NAVIX: Scaling Minigrid Environments with JAX

Eduardo Pignatelli
University College London
e.pignatelli@ucl.ac.uk

Jarek Liesen
BCCN Berlin
jarek@bccn-berlin.de

Robert Tjarko Lange
Technical University Berlin
robert.t.lange@tu-berlin.de

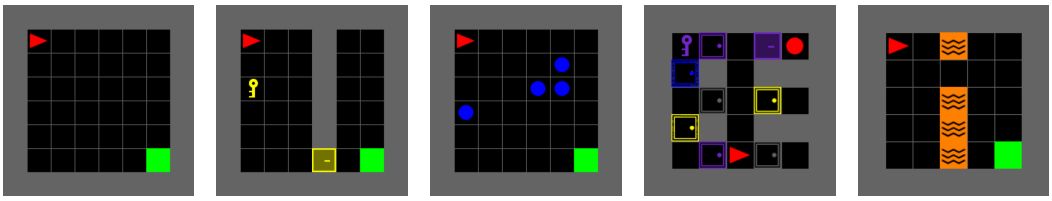
Chris Lu
University of Oxford
christopher.lu@eng.ox.ac.uk

Pablo Samuel Castro
Google DeepMind
Mila, Université de Montréal
psc@google.com

Laura Toni
University College London
l.toni@ucl.ac.uk

Abstract

As Deep Reinforcement Learning (Deep RL) research moves towards solving large-scale worlds, efficient environment simulations become crucial for rapid experimentation. However, most existing environments struggle to scale to high throughput, setting back meaningful progress. Interactions are typically computed on the CPU, limiting training speed and throughput, due to slower computation and communication overhead when distributing the task across multiple machines. Ultimately, Deep RL training is CPU-bound, and developing batched, fast, and scalable environments has become a frontier for progress. Among the most used Reinforcement Learning (RL) environments, Minigrid is at the foundation of several studies on exploration, curriculum learning, representation learning, diversity, meta-learning, credit assignment, and language-conditioned RL, and still suffers from the limitations described above. In this work, we introduce NAVIX[1], a re-implementation of Minigrid in JAX. NAVIX achieves over 160 000× speed improvements in batch mode, supporting up to 2048 agents in parallel on a single Nvidia A100 80 GB. This reduces experiment times from one week to 15 minutes, promoting faster design iterations and more scalable RL model development.



(a) 128.98× (b) 26.47× (c) 41.17× (d) 19.72× (e) 45.71×

Figure 1: Speedups for five of the NAVIX environments with respect to their Minigrid equivalent, using the protocol in Section 4.1. (a) `Empty-8x8-v0`, (b) `DoorKey-8x8-v0`, (c) `Dynamic-Obstacles-8x8-v0`, (d) `KeyCorridorS3R3-v0`, (e) `LavaGapS7-v0`.

[1] https://github.com/epignatelli/navix

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Track on Datasets and Benchmarks.

1 Introduction

Deep RL is notoriously sample inefficient (Kaiser et al., 2019; Wang et al., 2021; Johnson et al., 2016; Küttler et al., 2020). Depending on the complexity of the environment dynamics, the observation space, and the action space, agents often require between 10^7 to 10^9 interactions or even more for training up to a good enough policy. Therefore, as Deep RL moves towards tackling more complex environments, leveraging efficient environment implementations is an essential ingredient of rapid experimentation and fast design iterations.

However, while the efficiency and scalability of solutions for *agents* have improved massively in recent years (Schulman et al., 2017; Espeholt et al., 2018; Kapturowski et al., 2018), especially due to the scalability of the current deep learning frameworks (Abadi et al., 2016; Paszke et al., 2019; Ansel et al., 2024; Bradbury et al., 2018; Sabne, 2020), environments have not kept pace. They are mostly based on CPU, cannot adapt to different types of devices, and scaling often requires complex distributed systems, introducing design complexity and communication overhead. Overall, deep RL experiments are CPU-bound, limiting both speed and throughput of RL training.

Recently, a set of GPU-based environments (Freeman et al., 2021; Lange, 2022; Weng et al., 2022; Koyamada et al., 2023; Rutherford et al., 2023a; Nikulin et al., 2023; Matthews et al., 2024; Bonnet et al., 2024; Lu et al., 2023; Liesen et al., 2024b) and frameworks (Hessel et al., 2021; Lu et al., 2022; Liesen et al., 2024a; Toledo, 2024; Nishimori, 2024; Jiang et al., 2023) has sparked raising interest, proposing JAX-based, batched implementations of common RL environments that can significantly increase the speed and throughput of canonical Deep RL algorithms. This enables large-scale parallelism, allowing the training of thousands of agents in parallel on a single accelerator, significantly outperforming traditional CPU-based environments, and fostering meta-RL applications.

In this work, we build on this trend and focus on the Minigrid suite of environments (Chevalier-Boisvert et al., 2024), due to its central role in the Deep RL literature. MiniGrid is fundamental to many studies. For instance, Zhang et al. (2020); Zha et al. (2021); Mavor-Parker et al. (2022) used it to test new exploration strategies; Jiang et al. (2021) for curriculum learning; Zhao et al. (2021) for planning; Paischer et al. (2022) for representation learning, Flet-Berliac et al. (2021); Guan et al. (2022) for diversity. Parisi et al. (2021) employed Minigrid to design meta and transfer learning strategies, and Mu et al. (2022) to study language grounding.

However, despite its ubiquity in the Deep RL literature, Minigrid faces the limitations of CPU-bound environments. We bridge this gap and propose NAVIX, a reimplementation of Minigrid in JAX that leverages JAX's intermediate language representation to migrate the computation to different accelerators, such as GPUs, and TPUs.

Our results show that NAVIX is over $44\times$ times faster than the original Minigrid implementation, in common Deep RL settings (Section 4.1), 5.4 times faster when running a single environment (Appendix I), and increases the throughput by over $10^6\times$ (Section 4.2). Composed together, they produce speed-ups of over $160\,000\times$ (Section 4.2), turning 1-week experiments into 15 minutes ones. We show the scaling ability of NAVIX by training over 2048 PPO agents in parallel (Section 4.2), each using their own subset of environments, all on a single Nvidia A100 80 GB.

The main contributions of this work are the following:

1. A fully JAX-based implementation of environment configurations that reproduces exactly the original Minigrid Markov Decision Processes (MDPs) and Partially-observable MDPs (POMDPs).
2. A description of the design philosophy, the design pattern and principles, the organisation, and the components of NAVIX, which, together with the online documentation, form an instruction manual to use and extend NAVIX.
3. A set of RL algorithm baselines for all environments in Section 4.3.

2 Related work

JAX-based environments. The number of JAX-based reimplementations of common environments is in a bullish trend. Freeman et al. (2021) provide a fully differentiable physics engine for robotics, including MJX, a reimplementation of MujoCo (Todorov et al., 2012). Lange (2022) reimplements

several gym (Brockman et al., 2016) environments, including classic control, Bsuite (Osband et al., 2020), and MinAtar (Young & Tian, 2019),

Koyamada et al. (2023) reimplement many board games, including backgammon, chess, shogi, and go. Lu et al. (2023) provides JAX implementations of POPGym (Morad et al., 2023), which contains partially-observed RL environments. Matthews et al. (2024) reimplement Crafter (Hafner, 2021). Bonnet et al. (2024) provides JAX implementations of combinatorial problems frequently encountered in industry, including bin packing, capacitated vehicle routing problem, PacMan, Sokoban, Snake, 2048, Sudoku, and many others. Rutherford et al. (2023b) reimplement a set of multi-agent environments, including a MiniGrid-inspired implementation of the Overcooked benchmark.

Yet, none of these works proposes a reimplementation of Minigrid. Weng et al. (2022) is the only one providing a single environment of the suite, *Empty*, but it is only one of the many, most commonly used environments of the suite, and arguably the simplest one.

Batched MiniGrid−like environments. Two works stand out for they aim to partially reimplement MiniGrid. Jiang et al. (2023) present `AMaze`, a fully batched implementation of a partially observable maze environment, with MiniGrid−like sprites and observations. However, like Weng et al. (2022), the work does not reimplement the full Minigrid suite. Nikulin et al. (2023) proposes `XLand-Minigrid`, a suite of grid-world environments for meta RL. Like (Jiang et al., 2023), `XLand-Minigrid` reproduces Minigrid-like observations but focuses on designing a set of composable rules that can be used to generate a wide range of environments, rather than mirroring the original Minigrid suite while reimplementing it in JAX.

To conclude, Minigrid is a fundamental tool for Deep RL experiments, at the base of a high number of studies, as we highlighted in Section 1. It is easy to use, easy to extend, and provides a wide range of environments of scalable complexity that are easy to inspect for a clearer understanding of an algorithm dynamics, pitfalls, and strengths.

Nevertheless, none of the works above provides a full, batched reimplementation of Minigrid in JAX that mirrors the original suite in terms of environments, observations, state transitions, and rewards. Instead, we propose a full JAX-based reimplementation of the Minigrid suite with identical semantics for observations, actions, rewards, and terminations.

3 NAVIX: design philosophy and principles

In this section we describe:

- ***(i)*** the design philosophy and pattern of NAVIX in Section 3.1, and
- ***(ii)*** the design principles at its foundation in Sections 3.2.1 and 3.2.2.

In particular, in Section 3.2.2, we highlight *why* a JAX port of Minigrid is not trivial. Among others, the obstacles to transform a stateful program, where a function is allowed to change elements that are not an input of the function, into a stateless one, where the outputs of functions depend solely on the inputs; and the restrictions in the use of `for` loops and control flow primitives, such as `if` statements.[2]

3.1 Design pattern

NAVIX is broadly inspired by the ECSM, a design pattern widely used in video game development. In an ECSM, entities – the *objects* on the grid in our case – are composed of components – the *properties* of the object. Each property holds data about the entity, which can then be used to process the game state. For example, an entity `Player` is composed of components `Positionable`, `Holder`, `Directional`, each of which injects properties into the entity: the `Positionable` component injects the `Position` property, holding the coordinates of the entity (e.g., a player, a door, a key) on the grid, the `Holder` component injects the `Pocket` property, holding the id of the entity that the agent holds, and so on. A full list of components and their properties is provided in Table 1. This compositional layout allows to easily generate the wide range of combinations of tasks that Minigrid offers, and to easily extend the suite with new environments.

[2]See `https://jax.readthedocs.io/en/latest/notebooks/Common_Gotchas_in_JAX.html`.

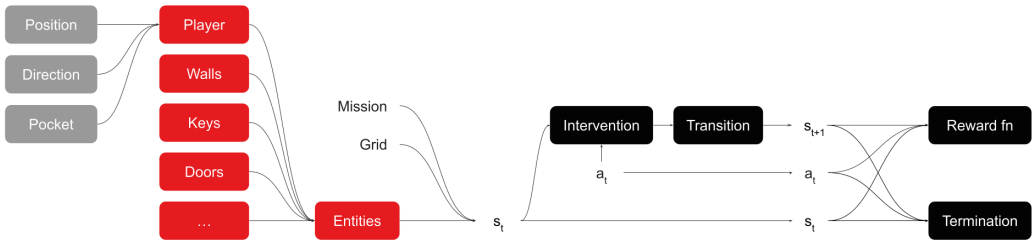


Figure 2: Information flow of the Entity-Component-System Model (ECSM) in NAVIX. Entities (*Player, Walls, Keys, Doors, ...*) are composed of components (*Position, Direction, Pocket*), which hold the data of the entity. Systems (*Intervention, Transition, Rewards, Terminations*) are functions that operate on the collective state of all entities and components.

Component	Property	Shape	Description
`Positionable`	`Position`	`f32[2]`	*Coordinates of the entity on the grid.*
`Directional`	`Direction`	`i32[]`	*Direction of the entity.*
`HasColour`	`Colour`	`u8[]`	*Colour of the entity.*
`Stochastic`	`Probability`	`f32[]`	*Probability that the entity emits an event.*
`Openable`	`State`	`bool[]`	*State of the entity, e.g., open or closed.*
`Pickable`	`Id`	`i32[]`	*Id of the entity that the agent can pick up.*
`HasTag`	`Tag`	`i32[]`	*Categorical value identifying the entity class.*
`HasSprite`	`Sprite`	`u8[32x32x3]`	*Sprite of the entity in RGB format.*
`Holder`	`Pocket`	`i32[]`	*Id of the entity that the agent holds.*

Table 1: List of **Components** in NAVIX. Each component provides a property (or a set of). These properties hold the data that can be accessed and manipulated by the systems (see Table 3) to provide observations, rewards, and state transitions.

Entity	Components	Description
`Wall`	`[HasColour]`	*An entity that blocks the agent's movement.*
`Player`	`[Directional, Holder]`	*An entity that can interact with the environment.*
`Goal`	`[HasColour, Stochastic]`	*An entity that the agent can to reach to receive a reward.*
`Key`	`[Pickable, HasColour]`	*An entity that can be picked up. Can open doors.*
`Door`	`[Openable, HasColour]`	*An entity that can be opened and closed by the agent.*
`Lava`	`[]`	*An entity that the agent has to avoid.*
`Ball`	`[HasColour, Stochastic]`	*An entity that the agent can push.*
`Box`	`[HasColour, Holder]`	*An entity that the agent can push.*

Table 2: List of **Entities** in NAVIX, together with the components that characterise them. By default, all entities already possess `Positionable`, `HasTah`, and `HasSprite` components, in addition to those reported in the table.

System	Function	Description
`Intervention`	$I : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$	*Updates the state according to the agent's actions.*
`Transition`	$P : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$	*Updates the state according to the MDP dynamics.*
`Observation`	$O : \mathcal{S} \to \mathcal{O}$	*The observation kernel;*
`Reward`	$R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$	*The Markovian reward function.*
`Termination`	$\gamma : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{B}$	*The termination function.*

Table 3: List of **Systems** in NAVIX. A state $s \in \mathcal{S}$ is a tuple containing: the set of entities, the static grid layout, and the mission of the agent.

Entities are then processed by *systems*, which are functions that operate on the collective state of all entities and components. For example, the *decision* system may update the state of the entities according to the actions taken by a player. The *transition* system may update the state of the entities according to the MDP state transitions. The *observation* system generates the observations that the agents receive, and the *reward* system computes the rewards that the agents receive, and so on. We provide a full list of implemented systems in Appendix A.

To develop a better intuition of what these elements are and how they interact, Figure 2 shows the information flow of the ECSM in NAVIX.

3.2 Design principles

On this background, two principles are at the foundation of NAVIX, and the key aspects that characterise it:

- ***(i)*** NAVIX aims to exactly match the semantics of Minigrid (Section 3.2.1) with identical observations, actions, rewards, and terminations;
- ***(ii)*** every environment is fully jittabile, i.e., that can be compiled into more efficient instructions using JAX, and differentiable (Section 3.2.2), to exploit the full set of features that JAX offers.

3.2.1 NAVIX matches MiniGrid

NAVIX matches the original Minigrid suite in terms of environments, observations, state transitions, rewards, and actions. We include the most commonly used environments of the suite (see Table 14, Appendix I), and provide a set of baselines for the implemented environments in see Section 4 and Table 14, Appendix I.

Formally, a NAVIX environment is a tuple $\mathcal{M} = (h, w, T, \mathcal{O}, \mathcal{A}, \mathcal{R}, d, O, R, \gamma, P)$. Here, h and w are the height and width of the grid, T is the number of timesteps before timeout; $\mathcal{O}$ is the observation space, $\mathcal{A}$ is the action space, $\mathcal{R}$ is the reward space; γ is the discount factor. O is the observation function, R is the reward function, d is the termination function, and P is the transition function.

Reward functions A key difference between NAVIX and MiniGrid, by design, is that the latter uses a non-Markovian reward function. In fact, Minigrid dispenses a reward of 0 everywhere, except at task completion, where it is inversely proportional to the number of steps taken by the agent to reach the goal:

$$r_t = R(s_t, a, s_{t+1}) - 0.9 * (t + 1)/T, \tag{1}$$

Here R is the reward function, s_t is the state at time t, a is the action taken at time t, s_{t+1} is the state at time $t + 1$, and T is the number of timesteps before timeout. Notice the dependency on the number of steps t, which makes the reward non-Markovian.

The use of a non-Markovian reward function is not a mild assumption as most of the RL algorithms assume Markov rewards (Schulman et al., 2017; Haarnoja et al., 2018b; van Hasselt et al., 2016). This might call into question the validity of the historical results obtained with MiniGrid, and the generalisation of the results to other environments.

However, the necessity to align the Minigrid reward function with Markov assumptions is in stark contrast with the principle to reproduce the exact reward semantics of MiniGrid. Since this is a point of difference that might invalidate our claim that NAVIX is a semantically compatible replacement for MiniGrid, we leverage the modularity of NAVIX and supply two ready-to-use reward functions. These functions are variables that can be easily changed at the time of the creation of the environment. In the first function, we depart from the original Minigrid reward function and use a Markovian reward function, which is 0 everywhere, 1 at task completion, and $-c$ every at every timestep if the agent performs an action different from the `do-nothing` action. Here, where c is a constant action cost. In the second version, we replicate the reward function of Minigrid in Equation (1). We analyse the impact of Markovianity on the training of a PPO agent in Appendix B by measuring how the task completion rate varies during training for each reward function.

3.2.2 Stateless and fully jittable

While we aim to match Minigrid in terms of environments, observations, state transitions, rewards, and actions, the API of NAVIX is different, as it must align with JAX requirements for the environment to be fully jittable. In fact, NAVIX environments can be compiled using Accelerated Linear Algebra (XLA) – an open-source compiler for machine learning that optimises Python code for high-performance execution across different hardware platforms including GPUs, CPUs, TPUs and other ML accelerators. This includes both simply jitting – i.e., compiling just-in-time – the `step` function, and jitting the entire training sequence (Lu et al., 2022), assuming that the agent is also implemented in JAX. XLA compilation increases the throughput of experiments massively, allowing for the training of thousands of agents in parallel on a single accelerator, compared to a few that are possible with traditional CPU-based environments. We show the scalability of NAVIX in Section 4.

For environments to be fully jittable, the computation must be stateless. For this reason, we need to define an environment *state-object*: the *timestep*. The timestep is a tuple $(t, o_t, a_t, r_{t+1}, \gamma_{t+1}, s_t, i_{t+1})$, where t is the current time – the number of steps elapsed from the last reset – o_t is the observation at time t, a_t is the action taken after o_t, r_{t+1} is the reward received after a_t, γ_{t+1} is the termination signal after a_t, s_t is the true state of the environment at time t, and i_{t+1} is the info dictionary, useful to store accumulations, such as returns.

This structure is necessary to guarantee the same return schema for both the `step` and the `reset` methods, and allows the environment to autoreset, and avoid conditional statements in the agent code, which would prevent the environment from being fully jittable.

At the beginning of the episode, the agent samples a starting state from the starting distribution $P_0 : \mathcal{S} \to \mathcal{S}$ using the `reset(key)` method, where `key` is a key for a stateless random number generator. Since there is no action and reward at the beginning of the episode, we pad with -1 and 0, respectively. Given an action a_t, the agent can interact with the environment by calling the `step(timestep, action, key)` method. The agent then receives a new state of the environment (a new timestep) and can continue to interact as needed. Code 1 shows an example of how to interact with a jitted NAVIX environment. More examples will be provided online.

```python
import navix as nx

# init a NAVIX environment
env = nx.make("Navix-KeyCorridorS6R3-v0")

# sample a starting state
timestep = env.reset(key)
for _ in range(1000):
    # sample a random key
    key, subkey = jax.random.split(key)
    # sample a random action
    action = jax.random.randint(subkey, (1,), 0, 4)
    # interact with the environment
    timestep = jax.jit(env.step)(timestep, action)  # autoresets when done
```

Code 1: Example code to interact with a jitted NAVIX environment.

Notice that the syntax is similar to the original MiniGrid, including the environment *id*, which simply replaces "MiniGrid" with "Navix". The only differences are in the use of an explicit random key for the stateless random number generator, and the fact that the `step` method also takes the current timestep as input, to guarantee the statelessness of the computation.

The schema in Code 1 is an effective template for any kind of agent implementation, including non JAX-jittable agents. However, while this already improves the speed of environment interactions compared to MiniGrid, as shown in Section 4.1, the real speed-up comes jitting the whole iteration loop. In Appendix F we provide additional reusable patterns that are useful in daily RL research, such as how to jit the training loop, how to parallelise the training of multiple agents, and how to run hyperparameter search in batch mode.

In addition, in Appendix H we provide a guide on how to extend NAVIX, including new environments, new observations, new rewards, and new termination functions. This is a fundamental aspect to reflect the flexibility of the original Minigrid suite, which is easy to extend and modify.

4 Experiments

This section aims to show the advantages of NAVIX compared to the original Minigrid implementation, and provides the community with a set of baselines for all environments. It does the former by comparing the two suites, for all environments, both in terms of speed improvements and throughput. For the latter, we train a set of baselines for all environments, and provide a scoreboard that stores the results for all environments. All experiments are run on a single Nvidia A100 80Gb, and Intel(R) Xeon(R) Silver 4310 CPU @ 2.10GHz and 128Gb of RAM.

4.1 Speed

We first benchmark the raw speed improvements of NAVIX compared to the original Minigrid implementation, in the most common settings. For each NAVIX environment and its Minigrid equivalent, we run $1K$ steps with 8 parallel environments, and measure the wall time of both. Notice that this is the mere speed of the environment, and does not include the agent training.

We show results in Figure 3, and observe that NAVIX is over $44\times$ times faster than the original Minigrid implementation on average. These improvements are due to both the migration of the computation to the GPU via XLA, which optimises the computation graph for the specific accelerator, and the batching of the environments. In Figure 15, Appendix I we ablate the batching, with no parallel environments, and show that the biggest contribution for the speedup is due to efficient batching.

To better understand how the speedup varies with the number of training steps, and to make sure that the $1K$ steps used in the previous experiment are representative of the general trend, we measure the speed improvements for different lengths of the training runs. We run $1K$, $10K$, $100K$, and $1M$ steps for the `MiniGrid-Empty-8x8-v0` environment and its NAVIX equivalent, and measure the wall time of both.

Results in Figure 4 show that NAVIX is consistently faster than the original Minigrid implementation, regardless of the number of steps. Both Minigrid and NAVIX show a linear increase in the wall time with the number of steps.

4.2 Throughput

While NAVIX provides speed improvements compared to the original Minigrid implementation, the real advantage comes from the ability to perform highly parallel training runs on a single accelerator. In this experiment, we test how the computation scales with the number of environments.

We first test the limits of NAVIX by measuring the computation while varying the number of environments that run in parallel. MiniGrid uses `gymnasium`, which parallelises the computation with *Python*'s multiprocessing library. NAVIX, instead, uses JAX's native `vmap`, which directly vectorises the computation. We confront the results with the original Minigrid implementation, using the `MiniGrid-Empty-8x8-v0` environment.

Results in Figure 5 show that the original Minigrid implementation cannot scale beyond 16 environments on 128GB of RAM, for which it takes around 1s to complete $1K$ unrolls. On the contrary, NAVIX can run up to 2^{21} (over $2M$) environments in parallel on the same hardware, with a wall time almost always below 1s. In short, NAVIX achieves a throughput over 10^5 orders of magnitude higher than the original Minigrid implementation.

Secondly, we simulate the very common operation of training many PPO agents, each with their own subset of 16 environments. However, with NAVIX, we can do this in parallel. We use the `Empty-8x8-v0` environment, and train the agent for $1M$ steps. Results are shown in Figure 6.

Overall, we observe that, on a single NVIDIA A100 80GB, stepping a batch of $2048 \times 16 = 32\,768$ NAVIX environments for 10^6 transitions takes $49s$. This is a throughput of $2048 \times 16 \times 1M/49s = 6.7 \times 10^8$ environment steps/s. On the other hand, the original Minigrid reference with a CleanRL

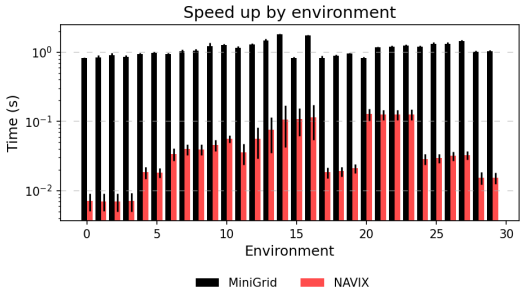


Figure 3: Speedup of NAVIX compared to the original Minigrid implementation, for the implemented environments. The identifiers on the x-axis correspond to the environments as reported in Table 13, Appendix I. Results are the average across 5 runs.

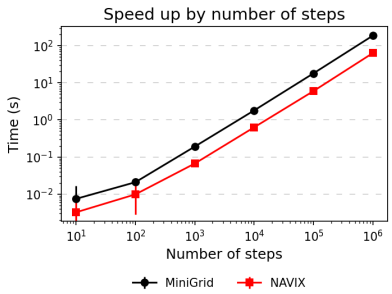


Figure 4: Variation of the speedup of NAVIX compared to the original Minigrid implementation.

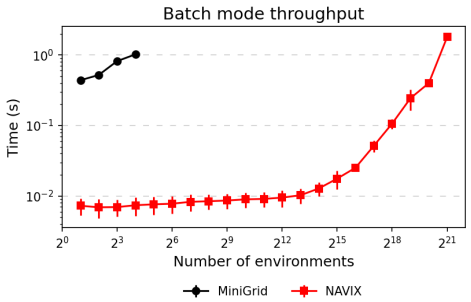


Figure 5: Wall time of $1K$ unrolls for both NAVIX and Minigrid in batch mode.

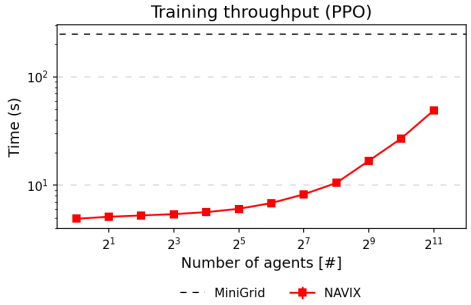


Figure 6: Computation costs with growing batch sizes.

baseline, executed on the same host with a dual-socket Intel® Xeon® Silver 4310 (24 cores, 48 threads at 2.10 GHz), takes $240s$ for a *single* PPO agent. This is a throughput of $1 \times 1M/240s = 4.2 \times 10^3$ environment steps/s. In other words, NAVIX achieves a speedup of $6.7{\times}10^8/4.2{\times}10^4 =\approx 160\,000\times$. All timings exclude the initial XLA compilation and a one-episode warm-up for fairness. This comparison fixes the compute budget to a single node to highlight the throughput boost that NAVIX unlocks. Matching the same wall-clock throughput with CPU-only Minigrid would require on the order of 10^5 concurrently active environments spread across many machines – incurring in network synchronisation overheads that our single-GPU setup avoids, and costs are that far beyond those of a single NVIDIA A100 80 GB node.

To complement these results, and give readers a more comprehensive expectation of NAVIX's performance, we also performed the speed and throughput experiments on consumer-grade hardware. Results in Figures 9, 10, 11, and 12, Appendix C, confirm this trends of this section.

4.3 Baselines

We provide additional baselines using the implementations of PPO (Schulman et al., 2017), Double DQN (DDQN) (Hasselt et al., 2016), Soft Actor Critic (SAC) (Haarnoja et al., 2018a), IQN (Dabney et al., 2018), and PQN (Gallici et al., 2025) in Rejax (Liesen et al., 2024a)[3]. We optimize hyperparameters (HP) for each algorithm and environment combination using 32 iterations of random search. Each HP configuration is evaluated with 16 different initial seeds. The HP configuration with the highest average final return is selected. The specific hyperparameters we searched for are shown in Table 12, Appendix E. The definitive list of hyperparameters used in the experiments can be found in Section E.

[3]https://github.com/keraJLi/rejax

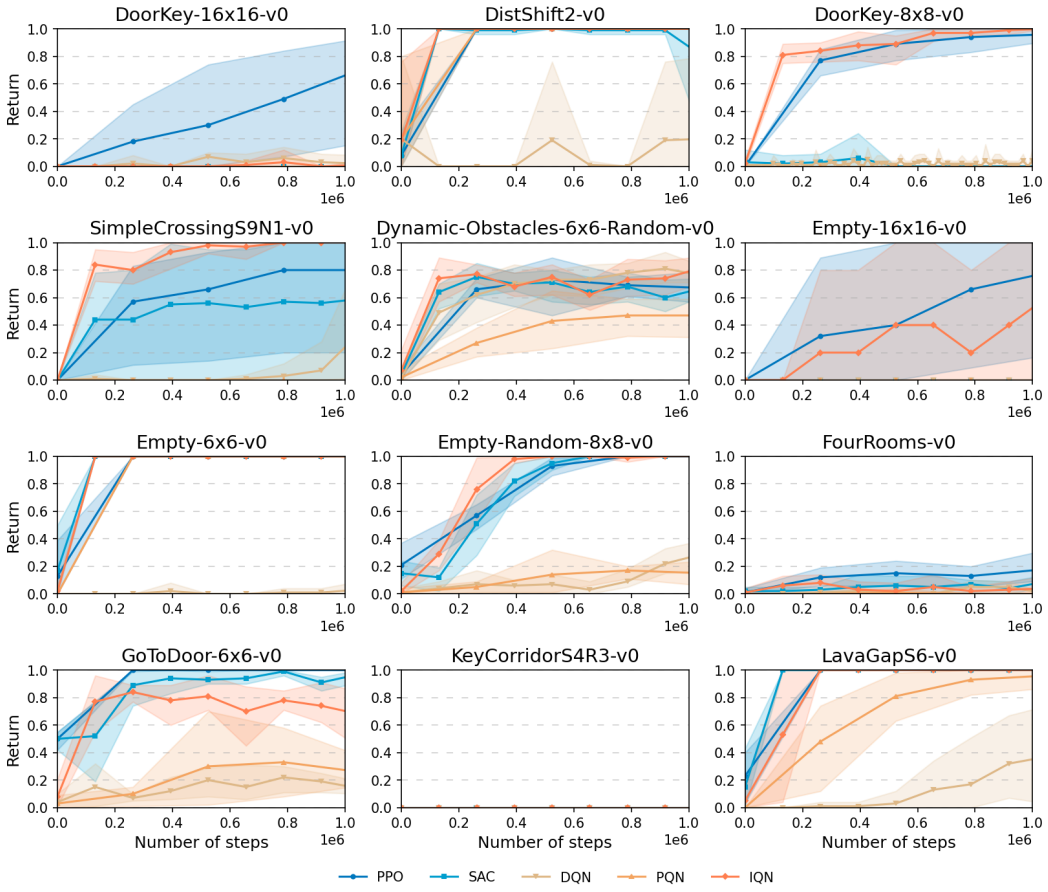


Figure 7: Episodic returns for a sample of NAVIX environments for DDQN, PPO and SAC baselines. Lines are average over 32 seeds, and shaded areas show 5-95 percentile confidence intervals.

We run the baselines for $1M$ steps, across 32 seeds, with the tuned hyperparameters for the environments shown in Figure 7. All algorithms use networks with two hidden layers of 64 units. Instead of alternating between a single environment step and network update, DQN and SAC instead perform 128 parallel environment steps and 128 network updates, each with a new minibatch. We found that this significantly improves the runtime while leaving the final performance unaffected.

5 Broader Impact

NAVIX lowers the hardware barrier to entry for RL research by providing a GPU-batched, JAX-native re-implementation of the Minigrid environment suite. Because NAVIX remains fully open-source (Apache 2.0) and can be run on commodity GPU hardware, instructors and students at resource-constrained institutions can reproduce state-of-the-art Minigrid agents on a single desktop GPU, democratizing access to RL education and experimentation.

However, higher-throughput simulation can also potentially accelerate the development cycle of autonomous navigation agents that could be deployed for disallowed or malicious purposes. Nevertheless, NAVIX itself does not contain any trained policies, faster training may facilitate downstream misuse.

6 Conclusions

We introduced NAVIX, a reimplementation of the Minigrid environment suite in JAX that leverages JAX's intermediate language representation to migrate the computation to different accelerators, such

as GPUs and TPUs. We described the design pattern, highlighting the connections to the ECSM, and the correspondence between the structure of its functions and the mathematical formalism of RL. We presented the environment interface, the list of available environments.

We showed the speed improvements of NAVIX compared to the original Minigrid implementation, and the scalability of NAVIX with respect to the number of agents that can be trained in parallel, or the number of environments that can be run in parallel.

Overall, NAVIX can be over $160\,000\times$ faster than the original Minigrid implementation, turning 1-week experiments into 15-minute ones. We noticed that these results assume access to a modern GPU, and extending NAVIX to the few remaining Minigrid variants and to training pipelines that cannot JIT-compile the full loop is left for future work.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 929–947, 2024.

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, 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/google/jax`.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. arxiv. *arXiv preprint arXiv:1606.01540*, 10, 2016.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *Advances in Neural Information Processing Systems*, 36, 2024.

Will Dabney, Mark Rowland, Marc Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pp. 1407–1416. PMLR, 2018.

Yannis Flet-Berliac, Johan Ferret, Olivier Pietquin, Philippe Preux, and Matthieu Geist. Adversarially guided actor-critic. *arXiv preprint arXiv:2102.04376*, 2021.

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. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=7IzeL0kflu.

Lin Guan, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging approximate symbolic models for reinforcement learning via skill diversity. In *International Conference on Machine Learning*, pp. 7949–7967. PMLR, 2022.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pp. 1861–1870. Proceedings of Machine Learning Research, 2018a.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018b.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. AAAI'16, pp. 2094–2100. AAAI Press, 2016.

Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, pp. 4940–4950. Proceedings of Machine Learning Research, 2021.

Minqi Jiang, Michael Dennis, Edward Grefenstette, and Tim Rocktäschel. minimax: Efficient baselines for autocurricula in jax. 2023.

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *Ijcai*, volume 16, pp. 4246–4247, 2016.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

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.

Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. *Advances in Neural Information Processing Systems*, 33:7671–7684, 2020.

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

Jarek Liesen, Chris Lu, and Robert Lange. rejax, 2024a. URL https://github.com/keraJLi/rejax.

Jarek Liesen, Chris Lu, Andrei Lupu, Jakob N Foerster, Henning Sprekeler, and Robert T Lange. Discovering minimal reinforcement learning environments. *arXiv preprint arXiv:2406.12589*, 2024b.

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.

Chris Lu, Yannick Schroecker, Albert Gu, Emilio Parisotto, Jakob Foerster, Satinder Singh, and Feryal Behbahani. Structured state space models for in-context reinforcement learning. *arXiv preprint arXiv:2303.03982*, 2023.

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. *arXiv preprint arXiv:2402.16801*, 2024.

Augustine Mavor-Parker, Kimberly Young, Caswell Barry, and Lewis Griffin. How to stay curious while avoiding noisy tvs using aleatoric uncertainty estimation. In *International Conference on Machine Learning*, pp. 15220–15240. PMLR, 2022.

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.

Jesse Mu, Victor Zhong, Roberta Raileanu, Minqi Jiang, Noah Goodman, Tim Rocktäschel, and Edward Grefenstette. Improving intrinsic exploration with language abstractions. *Advances in Neural Information Processing Systems*, 35:33947–33960, 2022.

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.

Soichiro Nishimori. Jax-corl: Clean sigle-file implementations of offline rl algorithms in jax. 2024. URL https://github.com/nissymori/JAX-CORL.

Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvári, Satinder Singh, Benjamin Van Roy, Richard Sutton, David Silver, and Hado van Hasselt. Behaviour suite for reinforcement learning. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rygf-kSYwH.

Fabian Paischer, Thomas Adler, Vihang Patil, Angela Bitto-Nemling, Markus Holzleitner, Sebastian Lehner, Hamid Eghbal-Zadeh, and Sepp Hochreiter. History compression via language models in reinforcement learning. In *International Conference on Machine Learning*, pp. 17156–17185. PMLR, 2022.

Simone Parisi, Victoria Dean, Deepak Pathak, and Abhinav Gupta. Interesting object, curious agent: Learning task-agnostic exploration. *Advances in Neural Information Processing Systems*, 34: 20516–20530, 2021.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Gardar Ingvarsson, Timon Willi, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Tjarko Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktaschel, Chris Lu, and Jakob Nicolaus Foerster. Jaxmarl: Multi-agent rl environments in jax. *arXiv preprint arXiv:2311.10090*, 2023a.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Gardar Ingvarsson, Timon Willi, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, et al. Jaxmarl: Multi-agent rl environments in jax. *arXiv preprint arXiv:2311.10090*, 2023b.

Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

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

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *International conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012.

Edan Toledo. Stoix: Distributed Single-Agent Reinforcement Learning End-to-End in JAX, April 2024. URL `https://github.com/EdanToledo/Stoix`.

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

Jane X Wang, Michael King, Nicolas Pierre Mickael Porcel, Zeb Kurth-Nelson, Tina Zhu, Charlie Deck, Peter Choy, Mary Cassin, Malcolm Reynolds, H. Francis Song, Gavin Buttimore, David P Reichert, Neil Charles Rabinowitz, Loic Matthey, Demis Hassabis, Alexander Lerchner, and Matthew Botvinick. Alchemy: A benchmark and analysis toolkit for meta-reinforcement learning agents. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL `https://openreview.net/forum?id=eZu4BZxlRnX`.

Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, et al. Envpool: A highly parallel reinforcement learning environment execution engine. *Advances in Neural Information Processing Systems*, 35:22409–22421, 2022.

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

Daochen Zha, Wenye Ma, Lei Yuan, Xia Hu, and Ji Liu. Rank the episodes: A simple approach for exploration in procedurally-generated environments. *arXiv preprint arXiv:2101.08152*, 2021.

Tianjun Zhang, Huazhe Xu, Xiaolong Wang, Yi Wu, Kurt Keutzer, Joseph E Gonzalez, and Yuandong Tian. Bebold: Exploration beyond the boundary of explored regions. *arXiv preprint arXiv:2012.08621*, 2020.

Mingde Zhao, Zhen Liu, Sitao Luan, Shuyuan Zhang, Doina Precup, and Yoshua Bengio. A consciousness-inspired planning agent for model-based reinforcement learning. *Advances in neural information processing systems*, 34:1569–1581, 2021.

A Details on NAVIX systems

Systems are *functions* that operate on the collective state of all entities, defining the rules of the interactions between them. In designing NAVIX, we aimed to maintain a bijective relationship between the systems and their respective mathematical formalism in RL. This makes it easier to translate the mathematical formalism into code, and vice versa, connecting the implementation to the theory. NAVIX includes the following systems:

1. `Intervention`: a function that updates the state of the entities according to the actions taken by the agents.
2. `Transition`: a function that updates the state of the entities according to the MDP state transitions.
3. `Observation`: a function that generates the observations that the agents receive.
4. `Reward`: a function that computes the rewards that the agents receive.
5. `Termination`: a function that determines if the episode is terminated.

We now describe the systems formally.

The intervention is a function $I : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ that updates the state of the entities according to the actions taken by the agents. This corresponds to the canonical decision in an MDP.

The transition is a function $\mu : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ that updates the state of the entities according to the MDP state transitions. This corresponds to the canonical state transition kernel in an MDP.

The observation is a function $O : \mathcal{S} \to \mathcal{O}$ that generates the observations that the agents receive. NAVIX includes multiple observation functions, each generating a different type of observation, for example, a first-person view, a top-down view, or a third-person view, both in symbolic and pixel format. We provide both full and partial observations, allowing to cast a NAVIX environment both as an MDP or as a POMDP, depending on the needs of the algorithm. This follows the design of the original MiniGrid suite.

The reward is a function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that computes the rewards that the agents receive. Likewise, the termination is a function $\gamma : \mathcal{S} \to \{0, 1\}$ that determines if the episode is terminated. We include both the reward and the termination functions necessary to reproduce all MiniGrid environments. Both these systems rely on the concept of *events*, representing a goal to achieve. An *event* is itself an entity signalling that a particular state of the environment has been reached. For example, it can indicate that the agent has reached a particular cell, has picked up a particular object, or that the agent performed a certain action in a particular state.

We provide a summary of the implemented systems in NAVIX in Tables 4, 5, and 6 for the observation, reward, and termination systems, respectively.

Observation function	Shape	Description
`symbolic`	`i32[H, W, 3]`	*The canonical grid encoding observation from MiniGrid.*
`symbolic_first_person`	`i32[R, R, 3]`	*A first-person view of the environment in symbolic format.*
`rgb`	`u8[32 * H, 32 * W, 3]`	*A fully visible image of the environment in RGB format.*
`rgb_first_person`	`u8[32 * R, 32 * R, 3]`	*A first-person view of the environment in RGB format.*
`categorical`	`i32[H, W]`	*A grid of entities ID in the environment.*
`categorical_first_person`	`i32[R, R]`	*A first-person view of the grid of entities ID.*

Table 4: Implemented observation functions in NAVIX.

Reward function	Description
`on_goal_reached`	$+1$ *when a Goal entity and a Player entity have the same position*
`on_lava_fall`	-1 *when a Lava entity and a Player entoty have the same position*
`on_door_done`	$+1$ *when the* `done` *action is performed in front of a door with the colour specific in the* `mission`
`free`	0 everywhere
`action_cost`	$-cost_a$ *at every action taken, except* `done`
`time_cost`	$-cost_t$ *at every step*

Table 5: Implemented reward functions in NAVIX.

Termination function	Description
`on_goal_reached`	*Terminates when a Goal entity and a Player entity have the same position*
`on_lava_fall`	*Terminates when a Lava entity and a Player entity have the same position*
`on_door_done`	*Terminates when the* `done` *action is performed in front of a door with the colour specific in the* `mission`
`free`	0 everywhere

Table 6: Implemented termination functions in NAVIX.

B Impact of reward Markovianity on PPO training

As described in Section 3.2.1, NAVIX replicates the semantics of MiniGrid in terms of environments, observations, state transitions rewards, and actions. However, while developing, we noticed that MiniGrid's reward function is non-Markovian, despite most of the RL algorithms assuming Markov rewards (Schulman et al., 2017; Haarnoja et al., 2018b; van Hasselt et al., 2016). This might call into question the validity of the historical results obtained with MiniGrid, and the generalisation of the results to other environments.

Here, we analyse the impacts of Markovianity on the training of a PPO agent, and how the task completion rate varies during training, with a Markov and a non-Markov reward function. We use the following reward function, respectively:

$$r_t = R(s_t, a, s_{t+1}) \tag{2}$$

$$r_t = R(s_t, a, s_{t+1}) - 0.9 * (t+1)/T. \tag{3}$$

Equation (2) represents the Markovian reward function, while Equation 3 the non-Markovian one.

Results in figure 8 shows that, for some environments, in particular those where PPO converges to near-optimality, the success rate presents a higher variance with non-Markov rewards. To avoid

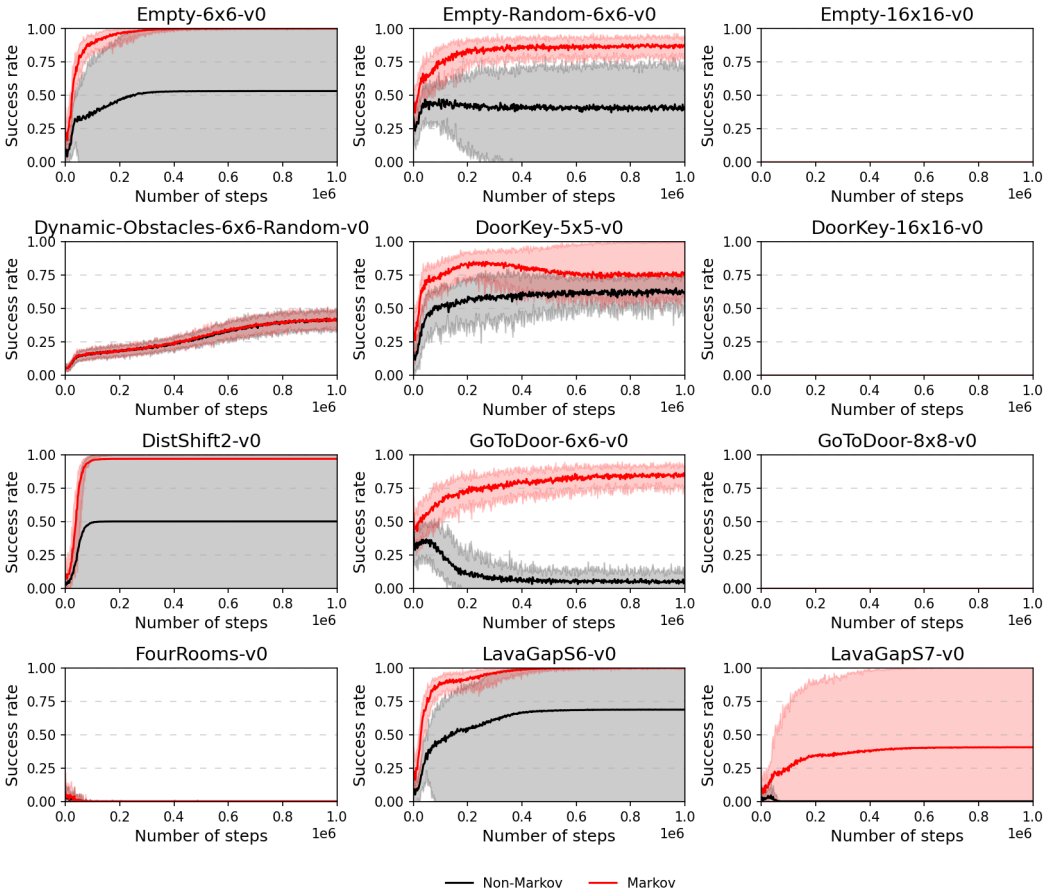


Figure 8: Task completion rate of a PPO agent across environments with a Markov (red) and a non-Markov (black) reward function.

failures when reproducing prior MiniGrid studies, this experiment calls researchers to pay particular attention to choose the original non-Markovian reward function of MiniGrid, also available in NAVIX.

C Performance on consumer-grade hardware

Despite NAVIX being designed for training at scale, to give readers a more comprehensive expectation of its performance, we tested the speed and throughput gain in Section 4.1 and 4.2 on a consumer-grade level hardware. The experiment is carried on a consumer desktop with an i7-11700 @ 2.50GHz CPU, 32Gb of RAM, and an Nvidia RTX A4000 with 16Gb of VRAM. All results are averaged across 5 runs.

The evidence in Figure 9 and 10 confirms the trend of Section 4.1. NAVIX is over 7 times faster than the original MiniGrid implementation on average across environments.

Figure 11 and 12, instead, replicate the experiments in Section 4.2. Results show consistent performance on a consumer-grade machine. On a consumer-grade machine, NAVIX can run up to 2^18 (over $260K$) environments in parallel on a single Nvidia RTX A4000 with 16Gb of VRAM, and over 512 full PPO agents.

While it is expected that consumer-hardware cannot keep up with the high-end hardware designed for computation at scale, NAVIX still shows a speedup of approximately $34\,000\times$.

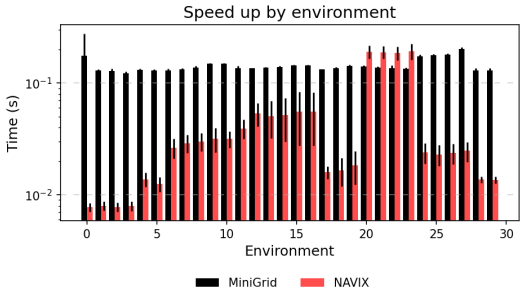


Figure 9: Speedup of NAVIX compared to the original Minigrid implementation on a commercial-grade desktop. The identifiers on the x-axis correspond to the environments as reported in Table 13, Appendix I. Results are the average across 5 runs.

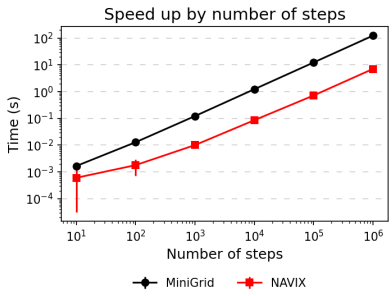


Figure 10: Variation of the speedup of NAVIX compared to the original Minigrid implementation on a consumer-grade machine.

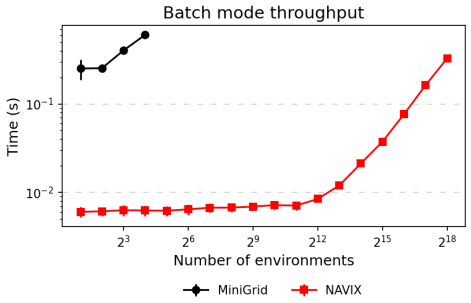


Figure 11: Wall time of $1K$ unrolls for both NAVIX and MiniGrid in batch mode on a consumer-grade machine.

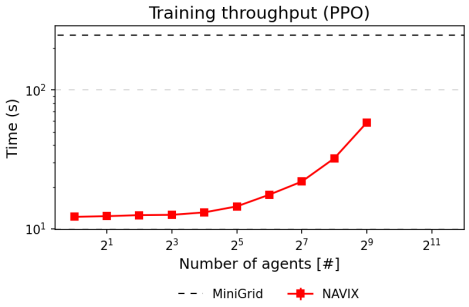


Figure 12: Computation costs with growing batch sizes on a consumer-grade machine. The horizontal dashed line shows the MiniGrid time to train a single PPO agent.

D Impact of the design pattern

The speedup of NAVIX does not only depend on JAX, but also on the transition to an ECS design pattern. To marginalise the performance gains over the design patter, we have compared the performance of NAVIX on CPU and GPU: the exact same Python program, with the exact same design pattern, but compiled and run on different accelerators.

We measure speed and throughput as described in Section 4.1 and Section 4.2. The experiment is carried on a desktop with an i7-11700 @ 2.50GHz CPU, 32Gb of RAM, and an Nvidia RTX A4000 with 16Gb of VRAM. Figures 13 and 14 show the speedup at different number of environment steps, while the second table shows the throughput capacity. Results suggest that when the number of environments is small (<32), NAVIX CPU is more or as efficient as on GPU, and allows for a larger batch size due to the higher memory capacity (RAM) compared to the GPU memory capacity (VRAM). However, as the batch size grows, the GPU scales better. Its performance remains constant up until around 2^12 environments, after which starts to grow linearly.

This confirms that the performance gains of NAVIX derive from a systemic transition to a different framework in JAX, and not only from the transition to different types of accelerators. The higher throughput of NAVIX on CPU derives from a higher RAM capacity compared to the VRAM of the GPU.

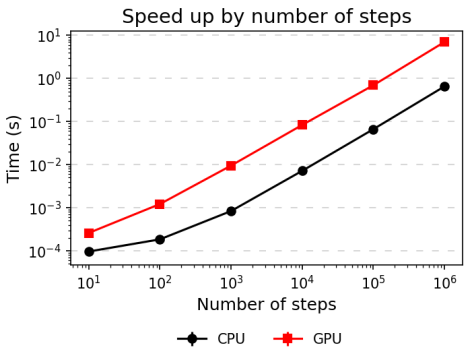


Figure 13: Computation speed across number different number of sequential steps between a CPU- and GPU-compiled NAVIX program.

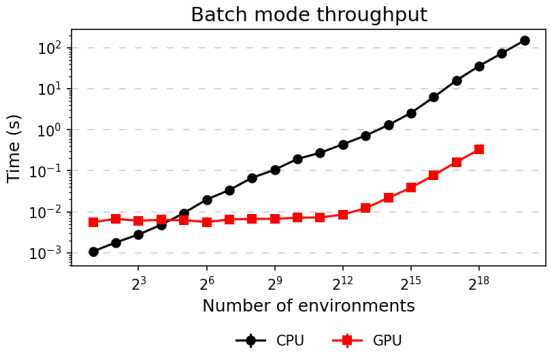


Figure 14: Computation costs across growing batch sizes between a CPU- and GPU-compiled NAVIX program.

E Details on baselines

Here we report the hyperparameters used to run the baselines in Section 4.3, for each algorithm. To allow the table to fit into the page horizontally we shorten the names of each environment and present a full mapping below:

1. FR: `Navix-FourRooms-v0`,
2. DK8: `Navix-DoorKey-8x8-v0`,
3. DK16: `Navix-DoorKey-16x16-v0`,
4. GD6: `Navix-GoToDoor-6x6-v0`,
5. SC9: `Navix-SimpleCrossingS9N1-v0`,
6. E6: `Navix-Empty-6x6-v0`,
7. E16: `Navix-Empty-16x16-v0`,
8. ER8: `Navix-Empty-Random-8x8-v0`,
9. LG6: `Navix-LavaGapS6-v0`,
10. KC43: `Navix-KeyCorridorS4R3-v0`,
11. DO6R: `Navix-Dynamic-Obstacles-6x6-Random-v0`,
12. DS2: `Navix-DistShift2-v0`.

PPO	DS2	DK16	DK8	DO6R	E16	E6	ER8	FR	GD6	KC43	LG6	SC9
activation	swish	tanh	tanh	tanh	tanh	relu	swish	swish	tanh	tanh	tanh	swish
num_envs	256	16	16	16	16	16	64	128	32	128	16	64
num_steps	64	128	128	128	128	256	128	32	32	64	128	128
num_epochs	16	2	2	16	2	4	8	2	8	4	8	4
num_minibatches	1	1	1	8	1	16	1	8	32	1	8	4
learning_rate	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003
max_grad_norm	10	1	1	5	1	1	10	5	10	10	0.5	0.5
total_timesteps	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M
eval_freq	2K	2K	2K	2K	2K	2K	2K	2K	2K	2K	2K	2K
gamma	0.95	0.95	0.95	0.99	0.95	0.99	0.99	0.99	0.99	0.95	0.99	0.99
gae_lambda	0.95	0.9	0.9	0.99	0.9	0.9	0.8	0.99	0.95	0.95	0.99	0.9
clip_eps	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
ent_coef	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
vf_coef	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
normalize_observations	true	false	false	true	false	true	true	false	true	true	true	true

Table 7: Hyperparameters for the PPO agent. Numeric abbreviations: $1\text{M} = 1\,048\,576$, $256\text{K} = 262\,144$.

SAC	DS2	DK16	DK8	DO6R	E16	E6	ER8	FR	GD6	KC43	LG6	SC9
activation	swish	swish	swish	tanh	relu	tanh	tanh	tanh	tanh	relu	swish	tanh
num_envs	128	128	128	128	128	128	128	128	128	128	128	128
buffer_size	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K
fill_buffer	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K
batch_size	256	128	128	512	512	512	512	256	256	128	1028	1028
learning_rate	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003
num_epochs	128	128	128	128	128	128	128	128	128	128	128	128
total_timesteps	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M
eval_freq	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K
gamma	0.8	0.8	0.8	0.8	0.8	0.8	0.95	0.9	0.9	0.8	0.8	0.8
polyak	0.9	0.995	0.995	0.95	0.995	0.95	0.9	0.995	0.995	0.95	0.9	0.9
target_entropy_ratio	0.6	0.9	0.9	0.5	0.5	0.5	0.3	0.9	0.9	0.4	0.6	0.6
normalize_observations	true	false	false	false	false	false	false	false	false	true	true	true

Table 8: Hyperparameters for the SAC agent. Numeric abbreviations: $1\text{M} = 1\,048\,576$, $128\text{K} = 131\,072$, $8\text{K} = 8\,192$.

DQN	DS2	DK16	DK8	DO6R	E16	E6	ER8	FR	GD6	KC43	LG6	SC9
activation	swish	relu	relu	relu	swish	swish	swish	relu	swish	swish	swish	tanh
num_envs	10	10	10	10	10	10	10	10	10	10	10	10
num_epochs	1	1	1	1	1	1	1	1	1	1	1	1
buffer_size	128K	128K	64K	128K	128K	128K	128K	128K	128K	128K	128K	128K
fill_buffer	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K
batch_size	1,024	512	128	1,024	2,048	1,024	1,024	256	1,024	128	512	512
learning_rate	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003
max_grad_norm	0.5	1	10	0.5	10	0.5	1	10	0.5	10	1	1
total_timesteps	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M
eval_freq	128K	128K	16K	128K	128K	128K	128K	128K	128K	128K	128K	128K
gamma	0.95	0.95	0.95	0.95	0.95	0.95	0.9	0.9	0.95	0.995	0.9	0.95
eps_start	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
eps_end	0.01	0.01	0.05	0.01	0.01	0.01	0.05	0.01	0.01	0.01	0.05	0.05
exploration_fraction	0.3	0.5	0.5	0.5	0.5	0.3	0.1	0.1	0.3	0.3	0.1	0.3
target_update_freq	8K	2K	2K	512	1K	8K	1K	2K	8K	8K	4K	512
ddqn	true	true	true	true	true	true	true	true	true	true	true	true
normalize_observations	true	false	false	true	true	true	false	false	true	false	true	false

Table 9: Hyperparameters for the DQN agent. Numeric abbreviations: $1\text{M} = 1\,048\,576$, $128\text{K} = 131\,072$, $64\text{K} = 65\,536$, $16\text{K} = 16\,384$, $8\text{K} = 8\,192$, $4\text{K} = 4\,096$, $2\text{K} = 2\,048$.

IQN	DS2	DK16	DK8	DO6R	E16	E6	ER8	FR	GD6	KC43	LG6	SC9
activation	relu	tanh	tanh	tanh	swish	relu	swish	relu	relu	relu	swish	tanh
num_envs	10	10	10	10	10	10	10	10	10	10	10	10
num_epochs	1	1	1	1	1	1	1	1	1	1	1	1
buffer_size	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K
fill_buffer	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K	8K
batch_size	512	256	256	256	256	512	256	512	256	256	512	128
learning_rate	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003
kappa	1.0	0.5	0.5	0.5	3.0	1.0	3.0	2.0	0.5	0.5	1.0	2.0
num_tau_samples	32	16	16	16	16	32	16	32	16	16	16	32
num_tau_prime_samples	64	64	64	64	64	64	64	16	64	64	32	64
total_timesteps	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M
eval_freq	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K	128K
gamma	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.95	0.9	0.9	0.9	0.9
eps_start	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
eps_end	0.01	0.01	0.01	0.01	0.1	0.1	0.1	0.05	0.01	0.01	0.1	0.05
exploration_fraction	0.1	0.1	0.1	0.1	0.3	0.1	0.3	0.7	0.1	0.1	0.7	0.1
target_update_freq	1K	512	512	512	4K	1K	4K	512	512	512	2K	1K
normalize_observations	false	true	true	true	true	false	true	false	true	true	false	true

Table 10: Hyperparameters for the IQN agent. Numeric abbreviations: $1\text{M} = 1\,048\,576$, $128\text{K} = 131\,072$, $8\text{K} = 8\,192$, $4\text{K} = 4\,096$, $2\text{K} = 2\,048$, $1\text{K} = 1\,024$.

PQN	DS2	DK16	DK8	DO6R	E16	E6	ER8	FR	GD6	KC43	LG6	SC9
num_envs	16	512	512	512	512	16	128	32	128	128	8	128
num_steps	128	128	128	128	16	128	16	512	16	512	512	16
num_epochs	8	16	16	16	1	8	2	1	2	1	16	2
num_minibatches	128	64	64	64	128	128	128	128	128	256	64	128
learning_rate	0.000246	0.000126	0.000126	0.000126	0.000536	0.000246	0.000104	0.000179	0.000179	0.000211	0.000242	0.000104
max_grad_norm	5	1	1	1	0.5	5	1	10	1	1	5	1
total_timesteps	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M
eval_freq	256K	256K	256K	256K	256K	256K	256K	256K	256K	256K	256K	256K
gamma	0.9	0.8	0.8	0.8	0.8	0.9	0.99	0.99	0.99	0.95	0.8	0.99
td_lambda	0.8	0.2	0.2	0.2	0.8	0.8	0.8	0.4	0.8	0.8	0.8	0.8
eps_start	1	1	1	1	1	1	1	1	1	1	1	1
eps_end	0.1	0.1	0.1	0.1	0.1	0.1	0.05	0.05	0.05	0.05	0.1	0.05
exploration_fraction	0.3	0.3	0.3	0.3	0.5	0.3	0.2	0.7	0.2	0.4	0.2	0.2
normalize_observations	false	true	true	true	false	false	true	true	true	true	true	true

Table 11: Hyperparameters for the PQN agent. Columns use the short env IDs defined in App. E. Numeric abbreviations: $1\text{M} = 1\,048\,576$, $256\text{K} = 262\,144$.

Algorithm	Fitted hyperparameters
PPO	#envs, #steps, #epochs, #minibatches, discount factor, λ (GAE), grad. norm clip, norm. obs., activation function
DQN	batch size, target network update freq., discount factor, exploration fraction, final ϵ, grad. norm clip, norm. obs., activation function
SAC	batch size, discount factor, τ (Polyak update), target entropy ratio, norm. obs., activation function

Table 12: Fitted hyperparameters for PPO, DQN, and SAC. Details on each hyperparameter set, for each environment and each algorithm are available at `https://github.com/epignatelli/navix/tree/speedup/baselines/rejax/configs`.

F Reusable patterns

Here we provide some useful patterns that users can reuse as-they-are or modify to suit their needs. In particular, we show how to jit the full interaction loop of a NAVIX environment in Code 2, and how to run multiple seeds in parallel in Code 3. Further examples, including how to jit a whole training loop with a JAX-based agent, and how to automate hyperparameter search, are available in the NAVIX documentation at `https://epignatelli/navix/examples/getting_started.html`.

F.1 Jitting full interaction loops

```
import navix as nx

# init a NAVIX environment
env = nx.make("Navix-KeyCorridorS6R3-v0")

# sample a starting state
timestep = env.reset(key)

# jitting the step function
step_env = jax.jit(env.step)

# unroll the environment for 1000 steps
timestep, _ = jax.lax.scan(
    lambda timestep, _: (unroll(timestep, i % 6), ()),
    timestep,
    (timestep, jnp.arange(1000))
)
```

Code 2: Example code to jit a `Navix-Empty-5x5-v0` environment.

F.2 Running multiple seeds in parallel

```
import navix as nx

env = nx.make("Navix-KeyCorridorS6R3-v0")

# define the run function
def run(key):
    def step(state, action):
        timestep, key = state
        key, subkey = jax.random.split(key)
        action = jax.random.randint(subkey, (), 0, env.action_space.n)
        return (env.step(timestep, action), key), ()


    # unroll the environment for 1000 steps
    timestep = env.reset(key)
    timestep, _ = jax.lax.scan(
        step,
        timestep,
        ((timestep, key) jnp.arange(1000)),
    )
    return timestep

seeds = jax.random.split(jax.random.PRNGKey(0), 1000)
batched_end_steps = jax.jit(jax.vmap(run))(seeds)
```

Code 3: Example code to jit a `Navix-Empty-5x5-v0` environment.

G Customising NAVIX environments

NAVIX is designed to be highly customisable, allowing users to create new environments by combining existing entities and systems. In this section, we provide examples of how to customise NAVIX environments by using different *systems*.

For example, to create a new environment where the agent has to reach a goal while avoiding lava, we can combine the `Goal` and `Lava` entities with the `Reward` system:

```
import navix a nx

reward_fn = nx.rewards.compose(
    nx.rewards.on_goal_reached(),
    nx.rewards.on_lava_fall()
)

env = nx.make(
    "Navix-Empty-5x5-v0",
    reward_fn=reward_fn)
```

Code 4: Example code to create a `Navix-Empty-5x5-v0` environment with a custom reward function. See Table 5 for a list of implemented reward functions.

Alternatively, to use a different observation function, we can use the `Observation` system:

```
import navix as nx

env = nx.make(
    "Navix-Empty-5x5-v0",
    observation_fn=nx.observations.rgb())
```

Code 5: Example code to create a `Navix-Empty-5x5-v0` environment with a custom observation function. See Table 4 for a list of implemented observation functions.

Finally, to terminate the environment, for example, only when the agent reaches the goal, but not when it falls into the lava, we can use the `Termination` system:

```
import navix as nx

env = nx.make(
    "Navix-Empty-5x5-v0",
    termination_fn=nx.terminations.on_goal_reached())
```

Code 6: Example code to create a `Navix-Empty-5x5-v0` environment with a custom termination function. See Table 6 for a list of implemented termination functions.

These examples can be extended to create more complex environments by combining different systems for the same environment configuration.

H Extending NAVIX environments

NAVIX is designed to be easily extensible. Users can create new entities, components, systems, and full environments by implementing the necessary functions. In this section, we provide **templates** to extend NAVIX environments. In particular, Code 7 shows how to create a custom environment, Code 8 shows how to create a custom component, Code 9 shows how to create a custom entity, and Code 10 shows how to create custom systems.

```python
import jax, navix as nx

class CustomEnv(nx.Environment):
    def _reset(self, key: jax.Array) -> nx.Timestep:
        """Reset the environment."""
        # create your grid, place your entities, define your mission
        return timestep


nx.registry.register_env(
    "CustomEnv",
    lambda *args, **kwargs: CustomEnv.create(
        observation_fn=nx.observations.symbolic(),
        reward_fn=nx.rewards.on_goal_reached(),
        termination_fn=nx.terminations.on_goal_reached(),
    )
)
```

Code 7: Example code to extend NAVIX by creating a custom environment. The `_reset` function allows to generate a custom starting state, after which the environment will evolve according to the usual systems: intervention, transition, reward and termination functions. Notice that it is convenient to use the environment constructor `create` to automatically set non-orthogonal properties (e.g. observation space and observation function).

```python
import jax, navix as nx

class CustomComponent(nx.Componnet):
    """My custom component."""

    custom_property: jax.Array = nx.components.field(shape=())
```

Code 8: Example code to extend NAVIX by creating a custom component. Notice that the property must have a type annotation and specify a shape.

```python
import jax, navix as nx

class CustomEntity(nx.Entity, CustomComponent):
    """My custom entity."""

    @property
    def walkable(self) -> jax.Array:
        return jnp.broadcast_to(jnp.asarray(False), self.shape)

    @property
    def transparent(self) -> jax.Array:
        return jnp.broadcast_to(jnp.asarray(False), self.shape)

    @property
    def sprite(self) -> jax.Array:
        sprite = # the address of your sprite, e.g., SPRITES_REGISTRY[Entities.WALL]
        return jnp.broadcast_to(sprite[None], (*self.shape, *sprite.shape))

    @property
    def tag(self) -> jax.Array:
        entity_id = # the id of your entity, e.g., EntityIds.WALL
        return jnp.broadcast_to(entity_id, self.shape)
```

Code 9: Example code to extend NAVIX by creating a custom entity. Notice that four properties must be implemented: `walkable`, `transparent`, `sprite`, and `tag`.

```python
import jax, navix as nx

def my_reward_function(state: nx.State, action: nx.Action, new_state: nx.State) -> jax.Array:
    """My custom reward function."""
    # do stuff
    return reward  # f32[]

def my_termination_function(state: nx.State, action: nx.Action, new_state: nx.State) -> jax.Array:
    """My custom termination function."""
    # do stuff
    return termination  # bool[]

def my_observation_function(state: nx.State) -> jax.Array:
    """My custom observation function."""
    # do stuff
    return observation  # f32[]

def my_intervention_function(state: nx.State, action: nx.Action) -> nx.State:
    """My custom intervention function."""
    # do stuff
    return new_state  # State

def my_transition_function(state: nx.State) -> nx.State:
    """My custom transition function."""
    # do stuff
    return new_state  # State
```

Code 10: Example code to extend NAVIX by creating custom systems.

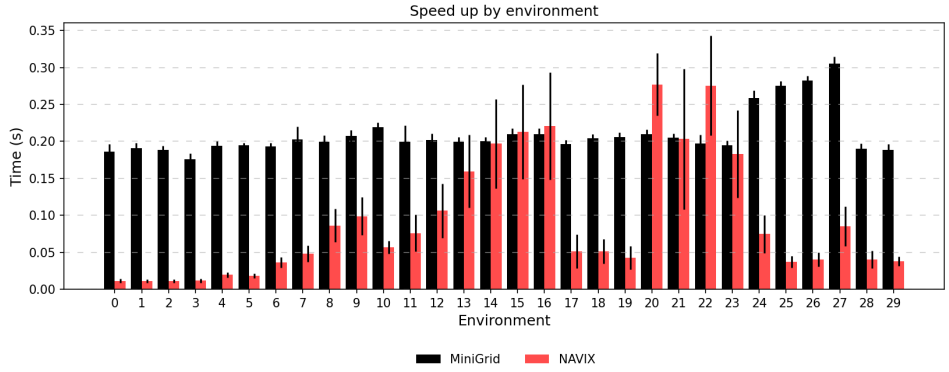


Figure 15: **Ablation.** Speedup of NAVIX compared to the original Minigrid implementation without batching. The identifiers on the x-axis correspond to the environments as reported in Table 13. Lower is better.

I Additional Tables

X tick	Env id
0	Navix-Empty-5x5-v0
1	Navix-Empty-6x6-v0
2	Navix-Empty-8x8-v0
3	Navix-Empty-16x16-v0
4	Navix-Empty-Random-5x5
5	Navix-Empty-Random-6x6
6	Navix-DoorKey-5x5-v0
7	Navix-DoorKey-6x6-v0
8	Navix-DoorKey-8x8-v0
9	Navix-DoorKey-16x16-v0
10	Navix-FourRooms-v0
11	Navix-KeyCorridorS3R1-v0
12	Navix-KeyCorridorS3R2-v0
13	Navix-KeyCorridorS3R3-v0
14	Navix-KeyCorridorS4R3-v0
15	Navix-KeyCorridorS5R3-v0
16	Navix-KeyCorridorS6R3-v0
17	Navix-LavaGapS5-v0
18	Navix-LavaGapS6-v0
19	Navix-LavaGapS7-v0
20	Navix-SimpleCrossingS9N1-v0
21	Navix-SimpleCrossingS9N2-v0
22	Navix-SimpleCrossingS9N3-v0
23	Navix-SimpleCrossingS11N5-v0
24	Navix-Dynamic-Obstacles-5x5
25	Navix-Dynamic-Obstacles-6x6
26	Navix-Dynamic-Obstacles-8x8
27	Navix-Dynamic-Obstacles-16x16
28	Navix-DistShift1-v0
29	Navix-DistShift2-v0

Table 13: Correspondence between the x-ticks in Figure 3 and the environment ids.

Table of environments available in NAVIX.

Env-id	Class	Height	Width	Reward
Navix-Empty-5x5-v0	Empty	5	5	R_1
Navix-Empty-6x6-v0	Empty	6	5	R_1
Navix-Empty-8x8-v0	Empty	8	8	R_1
Navix-Empty-16x16-v0	Empty	16	16	R_1
Navix-Empty-Random-5x5	Empty	5	5	R_1
Navix-Empty-Random-6x6	Empty	6	6	R_1
Navix-Empty-Random-8x8	Empty	8	8	R_1
Navix-Empty-Random-16x16	Empty	16	16	R_1
Navix-DoorKey-5x5-v0	DoorKey	5	5	R_1
Navix-DoorKey-6x6-v0	DoorKey	6	6	R_1
Navix-DoorKey-8x8-v0	DoorKey	8	8	R_1
Navix-DoorKey-16x16-v0	DoorKey	16	16	R_1
Navix-DoorKey-Random-5x5	DoorKey	5	5	R_1
Navix-DoorKey-Random-6x6	DoorKey	6	6	R_1
Navix-DoorKey-Random-8x8	DoorKey	8	8	R_1
Navix-DoorKey-Random-16x16	DoorKey	16	16	R_1
Navix-FourRooms-v0	FourRooms	17	17	R_1
Navix-KeyCorridorS3R1-v0	KeyCorridor	3	7	R_1
Navix-KeyCorridorS3R2-v0	KeyCorridor	5	7	R_1
Navix-KeyCorridorS3R3-v0	KeyCorridor	7	7	R_1
Navix-KeyCorridorS4R3-v0	KeyCorridor	10	10	R_1
Navix-KeyCorridorS5R3-v0	KeyCorridor	13	13	R_1
Navix-KeyCorridorS6R3-v0	KeyCorridor	16	16	R_1
Navix-LavaGap-S5-v0	LavaGap	5	5	R_2
Navix-LavaGap-S6-v0	LavaGap	6	6	R_2
Navix-LavaGap-S7-v0	LavaGap	7	7	R_2
Navix-Crossings-S9N1-v0	Crossings	9	9	R_2
Navix-Crossings-S9N2-v0	Crossings	9	9	R_2
Navix-Crossings-S9N3-v0	Crossings	9	9	R_2
Navix-Crossings-S11N5-v0	Crossings	11	11	R_2
Navix-Dynamic-Obstacles-5x5	Dynamic-Obstacles	5	5	R_3
Navix-Dynamic-Obstacles-5x5	Dynamic-Obstacles	5	5	R_3
Navix-Dynamic-Obstacles-6x6	Dynamic-Obstacles	6	6	R_3
Navix-Dynamic-Obstacles-6x6	Dynamic-Obstacles	6	6	R_3
Navix-Dynamic-Obstacles-8x8	Dynamic-Obstacles	8	8	R_3
Navix-Dynamic-Obstacles-16x16	Dynamic-Obstacles	16	16	R_3
Navix-DistShift1-v0	DistShift	6	6	R_2
Navix-DistShift2-v0	DistShift	8	8	R_2
Navix-GoToDoor-5x5-v0	GoToDoor	5	5	R_1
Navix-GoToDoor-6x6-v0	GoToDoor	6	6	R_1
Navix-GoToDoor-8x8-v0	GoToDoor	8	8	R_1

Table 14: List of environments available in NAVIX. *Env-id* denotes the id to instantiate the environment. Here, R_1 is the reward function for goal achievement – 1 when the agent is on the green square, and 0 otherwise. R_2 is the reward function for goal achievement and lava avoidance – 1 when the agent is on the green square, -1 when the agent is on the lava square, and 0 otherwise. R_3 is the reward function for goal achievement and dynamic obstacles avoidance – 1 when the agent is on the green square, -1 when the agent is hit by a flying object, and 0 otherwise. All environments terminate when the reward is not 0, for example, on goal achievement, or on lava collision.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract claims that: (i) NAVIX is a direct drop-in JAX re-implementation of MiniGrid, and (ii) it achieves $> 160\,000\times$ simulated-step throughput in batch mode and supports $2\,048$ agents on one A100 80 GB. Section 4.1 reports a 4.4×10^1 average speedup across environments without accounting for the increased throughput. Section 4.2 demonstrates by batching the computation also across multiple agents, up to $2\,048$, NAVIX can achieve a $> 160\,000\times$ speedup compared to the MiniGrid version, which is instead limited in scalability by its CPU-based architecture.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We remark here the dependence on modern GPUs, i.e., NVidia A100 80GB or better, to reproduce the results. Furthermore, NAVIX is limited by the omission of a few MiniGrid variants that cannot be immediately jittable. Both limitations are stated in Section 6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper does not include theoretical results.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Instructions on how to reproduce all experiments can be found in Section 4. Hyperparameters for the baselines can be found as described in Appendix E and in the config files at https://github.com/epignatelli/navix/tree/speedup/baselines/rejax/configs.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The full NAVIX code is avilable at https://github.com/epignatelli/navix. Documentation is available at https://epignatelli/navix. The folder https://github.com/epignatelli/navix/tree/speedup/benchmarks contains the code to run the experiments in Section 4.1 and 4.2. The code to reproduce the baselines is available at https://github.com/epignatelli/navix/tree/speedup/baselines/rejax.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Details on hyperparameters and training configurations can be found as described in Appendix E and at https://github.com/epignatelli/navix/tree/speedup/baselines/rejax/configs.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Results are accompanied by error bars, and their meaning is reported in the correspondig sections. In particular, shaded areas in Figure 7 show 5-95 percentile confidence intervals, as described in Section 4.3. Error bars in Figures 3, 4, 5, and6 represent standard errors as described in Section 4.1.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All experiments are run on a single Nvidia A100 80Gb, and Intel(R) Xeon(R) Silver 4310 CPU @ 2.10GHz and 128Gb of RAM., as stated in Section 4.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification:

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.

- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Section 5 analyses the broader impacts of NAVIX, including the positive effects of democratizing RL research, lowering energy use—and and the negative possibilities such as enabling faster development of malicious autonomous agents.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: NAVIX releases only an environment library, not a pretrained model or large scraped dataset. The code confers no novel dual-use capability beyond the long-established MiniGrid baseline, so additional safeguards are unnecessary.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: NAVIX is released under Apache 2.0, as stated on the main repository https://github.com/epignatelli/navix/blob/main/LICENSE, with the license file and copyright header included in the repository. MiniGrid is released under MIT license, as stated in the corresponding repo `https://github.com/Farama-Foundation/Minigrid/blob/master/LICENSE`. JAX is released under Apache 2.0 as stated in the corresponding repository `https://github.com/jax-ml/jax/blob/main/LICENSE`.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: Footnote 1 links to the public GitHub repository: https://github.com/epignatelli/navix. Code is everywhere documented, and API docs and installation instructtions are available at https://epignatelli/navix.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: No human-subject or crowd-sourcing studies were conducted.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.

- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: Not applicable—no human-subject research was performed.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification:

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.