will pursue running enemies within their sight range
SMAX enemy units will keep firing at a unit they have shot at before
By always passing through the centre of the map, SMAX AI can find enemies while being decentralised

Figure 2: Explanation of the operation of the SMAX heuristic AI.

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. For the SMACv2 scenarios, the start positions are generated as in SMACv2, with the small difference that the 'surrounded' start positions now treat allies and enemies identically, rather than always spawning allies in the middle of the map. This symmetry guarantees that a 50% win rate is always achievable.

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

Table 1: SMAX scenarios. The first section corresponds to SMAC scenarios, while the second corresponds to SMACv2.

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

2.2 Spatial-Temporal Representations of Matrix Games (STORM)

This environment features directional agents within an 8x8 grid world with a restricted field of view. For a visual description, see Figure 3. 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 is 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 [8] 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 [5].

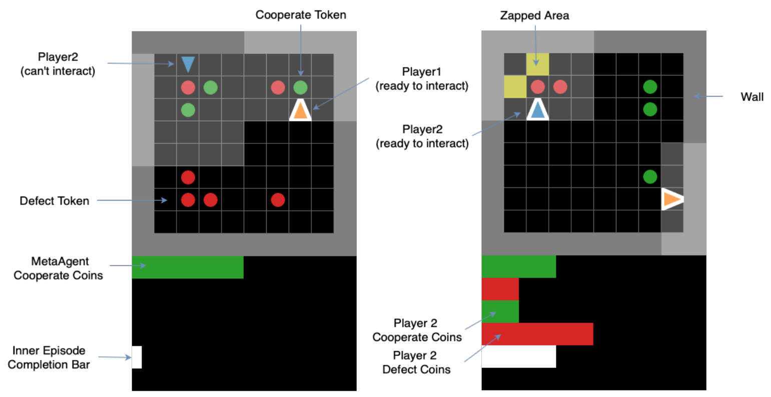


Figure 3: Annotated Image of IPDiTM renders, demonstrating the objects within the game

2.3 Coin Game

Coin Game is a two-player grid-world environment which emulates social dilemmas such as the iterated prisoner's dilemma [13]. Used as a benchmark for the general-sum setting, it expands on simpler social dilemmas by adding 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.

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.

2.4 Switch Riddle

Originally used to illustrate the Differentiable Inter-Agent Learning algorithm [6], 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 [18].

2.5 Hanabi

Hanabi is a fully cooperative partially observable multiplayer card game, where players can observe 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 ad-hoc teamplay research. Our implementation is inspired by the Hanabi Learning Environment [2] 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++ and split over dozens of files, our implementation is a single easy-to-read Python file, which simplifies interfacing with the library and running experiments.

2.6 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 aim is to quickly prepare and deliver soup, which involves putting three onions in a pot, cooking the soup, and serving it into bowls. Two agents, or *cooks*, must coordinate to effectively divide the tasks to maximise their common reward signal. Our implementation mimics the original from Overcooked-AI [3], 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 [9].

2.7 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 [11]. The code is structured to allow for straightforward extensions, enabling further tasks to be added.

2.8 Multi-Agent Brax (MABrax)

MABrax is a derivative of Multi-Agent MuJoCo [11], an extension of the MuJoCo Gym environment [15] that is commonly used for benchmarking continuous multi-agent robotic control. Our implementation utilises Brax[7] 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 **??**, are: `ant_4x2`, `halfcheetah_6x1`, `hopper_3x1`, `humanoid_9|8`, and `walker2d_2x3`. The task descriptions mirror those from Gymnasium-Robotics [4].

3 JaxMARL's API

The interface of JaxMARL is inspired by PettingZoo [14] and Gymnax. We designed it to be a simple and easy-to-use interface for a wide-range of MARL problems. An example of instantiating

```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 4: An example of JaxMARL's API, which is flexible and easy-to-use.

an environment from JaxMARL's registry and executing one transition is presented in Figure 4. 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 arrays 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 (e.g. turn-based games).

To ensure clarity and reproducibility, we keep strict registration of environments with suffixed version numbers, for example "MPE Simple Spread V3". Whenever JaxMARL environments correspond to existing CPU-based implementations, the version numbers match.

4 Value-Based MARL Methods and Implementation details

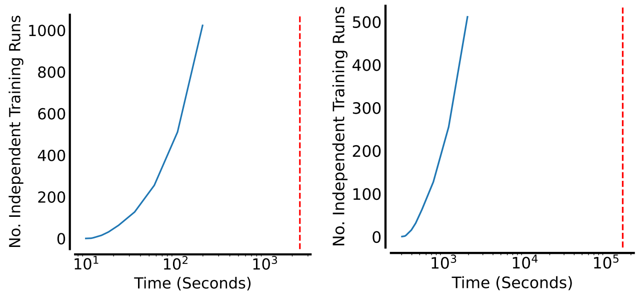
Key features of our framework include parameter sharing, a recurrent neural network (RNN) for agents, an epsilon-greedy exploration strategy with linear decay, a uniform experience replay buffer, and the incorporation of Double Deep Q-Learning (DDQN) [17] techniques to enhance training stability. We stored the replay buffer in GPU memory using Flashbax [16].

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,



(a) MPE Simple Spread (b) SMAX `2s3z`

Figure 5: Time taken to train a varying number of seeds in parallel on the same device for JaxMARL IPPO (in blue) compared to the time taken to train one seed with MARLLIB (shown as the red dashed line)

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.

Issues found when using Q-Learning in an end-to-end GPU setting. As discussed in the paper's results section, PPO demonstrates a clear advantage over Q-Learning for our benchmarked environments, both in agent performance and training runtime. The speed differential is caused by the optimal sampling/replay ratio for Q-Learning methods becoming rapidly unbalanced as the number of parallel environments increases, which requires us to use fewer parallel environments than we use with PPO. PPO also has a major advantage over Q-Learning in that it does not use a replay buffer, which can occupy a significant amount of GPU memory. Secondly, our experiments empirically showed PPO to be more stable during training.

A possible workaround is to increase the replay ratio by performing multiple update steps per training episode, which nevertheless affects computational efficiency. A better solution is to implement a distributed framework, separating the learning and sampling process, which is also out-of-scope for this work.

5 Speed Comparison

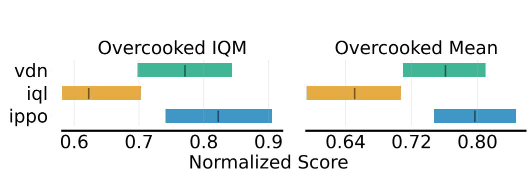
The runs reported in Figures 3 and 5(c) were all run on the same system featuring two NVIDIA GeForce RTX 4090s (although only one was used for training), an Intel(R) Xeon(R) Silver 4316 CPU (20 cores with 40 threads), and 132 GB of RAM. We report the average environment steps per second over the entire RL training process, which for JaxMARL includes any compilation time. For Table 3, 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.

In Figure 5 we repeat the analysis, reported in the main paper for QMIX, of JaxMARL's ability to train multiple seeds in parallel for IPPO. Training this way allows training agents many thousands of times faster, with a 12500x speed up in the MPE simple spread environment.

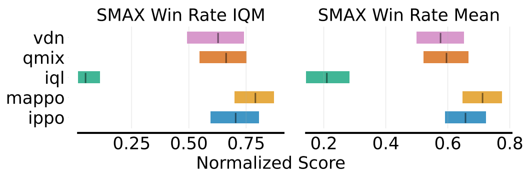
6 Training & Correctness Results

6.1 Overcooked

We train IPPO, VDN and IQL agents in Overcooked and present their aggregate performance in Figure 6a. IPPO performs better than the Q-Learning methods in inter-quartile mean and mean, in line with our more general findings. During training, we use the same shaped reward as stated



(a) Overcooked aggregate performance



(b) SMAX aggregate performance

Figure 6: Aggregate performance in Overcooked and SMAX for a range of algorithms. Performance is aggregated across 10 seeds and error bars represent 95% bootstrapped confidence intervals as recommended in [1].

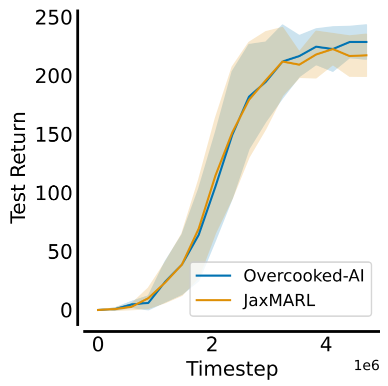


Figure 7: Evaluation performance throughout training of an IPPO policy trained with JaxMARL on our Overcooked Cramped Room scenario implementation and the original [3]. The similarity in performance demonstrates correspondence.

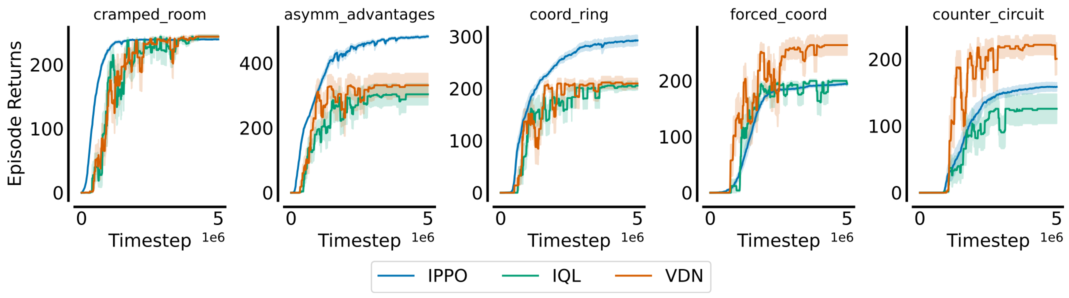


Figure 8: Evaluation of all algorithms in Overcooked scenarios. These scores are obtained after our own hypeparameter tuning, which held to better performances than the using original hyperparameters from Overcooked paper.

in the original Overooked paper [3], which is added to the score of the game with a factor that is decayed from 1 to 0 during the first half of training. We don't train MAPPO and QMIX for this task because, in Overcooked, agents can observe the entire state of the map. Therefore, there is no partial observability that can be improved through centralized training. We demonstrate correspondence by training an IPPO policy with JaxMARL on our implementation and evaluating the policy over 10 rollouts for both our Overcooked implementation and the original. Results are shown in Figure 8 with the similarity in performance demonstrating their equivalence.

6.2 MABrax

The performance of IPPO on `ant_4x2`, `humanoid_9|8`, `hopper_3x1` and `walker2d_2x3` is reported in Figure 9, with hyperparameters reported in Table 2.

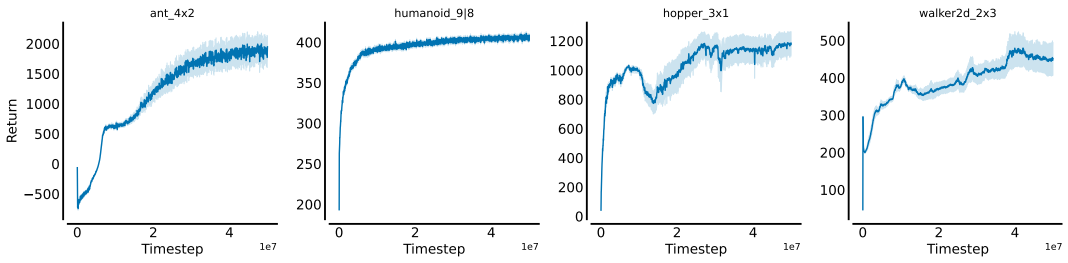


Figure 9: Performance of IPPO on MABrax Tasks

6.3 MPE

Performance of Q-Learning baselines in all the MPE scenarios are reported in **??**. The upper row represents cooperative scenarios, with results for all our Q-learning baselines reported. The bottom row refers to competitive scenarios, and results for IQL are divided by agent types. Hyperparameters are given inTable 7

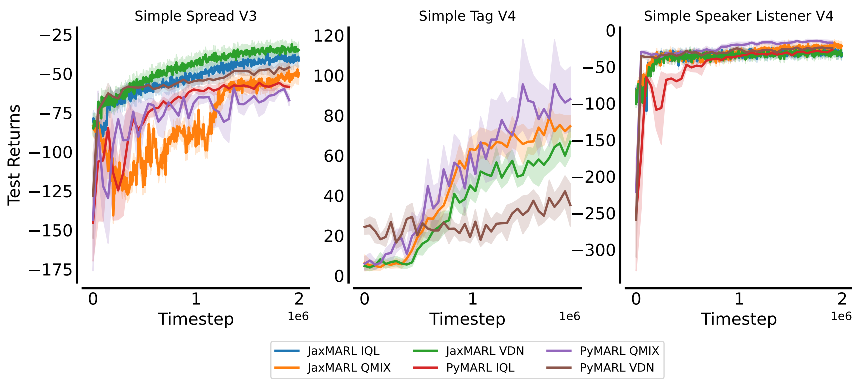


Figure 10: Comparison of the performances of Q-Learning baselines in PyMARL and JaxMARL in two cooperative scenarios of MPE (Spread and Speaker Listener) and one competitive scenario (Simple Tag). For Simple Tag, we pre-trained a prey in JaxMARL and then trained agents to compete with it in both PyMARL and JaxMARL. Despite the small differences in the obtained returns in the two frameworks, the algorithms show similar learning dynamics, and the final ordering is preserved, validating our environment and algorithm implementations.

6.4 SMAX

The performance of different algorithms in SMAX versus MAPPO in SMAC is shown in Figure 13. Hyperparameters for IPPO and the Q-learning methods are given in Table 4 and Table 8 respectively.

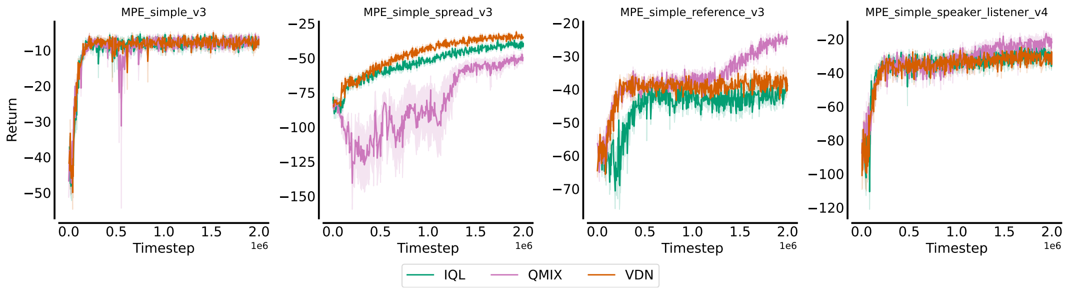


Figure 11: Evaluation of performances of QLearning in all the MPE cooperative scenarios.

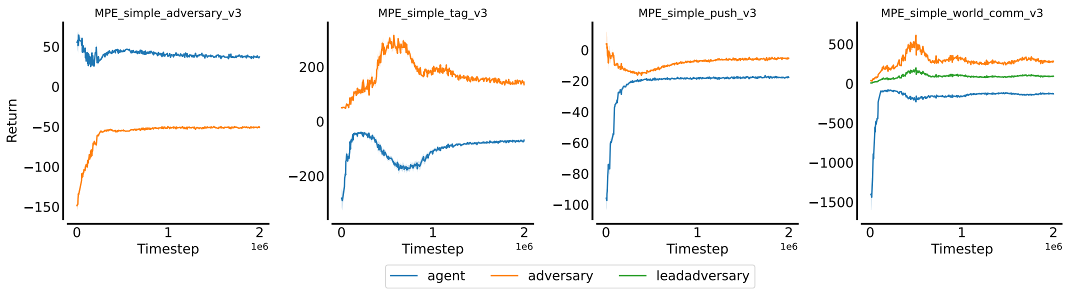


Figure 12: Evaluation of performances of IQL in all the MPE competetive scenarios. All the competetive agents are trained independently together. "Agent" and "Adversary" are teams, not single agents.

Some maps are significantly more difficult in SMAX, such as `10m_vs_11m`, whereas some are much easier such as `3s_vs_5z`.

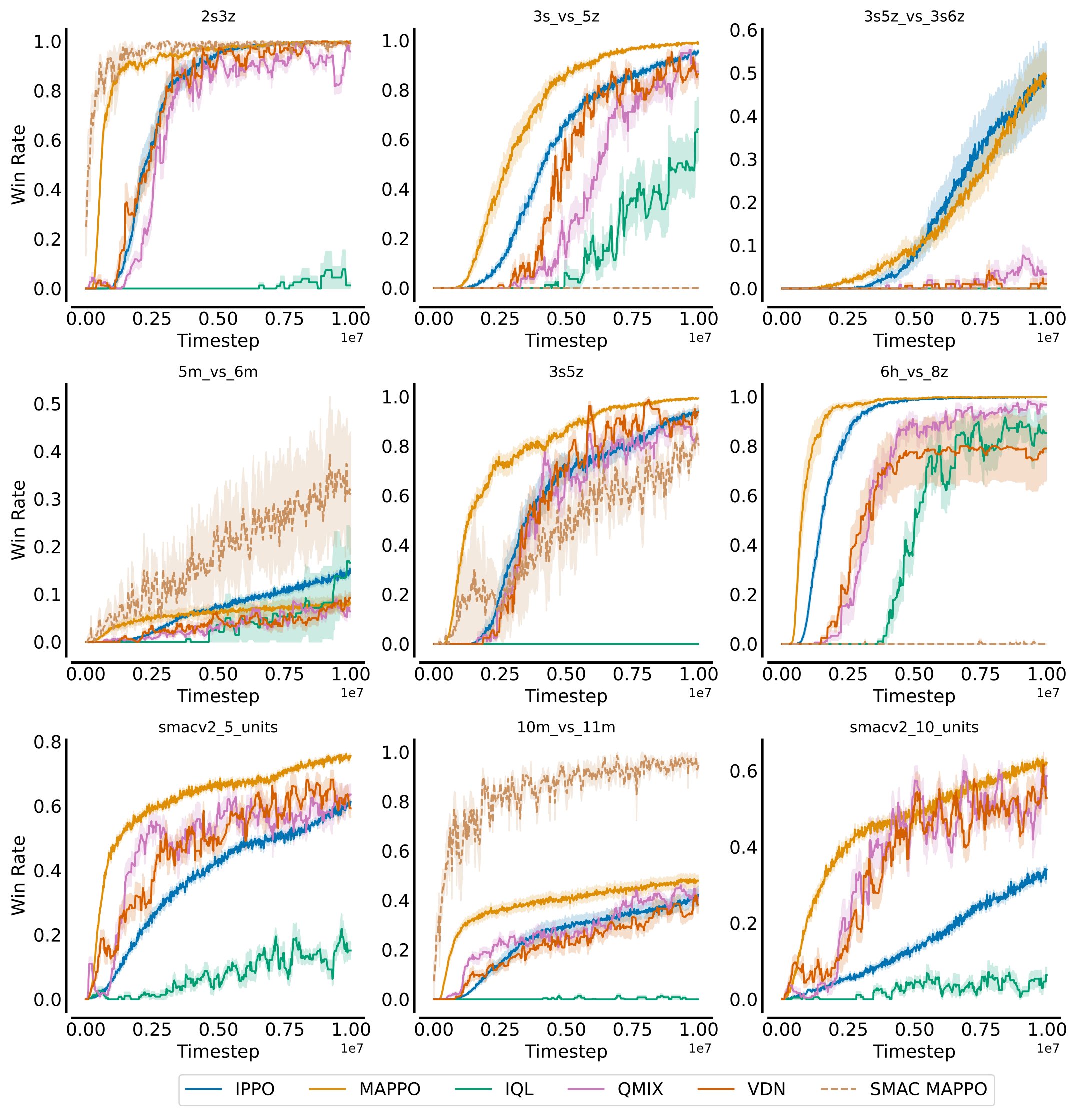


Figure 13: Comparison of IPPO, MAPPO, IQL, QMIX, VDN in SMAX with MAPPO in SMAC.

7 Hyperparameters

Value	**Ant**	**HalfCheetah**	**Walker**
VF_COEF	4.5	0.14	1.9
ENT_COEF	2×10^{-6}	4.5×10^{-3}	1×10^{-3}
LR	1×10^{-3}	6×10^{-4}	7×10^{-3}
NUM_ENVS	64	–	–
NUM_STEPS	300	–	–
TOTAL_TIMESTEPS	1×10^{8}	–	–
NUM_MINIBATCHES	4	–	–
GAMMA	0.99	–	–
GAE_LAMBDA	1.0	–	–
CLIP_EPS	0.2	–	–
MAX_GRAD_NORM	0.5	–	–
ACTIVATION	tanh	–	–
ANNEAL_LR	True	–	–

Table 2: MABrax Hyperparameters, where – indicates repeated parameters

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 MPE IPPO

Hyperparameter	**Value**
LR	0.004
NUM_ENVS	64
NUM_STEPS	128
TOTAL_TIMESTEPS	1×10^{7}
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

Hyperparameter	**Value**
LR	5×10^{-4}
NUM_ENVS	1024
NUM_STEPS	128
TOTAL_TIMESTEPS	1×10^{10}
UPDATE_EPOCHS	4
NUM_MINIBATCHES	4
GAMMA	0.99
GAE_LAMBDA	0.95
CLIP_EPS	0.2
ENT_COEF	0.01
VF_COEF	0.5
MAX_GRAD_NORM	0.5
ACTIVATION	relu
ANNEAL_LR	True
NUM_FC_LAYERS	2
LAYER_WIDTH	512

Table 5: Hyperparameters for Hanabi IPPO

Hyperparameter	**Value**
LR	0.0005
NUM_ENVS	64
NUM_STEPS	256
TOTAL_TIMESTEPS	5×10^{6}
UPDATE_EPOCHS	4
NUM_MINIBATCHES	16
GAMMA	0.99
GAE_LAMBDA	0.95
CLIP_EPS	0.2
ENT_COEF	0.01
VF_COEF	0.5
MAX_GRAD_NORM	0.5
ACTIVATION	relu
ANNEAL_LR	True

Table 6: Hyperparameters for Overcooked IPPO

Hyperparameter	Value
NUM_ENVS	8
NUM_STEPS	26
BUFFER_SIZE	5000
BUFFER_BATCH_SIZE	32
TOTAL_TIMESTEPS	2×10^6
HIDDEN_SIZE	64
MIXER_EMBEDDING_DIM*	32
MIXER_HYPERNET_HIDDEN_DIM*	128
MIXER_INIT_SCALE*	0.001
EPS_START	1.0
EPS_FINISH	0.05
EPS_DECAY	0.1
MAX_GRAD_NORM	25
TARGET_UPDATE_INTERVAL	200
TAU	1.0
NUM_MINI_EPOCHS	1
LR	0.005
LEARNING_STARTS	10000
LR_LINEAR_DECAY	True
GAMMA	0.9

Table 7: QLearning Hyperparameters in MPE
(* Hyperparameters specific to QMix.)

Hyperparameter	Value
NUM_ENVS	16
NUM_STEPS	128
BUFFER_SIZE	5000
BUFFER_BATCH_SIZE	32
TOTAL_TIMESTEPS	1×10^7
HIDDEN_SIZE	512
MIXER_EMBEDDING_DIM*	64
MIXER_HYPERNET_HIDDEN_DIM*	256
MIXER_INIT_SCALE*	0.001
EPS_START	1.0
EPS_FINISH	0.05
EPS_DECAY	0.1
MAX_GRAD_NORM	10
TARGET_UPDATE_INTERVAL	10
TAU	1.0
NUM_MINI_EPOCHS	8
LR	0.00005
LEARNING_STARTS	10000
LR_LINEAR_DECAY	False
GAMMA	0.99

Table 8: QLearning Hyperparameters in Smax
(* Parameters specific to QMix.)

Hyperparameter	Value
NUM_ENVS	32
NUM_STEPS	1
BUFFER_SIZE	1×10^5
BUFFER_BATCH_SIZE	128
TOTAL_TIMESTEPS	5×10^6
HIDDEN_SIZE	64
EPS_START	1.0
EPS_FINISH	0.05
EPS_DECAY	0.2
MAX_GRAD_NORM	1
TARGET_UPDATE_INTERVAL	10
TAU	1.0
NUM_MINI_EPOCHS	4
LR	0.000075
LEARNING_STARTS	1000
LR_LINEAR_DECAY	True
GAMMA	0.99

Table 9: QLearning Hyperparameters in Overcooked

References

[1] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems*, 34:29304–29320, 2021.

[2] 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.

[3] 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.

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

[5] Benjamin Ellis, Jonathan Cook, Skander Moalla, Mikayel Samvelyan, Mingfei Sun, Anuj Mahajan, Jakob Foerster, and Shimon Whiteson. Smacv2: An improved benchmark for cooperative multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] Edan Toledo, Laurence Midgley, Donal Byrne, Callum Rhys Tilbury, Matthew Macfarlane, Cyprien Courtot, and Alexandre Laterre. Flashbax: Streamlining experience replay buffers for reinforcement learning with jax, 2023.

[17] 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.

[18] 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.