JAXNASIUM: EMBRACING PYTREES FOR ANY-AGENT, CROSS-SUITE REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Hardware-accelerated reinforcement learning in JAX has gained rapid popularity in recent years and has proven to be a successful method of scaling online reinforcement learning. The resulting ecosystem, however, is fragmented: environment suites define their own interfaces, and algorithm implementations are typically written for a single suite, a single type of space, and either JAX-native or host-based environments. Here we present Jaxnasium, a library for JAX-based reinforcement learning, aiming to make the field more accessible. Jaxnasium represents observation and action spaces as PyTrees, and uses this structure to build networks and support multi-agent processing. Through this, its near-single-file implementations run on discrete, continuous, image, and composite spaces, as well as in multi-agent environments, without any changes to the algorithm code. The same files serve as readable, research-friendly algorithms, as well as importable algorithms for more applied workloads. Through a minimal Gymnasium-style interface and wrappers for 11 suites, over 300 environments are available on which a small PPO implementation with default hyperparameters is able to run. Additionally, through a simple wrapper, we can run Gymnasium and EnvPool environments with the same algorithm and training loop, with no performance loss compared to architectures specifically built for training JAX algorithms on host-based environments. Jaxnasium further provides tools for multi-seed evaluation, hyperparameter sweeps, and project scaffolding. Jaxnasium is open-sourced at `https://anonymous.4open.science/r/jaxnasium-2909`.

1 INTRODUCTION

Running the entire reinforcement learning loop on the accelerator has reshaped what is possible for many researchers. When both the environment and the agent are written in JAX (Bradbury et al., 2018), rollouts and updates can together be compiled and vectorized without host (CPU) round-trips, which can yield speed-ups of several orders of magnitude (Hessel et al., 2021; Lu et al., 2022). A broad ecosystem of JAX-native environments has emerged, including Gymnax (Lange, 2022), Brax (Freeman et al., 2021), Jumanji (Bonnet et al., 2024), Pgx (Koyamada et al., 2023), JaxMARL (Rutherford et al., 2024), XLand-MiniGrid (Nikulin et al., 2023), and Craftax (Matthews et al., 2024), each with its own interface and conventions for state, time steps and spaces. This expansion added to the already large landscape of reinforcement learning.

Algorithm implementations, however, have not kept pace with this breadth. They tend to be written for a specific environment suite and/or space type. As a result, even within the same library, an algorithm is often re-implemented for discrete and continuous actions, for image and vector observations, for single- and multi-agent settings. Libraries serve different users in different ways: research-oriented libraries provide readable single- or near-single-file implementations that are easy to modify (Huang et al., 2022; Lu et al., 2022), while production-oriented libraries (Raffin et al., 2021; Liang et al., 2018) offer a single importable algorithm that handles many environments, at the cost of spreading the implementation over many interacting and inherited components, making them less accessible for researchers who want to adapt the underlying algorithms.

A new divide now separates JAX-native environments from the many environments that still run on the host CPU, such as those in Gymnasium (Towers et al., 2025) and EnvPool (Weng et al., 2022). For the former, the entire training loop can run on the accelerator; for the latter, the environment

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

steps on the host and only the network runs on the accelerator. These two settings call for different training architectures (originally proposed as Anakin and Sebulba in Hessel et al. (2021)), and different implementations are then again written for each of them. As a result, the starting point for researchers is typically an implementation that supports a subset of tasks or is difficult to modify.

We introduce **Jaxnasium**, a lightweight RL library built on Equinox (Kidger & Garcia, 2021), with the aim of making JAX-based RL more accessible. At its core are near-single-file algorithm implementations that operates many environments. They build on PyTrees, JAX's notion of arbitrarily nested structures (including structures built with standard Python containers such as dictionaries, tuples and lists). In Jaxnasium, observation and action spaces are PyTrees of primitive spaces, and each algorithm constructs its networks from the structure of these spaces without any complicated machinery. A single implementation trains on discrete, continuous, image and composite spaces alike, and even seamlessly extends to multi-agent settings, where each agent simply has its own entry in the space PyTree. Because JAX provides simple operations on PyTrees, this generally requires little special-case code, and our implementations remain readable and modifiable for algorithm researchers.

Building on Equinox further lets us write algorithms as classes while respecting JAX's functional requirements, giving practitioners the familiar interface of libraries such as Stable-Baselines3. To let these algorithms run on many environments, Jaxnasium defines a minimal, Gymnasium-style environment interface and currently wraps over 300 environments from different domains into it, on which a single small PPO implementation can operate (Figure 1). Although the algorithms target end-to-end JAX training for maximum performance, host-based environments (like Gymnasium and EnvPool) run within the same compiled training loop through a wrapper rather than a separate implementation, matching or exceeding the throughput of implementations built specifically for host-based environments (Section 5).

Concretely, our contributions are:

- **Space-agnostic algorithms.** Near-single-file implementations whose networks are constructed from the space PyTrees, so that each algorithm supports arbitrary observation and action spaces and extends to multi-agent settings without code changes. The same code serves as a readable reference for researchers and as an importable, Stable-Baselines-style algorithm for practitioners.
- **One implementation for JAX-native and host environments.** Gymnasium and EnvPool environments run inside the same compiled training loop as JAX-native ones. We show that this loses no throughput compared to implementations dedicated to either setting.
- **A common environment interface.** A minimal, Gymnasium-style functional interface with PyTree spaces, and wrappers that bring 11 existing JAX environment suites under it.
- **Research tooling.** A command-line interface that scaffolds new projects, and utilities for multi-seed evaluation and hyperparameter sweeps (grid, random, Sobol and one-at-a-time), and a small `tree` package for convenience operations over PyTrees.

```python
import jaxnasium as jym
import jaxnasium.algorithms as jxa

ppo = jxa.PPO()
envs = jym.registry.registered_envs

for env_id in envs:
    env = jym.make(env_id)
    agent, metrics = ppo.train(key, env)

    # agent.save(...)
    # agent.evaluate(...)
```

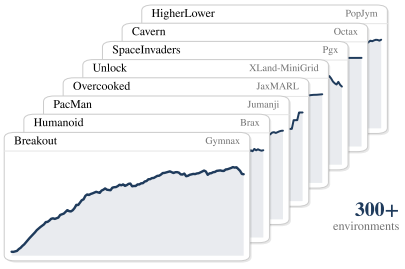


Figure 1: Jaxnasium registers 333 environments and includes algorithm implementations that can train on over 300 of them without any changes. The functionality is familiar to those who have used well-known libraries such as Gymnasium and Stable-Baselines.

2 BACKGROUND & LANDSCAPE

Reinforcement learning has notoriously been data-hungry (Kaiser et al., 2024) and as a result, accelerating data generation has been an important challenge in the field. Because early environment suites had been implemented primarily in Python, a natural optimization strategy has been to (re)implement environments in faster compiled languages such as C or C++ (Weng et al., 2022; Cobbe et al., 2019; Suarez, 2025; Shacklett et al., 2023).

While these efforts dramatically reduce the CPU-side overhead, they cannot address the fundamental issue that data has to repeatedly travel between the host CPU (where trajectories are being generated in the environment), and the accelerator device (where the policy generates actions and the networks are updated). Additionally, as accelerators are only part of the reinforcement learning process, the field essentially "lost the hardware lottery" (Scarfe et al., 2025), as it struggled to benefit from the massive GPU scaling that propelled computer vision and natural language processing following the ImageNet moment (Deng et al., 2009; Krizhevsky et al., 2012).

To bypass the communication bottleneck, one may attempt to execute the environment transitions directly on the accelerator. WarpDrive (Lan et al., 2022) demonstrated this capability by writing the environment dynamics directly in CUDA kernels. However, writing custom CUDA kernels introduced engineering friction for the machine learning community. JAX (Bradbury et al., 2018) emerged as a compelling alternative, allowing researchers to write familiar high-level Python code that could automatically be vectorized and compiled to accelerators via XLA.

End-to-End Reinforcement Learning in JAX Through JAX, accelerator-native environments enabled unified training pipelines without device communication overhead. Hessel et al. (2021) formalized this idea for reinforcement learning and introduced the **Anakin** architecture, which trained agents end-to-end on accelerators when environments were JIT-compatible through JAX. The same paper also introduced the **Sebulba** architecture for traditional environments in which data generation is still executed on the host device. Such an architecture was (and is) still relevant, as the vast majority of environments are not written to be fully JIT-compatible.

The Anakin architecture was later popularized as PureJAXRL (Lu et al., 2022), which provided simple single-device implementations (in the spirit of CleanRL (Huang et al., 2022)) for algorithms such as PPO (Schulman et al., 2017). They showed how this method was able to train reinforcement learning agents multiple orders of magnitude faster, possibly reducing experiment training times from days to minutes.

Functional Environments Standard reinforcement learning APIs, most prominently Gymnasium (Towers et al., 2025) and `dm-env` (Muldal et al., 2019), rely on stateful object-oriented designs with in-place mutations. Because JAX relies on pure, side-effect-free functions, the traditional interface was incompatible with PureJAXRL-style training. Although Gymnasium eventually introduced an experimental functional interface `FuncEnv`, it arrived after several JAX-native suites (Lange, 2022; Freeman et al., 2021) had been available, and its supporting utilities are not designed to be JIT-compatible. As a result, this interface has seen little adoption.

JAX JAX (Bradbury et al., 2018) is a Python library focused on efficient array operations and built around composable function transformations. These function transformations are most notably Just-In-Time compilation via XLA (`jax.jit`), automatic vectorization (`jax.vmap`), and automatic differentiation (`jax.grad`). In reinforcement learning, `jax.jit` allows an entire training loop to be compiled for the accelerator, and `jax.vmap` turns a single-environment step function into a batched one without modification. These function transformations, however, come with constraints: the function inputs and outputs must be numerical arrays or scalars, and those arrays must have shapes that are known at compile time; and finally, the functions must be pure, free of side-effects. Any state, like the environment state, must therefore be passed into a function explicitly and returned from it.

PyTrees Such state is rarely just a single array. To accommodate more complex structures of arrays, JAX introduces the concept of PyTrees and allows PyTrees of numerical arrays as valid inputs for its transformations. A PyTree is a (nested) tree structure whose internal nodes are registered containers (such as `tuple` and `dict` by default) and whose leaves are anything not registered

as such a container. JAX operates over these PyTrees through defined flattening and unflattening rules, which decompose a tree into a flat sequence of leaves (on which to operate) alongside the structure metadata required for reconstruction. Outside of transformations, PyTree utilities such as `jax.tree.map` accept leaves of any type, which provides a convenient way to operate over any nested structure (Section 3.2).

Equinox To bridge between JAX's functional paradigm and object-oriented code, Kidger & Garcia (2021) introduced Equinox. Its core building block, `equinox.Module`, is an enhanced Python dataclass that is automatically registered as a JAX PyTree. Within a module, array attributes (such as neural network weights) are treated as transformable PyTree leaves, while non-array attributes (such as configuration metadata) can be marked as part of the static tree structure or filtered out. Because Equinox modules are valid JAX PyTrees, methods defined on them remain functional, and instances can freely cross `jax.jit`, `jax.vmap`, and `jax.grad` boundaries. Jaxnasium adopts this object-functional architecture and builds its environments, wrappers, networks and algorithms as Equinox modules.

3 Environments as PyTrees

Currently, many JAX environment suites have been created, and many have established their own JAX-compatible interface. Although some suites borrowed their interface from existing libraries (Matthews et al., 2024; Tomilin et al., 2026; Morad et al., 2023), many suites have opted to create their own (Lange, 2022; Bonnet et al., 2024; Freeman et al., 2021; Koyamada et al., 2023; Rutherford et al., 2024; Nikulin et al., 2023; Pignatelli et al., 2025; Radji et al., 2025). Similar to traditional suites, multi-agent reinforcement learning has also received its own functional interface (Rutherford et al., 2024; de Kock et al., 2023), primarily following the PettingZoo (Terry et al., 2021) convention. Using several suites from the same code therefore requires wrappers that bridge these differences (Toledo, 2025).

Jaxnasium is not an environment suite. The interface introduced here primarily exists to provide the algorithms of Section 4 a single target. Those algorithms primarily require: i) a functional, JIT-compatible signature; ii) separate termination and truncation signals, so that value estimates can be bootstrapped correctly; and iii) observation and action spaces that are arbitrary PyTrees, so that composite spaces and multi-agent settings share one abstraction. Beyond these requirements, we made two choices for convenience: the interface stays close to Gymnasium (Towers et al., 2025), so that it is familiar to most practitioners, and environments are PyTrees themselves, so that their parameters travel with the object and they can freely pass through JAX function transformations. Section 5 describes how existing suites are wrapped into this interface.

3.1 Environment Interface

Jaxnasium largely follows the functional design of Gymnax (Lange, 2022): `reset` returns an initial observation together with the environment state, and `step` takes that state explicitly and returns the next one. For the output of `step`, we primarily mirror Gymnasium (Towers et al., 2025) and separate natural termination from truncation, e.g. by a time limit. [1]

$$(\text{obs}, \text{reward}, \text{terminated}, \text{truncated}, \text{info}), s_{t+1} \leftarrow \texttt{step}(\text{key}, s_t, a)$$

Environments reset automatically when an episode ends, as is usual in vectorized training. Since the returned observation then belongs to the next episode, the final observation of the finished episode is stored in `info`, so that algorithms can still correctly bootstrap from it after truncation. Listing 1 shows the resulting interaction loop.

Gymnax passes the environment parameters as a separate argument to `step` and `reset` and treats the environment object itself as static under `jax.jit`. Jaxnasium instead defines environments and wrappers as Equinox modules (Kidger & Garcia, 2021), such that parameters are attributes on the object. An environment is then itself a PyTree and can be passed through `jax.jit` and `jax.vmap` like any other value. For settings that require changing parameters on every call, such as meta-RL, we provide a wrapper for the Gymnax-style parameter argument.

[1] Recent versions of Gymnax (from v1.0) likewise return separate termination and truncation flags.

```python
import jax
import jaxnasium as jym

key = jax.random.PRNGKey(0)
env = jym.make("CartPole-v1")

key, reset_key, step_key = jax.random.split(key, 3)
obs, state = env.reset(reset_key)
action = env.action_space.sample(step_key)

(obs, reward, terminated, truncated, info), state = env.step(
    step_key, state, action
)
```

Listing 1: Interacting with a Jaxnasium environment. Apart from the explicit PRNG keys and environment state required by JAX, the calls mirror Gymnasium.

3.2 PyTrees of Spaces

For environment space specifications, Jaxnasium provides standard observation and action spaces in the form of `Box`, `Discrete`, and `MultiDiscrete`, preserving the same abstractions that have long been established and understood (Brockman et al., 2016). Jaxnasium only deviates from Gym(nasium) spaces for some methods, e.g. to support JIT-compatible sampling.

The bigger architectural change in Jaxnasium occurs in composite spaces. In Gymnasium, composite spaces are defined via dedicated container classes such as `gymnasium.spaces.Dict` or `gymnasium.spaces.Tuple`, which requires algorithms to understand and unpack these types of spaces within their training pipelines. Instead, Jaxnasium has no composite space classes and instead embraces JAX's PyTree abstraction: *any arbitrary PyTree whose leaves are primitive spaces is a valid composite space*. Because Python dictionaries, tuples, and lists are already registered PyTrees, users can structure complex observation and action spaces using standard Python idioms, and operations on a composite space are then simply a mapping over the tree:

```python
import jax
import jaxnasium as jym

action_space = {
    "arrow_keys": jym.spaces.Discrete(4),
    "mouse": (
        jym.spaces.Box(-1.0, 1.0, (2,)),
        jym.spaces.Discrete(2)
    ),
}
keys = jym.tree.split_key_like(key, action_space)
jax.tree.map(lambda s, k: s.sample(k), action_space, keys)
# > {"arrow_keys": 2, "mouse": (Array([0.2, -0.4]), 1)}
```

This choice has two advantages for algorithms: 1) because the observation and action spaces are PyTrees, networks can, without any special-case code, construct matching structured networks; observation leaves can produce encoders, and action leaves can produce output heads (Section 4.1). 2) we need no separate abstraction or interface for multi-agent environments. A multi-agent environment is simply an environment where spaces and returned rewards are PyTrees with one entry per agent at the top level of the PyTree. Agents may have different spaces, and these spaces may themselves be composite. Section 4.2 shows how algorithms use this structure.

4 Algorithms over PyTrees

As outlined in the introduction, reinforcement learning libraries tend to offer either research-focused, readable implementations of single- or near-single-file algorithms (Huang et al., 2022; Lu et al., 2022), or production-focused importable algorithms that handle more environment types from the same exported class (Raffin et al., 2021; Raffin, 2020; Liang et al., 2018). Jaxnasium aims to offer

both from the same code. We provide small implementations of PPO (Schulman et al., 2017), SAC (Haarnoja et al., 2018), DQN (Mnih et al., 2013), and PQN (Gallici et al., 2025), separating only networks and container objects such as replay buffers and normalization classes.

In addition, however, while retaining a fully JIT-compatible training loop for end-to-end reinforcement learning (Lu et al., 2022; Hessel et al., 2021), Jaxnasium algorithms are built as importable classes (Equinox Modules (Kidger & Garcia, 2021)) with the library being available through pip. As such, similar to the second category of libraries, Jaxnasium algorithms are usable through a friendly API similar to Stable-Baselines. Algorithms are implemented such that they can learn over various action spaces, and because their networks are separated and constructed over the PyTree of spaces (Sections 3.2 and 4.1), all these algorithms can operate over almost any type of environment.

4.1 PyTrees of Networks

Because Jaxnasium represents composite spaces as native PyTrees (Section 3.2), neural network architectures can mirror this tree structure by simply mapping over the tree structure. The default agent network of Jaxnasium consists of three parts (Figure 2). An observation network maps over the observation space and creates an encoder for each leaf. The resulting features are concatenated and passed through a shared body, by default a SimBa (Lee et al., 2025a) network. An output network then maps over the action space and creates a head for each leaf, with a distribution matching its type, e.g. categorical for discrete and Gaussian for continuous actions.

While this is how algorithms initialize themselves by default, each part of the network can be replaced and we provide additional architectures to insert into the agent networks, such as BroNet (Nauman et al., 2024), a CNN, or an MLP. However, the algorithms themselves only expect a callable module and as such, the default agent network is purely optional. For those more invested in the Flax (Heek et al., 2026) ecosystem, we provide examples in our documentation on how to use custom Flax networks within a Jaxnasium algorithm.

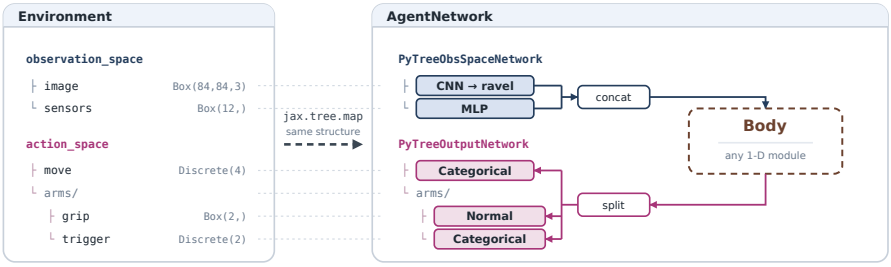


Figure 2: The default agent (policy) network. An observation network creates an encoder for each observation leaf, a shared body processes the concatenated features, and an output network creates a head for each action leaf, with a distribution matching the leaf's space. While the structure is useful for mapping spaces onto networks, it remains optional. Critic networks are created similarly, but with different output heads.

4.2 PyTrees of Agents (Multi-Agent)

This PyTree formulation for spaces extends naturally to multi-agent reinforcement learning, without any new abstractions. As described in Section 3.2, a multi-agent environment exposes its observations, actions and rewards as PyTrees whose first level indexes the agents. When an algorithm is trained on such an environment, it creates one agent per entry of this first level, each seeing a single-agent view of the environment with only its own spaces. Typical operations on a single agent are now automatically dispatched over each agent; the transformation to multi-agent execution occurs purely at the PyTree boundary. As a result, the entire algorithm and agent logic can remain written as single-agent code, coupling these two fields back together into the same programming model.

When all agents have identical shapes, their operations are vectorized with `jax.vmap`; otherwise, they run per agent. Hyperparameters can be set per agent by passing a PyTree with the agent structure, as long as they do not change the structure of the shared training loop (Listing 2).

```python
env = jym.make("MPE_simple_spread_v3")
env.action_space
# > {"agent_0": Discrete(5), "agent_1": Discrete(5), ...}

ppo = jxa.PPO(gamma={"agent_0":0.99, "agent_1":0.99, "agent_2":0.95})
agent, metrics = ppo.train(key, env)  # one PPO agent per entry
agent.get_action(key, obs)
# > {"agent_0": ..., "agent_1": ..., ...}
```

Listing 2: Training PPO on a multi-agent environment. The algorithm is used unchanged; the per-agent discount factors are optional and illustrate heterogeneous configuration.

By default, our multi-agent transformation yields independent learners: each agent optimizes its own objective with its own networks (IPPO (de Witt et al., 2020)). Centralized critic observations in the style of MAPPO are additionally supported, and we provide documentation examples on mirroring networks across agents.

5 Environment Interoperability

Similar to earlier efforts (Toledo, 2025), to allow the algorithms of Section 4 to train seamlessly on as many environments as possible, Jaxnasium provides wrappers to convert many existing environment suites to the expected interface (Section 3.1). In addition to wrappers for existing JAX suites, which allow training pipelines to be fully compiled to the accelerator, Jaxnasium provides wrappers for host-based environments, which run Gymnasium (Towers et al., 2025) and EnvPool (Weng et al., 2022) environments from within the same compiled training loop. As such, rather than requiring separate code implementations for host-bound and JAX-native setups, Jaxnasium consolidates the two regimes in a single simple research-friendly implementation.

JAX-Native Suites Jaxnasium currently registers 333 environments from eleven existing JAX suites, and exposes a familiar `make` command to initialize each environment and wrap it automatically with the appropriate wrapper. The wrappers translate each suite's signature, time steps and spaces into the Jaxnasium interface, and pass on action masks where a suite provides them. A full list of current environment suites, covering various domains, is given in Appendix A. Included algorithms can operate on over 300 of these environments directly in an end-to-end training loop (Figure 1). However, the environment registry and wrappers may also be consumed without Jaxnasium algorithms. While more wrappers are likely to be included in the future, new environments may also be created directly as Jaxnasium environments (Anonymized, 2025a;b; Anonymized2, 2025).

Host-Based Environments Many reinforcement learning environments are not written in JAX and excluding them would omit a large number of possible benchmarks. Rather than providing a separate implementation to run these host-based environments (Hessel et al., 2021; Huang et al., 2022; Toledo, 2024), Jaxnasium provides a wrapper to run (vectorized) Gymnasium (Towers et al., 2025) or EnvPool environments (Weng et al., 2022) to transition these environments through JAX callbacks. The wrapper converts the host spaces to Jaxnasium spaces and extracts the final observation of finished episodes, so that automatic resets and bootstrapping behave as they do for JAX-native environments. EnvPool environments can alternatively be stepped through EnvPool's own XLA interface. To the algorithm, the wrapped environment is an ordinary environment: rollout, update and network code are unchanged, and only the environment transition runs on the host.

Performance Although end-to-end training in JAX using native JAX environments is known to boost performance, it is reasonable to ask whether our simplified callback wrapper approach sacrifices performance compared with keeping the implementations separate. To test this, we compare PPO throughput against Stable-Baselines3 (Raffin et al., 2021), the PyTorch and JAX implementations of CleanRL (Huang et al., 2022), and the Anakin and Sebulba implementations of Stoix (Toledo, 2024), on four backends: JAX-native environments, Gymnasium, EnvPool, and EnvPool through its XLA interface. All implementations use the same hyperparameters: rollouts of 4096 transitions, 4 epochs of 4 minibatches, 64×64 networks, and observation and reward normalization. Each configuration trains for $2 \cdot 10^5$ environment steps with 8, 512 or 2048 parallel

environments. We report environment steps per second over the full training run, excluding compilation, as the mean of five runs after one discarded warm-up run, and take the geometric mean over the five classic-control environments and Catch. All runs use a single NVIDIA RTX 5090.

The results of these runs are reported in Figure 3. Jaxnasium matches or exceeds the fastest baseline in every environment backend. For the settings we tested, a separate implementation for host-based environments is therefore not necessary. We did not test multi-device setups, where asynchronous designs such as Sebulba (Toledo, 2024) may pay off, but believe this shows Jaxnasium can be a good starting point for the compute available to many research labs. The absolute throughput per library on further (more complex) environments is given in Appendix C.

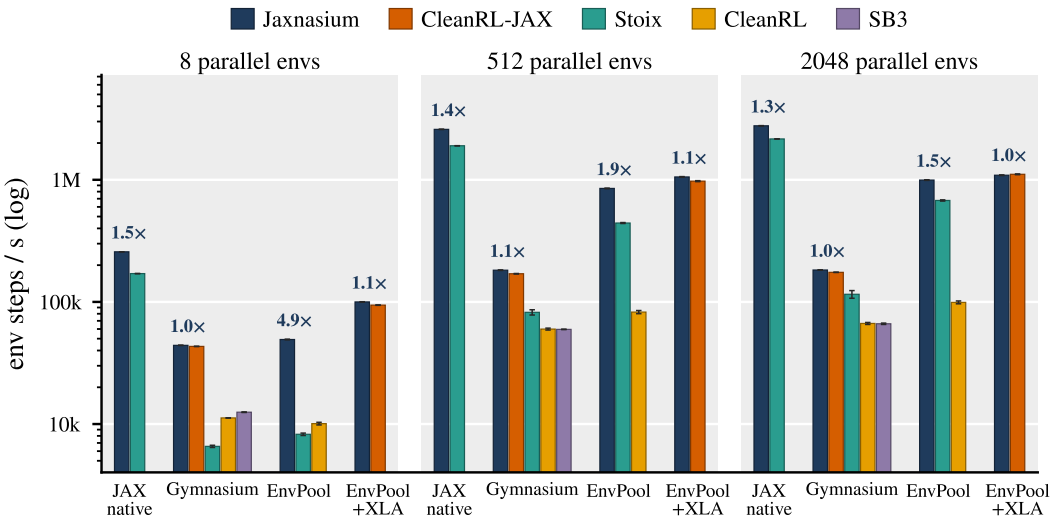


Figure 3: Average throughput of Jaxnasium and other existing established libraries on various environment backends with a small PPO network for each. Not all libraries have an implementation available to run on each environment backend, hence these bars are omitted from the graph. Jaxnasium, despite using a single small implementation to run on all types of environments, matches or exceeds the throughput of other libraries. Because not many environments have been implemented in all these backends, this graph only averages over 6 simple environments, with a cheap step function. Appendix C lists extra environments, which paint roughly the same picture.

6 Research Tooling

Beyond environments and algorithms, running reinforcement learning experiments involves recurring work: setting up a project, evaluating over many seeds, and sweeping hyperparameters across environments and algorithms. Jaxnasium includes lightweight tools for each of these. While many of these recurring tasks, particularly project setup and setting up evaluations, are increasingly generated by large language models, details such as seeding and aggregation over runs can affect experimental conclusions. We hope that providing default implementations for these tasks provides researchers with tools to create a more controlled and deterministic result.

Project Scaffolding with `create-rl-app` Inspired by the web development community, Jaxnasium ships a command-line interface, `create-rl-app`, that scaffolds new projects and copies templates into existing ones. This offers a simplified way to use Jaxnasium environment templates or algorithms as a starting point for further research.

```
uvx jaxnasium init          # new project with training script
uvx jaxnasium add ppo       # ppo.py and its networks, ready to modify
uvx jaxnasium add sweep     # template for a multi-environment sweep
```

While `create-rl-app` is integrated into Jaxnasium, it is released as a standalone package and may be included by anyone to create a CLI to share their templates, simply by defining templates in a configuration `toml` file.

Evaluation and Sweeps Reliable conclusions in reinforcement learning require many seeds and careful aggregation (Henderson et al., 2019; Agarwal et al., 2021). Jaxnasium's evaluation module provides intuitive tooling for sweeping over arguments of any function; typically a training run over various algorithms, hyperparameters, and/or environments. The module provides several standard search methods such as grid search, random search, and Sobol search, as well as evaluation metrics for reinforcement learning (Agarwal et al., 2021). An example sweep configuration is given in Appendix B.

Other Utilities The `jaxnasium.tree` module offers operations over PyTrees that JAX does not provide directly but that are common in our reinforcement learning workload, such as stacking and unstacking trees, mapping over the first level of a tree (the agent level in multi-agent settings), and splitting PRNG keys to match a tree's structure. Furthermore, environment wrappers cover vectorization, episode logging, observation and reward normalization, observation flattening and discretization of continuous action spaces. Trained agents can be saved and loaded and have normalization built in, and training functions can be compiled ahead of time to separate compilation from run time.

7 Limitations & Future Work

Currently, Jaxnasium implements four algorithms. While we aim to include more algorithms in the future, we also aim to update algorithms with newly found improvements, rather than sticking to the exact original implementation. As a result, to keep the library maintainable, Jaxnasium is unlikely to offer as many algorithms as some other libraries do (Toledo, 2024; Raffin et al., 2021). Additionally, while Jaxnasium algorithms can vectorize their training runs and shard them over multiple accelerators through JAX, its main focus remains simple setups for research-sized labs; large distributed (production-ready) reinforcement learning experiments are likely better solved with different libraries (Liang et al., 2018; Toledo, 2024; de Kock et al., 2023).

While our multi-agent transformation allows single-agent implementations to automatically operate on multi-agent environments, this is limited to individual multi-agent learning (de Witt et al., 2020). While we provide examples in our documentation for CTDE learning, complex multi-agent structures would require tweaking of the algorithms (Rashid et al., 2020).

Jaxnasium will be used as a research library. Future research will look into what engineering tricks have worked on various algorithms and how we can best operate over a wide variety of different tasks (Lee et al., 2025a;b; Hessel et al., 2018). The aim is not to keep algorithms static, but rather implement newly found improvements to provide up-to-date algorithms, with proper defaults, that can operate on a wide variety of environments. The goal is that this provides researchers from various fields with access to a set of proper baselines and starting points for their research.

8 Conclusion

We presented Jaxnasium, a library aiming to make JAX-based reinforcement learning more accessible and to broaden its use to environments not written in JAX, without code changes. Its central idea is to treat observation and action spaces as PyTrees and to let the components around an algorithm, rather than the algorithm itself, follow this PyTree structure. As a result, near-single-file implementations of various algorithms train on discrete, continuous, image and composite spaces, and even freely upgrade to the multi-agent setting, without changes to their code. These algorithm implementation files serve as readable references for researchers as well as importable algorithms for practitioners. Through wrappers, they apply to over 300 registered environments, and additionally on host-based Gymnasium and EnvPool environments without sacrificing performance compared to algorithms designed for this setting. Together with tools for scaffolding projects, multi-seed evaluation and hyperparameter sweeps, we hope Jaxnasium provides researchers from various fields with a good starting point for running reinforcement learning agents on any type of environment.

AI USE STATEMENT

In this work, we used generative AI tools for polishing the paper, drafting some paragraphs and suggest (alternative) structures for the text, generating an image, aiding in creating plots, and, most relevant for this paper, aiding in coding: particularly in the later stages of development and for writing tests. Primarily, AI was used to get a quick feeling for how feasible certain features would be to implement and to evaluate and discuss different solutions. Generated code has been carefully evaluated and great care has been put into ensuring that all AI generated code remained understandable and human maintainable.

We have not used generative AI tools for supporting qualitative and thematic data analysis and designing or providing feedback on research. Furthermore, generating synthetic data sets, helping develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, proposing or refining hypotheses, methodology or experiments, implementing methods, assisting with translation, and cleaning and reformatting datasets, are not applicable to this work.

We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 2021.

Anonymized Anonymized. Chargax: A JAX accelerated EV charging simulator. *Reinforcement Learning Journal*, 6:364–383, 2025a.

Anonymized Anonymized. Econojax: A fast & scalable economic simulation in jax. In *Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '25, pp. 1679–1687, Richland, SC, 2025b. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9798400714269.

Anonymized2 Anonymized2. AI for global climate cooperation: Modeling global climate negotiations, agreements, and long-term cooperation in RICE-n. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76332–76360. PMLR, 13–19 Jul 2025.

Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Sasha Abramowitz, Paul Duckworth, Vincent Coyette, Laurence I. Midgley, Elshadai Tegegn, Tristan Kalloniatis, Omayma Mahjoub, Matthew Macfarlane, Andries P. Smit, Nathan Grinsztajn, Raphael Boige, Cemlyn N. Waters, Mohamed A. Mimouni, Ulrich A. Mbou Sob, Ruan de Kock, Siddarth Singh, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. Jumanji: a diverse suite of scalable reinforcement learning environments in jax, 2024. URL https://arxiv.org/abs/2306.09884.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

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

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

Ruan de Kock, Omayma Mahjoub, Sasha Abramowitz, Wiem Khlifi, Callum Rhys Tilbury, Claude Formanek, Andries P. Smit, and Arnu Pretorius. Mava: a research library for distributed multi-agent reinforcement learning in jax. *arXiv preprint arXiv:2107.01460*, 2023. URL https://arxiv.org/pdf/2107.01460.pdf.

Christian Schroeder de Witt, Tarun Gupta, Denys Makoviichuk, Viktor Makoviychuk, Philip H. S. Torr, Mingfei Sun, and Shimon Whiteson. Is independent learning all you need in the starcraft multi-agent challenge?, 2020. URL https://arxiv.org/abs/2011.09533.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

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

Matteo Gallici, Mattie Fellows, Benjamin Ellis, Bartomeu Pou, Ivan Masmitja, Jakob Nicolaus Foerster, and Mario Martin. Simplifying deep temporal difference learning, 2025. URL https://arxiv.org/abs/2407.04811.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL https://arxiv.org/abs/1801.01290.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2026. URL http://github.com/google/flax.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2019. URL https://arxiv.org/abs/1709.06560.

Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart, 2018. URL https://arxiv.org/abs/1809.04474.

Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning, 2021. URL https://arxiv.org/abs/2104.06272.

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. URL http://jmlr.org/papers/v23/21-1342.html.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari, 2024. URL https://arxiv.org/abs/1903.00374.

Patrick Kidger and Cristian Garcia. Equinox: neural networks in JAX via callable PyTrees and filtered transformations. *Differentiable Programming workshop at Neural Information Processing Systems 2021*, 2021.

Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-accelerated parallel game simulators for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 45716–45743, 2023.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

Tian Lan, Sunil Srinivasa, Huan Wang, and Stephan Zheng. Warpdrive: Fast end-to-end deep multi-agent reinforcement learning on a gpu. *Journal of Machine Learning Research*, 23(316):1–6, 2022. URL http://jmlr.org/papers/v23/22-0185.html.

Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022. URL http://github.com/RobertTLange/gymnax.

Hojoon Lee, Dongyoon Hwang, Donghu Kim, Hyunseung Kim, Jun Jet Tai, Kaushik Subramanian, Peter R. Wurman, Jaegul Choo, Peter Stone, and Takuma Seno. Simba: Simplicity bias for scaling up parameters in deep reinforcement learning, 2025a. URL https://arxiv.org/abs/2410.09754.

Hojoon Lee, Youngdo Lee, Takuma Seno, Donghu Kim, Peter Stone, and Jaegul Choo. Hyperspherical normalization for scalable deep reinforcement learning, 2025b. URL https://arxiv.org/abs/2502.15280.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018. URL https://arxiv.org/pdf/1712.09381.

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.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2024.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

Steven Morad, Ryan Kortvelesy, Matteo Bettini, Stephan Liwicki, and Amanda Prorok. POPGym: Benchmarking partially observable reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=chDrutUTs0K.

Alistair Muldal, Yotam Doron, John Aslanides, Tim Harley, Tom Ward, and Siqi Liu. dm_env: A python interface for reinforcement learning environments, 2019. URL http://github.com/deepmind/dm_env.

Michal Nauman, Mateusz Ostaszewski, Krzysztof Jankowski, Piotr Miłoś, and Marek Cygan. Bigger, regularized, optimistic: scaling for compute and sample-efficient continuous control, 2024. URL https://arxiv.org/abs/2405.16158.

Alexander Nikulin, Vladislav Kurenkov, Ilya Zisman, Viacheslav Sinii, Artem Agarkov, and Sergey Kolesnikov. XLand-minigrid: Scalable meta-reinforcement learning environments in JAX. In *Intrinsically-Motivated and Open-Ended Learning Workshop, NeurIPS2023*, 2023. URL https://openreview.net/forum?id=xALDC4aHGz.

Eduardo Pignatelli, Jarek Liesen, Robert Lange, Chris Lu, Pablo Samuel Castro, and Laura Toni. Navix: Scaling minigrid environments with jax. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-4437. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c108bda95007522456be2e4b0794b836-Paper-Datasets_and_Benchmarks_Track.pdf.

Waris Radji, Thomas Michel, and Hector Piteau. Octax: Accelerated chip-8 arcade environments for reinforcement learning in jax, 2025.

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

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL https://jmlr.org/papers/v22/20-1364.html.

Tabish Rashid, Gregory Farquhar, Bei Peng, and Shimon Whiteson. Weighted qmix: Expanding monotonic value function factorisation for deep multi-agent reinforcement learning, 2020. URL https://arxiv.org/abs/2006.10800.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garðar Ingvarsson, Timon Willi, Ravi Hammond, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Tjarko Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktäschel, Chris Lu, and Jakob Nicolaus Foerster. Jaxmarl: Multi-agent rl environments and algorithms in jax. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

Tim Scarfe, Jakob Foerster, and Chris Lu. Imagenet moment for reinforcement learning?, feb 2025. URL https://www.dropbox.com/scl/fi/yqjszhntfr00bhjh6t565/JAKOB.pdf?rlkey=scvny4bnwj8th42fjv8zsfu2y&e=1&dl=0. Machine Learning Streettalk podcast episode.

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

Brennan Shacklett, Luc Guy Rosenzweig, Zhiqiang Xie, Bidipta Sarkar, Andrew Szot, Erik Wijmans, Vladlen Koltun, Dhruv Batra, and Kayvon Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Trans. Graph.*, 42(4), 2023.

Joseph Suarez. Pufferlib 2.0: Reinforcement learning at 1m steps/s. In *Reinforcement Learning Conference*, 2025. URL https://openreview.net/forum?id=qRyteMTgn0.

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.

Edan Toledo. Stoix: Distributed single-agent reinforcement learning end-to-end in jax, April 2024. URL https://github.com/EdanToledo/Stoix.

Edan Toledo. Stoa: A jax-native interface for reinforcement learning environments, 2025. URL https://github.com/EdanToledo/Stoa.

Tristan Tomilin, Mourad Boustani, Mickey Beurskens, and Thiago D. Simão. Crax: Fast safe reinforcement learning benchmarking. *arXiv preprint arXiv:2606.20376*, 2026.

Mark Towers, Ariel Kwiatkowski, John Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Kallinteris Andreas, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, J Terry, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-4916. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/d7ff1795e8527f6443371c3933bdb52b-Paper-Datasets_and_Benchmarks_Track.pdf.

Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng Yan. EnvPool: A highly parallel reinforcement learning environment execution engine. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 22409–22421. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/8caaf08e49ddbad6694fae067442ee21-Paper-Datasets_and_Benchmarks.pdf.

Kevin Zakka, Baruch Tabanpour, Qiayuan Liao, Mustafa Haiderbhai, Samuel Holt, Jing Yuan Luo, Arthur Allshire, Erik Frey, Koushil Sreenath, Lueder A. Kahrs, Carlo Sferrazza, Yuval Tassa, and Pieter Abbeel. Mujoco playground: An open-source framework for gpu-accelerated robot learning and sim-to-real transfer., 2025. URL https://github.com/google-deepmind/mujoco_playground.

A Included environments

Table 1: Registered environments per suite currently available in Jaxnasium. For convenience, Jaxnasium bundles the 5 classic-control environments; other libraries have to be installed and can subsequently be initialized with `jym.make(...)`. More environments will likely be added as they become available.

Suite	Domain	Envs
Brax (Freeman et al., 2021)	Continuous control	10
Craftax (Matthews et al., 2024)	Open-ended survival	4
Gymnax (Lange, 2022)	Classic control, bsuite, MinAtar	24
JaxMARL (Rutherford et al., 2024)	Multi-agent coordination	29
Jumanji (Bonnet et al., 2024)	Combinatorial problems	26
Navix (Pignatelli et al., 2025)	Grid worlds	41
Octax (Radji et al., 2025)	Arcade games (CHIP-8)	40
Pgx (Koyamada et al., 2023)	Board games, MinAtar	20
Playground (Zakka et al., 2025)	Robotics	50
PopJym (Morad et al., 2023)	Partial observability	40
XLand-MiniGrid (Nikulin et al., 2023)	Meta-RL grid worlds	44
Jaxnasium	Classic control	5
Total		333

B Example of a sweep configuration

Sweeps are composed from search stages: grid search, one-at-a-time search, random search and quasi-random Sobol search. Stages can be nested, for instance a grid over environments with a random search over learning rates inside it, and can branch per algorithm, so that each algorithm is searched over its own hyperparameters (Listing 3). A sweep is a list of independent jobs that can be run in a loop; or, more typically, indexed by a Slurm job array on a cluster.

```python
from jaxnasium.eval import AlgorithmEvaluation, GridSearch, SobolSearch, Sweep

evaluation = AlgorithmEvaluation(jax.random.split(key, 10))  # 10 seeds
sweep = Sweep(
    evaluation,
    GridSearch({"env": ["CartPole-v1", "Acrobot-v1"]}),
    GridSearch({"algorithm": {
        "PPO": [{"algorithm": PPO()},
                SobolSearch({"clip_coef": (0.1, 0.3)}, 32, seed=key)],
        "SAC": [{"algorithm": SAC()},
                SobolSearch({"tau": (1e-3, 5e-2)}, 32, seed=key)],
    }}),
)
result = sweep[job_index]()  # e.g. job_index = $SLURM_ARRAY_TASK_ID
```

Listing 3: A sweep over two environments and two algorithms, each with its own Sobol search, evaluated over ten seeds per configuration. `AlgorithmEvaluation` is a pre-implemented function for running a standard train+evaluate loop. However, any function can be defined and swept over.

C EXTRA ENVIRONMENTS FOR BACKEND PERFORMANCE COMPARISON

The figures below provide extra benchmarking runs over various backends, comparable to Figure 3. Figure 3 was an average over only 6 simple environments (the 5 classic-control environments and Catch), whereas the environments below are more complex but do not have an implementation ready in each of the three backends (JAX-native, Gymnasium, EnvPool). Overall, these results demonstrate a similar picture: running the JAX-native version is typically fastest, but when the environment is not available in JAX, using our wrapper does not lead to diminished performance compared to using a JAX agent that is not built for end-to-end training in JAX.

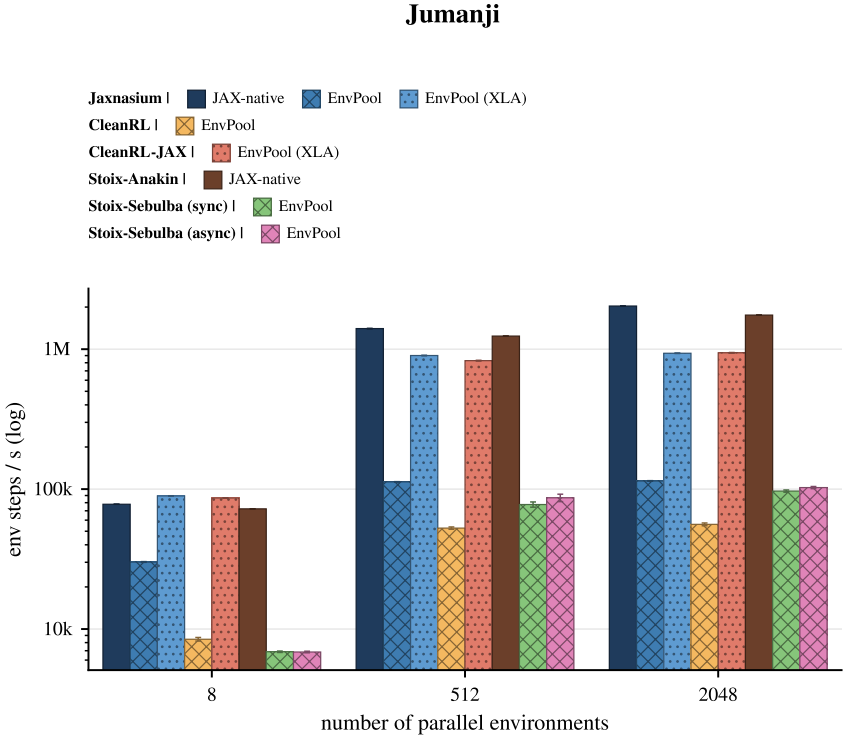


Figure 4: Throughput tested for various backends and different libraries on the Jumanji suite. Averaged over 5 runs. Standard deviation is plotted, but is negligible.

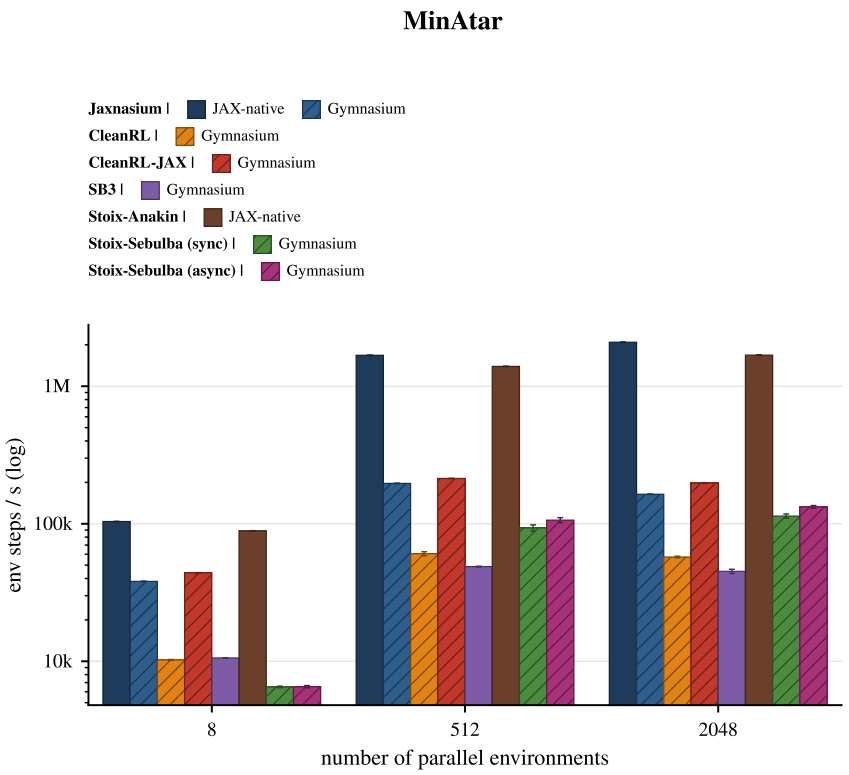


Figure 5: Throughput tested for various backends and different libraries on the MinAtar suite. Averaged over 5 runs. Standard deviation is plotted, but is negligible.

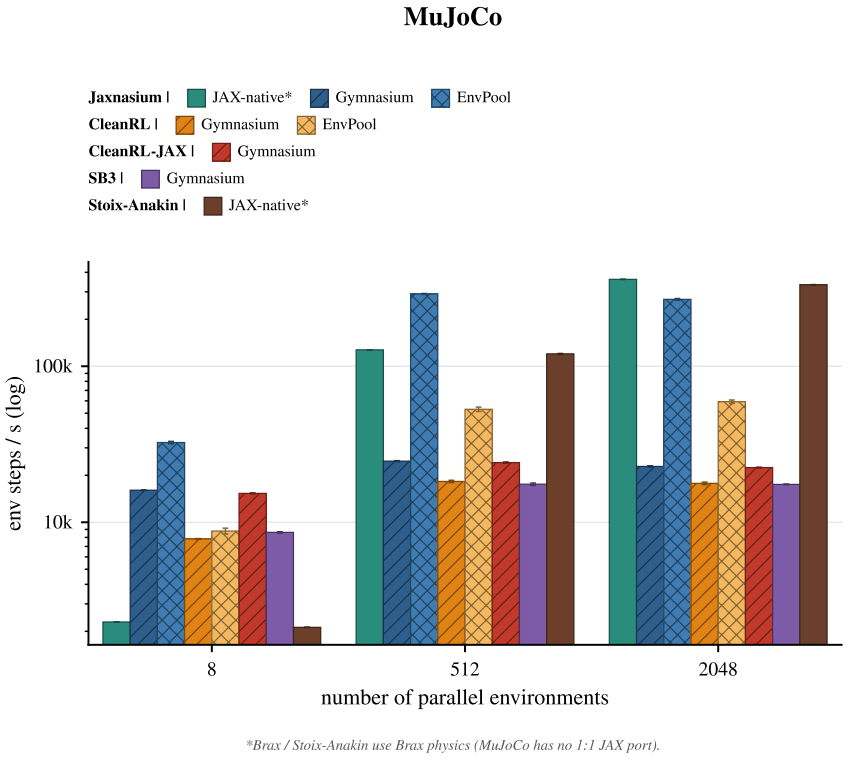


Figure 6: Throughput tested for various backends and different libraries on the MuJoCo and Brax suites. Averaged over 5 runs. Standard deviation is plotted, but is negligible.