Accelerating Swarms : Harnessing Hardware Acceleration and Parallelization in Multi-Agent Reinforcement Learning

T. Gavin[*x+], S. Lacroix[x], and M. Bronz[+]
[*]*IAS*, *Thales LAS*, Rungis, France
[x]*RIS*, *LAAS-CNRS*, Toulouse, France
[+]*Fédération ENAC ISAE-SUPAERO ONERA*, *Université de Toulouse*, France
timothee.gavin@thalesgroup.fr , simon.lacroix@laas.fr , murat.bronz@enac.fr

ABSTRACT

In Multi-Agent Reinforcement Learning, researchers often face a challenging trade-off: the use of complex environments that demand substantial computational resources, or simpler dynamics for expedited execution, albeit at the cost of transferability to more realistic tasks. This article delves into the potential of vectorizable environments, which enable parallel environment rollout and fully harness the parallelization capabilities of modern GPUs. We present a comparison of various RL environment libraries, highlighting their features and limitations for end-to-end hardware-accelerated training pipelines. We observe that the most commonly used RL algorithm libraries have yet to fully embrace end-to-end hardware-accelerated training pipeline and the limited cross-compatibility between the frameworks used for hardware acceleration and parallelization in machine learning: PyTorch, TensorFlow, and JAX, limits the mix-and-match options for RL environments and algorithms libraries.

1 INTRODUCTION

A swarm of drones is a prime example of the Decentralized Partially Observable Markov Decision Processes (Dec-POMDPs) problem that can be addressed using Multi-Agent Reinforcement Learning (MARL). In this application, each drone operates as an individual agent with its own set of actions and observations. The drones must collaborate to achieve a common goal, such as area surveillance or delivery tasks, despite having only partial and local observations of the environment. The inherent non-stationarity and decentralized partial observability in such a setup makes MARL a suitable approach. Through MARL, each drone learns from its interactions with the environment and other drones, improving its policy over time to contribute effectively to the overall objective of the swarm. However, generating environment data can be slow, especially for complex environments.

Reinforcement Learning (RL) is notoriously sample inefficient, a problem worsened in MARL due to non-stationarity and decentralized partial observability. Workloads for RL agents are steadily increasing as we tackle more complex problems that require millions of training steps. This issue is worsened by the need for hyperparameter tuning in deep RL, which requires time-consuming repeated training sessions.

To address this issue, research has focused on scalable RL frameworks. Current training pipelines use CPUs for simulating environment physics, calculating rewards, and running the environment, while GPUs accelerate neural network models during training and inference. Increasing environment throughput often involves parallelizing the environment; prior work has used multiple CPU cores and multi-threading to run multiple environment instances in parallel to speed up rollouts. However, this approach is limited by the number of cores and memory usage. Therefore, training in complex environments requires scaling up on massive distributed systems, such as large clusters with thousands of CPUs. Despite the use of these powerful systems, training times remain lengthy due to the data transfer bottleneck between the CPU, where the environment is simulated, and the GPU, where the agents are trained and evaluated. This has led to a trade-off in RL research: use complex environments requiring large compute resources, making it inaccessible to researchers with limited resources, or rely on simpler dynamics for faster execution, which makes the transferability of results to more realistic tasks more challenging.

To overcome these challenges, researchers are exploring the use of hardware accelerators to create end-to-end training pipelines. This approach leverages the parallelization capabilities of modern GPUs to achieve orders of magnitude faster training pipelines. This could significantly advance RL research by speeding up the testing and iteration of ideas, lowering computational barriers for in-depth MARL research, and allowing experiments that previously required a data center to be conducted locally on desktops or small GPU servers.

However, this article aims to show that the most commonly used RL libraries have yet to fully embrace end-to-end hardware-accelerated training pipelines. Hardware acceleration in the field of machine learning (ML) necessitates a choice between the three leading frameworks: PyTorch [1], TensorFlow [2], and JAX [3], which have limited cross-compatibility. This means that if an RL environment library adopts one of these frameworks, it may face compatibility issues with RL algorithms that are designed using the other frameworks. This limits the mix-and-match options for RL environments and algorithms libraries.

2 BACKGROUND

2.1 *Reinforcement Learning*

Reinforcement Learning is a type of machine learning for sequential decision-making. In a *rollout* phase, an *agent* interacts with an uncertain *environment* which provides it with a *partial observations* of its state, takes a series of *actions* following a *policy* and receives a scalar feedback in the form of *rewards*. These sequences of observe-act-reward, repeated over time, form the *rollouts*. The collected rollouts are then used to update the policy in a learning phase, which will then be employed in the rollout phase of the next training iteration.

2.2 *Multi-Agent Reinforcement Learning*

Single-agent RL methods often fail in multi-agent settings due to the well-known curse of dimensionality and non-stationarity. Recently, MARL approaches addressed these issues with the Centralized Training, Decentralized Execution (CTDE) approach. By training the agents in a centralized manner to leverage global information and then executing the learned policies in a decentralized manner, CTDE ensures scalability and robustness. Several state-of-the-art MARL algorithms have been developed under the CTDE framework, including value-based methods like QMIX [17] and VDAC [18], and policy gradient methods like MADDPG [19], MAPPO [20], and multi-agent versions of SAC [21].

2.3 *Environments*

Simulations are integral to RL for ensuring safety and enhancing iteration speed. They provide a risk-free testing ground, preventing real-world mishaps like crashes or environmental damage. Additionally, simulations are efficient and scalable, enabling rapid trial-and-error iterations, which are essential for the learning process in RL. A wide range of simulators were designed to help develop RL and MARL algorithms and solve problems for real-life drone applications. The degree of realism of these environments can vary. On one end, we have environments with complex physics engines that simulate low-level tasks, focusing on issues related to sensing and low-level control. On the other end, we have high-level simulations with often 2D dynamics engines that focus on high-level tasks like coordination for multi-agent operations. Table 1 lists a set of environments and highlights their features related to drone-oriented tasks.

3 THREE ML FRAMEWORKS

Hardware acceleration in the field of machine learning, whether for environments or training algorithms, necessitates a choice between the three leading frameworks: PyTorch [1], TensorFlow [2], and JAX [3]. In this section, we describe each framework. Table 2 summarizes the various features of each framework.

Environment	State[a]	Action[b]	PartObs[c]	Random[d]	Physics[e]	Drone[f]	MARL[g]	#Agents[h]	Collab[i]	Advers[j]	Hetero[k]	Comm[l]
gymnasium [4]	C	C	✓	✓	Any	✓	✗	-	-	-	-	-
PettingZoo [5]	C	C	✓	✓	2D	✗	✓	<100	✓	✓	✓	✓
Pybullet-drones [6]	C	C	✗	✓	3D	✓	✓	<10	✓	✗	✗	✗
EnvPool [7]	C	C	✗	✓	2D/3D	✗	✗	-	-	-	-	-
MAgent [8]	D	D	✗	✓	Grid	✗	✓	>1000	✗	✓	✗	✗
brax [9]	C	C	✗	✓	3D	✗	✗	-	-	-	-	-
gymnax [10]	C	C	✗	✓	2D	✗	✗	-	-	-	-	-
IsaacSIM [11]	C	C	✓	✓	3D	✓	✓	<10	✓	✗	✗	✗
Gigastep [12]	C	C	✓	✓	2D	✗	✓	>1000	✓	✓	✓	
VMAS [13]	C	C	✓	✓	2D	✗	✓	<100	✓	✓	✓	✓
JaxMARL [14]	C	C	✓	✓	2D/3D	✗	✓	<10	✓	✓	✓	✓
Jumanji [15]	C	C	✗	✓	2D	✗	✓	-	✓	✓	✗	✓
CrazyRL [16]	C	C	✗	✗	3D	✓	✓	<10	✓	✗	✗	✗

[a] Continuous (C) or discrete (D) state space
[b] Continuous (C) or discrete (D) actions space
[c] Environments support partial observability
[d] Environments support domain randomization
[e] Type of physics engine used for the simulation
[f] Environment natively handle drone oriented tasks
[g] Native support for multi-agent reinforcement learning
[h] Number of agents generally supported
[i] Collaborative multi-agent tasks
[j] Adversarial multi-agent tasks
[k] Handle heterogeneous agents
[l] Handle Inter-Agents communications

Table 1: Comparison of different RL environments features

Framework	PyTorch	TensorFlow	JAX
Neural Networks	torch.nn	Keras	Flax, Haiku
Gradient processing and optimization	torch.optim	tf.optimizers or tf.train	Optax
Checkpointing	torch.save, torch.load	tf.train.Checkpoint, tf.train.CheckpointManager	Orbax
Testing utilities	unittest, pytest	tf.test	Chex
Distributions	torch.distributions	TensorFlow Probability	Distrax, TensorFlow Probability
Parallelization	Not automatic	Not automatic	Automatic
Target Audience	Researcher, Developers	Researcher, Developers	Researchers
Maturity of Development	Mature	Mature	Not Mature

Table 2: Comparison of the three different ML Frameworks

3.1 PyTorch

PyTorch is an open ML library developed by Meta AI Research, and now governed by PyTorch Foundation, a subsidiary of the Linux Foundation. PyTorch handles PyTorch tensors, similar to NumPy multidimensional arrays, but that can be operated on CUDA GPUs. Environments implemented using PyTorch tensors can be vectorized and hardware-accelerated, such as VMAS [13]. PyTorch benefits from a very consistent and mature ecosystem, with all main libraries needed for ML research being developed and maintained in the same package. It is an easily debuggable, hackable, and very flexible framework, and fits well into the Python ecosystem, allowing the use of Python debugger tools for debugging PyTorch code. It is targeted at both developers and researchers, who favor its Python-like imperative coding style, which makes it easy for developers to learn. Finally, PyTorch offers extensive community support and documentation and seamless deployment options.

3.2 TensorFlow

TensorFlow is an open ML library developed by Google. TensorFlow handles TensorFlow Tensors, similar to PyTorch ones, which can be operated on CUDA GPUs. Environments implemented using TensorFlow Tensors can be vectorized and hardware-accelerated, and TensorFlow provides an API as part of its RL library TF-Agents [22] to directly create parallelized and accelerated environments. However, it has yet to be broadly adopted. TensorFlow is known for its flexibility, scalability, and production readiness. TensorFlow also benefits from a very consistent ecosystem, with all main libraries needed for ML research being developed and maintained in the same package, except for neural network design, which is done with Keras' deep neural network API. However, we've seen a decline in the use of TensorFlow in both the industrial and research communities in the last few years, with researchers preferring the flexibility of PyTorch. Still, TensorFlow remains a dependable framework and offers extensive community support and documentation.

3.3 JAX

JAX is a Python library for accelerator-oriented array computation and program transformation, designed for high-performance numerical computing, built on top of Google's XLA (Accelerated Linear Algebra) library, and developed by researchers at Google and DeepMind. It's known for its automatic differentiation capabilities, which make it well-suited for optimization tasks such as training neural networks. JAX is also popular in the research community due to its auto-vectorization, device parallelism, and just-in-time compilation abilities, allowing to seamlessly parallelize Python code across multiple accelerators. JAX provides a NumPy-like interface, making it easy to transition from traditional numerical computing to more complex ML tasks. However, JAX is not an ML library, and its usage for machine RL requires dealing with various libraries, interoperable but independent [23, 24]. JAX is newer than PyTorch and TensorFlow and is celebrated for its immense scalability. However, it is at the edge of AI research and requires keeping pace with a rapidly evolving research ecosystem.

4 Hardware-accelerated Environments Recommended Features

Here, we elaborate on the design choices and features required in environment implementation to accelerate the training pipeline in RL and enable their use for multi-drone applications. Table 3 summarizes key features and lack thereof in existing environments.

4.1 Vectorization

Vectorization is a technique in RL that enables parallel simulations. In a training iteration, the rollout phase is typically the most time-consuming part. Vectorization mitigates this by adding a batch dimension to the rollout phase, through sampling multiple sub-environments at the same time. Vectorization is something that is now widely adopted in RL, with the most popular environment libraries such as Gymnasium [4] providing built-in vectorization.

4.2 Hardware acceleration

One straightforward method of vectorization is to simulate one environment instance per thread. However, this approach has limitations. The scalability is restricted by the number of physical cores in the system. Running a large number of threads can lead to synchronization issues, context-switching overhead, and memory bandwidth limitations. Moreover, the frequent switching between CPU cores, designed for sequential tasks, and GPUs, can be inefficient. It is possible to alleviate this, by leveraging the parallel processing and acceleration capabilities of GPUs.

GPU acceleration of the training environment is not something new. Gym-pybullet-drones [6] uses GPU-based rendering to speed up the simulation of multi-agent quadcopter control tasks. Moreover, a CUDA port of the Atari Learning Environment [26] was implemented by Dalton et al. However, these solutions still require data to be transferred back and forth between the CPU and the GPU when used in a training pipeline, introducing performance bottlenecks.

It is possible to eliminate those inefficiencies by keeping all the computations on the GPU [27]. GPU vectorization and parallelization allow seamless scaling to tens of thousands of parallel environments on accelerated hardware, speeding up training by order of magnitudes.

4.3 Python based

Python is an interpreted language, that is mostly known for its simplicity and rapidity of development rather than its execution performance. Complex environments implemented in Python require high CPU and memory requirements. The first attempts to alleviate this issue have been to implement the environments in C++. While it makes running the environments on the CPU much faster, it cannot be parallelized natively on the GPU, making it difficult to parallelize without large clusters of CPU. Currently, GPU Parallelization methods are typically implemented in CUDA C, such as [26]. However, the use of C++ and CUDA C considerably reduces their accessibility for most ML researchers who prefer Python [28]. Python environments enable a wider adoption in the MARL community.

New alternatives use Just-In-Time (JIT) compilation libraries, to compile Python code at execution. For example, WarpDrive [25] provides a Python interface and environment wrappers for implementing environments in CUDA C or Python using Numba, and automatically manages data transfer between the host and the device. Numba [29], optimizes Python code and allows easy Just-In-Time (JIT) compilation for Python array and numerical functions, significantly speeding up CPU code execution. It also supports CUDA GPU compilation for a limited subset of Python code, but this requires a solid understanding of the nuances and intricacies of the CUDA C language, as the majority of Numba GPU API functions map directly to Nvidia's CUDA C language. Tran-

Environment	Vector[a]	GPU[b]	Python[c]	PyTorch[d]	TensorFlow[e]	JAX[f]	Extens[g]	Open[h]	Compat[i]	Maint[j]
gymnasium [4]	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓
PettingZoo [5]	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓
Gym-Pybullet-drones [6]	✓	✗	✗	✗	✗	✗	✓	✓	✗	✓
EnvPool [7]	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓
MAgent (original) [8]	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗
brax [9]	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓
gymnax [10]	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓
IsaacSIM [11]	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓
Gigastep [12]	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓
VMAS [13]	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓
JaxMARL [14]	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓
Jumanji [15]	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓
CrazyRL (NumPy) [16]	✓	✗	✓	✗	✗	✗	✓	✓	✓	✗
CrazyRL (JAX) [16]	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗
WarpDrive [25]	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗

[a] Vectorization
[b] GPU acceleration and parallelization
[c] Python-based
[d] PyTorch-based
[e] TensorFlow-based
[f] JAX-based
[g] Extensible
[h] Open-Source
[i] Compatible with common APIs
[j] Maintained

Table 3: Comparison of different RL environments hardware-acceleration features

sitioning from CPU to GPU with Numba involves rewriting Python functions to manage kernel invocation and thread positioning, which is not straightforward.

As explained in Section 3, hardware acceleration in the field of machine learning in Python is instead typically done by adopting one of the three leading frameworks: PyTorch [1], TensorFlow [2], and JAX [3]. Vectorized environments with a batch dimension can be implemented using either PyTorch or TensorFlow tensors, allowing to parallelize and running them on GPU. However, the number of available PyTorch-based libraries is quite low, and no libraries were found using TensorFlow. An environment implemented using JAX arrays can be automatically vectorized to run a batch of this environment in parallel, and in comparison to the two other frameworks there have been more initiatives for parallelized environments using JAX presented in the literature, as JAX is very popular among researchers for its parallelization capabilities, and is seen as a successor of TensorFlow. However, there is limited cross-compatibility between these frameworks. Switching from one framework tensor representation to another requires moving the data on the CPU and inevitably introduces overheads in the training pipelines.

Each of these frameworks offers nowadays some JIT compilation functionality. Unlike Numba, JIT compilation with these frameworks for hardware accelerators doesn't require in-depth knowledge of low-level CUDA C details, and streamlines the process of adapting code for execution on a GPU automatically, simplifying the process. PyTorch offers limited JIT compilation with the TorchScript language, a typed subset of Python. It can be applied to any Python function that manipulates PyTorch tensors and supports PyTorch functions and many Python built-in functions. This is for example used in Isaac Gym [11], although the core functionalities are coded in C++ and CUDA C, a Python interface is provided for users to implement their own Python environment, and take advantage of TorchScript JIT to compile their Python functions to lower level scripts which are added to the parallelized training pipelines. TensorFlow allows using the XLA compiler through tf.function. It can be applied to any function that manipulates TensorFlow tensors, but it is not broadly used outside the optimization of neural network models and their training algorithms written with this framework [22]. In comparison, JAX JIT compilation using the XLA compiler is much more broadly adopted. It can be applied to any Python function that manipulates JAX arrays and automatically optimizes computations on hardware accelerators. One advantage over PyTorch is that JAX maintains most NumPy functions and their signatures, making it easier to directly adapt native Python code.

4.4 Compatible with common APIs

The Gym API [4], developed by OpenAI, has been a standard in the field since its release in 2016 and is the most used framework for coding simulators. DeepMind's environment API [23], known as dm_env, was introduced later in 2019 but is also broadly used. These APIs have played a significant role in shaping the development and standardization of RL environments. The majority of RL algorithm libraries, such as RLlib [30], Stable Baselines3 [31], and TorchRL [32], use these standardized APIs therefore, ensuring RL environments' compatibility with these standardized APIs is critical to ensure an easy integration in the RL ecosystem.

To enable batched execution of transition simulations, RL libraries are increasingly adopting *stateless* environments, where instead of environments keeping track of the last physical state encountered and relying on it to simulate the transition from state to state, stateless environments expect to be provided with the current state at each step, along with the action taken. With PyTorch, TensorFlow, or JAX, an algebraic operation can be seamlessly performed on scalars, vectors, or tensors, allowing the state to be easily batched to run multiple simulation steps in parallel. This is widely used in JAX because methods written this way are functionally pure and easily jittable. Gymnax [10] introduces such an API, where the Gym API is made stateless, and Jumanji [15] does a similar transformation for the dm_env API.

4.5 Open-source, Extensible, and Maintained

An environment library should be open-sourced and offer a high level of customization, to allow researchers and developers to inspect the code, ensuring that the algorithms and environments are implemented correctly and without hidden biases, to allow the research community to modify, adapt, or create custom scenarios that address their use cases, or to modify the code to specialize it for their application. Finally, as the RL research is rapidly evolving it is crucial that the library is still maintained, to ensure compatibility with the dependencies (in particular the last versions of Python) prevent issues that could arise from outdated code, and keep up with the advance and new features used in RL. Well-maintained code is also often a sign of trust within the community, showing that many users have adopted the library, are still contributing to it, and can provide active support.

5 Training on hardware-accelerated environments

Hardware-accelerated environments are increasingly gaining prominence in the single RL ecosystem. However, the existing RL libraries offering implementations of the state-of-the-art RL algorithm often fall short of effectively managing end-to-end training pipelines on accelerated hardware. This problem is further aggravated with MARL, as many of the most commonly used algorithm libraries lack native support for MARL. Table 4 summarizes how existing RL libraries support hardware-accelerated multi-agent environments.

One of the most popular RL libraries, RLlib [30], is implemented to facilitate highly distributed RL workloads. It can scale training from a single core to many thousands of

RL Libraries	Vector[a]	GPU[b]	Multi-GPU[c]	API[d]	MARL[e]	Audience[f]	PyTorch[g]	TensorFlow[h]	JAX[i]
Stable Baselines3 [31]	✓	limited	limited	gym	✗	D	✓	✗	limited
RLlib [30]	✓	✗	✗	gym	✓	D	✓	✓	✗
TorchRL [32]	✓	✓	✓	gym*	✓	R&D	✓	✗	limited
TF-Agents [22]	✓	✓	✗	dm_env	✗	D	✗	✓	✗
Acme [33]	✓	✓	✗	dm_env	✗	R	✗	✓	✓
CleanRL [34]	✓	with JAX	✗	any	limited	R	✓	✗	limited
PureJaxRL [35]	✓	✓	✗	gymnax*	✗	R	✗	✗	✓
JaxMARL [14]	✓	✓	✗	gymnax*	✓	R	✗	✗	✓
Mava [36]	✓	✓	✓	Jumanji	✓	R	✗	✗	✓

* provides wrappers for a limited number of additional APIs

[a] Handles vectorized environments
[b] Handles environments parallelized on GPU
[c] Handles environment parallelized on multiple GPUs
[d] Supported environment API
[e] Natively support for MARL
[f] Target Audience (R) Researchers (D) Developers
[g] PyTorch-based
[h] TensorFlow-based
[i] JAX-based

Table 4: Comparison of different RL libraries support for hardware-accelerated multi-agent environments

cores in a cluster. Yet, it does not support end-to-end training pipelines on accelerated hardware as even if the environment was running on a hardware accelerator, the training pipeline requires the rollout data to be processed on the CPU, before sending it back to the GPU, where the agents are trained. These transfers back and forth introduce large overhead in the training pipeline.

Another difficulty is, the limited cross-compatibility between the three leading frameworks used for hardware acceleration: PyTorch [1], TensorFlow [2], and JAX [3]. Switching from one framework tensor representation to another requires moving the data on the CPU and inevitably introduces some overhead in the training pipeline. This means that if an RL environment library adopts one of these frameworks, it will face compatibility issues with RL algorithms that are designed for the other frameworks. This limits the mix-and-match options for RL environments and algorithms libraries.

The most popular RL libraries, such as Stable Baselines3 [31], are implemented in PyTorch or TensorFlow, which means that they do not support JAX-based environments natively, and require converting JAX arrays to their tensor representation to be able to process them. This is the case also with TorchRL [32], while it natively only supports end-to-end hardware-accelerated pipelines for PyTorch-based environments such as VMAS [32, 13], it provides a generic environment API and wrappers to support other existing simulators, such as Jumanji environments easily [15], which is JAX-based. However, JAX-based (and TensorFlow-based) environments inevitably suffer from a consequent overhead due to the conversion from JAX array to PyTorch Tensors.

	Single-Agent RL				MARL					
RL Libraries	DQL	DDPG	PPO	SAC	IPPO	MAPPO	IQL	MADDPG	QMIX	SAC
Stable Baselines3 [31]	FF - R	FF - R	FF - R	FF - R	-	-	-	-	-	-
RLlib [30]	FF - R	FF - R	FF - R	FF - R	FF - R	FF - R	FF - R	FF - R	-	FF - R
TorchRL [32]	FF	FF	FF	FF	FF	FF	FF - R	FF	FF	FF
TF-Agents [22]	FF - R	FF - R	FF - R	FF - R	-	-	-	-	-	-
Acme [33]	FF	FF	FF	FF	FF	-	-	-	-	-
CleanRL [34]	FF	FF	FF - R	FF	FF	FF	-	-	-	-
PureJaxRL [35]	FF	-	FF - R	-	-	-	-	-	-	-
JaxMARL [14]	-	-	-	-	FF - R	FF - R	R	-	R	-
Mava [36]	-	-	-	-	FF - R	FF - R	R	-	-	FF

FF — The library provides a Feed-Forward version of the baseline algorithm
R — The library provides a Recurrent version of the baseline algorithm

Table 5: Comparison of how RL libraries cover the existing state-of-the-art RL and MARL algorithms

This lack of support for JAX-based environments by the most used and maintained RL libraries led the research community to create their own RL and MARL algorithms libraries implemented in JAX, such as PureJaxRL [35], Jax-MARL [14] and Mava [36]. Often, while implementing and providing new environment libraries in JAX, the research community provides their own re-implementation of the baseline algorithms in JAX [14, 15, 36]. This results in an uneven offer of JAX implementations of the state-of-the-art RL algorithms. Often, these implementations lack certain training options or features, and their design choices are shaped by the benchmarked environments that accompany the algorithms. For example, the MAPPO implementation of Mava does not use value normalization, even though it is a recommended feature that has been shown to never hurt training and often improves the final performance of MAPPO significantly [20]. Again, this problem is worse with MARL algorithms. In particular, existing MARL libraries in JAX often lack the recurrent versions of the baselines algorithms, which are used to address partial observability in Dec-POMDPs, an intricate aspect of MARL problems such as the control of swarms of drones. The JAX ecosystem still lacks a unified, reliable, tested, and documented collection of RL algorithms. Table 5 provides a summary of how existing RL libraries cover the existing state-of-the-art RL and MARL algorithms.

6 Conclusion

The potential of vectorizable environments to leverage modern GPUs for parallel environment rollout promises a substantial boost in computational efficiency. However, the current landscape of RL algorithm libraries and frameworks falls short of effectively managing end-to-end training pipelines on accelerated hardware. Some initiatives have been introduced for parallelized and hardware-accelerated environments using PyTorch and JAX to a larger extent, while TensorFlow usage has been on the decline. However, the limited cross-compatibility between these frameworks hinders the seamless mix-and-match of RL environments and algorithm libraries, as switching from one framework tensor representation to another requires moving the data on the CPU and inevitably introduces some overhead in the training pipeline. This forces the research community to implement RL algorithm libraries for each framework, but while the most popular RL algorithm libraries are implemented in PyTorch and TensorFlow they have yet to fully adopt GPU-parallelized environments, and in contrast, the libraries implemented in JAX natively handle parallelized environments but lack maturity. A problem accentuated in MARL, as it is a specialized branch of RL research.

References

[1] Adam Paszke, Sam Gross, Francisco Massa, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[3] James Bradbury, Roy Frostig, Peter Hawkins, et al. JAX: composable transformations of Python+NumPy programs, 2018.

[4] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, et al. Gymnasium, March 2023.

[5] J Terry, Benjamin Black, Nathaniel Grammel, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021.

[6] Jacopo Panerati, Hehui Zheng, SiQi Zhou, et al. Learning to Fly—a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7512–7519, September 2021.

[7] Jiayi Weng, Min Lin, Shengyi Huang, et al. EnvPool: A Highly Parallel Reinforcement Learning Environment Execution Engine. *Advances in Neural Information Processing Systems*, 35:22409–22421, December 2022.

[8] Lianmin Zheng, Jiacheng Yang, Han Cai, et al. MAgent: A Many-Agent Reinforcement Learning Platform for Artificial Collective Intelligence. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), April 2018.

[9] C. Daniel Freeman, Erik Frey, Anton Raichuk, et al. Brax – a differentiable physics engine for large scale rigid body simulation, June 2021.

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

[11] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, et al. Isaac gym: High performance gpu-based physics simulation for robot learning, August 2021.

[12] Mathias Lechner, Lianhao Yin, Tim Seyde, et al. Gigastep - One Billion Steps per Second Multi-agent Reinforcement Learning. *Advances in Neural Information Processing Systems*, 36:155–170, December 2023.

[13] Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. VMAS: A Vectorized Multi-agent Simulator for Collective Robot Learning. In Julien Bourgeois, Jamie Paik, Benoît Piranda, et al., editors, *Distributed Autonomous Robotic Systems*, pages 42–56, Cham, 2024. Springer Nature Switzerland.

[14] Alexander Rutherford, Benjamin Ellis, Matteo Gallici, et al. Jaxmarl: Multi-agent rl environments in jax, December 2023.

[15] Clément Bonnet, Daniel Luo, Donal Byrne, et al. Jumanji: a diverse suite of scalable reinforcement learning environments in jax, March 2023.

[16] Florian Felten, Coline Ledez, Pierre-Yves Houitte, et al. Crazyrl: A multi-agent reinforcement learning library for flying crazyflie drones. `https://github.com/ffelten/CrazyRL`, 2023.

[17] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, et al. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. pages 4295–4304. PMLR, July 2018.

[18] Jianyu Su, Stephen Adams, and Peter Beling. Value-Decomposition Multi-Agent Actor-Critics. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(13):11352–11360, May 2021.

[19] Ryan Lowe, YI WU, Aviv Tamar, et al. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[20] Chao Yu, Akash Velu, Eugene Vinitsky, et al. The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games. *Advances in Neural Information Processing Systems*, 35:24611–24624, December 2022.

[21] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. pages 1861–1870. PMLR, July 2018.

[22] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, et al. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018.

[23] DeepMind, Igor Babuschkin, Kate Baumli, et al. The DeepMind JAX Ecosystem, 2020.

[24] Jonathan Heek, Anselm Levskaya, Avital Oliver, et al. Flax: A neural network library and ecosystem for JAX, 2023.

[25] Tian Lan, Sunil Srinivasa, Huan Wang, and Stephan Zheng. WarpDrive: Extremely Fast End-to-End Deep Multi-Agent Reinforcement Learning on a GPU, October 2021.

[26] Steven Dalton and iuri frosio. Accelerating Reinforcement Learning through GPU Atari Emulation. In *Advances in Neural Information Processing Systems*, volume 33, pages 19773–19782. Curran Associates, Inc., 2020.

[27] Matteo Hessel, Manuel Kroiss, Aidan Clark, et al. Podracer architectures for scalable reinforcement learning. April 2021.

[28] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence. *Information*, 11(4):193, April 2020.

[29] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 1–6, New York, NY, USA, November 2015. Association for Computing Machinery.

[30] Eric Liang, Richard Liaw, Robert Nishihara, et al. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

[31] Antonin Raffin, Ashley Hill, Adam Gleave, et al. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[32] Albert Bou, Matteo Bettini, Sebastian Dittert, et al. Torchrl: A data-driven decision-making library for pytorch, November 2023.

[33] Matthew W. Hoffman, Bobak Shahriari, John Aslanides, et al. Acme: A research framework for distributed reinforcement learning, September 2020.

[34] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, et al. CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[35] Chris Lu, Jakub Kuba, Alistair Letcher, et al. Discovered Policy Optimisation. *Advances in Neural Information Processing Systems*, 35:16455–16468, December 2022.

[36] Ruan de Kock, Omayma Mahjoub, Sasha Abramowitz, et al. Mava: a research library for distributed multi-agent reinforcement learning in jax, December 2021.