JAXenstein: Accelerated Benchmarking for First-Person Environments

Ruo Yu Tao, George Konidaris
ruo_yu_tao@brown.edu

Department of Computer Science, Brown University

Abstract

The progression of reinforcement learning algorithms have been driven by challenging benchmarks. The rate in which a researcher can iterate on a problem setting directly impacts the speed of algorithm development. Modern machine learning has produced tools that allow for fast and scalable algorithm development like the JAX library. With the availability of these tools, a serious bottleneck in algorithm development is the availability of large and complex domains for experimentation. Most notably, the JAX reinforcement learning ecosystem does not have any benchmarks that test visual first-person tasks; these domains are crucial for testing both exploration and an agent's ability to overcome partial observability. We introduce JAXenstein[1] an open-source JAX-based benchmark that implements the Wolfenstein 3D rendering engine for fast and scalable experimentation in visual first-person tasks. JAXenstein is several times faster than comparable vision-based benchmarks, and is easily extensible to more complex first-person domains.

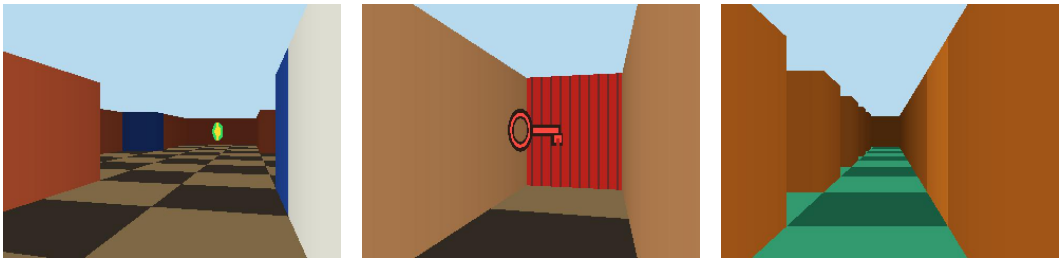
1 Introduction

The development of reinforcement learning algorithms have been driven by challenging benchmarks. Problem settings have preceded the development of algorithms, as they act as a foundation for algorithmic improvement. Most famously, the arcade learning environment (Bellemare et al., 2013) gave rise to deep Q-networks (Mnih et al., 2015) and discrete, deep value-based control, while the Mujoco benchmark (Todorov et al., 2012) was the testbed for deep continuous control and policy-gradient methods (Silver et al., 2014; Lillicrap et al., 2016; Schulman et al., 2017).

The speed of benchmarks directly affect the rate of experimentation and hence the speed of algorithmic development. A reinforcement learning research project will usually start with a faster, more lightweight benchmark, such as Cart Pole (Barto et al., 1983) or Mountain Car (Moore, 1990), to develop an algorithm before moving on to larger domains which require more time to train on. Historically, these larger domains requires much time to train algorithms. With the introduction of multiprocessing (Mnih et al., 2016) and GPU accelerated (Freeman et al., 2021) environments, the speed, efficiency and parallelizability of environments has made iterating on larger domains more accessible, albeit at the cost of more computation.

With the explosion in modern deep learning tools, a key bottleneck for algorithm development is the need for scalable, larger, more complex domains. Reinforcement learning experimentation has seen large speed and parallelization gains due to the adoption of the JAX (Bradbury et al., 2018) library, which can run experiments several times faster than one trained with traditional frameworks (Lu

[1] Code: https://anonymous.4open.science/r/JAXenstein-976D
Baselines: https://anonymous.4open.science/r/JAXenstein_baselines-C711



(a) `my-way-home` (b) `key-door` (c) `dmlab-static-03`

Figure 1: Example environments in the JAXenstein benchmark, entirely implemented in end-to-end JAX. Objects are rendered as fixed sprites, like in the original Wolfenstein 3D engine.

et al., 2022). This requires both the algorithm and environment be entirely implemented in this framework, which means benchmarks not implemented in JAX are unavailable or costly. Notably, first-person domains (Beattie et al., 2016; Wydmuch et al., 2019) that were originally introduced as candidate challenge problems have not been widely adopted in JAX frameworks due to the complexity in reimplementing rendering engines. These benchmarks are crucial for testing decision making in visual domains that both need to overcome partial observability and require good exploration.

We introduce JAXenstein: a lightweight JAX-based (Bradbury et al., 2018) implementation of the Wolfeinstein 3D engine for accelerated first-person vision-based environments. JAXenstein environments are simplified versions of first-person vision-based environments used in reinforcement learning benchmarking (Wydmuch et al., 2019; Beattie et al., 2016), foregoing visual rendering assets for speed. JAXenstein is entirely implemented in JAX with just-in-time (`jit`) compilation and vectorized mapping (`vmap`) support, and runs at $\sim 6\times$ the speed of the ViZDoom simulator on a single CPU core. Together with a purely end-to-end JAX pipeline, JAXenstein allows for fast, GPU-accelerated training many times faster than a comparable implementation with PyTorch.

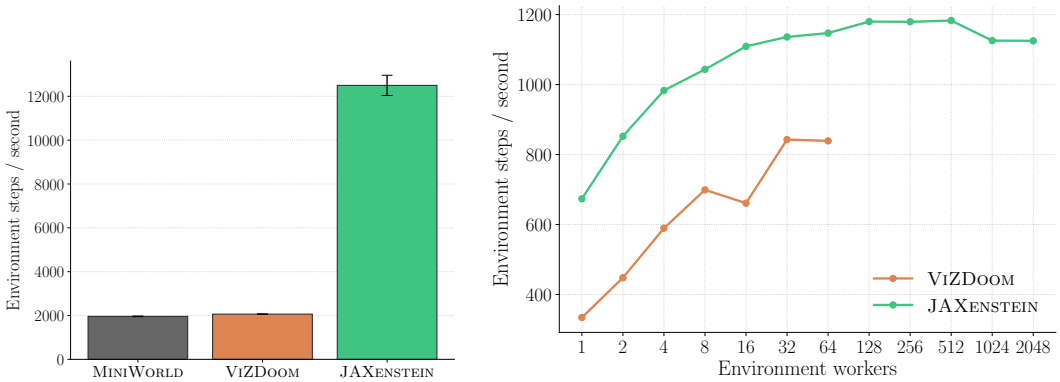
2 A simple JAX-native rendering engine

JAXenstein runs on a simple rendering engine that uses ray casting, first introduced in a production game in Wolfenstein 3D (id Software, 1992). JAXenstein uses this simple 3D rendering engine because its speed: ray casting is extremely fast in rendering frames for simple 3D environments based on tile maps, allowing for fast and scalable environment interactions. We show a few examples of this rendering in Figure 1. We use this front-end as a rendering engine to reimplement light and fast versions of existing first-person reinforcement learning benchmarks.

2.1 Ray casting

Ray casting produces the scene of an image from an observer by calculating, from each column in an image, the intersect of a ray cast from the observer to an intersection in the scene (e.g. a wall). We can use an appropriate shading color to give depth to the rasterization based on the distance of this intersection.

We calculate this intersection with the digital differential analyzer (DDA) algorithm (Watt, 2000). The DDA algorithm leverages the fact that we are in a tile-based world, and finds the intersection for a particular ray by "marching" along the tile boundaries that intersect with the ray. This requires far fewer steps in the ray casting algorithm as compared to marching along the ray in a set, small interval, and results in significantly fewer intersection checks.



(a) Steps-per-second comparison. JAXenstein is approximately 6× the speed of MiniWorld and ViZDoom.

(b) Training speed comparison between ViZDoom and JAXenstein. ViZDoom crashes after trying to parallelize > 64 environments.

Figure 2: Speed comparisons between JAXenstein and similar benchmarks.

2.2 Performance trade-offs

While ray casting with DDA is an extremely fast and simple algorithm for rendering, it does incur trade-offs. First and most importantly, it requires that the environment is a tile-based map. While fitting for many simple first-person environments, anything with more complex environment objects needs more complex algorithms that would necessarily slow down the simulator. This also implies that stairs, jumping, height differences are not possible to render with this engine. While extensions are possible for more complex objects (Mordvintsev, 2022), this comes at a computational cost that will affect per-timestep performance.

Nonetheless, this rendering engine is the first to introduce first-person environments into the JAX reinforcement learning ecosystem. Using this renderer, we port existing first-person environments used in reinforcement learning benchmarks for extremely fast and scalable experimentation.

3 JAXenstein: a benchmark suite for accelerated first-person environments

We introduce JAXenstein: a purely JAX-based benchmark for first-person environments based on the Wolfenstein 3D engine. This benchmark includes simplified reimplementations of popular first-person environments used throughout reinforcement learning research, such as ViZDoom (Wydmuch et al., 2019) and DeepMind Lab (Beattie et al., 2016), as well as completely new environments.

3.1 Runtime comparison

We conduct experiments to test the computational efficiency of JAXenstein compared to other benchmarks. We first compare JAXenstein run times to MiniWorld (Chevalier-Boisvert et al., 2023) and ViZDoom in Figure 2a. JAXenstein achieves around a 6× speed up in steps per second. Details of these runs are given in Section 5.

One reason JAXenstein achieves comparably faster speeds is due to the availability of native rendering at lower resolutions. Since images are usually downsampled to a smaller size in deep reinforcement learning algorithms, rendering at this smaller size allows for massive per-time-step speed ups, without the per-timestep cost of rescaling images.

We also compare a training time comparison when scaling up the number of parallel environments in Figure 2b. Using a base recurrent proximal policy optimization (PPO) (Schulman et al., 2017) algorithm over 1M steps, compare a pure JAX end-to-end training pipeline (Lu et al., 2022) compared

to ViZDoom with a Stable Baselines3 (Raffin et al., 2021) PPO implementation, the JAXenstein implementation achieves much higher environment steps per second. This, together with the ability for a pure JAX training loop to utilize `vmap` allows for both massive scaling and extremely fast experimentation with GPU acceleration. Training experiment details are in Section 5.2.

3.2 Environments

The ray casting engine affords many options for first-person environments. Since DDA ray casting uses a tile-based representation of the environment, this allows for a variety of options for reinforcement learning tasks, all with a common map definition format.

ASCII mazes JAXenstein includes functionality to convert any 2D maze defined over ASCII characters. We introduce the `simple` and `key-door` domains, which are simple navigation environments. JAXENSTSEIN allows for conversion of any ASCII maze into a first-person environment. All that is required to define an environment is to define a multi-line ASCII string which represents the map based on differing characters which map to elements in the map. A full definition of all characters are described in Section 6.1. Additionally, using this ASCII mapping we reimplement the MiniGrid `MiniGrid-KeyCorridorS4R3-v0` environment in JAXenstein as `key-corridor`.

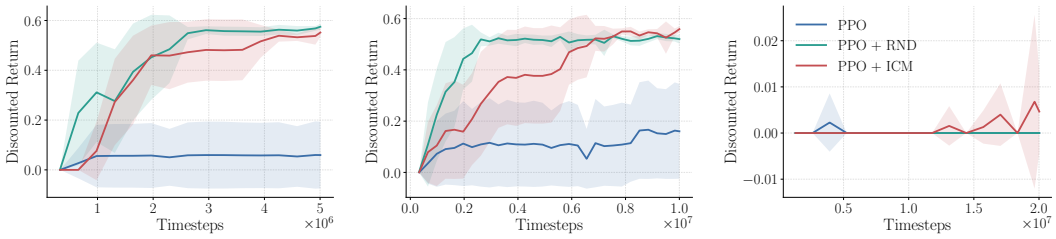
ViZDoom environments JAXenstein also includes its own version of ViZDoom (Wydmuch et al., 2019) environments. These include the My Way Home (`my-way-home`) and Health Gathering (`health-gathering`) environments. Environments that involve shooting and enemy interaction are also possible with the JAXenstein engine, but will be added in a later release.

DeepMind Lab navigation mazes To test hard first-person navigation environments, we port over the navigation domains from the DeepMind Lab benchmark (Beattie et al., 2016). These environments are over three maze configurations, with options `static` or `random` to denote whether the goal state is static or randomly sampled at each new episode. There are three maps (indexed `01`, `02` and `03`) of increasing size and complexity in this domain, giving us a total of six navigation mazes ported over from DeepMind Lab.

All implemented environments are listed in Table 1.

Group	Environment	ID	Description
Basic	Simple	`simple`	Small navigation task with one start and one goal.
Basic	Key-door	`key-door`	Collect a red key, open a red door, reach the goal.
MiniGrid	KeyCorridorS4R3	`key-corridor`	3×3 room key corridor with colored doors and a locked goal room.
ViZDoom	Health Gathering	`health-gathering`	Survive an acidic room by collecting medkits.
ViZDoom	My Way Home	`my-way-home`	Large maze with many starts, colored walls, and one goal.
DMLab	Static goal	`dmlab-static-{01,02,03}`	Fixed-goal mazes; `01` small, `02` medium, `03` large.
DMLab	Random goal	`dmlab-random-goal-{01,02,03}`	Same sizes; one randomly sampled goal candidate is active each episode.

Table 1: JAXenstein environments.



(a) `simple` (b) `key-door` (c) `dmlab-static-01`

Figure 3: Baseline results across JAXenstein environments. Runs were conducted on the recurrent PPO algorithm with different exploration strategies. Full experimental details in Section 7.

3.3 Baseline results

We run experiments with the recurrent PPO (Schulman et al., 2017) algorithm augmented with different exploration algorithms in Figure 3. Experiments were conducted on a selection of the environments available in the JAXenstein benchmark. `simple` and `key-door` were included as easy sanity-checking environments. Interestingly, performance of these three algorithms are all poor in the `dmlab-static-01` environment, which is the easiest maze navigation environment among all the benchmarks. This reveals the need for better memory mechanisms and exploration methods for first-person environments.

Environments in JAXenstein require both recurrency and exploration to solve tasks. Due to the large and complex observation space, and the partial observability of the environment, JAXenstein is a benchmark that tests the memory capabilities of an algorithm, as well as the exploration capabilities when dealing with partial observability. Even in the relatively small `simple` environment, an agent that does not leverage good exploration strategies are not able to learn the task. More involved exploration algorithms like random network distillation (Burda et al., 2019) and intrinsic curiosity module (Pathak et al., 2017) are required to solve this simple task due to its large and complex state/observation space.

4 Conclusion

We present JAXenstein, a fast and scalable visual first-person benchmark based on the Wolfenstein 3D rendering engine. JAXenstein is entirely implemented in JAX, allowing for fast GPU accelerated experimentation. JAXenstein reimplements a lightweight version of existing visual first-person environments from existing benchmarks like ViZDoom and DeepMind Lab, and also converts 2D gridworld environments into first-person. In the future, we hope to implement more functionality to the environment that would allow for more complex environments, like adding non-playable characters and movable objects.

References

Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013.

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

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2019.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

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.

id Software. Wolfenstein 3D, 1992.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun (eds.), *ICLR*, 2016.

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.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

Andrew William Moore. Efficient memory-based learning for robot control. Technical report, University of Cambridge, 1990.

Alexander Mordvintsev. Simple 3d visualization with jax raycasting, 2022. URL https://github.com/schmidtdominik/jax-raytracing.

Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*, 2017.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pp. I–387–I–395. JMLR.org, 2014.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

Alan Watt. *3D Computer Graphics*. Addison-Wesley, Harlow, England, 3rd edition, 2000. ISBN 0-201-39855-9.

Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. ViZDoom Competitions: Playing Doom from Pixels. *IEEE Transactions on Games*, 11(3):248–259, 2019. The 2022 IEEE Transactions on Games Outstanding Paper Award.

Supplementary Materials

The following content was not necessarily subject to peer review.

5 Runtime comparison details

All runtime comparisons were done on a single desktop with AMD Ryzen 9 5900X process with 32GB of system memory, and an NVIDIA RTX3090 with 24GB of VRAM.

5.1 Steps per second

We detail the runtime experiments in Figure 2a. A uniform random policy was unrolled for all 3 benchmark environments. JAXenstein ran the `my-way-home` environment, ViZDoom ran the `ViZDoomMyWayHome-v1` environment, whereas MiniWorld ran the `MiniWorld-Hallway-v0` environment. While the large discrepancy for MiniWorld may seem an issue, we note that the environment ran for this library was much simpler than the My Way Home environment, and should be much faster to run than a comparable version of My Way Home with the same engine, since the number of rooms is substantially lower. This environment should be considered an upper-bound for environment steps per second on the MiniWorld benchmark.

We also note that MiniWorld and ViZDoom also require an image resizing down to the 64×64 we require for training a reinforcement learning agent. We plot the same environment step comparison without image resizing in Figure 4. While we do see a modest increase in run time for ViZDoom, the JAXenstein library still achieves an approximately $3\times$ speed up over ViZDoom.

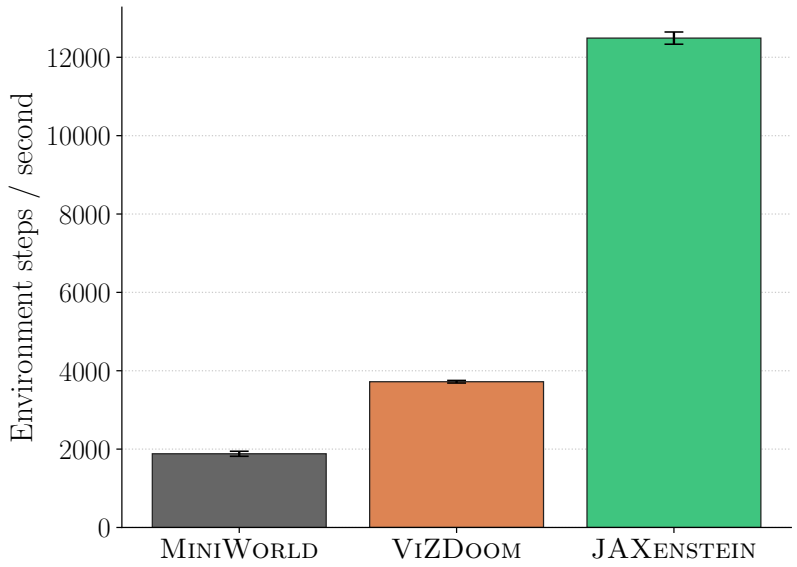


Figure 4: Steps per second without image resizing.

5.2 Training speed comparison

For the training speed comparison, we time the speed it takes different implementations of the recurrent PPO algorithms with different benchmarks to train for 1M steps. For the JAXenstein runs, we run an implementation of recurrent PPO in JAX (Lu et al., 2022) on the `my-way-home` environment. For the ViZDoom runs, we use an implementation of recurrent PPO in PyTorch (Raffin et al., 2021) with the ViZDoom benchmark on `VizdoomMyWayHome-v1`. For both implementations, we match network architectures and hyperparameters between the two environments, as well

as down-sample the ViZDoom environment frames to 64×64. As for the JAXenstein environment, we natively render at 64×64. Full implementation details are provided in the open-source baselines codebase.

6 JAXenstein environment details

6.1 ASCII maze details

Below we define the different characters and what they map to in a JAXenstein environment.

Table 2: JAXenstein ASCII map symbols.

Symbol	Meaning
`#`	Default static wall
`1-9`, generated DMLab symbols	Colored static wall
`.`	Floor
`space`	Floor
`S`	Spawn candidate
`G`	Yellow goal candidate
`r`	Red key
`b`	Blue key
`y`	Yellow key
`"`	Blue unlocked door
`\`	Yellow unlocked door
`R`	Red-locked door
`B`	Blue-locked door
`Y`	Yellow-locked door

7 Baseline experiment details

For these baseline experiments, we conduct experiments over 5 seeds and use the following set of hyperparameters for PPO:

Table 3: PPO hyperparameters.

Hyperparameter	Value
Rollout steps	128
PPO epochs	4
Image size	64×64
Learning rate	2.5×10^{-4}
GAE λ	0.95
Entropy coefficient	0.01
Value loss coefficient	0.5
Max gradient norm	0.5
GRU hidden size	256
CNN feature dimension	256
Previous action concatenation	True
Discount factor γ	Environment default

Environment-specific hyperparameters are listed in the open-source baselines codebase. We also use and sweep over the following hyperparameters for RND and ICM:

Table 4: RND hyperparameters.

Hyperparameter	Value(s)
RND reward coefficient	$\{0.1, 1.0, 10.0\}$
RND loss coefficient	$\{0.01, 0.1\}$
RND hidden size	128
RND output size	128
RND number of layers	2
Intrinsic advantage coefficient	1.0

Table 5: ICM hyperparameters.

Hyperparameter	Value(s)
ICM learning rate	$\{3 \times 10^{-4}, 1 \times 10^{-4}\}$
ICM reward coefficient	$\{0.1, 1.0, 10.0\}$
ICM latent size	128
ICM hidden size	256
ICM update epochs	1