The Journal of Open Source Software

BordAX: A High-Performance JAX Framework for Programmatic Reinforcement Learning

Roman Kniazev [1] and **Nathanaël Fijalkow** [1]

1 CNRS, LaBRI, University of Bordeaux, France

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor:

Submitted: 13 February 2026
Published: unpublished

Summary

BordAX is a Python framework for reinforcement learning (RL) built on JAX (Bradbury et al., 2018). It provides a modular, fully JIT-compilable training pipeline that supports multiple policy representations, including standard multilayer perceptrons (MLPs), differentiable decision trees (DTSemNet), and boolean function networks (HyperBool). BordAX currently implements Proximal Policy Optimization (PPO) (Schulman et al., 2017) and Deep Q-Networks (DQN) (Mnih et al., 2015), and is designed so that additional algorithms can be composed from interchangeable collector, batch-builder, and updater components. The framework supports both Gymnax (Lange, 2022) environments, which enable full JIT compilation of the training loop via `jax.lax.scan`, and standard Gymnasium (Brockman et al., 2016) environments. On a CartPole-v1 benchmark with identical hyperparameters, BordAX with Gymnax achieves approximately 3.2 times higher throughput than Stable-Baselines3 (Raffin et al., 2021).

Statement of need

Programmatic Reinforcement Learning (PRL) is a subfield of RL concerned with learning structured, interpretable policies such as decision trees, boolean expressions, or symbolic programs, rather than opaque neural networks (Landajuela et al., 2021; Verma et al., 2018). Researchers in PRL often need to train and evaluate several types of policy representations under the same algorithmic conditions. Existing RL frameworks are typically built around a fixed neural network policy and do not provide straightforward mechanisms for swapping in non-standard architectures like differentiable decision trees (Silva et al., 2020) or boolean networks.

Stable-Baselines3 (Raffin et al., 2021) is a widely used PyTorch-based RL library, but its reliance on Python-level iteration and PyTorch's eager execution model limits throughput. PureJaxRL (Lu, 2023) demonstrated that implementing the entire RL training loop in JAX and compiling it end-to-end via XLA yields large speedups, but it is structured as a collection of standalone scripts rather than a reusable library with modular components. Brax (Freeman et al., 2021) provides JIT-compiled environments and training but is focused on continuous control in simulated physics rather than serving as a general-purpose RL framework with pluggable policy types.

BordAX addresses these gaps by providing:

- A modular architecture in which the algorithm is defined as the composition of a data collector, a batch builder, and a parameter updater, making it straightforward to implement new algorithms or modify existing ones.
- First-class support for non-neural policy representations (differentiable decision trees and boolean function networks) alongside standard MLPs, enabling controlled comparisons across policy types.

- Full JIT compilation of the training loop when using Gymnax environments, and JIT compilation of the update step when using Gymnasium environments.
- A clean functional design compatible with JAX transformations (`jax.jit`, `jax.vmap`, `jax.grad`).

The target audience is researchers working on programmatic or interpretable reinforcement learning who need a fast, modular framework for experimenting with different policy representations and algorithms.

State of the field

Several frameworks exist for reinforcement learning in Python.

Stable-Baselines3 (Raffin et al., 2021) is among the most widely used. It provides reliable implementations of standard algorithms (PPO, DQN, SAC, TD3, A2C) built on PyTorch. Its design prioritizes usability and correctness, but training throughput is limited by Python-level environment stepping and PyTorch's eager execution.

PureJaxRL (Lu, 2023) showed that writing the entire training loop in JAX and compiling it with XLA can yield order-of-magnitude speedups over PyTorch-based frameworks. However, PureJaxRL is organized as individual training scripts rather than a library with reusable, composable components, and it does not support non-neural policy architectures.

Brax (Freeman et al., 2021) provides JIT-compiled physics simulation environments and training utilities for continuous control. It is tightly coupled to its own environment interface and is not designed as a general-purpose RL framework with interchangeable policy types.

Gymnax (Lange, 2022) offers JAX-based reimplementations of classic RL environments, enabling `jax.vmap` vectorization and `jax.lax.scan` loop compilation. It provides environments but not training infrastructure.

BordAX builds on Gymnax and JAX to provide a complete, modular training framework. Unlike PureJaxRL, BordAX factors the training pipeline into composable components (collectors, batch builders, updaters, loss functions) defined through abstract interfaces, allowing new algorithms to be added by implementing a small number of well-defined functions. Unlike Stable-Baselines3, BordAX achieves significantly higher throughput through JIT compilation. Unlike any of the above, BordAX includes built-in support for non-neural policy representations (differentiable decision trees and boolean function networks), which are relevant for research on interpretable RL (Silva et al., 2020; Topin et al., 2021; Verma et al., 2018).

Software design

BordAX is organized into five modules:

- **Agents** define the policy and value function interfaces. The abstract `Agent` class specifies `init`, `policy`, `action`, and `value` methods. Concrete implementations include `MLPPolicyValue` for discrete action spaces and `MLPPolicyValueContinuous` for continuous action spaces. Each agent accepts a `policy_architecture` parameter that selects among MLP, DTSemNet (differentiable decision tree), or HyperBool (boolean function network) policy heads, all implemented as Flax (Heek et al., 2020) modules. A `DQNAgent` provides Q-value estimation for off-policy learning.

- **Algorithms** are defined as named tuples of three components: a `Collector`, a `BatchBuilder`, and an `Updater`. Factory functions `ppo_algo()` and `dqn_algo()` wire these components together with appropriate hyperparameters and loss functions. This decomposition means that implementing a new algorithm requires writing at most three small, focused components. The `Algorithm.train_step` method orchestrates a single training iteration: collect data, build batches, and update parameters.

- **Data** provides the data collection and batching pipeline. `OnPolicyCollector` gathers rollouts using `jax.lax.scan` for JIT-compiled environments or a Python loop for Gymnasium environments. `EpsGreedyCollector` implements epsilon-greedy exploration for DQN. Batch builders are composable: `FullBufferBatch`, `MiniBatch`, and `NormalizeAdvantagesTargets` can be chained via `ComposedBatchBuilder`. A `ReplayBuffer` provides ring-buffer storage for off-policy methods.
- **Environments** provide an `EnvAdapter` abstraction over Gymnax and Gymnasium. `EnvGymnaxAdapter` wraps Gymnax environments with `jax.vmap` for vectorized execution and `jax.jit` for compilation. `EnvGymnasiumAdapter` wraps standard Gymnasium environments, handling the JAX-to-NumPy boundary. Both expose the same `reset`, `step`, `action_space`, and `obs_space` interface.
- **Training** provides the `Trainer` class, which manages the training loop including initialization, warmup (for off-policy methods), epoch execution, evaluation, logging (with optional Weights & Biases integration), and checkpointing (via Orbax). When the environment is JIT-compilable and the algorithm is on-policy, the entire `train_step` is JIT-compiled. Otherwise, only the update step is JIT-compiled.

A key design decision is the use of pure functional state passing throughout the framework. All state (environment state, training state, optimizer state, replay buffer) is passed explicitly, with no hidden mutation. This makes the code compatible with JAX's functional transformation model and ensures that the training loop can be compiled end-to-end when conditions allow.

Loss functions are implemented as callable classes inheriting from `LossFn`. PPO uses a composite loss comprising `SurrogateLoss` (clipped policy gradient), `ValueLoss`, and `EntropyLoss`, each accepting schedule functions for their coefficients. DQN uses `DQNLoss` with a configurable base loss (squared error or Huber loss). Optimization is handled by Optax (Hessel et al., 2020), and probability distributions are provided by Distrax (Budden et al., 2021).

Research impact statement

BordAX was developed to support research on programmatic reinforcement learning. Its modular design enables systematic comparison of different policy representations (neural networks, decision trees, boolean functions) under identical algorithmic and environmental conditions. The framework is publicly available on GitHub under the MIT license and includes automated tests with 77% code coverage.

On a CartPole-v1 benchmark using PPO with identical hyperparameters across five random seeds and 51,200 timesteps, BordAX with Gymnax completes training in 4.26 seconds (12,027 steps/s), compared to 13.79 seconds (3,714 steps/s) for Stable-Baselines3, a 3.2x throughput improvement. With Gymnasium environments, BordAX achieves 8,021 steps/s (2.2x speedup), demonstrating that JAX-based optimization of the update step alone provides a meaningful improvement. The benchmark script is included in the repository for reproducibility.

AI usage disclosure

GitHub Copilot was used to assist with code generation, documentation drafting, and paper authoring during the development of this project. All AI-generated content was reviewed, edited, and validated by the human authors, who made all core design and architectural decisions.

JOSS
The Journal of Open Source Software

Acknowledgements

The authors acknowledge the developers of JAX, Flax, Optax, Gymnax, Distrax, and the broader JAX ecosystem, on which BordAX is built.

References

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/jax-ml/jax

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *arXiv Preprint arXiv:1606.01540*. https://arxiv.org/abs/1606.01540

Budden, D., Hessel, M., Quan, J., Kapturowski, S., Baumli, K., Bhupatiraju, S., Guy, A., & King, M. (2021). *Distrax: Probability distributions in JAX*. http://github.com/google-deepmind/distrax

Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., & Bachem, O. (2021). *Brax – a differentiable physics engine for large scale rigid body simulation*. https://doi.org/10.48550/arXiv.2106.13281

Heek, J., Levskaya, A., Oliver, A., Ritter, M., Rondepierre, B., Steiner, A., & Zee, M. van. (2020). *Flax: A neural network library and ecosystem for JAX* (Version 0.5.0). http://github.com/google/flax

Hessel, M., Budden, D., Viola, F., Rosca, M., Sezener, E., & Hennigan, T. (2020). *Optax: Composable gradient transformation and optimisation, in JAX* (Version 0.0.6). http://github.com/deepmind/optax

Landajuela, M., Petersen, B. K., Kim, S., Santiago, C. P., Glatt, R., Mundhenk, N., Pettit, J. F., & Faissol, D. (2021). Discovering symbolic policies with deep reinforcement learning. *Proceedings of the 38th International Conference on Machine Learning*, 5979–5989.

Lange, R. T. (2022). *gymnax: A JAX-based reinforcement learning environment library* (Version 0.0.4). http://github.com/RobertTLange/gymnax

Lu, C. (2023). *PureJaxRL: High-performance reinforcement learning in pure JAX*. https://github.com/luchris429/purejaxrl

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. https://doi.org/10.1038/nature14236

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, *22*(268), 1–8. http://jmlr.org/papers/v22/20-1364.html

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv Preprint arXiv:1707.06347*. https://arxiv.org/abs/1707.06347

Silva, A., Gombolay, M., Killian, T., Jimenez, I., & Son, S.-H. (2020). Optimization methods for interpretable differentiable decision trees applied to reinforcement learning. *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 1855–1865.

Topin, N., Milani, S., Fang, F., & Veloso, M. (2021). Iterative bounding MDPs: Learning interpretable policies via non-interpretable methods. *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(11), 9923–9931.

Verma, A., Murali, V., Singh, R., Kohli, P., & Chaudhuri, S. (2018). Programmatically interpretable reinforcement learning. *Proceedings of the 35th International Conference on Machine Learning*, 5045–5054.