minimax: Efficient Baselines for Autocurricula in JAX

Minqi Jiang
FAIR at Meta AI
msj@meta.com

Michael Dennis
UC Berkeley

Edward Grefenstette
UCL

Tim Rocktäschel
UCL

Abstract

Unsupervised environment design (UED) is a form of automatic curriculum learning for training robust decision-making agents to zero-shot transfer into unseen environments. Such autocurricula have received much interest from the RL community. However, UED experiments, based on CPU rollouts and GPU model updates, have often required several weeks of training. This compute requirement is a major obstacle to rapid innovation for the field. This work introduces the `minimax` library for UED training on accelerated hardware. Using JAX to implement fully-tensorized environments and autocurriculum algorithms, `minimax` allows the entire training loop to be compiled for hardware acceleration. To provide a petri dish for rapid experimentation, `minimax` includes a tensorized grid-world based on `MiniGrid`, in addition to reusable abstractions for conducting autocurricula in procedurally-generated environments. With these components, `minimax` provides strong UED baselines, including new parallelized variants, which achieve over 120× speedups in wall time compared to previous implementations when training with equal batch sizes. The `minimax` library is available under the Apache 2.0 license at https://github.com/facebookresearch/minimax.

1 Introduction and Motivating Work

Figure 1: Wall time *speed-up factors* achieved by `minimax` relative to PyTorch reference implementations with equal batch sizes (mean and std of 10 runs). Experiments that took 100+ hours now finish in < 3 hours on a single GPU. Here, +S5 indicates the use of an S5 policy, and +P, parallel DCD. (See Section 4 for details.)

Autocurricula have proven highly effective in producing powerful deep reinforcement learning (RL) agents in complex multi-agent settings [54, 61, 2, 17, 58, 46]. Here, a self-organizing curriculum across co-players emerges as agents compete with one another over billions of iterations [28]. The burgeoning field of *Unsupervised Environment Design* [UED, 10] recently extends these ideas to the design of the training task itself. In UED, the learning agent or the *student*, plays against a *teacher* in a curriculum game. In each episode of this game, the teacher chooses training tasks or environments in order to maximize some metric based on the student's behavior. A principled choice

2nd Agent Learning in Open-Endedness Workshop (ALOE 2023).

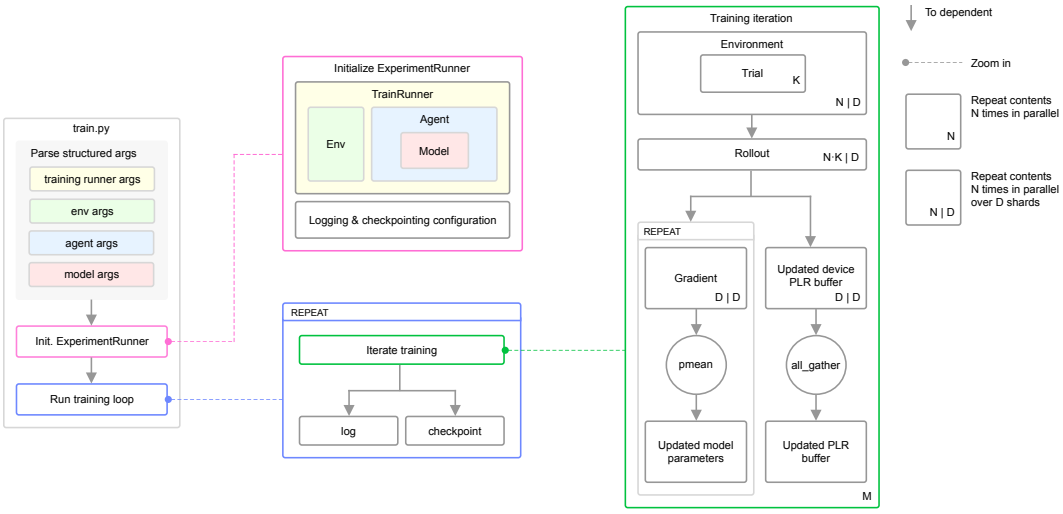

Figure 2: A high-level overview of the `minimax` library. The training iteration logic is fully-jitted.

for the teacher's objective is the student's regret [48, 10], leading to minimax-regret optimal students should the curriculum game reach an equilibrium.

Many works have demonstrated that UED produces more robust agents in terms of zero-shot transfer to out-of-distribution (OOD) tasks [10, 20, 36, 21, 57]. While empirically effective, this class of autocurriculum methods can be computationally expensive. Results on a common maze navigation benchmark, featured in the majority of works on UED, can require on the order of $10^8 - 10^9$ steps to reproduce. Such experiments can take up to one week to complete, when using the standard UED baselines from the open-source, PyTorch-based `dcd` codebase [20] with a V100 GPU—a considerable outlay of compute that is especially taxing for more GPU-poor academic labs. In this technical report, we present `minimax`, a fast and modular library for accelerating the pace of research in UED, as well as other forms of autocurricula [32, 39, 40, 12]. Crucially, `minimax` derives its speed from the powerful JAX library [14] for vector transforms on top of XLA [27]. In addition to a modular library of components for assembling, extending, and evaluating new autocurriculum methods, `minimax` features fully-tensorized implementations of the maze benchmark and standard UED baselines, as well as new parallelized and multi-device variants, achieving over $120\times$ speed-ups in wall time compared to those from `dcd`, as reported in Figure 1.

The main contributions of this work are presented in Sections 2 – 4:

- A detailed discussion of the design philosophy and high-level structure of the `minimax` library is provided in Section 2.
- `AMaze`, a fully-tensorized procedurally-generated, partially-observable maze navigation benchmark for use as a rapid petri dish for evaluating autocurricula is presented in Section 3.
- All `minimax` baselines, including novel parallelized and multi-device variants of PLR$^\perp$ and ACCEL, are presented and benchmarked in Section 4.

2 The `minimax` Library

A foundational principle behind `minimax` is that modularity is crucial for rapid experimentation. Thus, from an architectural standpoint, `minimax` centers around strongly decoupled and broadly interoperable atomic building blocks, consisting of training runners, environments, agents, and models. Each of these building block categories exists as its own submodule. Any component can be easily accessed via a registry interface (similar to the environment registry in `OpenAI Gym` [7]), which allows necessary dependencies to be explicitly stated and retrieved, e.g. setting and getting a policy model for a specific environment. In general, dependent components are made generic via dependency injection [51] where possible. An overview of `minimax` is presented in Figure 2.

In order to model the diverse world of autocurriculum methods in a modular way, `minimax` adopts the conceptual structure of *Dual Curriculum Design* [DCD, 20]. DCD describes many autocurricula,

including common UED baselines, as unfolding as a game between a student and two kinds of teachers: a generator that can adaptively design tasks and a curator that selectively curates those generated. For example, domain randomization [DR, 60, 45] corresponds to a random generator (i.e. no adaptation) with no curator; PAIRED [10], to an adaptive generator with no curator; and PLR [22, 20] to a curator with a random generator. Each high-level combination of DCD teachers can describe many subsets of UED methods. Thus, `minimax` structures autocurricula as curriculum games and supports extensions to multi-student or multi-teacher games. In the sections that follow, we discuss the role of each of `minimax`'s building blocks in constructing such autocurricula.

2.1 Runners

Each runner orchestrates a specific kind of curriculum game by coordinating agent and environment components through the registry system. Many autocurricula methods can already be implemented as simple extensions of the runners in the initial release of `minimax`: `DRRunner`, `PAIREDRunner`, and `PLRRunner`, corresponding to DCD autocurricula based on a random teacher, learned generator, and curator respectively. We refer to each algorithmic runner as a *training runner*. A separate `EvalRunner` performs evaluation of model checkpoints on a prespecified set of test environments.

To minimize configuration overhead, `minimax` follows a highly-declarative approach: A small bundle of logic in a single `train.py` file executes all algorithms. This design is enabled by a custom argument parser that defines how command-line arguments are packaged into initialization arguments for training runners, agents, models, and evaluators. In practice, this approach pays further dividends in simplifying the addition of new components by removing low-level argument passing from mind. Moreover, the final argument schema serves as a legible blueprint of the corresponding experiment.

Parsed argument groups are passed to their destinations by `ExperimentRunner`. In addition to managing logging, checkpointing, and periodic calls to `EvalRunner`, this higher-level runner progresses training by stepping the `run` method of the training runner, which executes one iteration of the autocurriculum (i.e. one rollout and update cycle for each participating agent). Each `run` method is fully jittable and can thus be `vmap` transformed to conduct parallel, independent training runs.[1]

2.2 Environments

The `minimax` library environment interface is based on that from the `Gymnax` [25] framework, but departs in several ways: First, `minimax` supports environment wrappers, which carry their own state. This state, is passed across timesteps via an additional `extra` dictionary in the tuple returned by environments' `step` and `reset` methods. This design shields wrapper-specific metadata from environment-specific metadata, typically conveyed within the `info` component of the tuple. Second, to measure how distributions of environment metrics evolve over time, `minimax` environments can optionally implement the `get_env_metrics` method, which returns a dictionary of environment attributes, e.g. number of walls in a maze. We now describe two more consequential design choices in how `minimax` handles environments.

Hierarchical parallelism Usefully, `minimax` directly supports environment parallelism across a hierarchy via the `BatchEnv` decorator class: agents (i.e. the population batch), evaluations, and environments. Specifically, the last two environment batch dimensions are flattened inside `BatchEnv` instances. This grouping allows tidy implementations of multi-student or multi-teacher training logic, as well as parallelizing environment logic across not only multiple environment instances, but also evaluations of each instance, e.g. to obtain a denoised estimate per instance. Crucially, environment logic is specified, as normally, for a single instance, and all parallelism is abstracted into `BatchEnv`.

UPOMDP as a first-class citizen Prior RL libraries are designed to support standard (partially-observable) Markov decision processes [MDPs, 41]. However, autocurriculum methods typically operate over an extension, called an *Underspecified POMDP* [UPOMDP, 10], which explicitly considers the set of configurable free parameters of the environment, which these prior libraries ignore. In `minimax`, environment parameters are separated into *static parameters*, which define fixed aspects set at initialization, e.g. maze size, and free parameters, which vary per instance and are

[1] However this should be avoided for some algorithms to allow `lax.cond` to perform efficient branching, e.g. as used in PLR$^{\perp}$ updates, that is currently foregone by XLA once inside `vmap`.

stored in the environment state. Thus, to fully exploit free parameters in generating autocurricula, `minimax` environments implement getter and setter methods for the environment state.

UED with a learned adversary, e.g. PAIRED, requires the teacher to make a sequence of design decisions in a separate MDP that results in producing a specific environment instance. To directly model such teacher MDPs, `minimax` provides the `UEDEnvironment` class, which mirrors the student's `Environment` interface for the teacher. Each `UEDEnvironment` class is directly registered to match with its corresponding `Environment` class. The `BatchUEDEnv` decorator class then takes the pair of environments and implements vectorized environment logic via a shared student and teacher interface: Once the teacher steps through its rollout, calling `BatchUEDEnv`'s `reset_student` method sets the student's UPOMDP to the final instance designed by the teacher. This approach allows for cleaner, modular development of student and teacher decision processes, while allowing for a simple, shared environment abstraction for use in training runners.

2.3 Agents and Models

In `minimax`, an agent corresponds to a specific algorithm for optimizing a sequential-decision making policy, while a model corresponds to a module implementing the underlying policy. Following a common pattern in `minimax`, models are dependency-injected into `Agent` instances. This design allows the same implementation of a `PPOAgent` to perform PPO over any policy model for any environment. In general, training runners operate over populations of agents, and any individual `Agent` can be transformed into a population via the `AgentPop` decorator class. Combined with the batch environment classes, these abstractions make it simple to extend future releases of `minimax` to support more complex multi-agent settings [19, 24, 61, 47].

3 Maze as an Accelerated Petri Dish

Mazes are a simple and effective setting for studying autocurricula: Mazes exhibit interpretable difficulty gradients (in terms of initial distance to the goal and number of obstacles) and agent behavior. Moreover, the combinatorial design space of potential obstacle configurations serves as a rich and easily extensible domain for UED. Importantly, many difficult problems in RL, such as decision-making under partial observability and generalization to unseen task instances can be directly studied in this domain. A large share of prior works on autocurricula have thus used 2D mazes to initially vet algorithmic ideas and other hypotheses [18, 42, 8, 63, 10, 22].

Thus, we include a procedurally-generated 2D maze environment in `minimax`. This environment, which we call `AMaze`, is a fully-tensorized implementation of goal-reaching mazes based on `MiniGrid` [9]. In its basic form, `AMaze` provides a challenging, partially-observable navigation task for RL agents. It can be configured to any rectangular grid layout with randomly sampled wall, goal, and agent positions (see Figure 3). Further, `AMaze` is fully compatible with the `UEDEnvironment` abstraction in `minimax`, allowing full support for UED with learned teachers and directly setting free parameters (i.e. the tile map). Most importantly, `AMaze` is fast. As seen in Table 1, even with an environment batch size of 1, `AMaze` runs 60% faster in terms of steps per second (SPS). As environment parallelism increases, `AMaze` takes greater advantage of XLA parallelism, achieving speedup factors of $136\times$ at 1024 parallel environments.

Table 1: Speed comparison of `AMaze` and `MiniGrid` mazes for varying degrees of parallelism.

# Environments	**1**	**32**	**256**	**1024**
Steps-per-second (SPS)				
`MiniGrid`	2k	12k	16k	16k
`AMaze`	3k	76k	582k	2M
Speedup	1.6×	6.5×	37×	136×

In order to compute shortest paths within a fully-jitted `run` method, `AMaze` makes use of Seidel's Algorithm [52], a matrix-based all pairs shortest path algorithm with time complexity $O(V^\omega \log V)$. Here, V is the number of vertices and $\omega < 2.373$ is the exponent in the computational complexity of matrix multiplication. This all-pairs approach further enables efficiently computing more exotic complexity metrics such as shortest-paths distributions [59] or resistance distance distributions [43].

Importantly, `AMaze` exactly replicates the `MiniGrid`-based mazes in prior works, including the full OOD benchmark [20], enabling direct comparisons with `minimax` implementations.

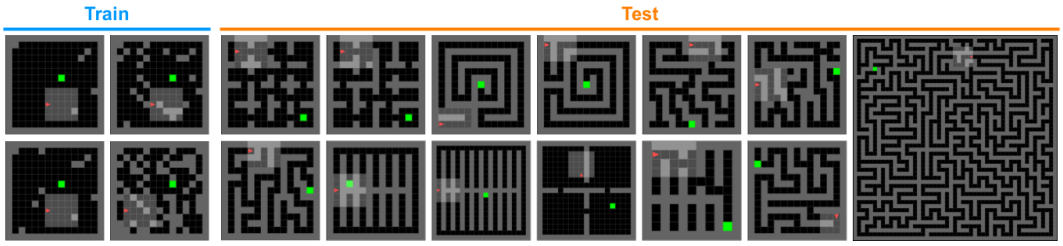

Figure 3: Example training and test environments in `AMaze`, a fully-tensorized maze environment in JAX.

Table 2: Comparison of wall time and task performance (mean and std of 10 runs) between `minimax` and `dcd`, based on training for 30k PPO updates. Corresponding runs use equal PLR replay rate and PPO minibatch and epoch settings. PLR$^{\parallel}$ and ACCEL$^{\parallel}$ compare to PLR$^{\perp}$ and ACCEL in `dcd` respectively.

	DR	**PAIRED**	**PLR$^{\perp}$**	**PLR$^{\parallel}$**	**ACCEL**	**ACCEL$^{\parallel}$**
Wall time (hours)						
`dcd`	63 ± 2	426 ± 47	119 ± 0	–	104 ± 8	–
`minimax`	3 ± 0	7 ± 0	5 ± 0	3 ± 0	4 ± 0	3 ± 0
Speedup	$20\times$	$62\times$	$26\times$	$36\times$	$24\times$	$32\times$
Solved rate						
`dcd`	0.62 ± 0.05	0.52 ± 0.13	0.71 ± 0.04	–	0.75 ± 0.03	–
`minimax`	0.55 ± 0.05	0.63 ± 0.04	0.70 ± 0.03	0.74×0.04	0.73 ± 0.05	0.74 ± 0.04
Relative solved rate	$0.88\times$	$1.22\times$	$0.99\times$	$1.04\times$	$0.99\times$	$1.00\times$

4 Efficient Baselines

The central aim of `minimax` is to provide fast open-source implementations of strong autocurriculum baselines. In this way, we seek to remove the computational overhead for researchers to collectively innovate on this exciting algorithmic frontier. Table 3 summarizes the current baselines in `minimax`. Notably, we introduce new parallelized variants of PLR$^{\perp}$ and ACCEL that, as described later in this section, achieve additional wall-time gains compared to the PyTorch [37] reference implementations [20, 36]. All evaluations used V100 GPUs and Intel Xeon E5-2698 v4 CPUs.

UED baselines Currently `minimax` contains baselines that are variants on three core algorithms: domain randomization (DR), PAIRED, and Prioritized Level Replay (PLR). Important variants of these include Robust PLR (PLR$^{\perp}$), which only updates the agent after replay episodes; ACCEL, which replaces PLR$^{\perp}$'s random search with evolutionary search over the level buffer; and Population PAIRED, which uses a learned teacher to maximize relative population regret, as defined by the maximum return achieved by any individual minus the mean performance across the population of $N \geq 2$ students. Our PAIRED evaluations compare to the stronger "high-entropy" baseline [34]. In `minimax`, the main rollout and update loop of each algorithm is fully jitted via JAX, and environment parallelism is accomplished via the vectorizing decorator classes described in Section 2. Importantly, `minimax` avoids the computational overhead of the multiple student rollouts required by PAIRED variants by parallelizing all student rollouts. Under equal batch sizes, these optimizations result in consistent wall time speedups of at least $20\times$ or higher across all baselines, in comparison to the `dcd` library, the previous reference implementation in PyTorch (see Tables 2 and 4), while maintaining comparable test performance on the full OOD maze benchmark (see Figure 7). Practically speaking, these speedups mean that experiments that once took hundreds of hours to finish can now be done in just a few hours on a single GPU. Figure 9 in Appendix A compares the absolute solved rate of each method. Appendix B details the choice of hyperparameters and model architectures.

Table 3: Baselines currently implemented in `minimax`.

Algorithm	**Reference**	**Runner**
DR	Tobin et al, 2019 [60]	`dr`
Minimax UED	Dennis et al, 2020 [10]	`paired`
PAIRED	Dennis et al, 2020 [10]	`paired`
Pop. PAIRED	Dennis et al, 2020 [10]	`paired`
PLR	Jiang et al, 2021 [22]	`plr`
Robust PLR	Jiang et al, 2021 [20]	`plr`
ACCEL	Parker-Holder et al, 2022 [36]	`plr`
Parallel PLR	Introduced in this work	`plr`
Parallel ACCEL	Introduced in this work	`plr`

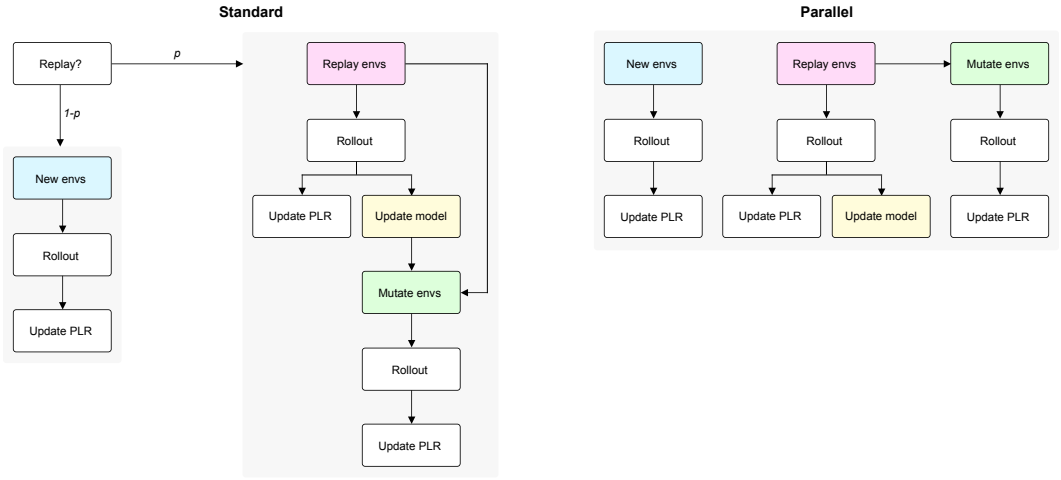

Figure 5: Left: The sequence of operations in standard implementations of PLR$^{\perp}$ and ACCEL. Right: PLR$^{\parallel}$ and ACCEL$^{\parallel}$ reduce wall time by executing rollouts for new levels, replay levels, and mutated levels in parallel.

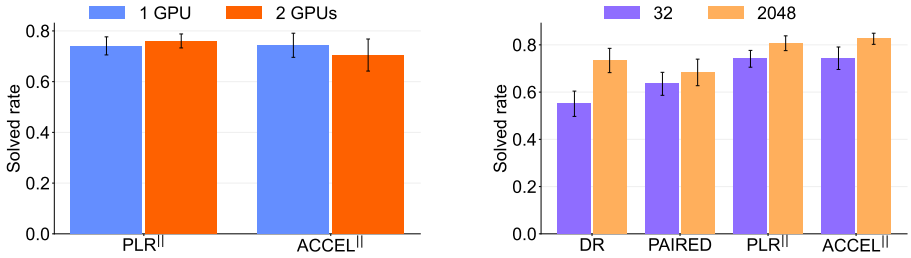

Figure 6: Solved rate over the full OOD maze benchmark for PLR$^{\parallel}$ and ACCEL$^{\parallel}$ using 32 parallel environments split across either one or two GPUs (left) and for each method for 32 vs. 2048 parallel environments (right). The plots show means and std across 10 training runs.

Parallel DCD Next, we seek to push speedups even higher by introducing Parallel PLR (PLR$^{\parallel}$) and Parallel ACCEL (ACCEL$^{\parallel}$). The key insight behind these two methods is that both search and replay steps of PLR$^{\perp}$-based methods can be run fully in parallel, allowing these methods to take full advantage of hardware-accelerated parallelism. Specifically, PLR$^{\parallel}$ doubles the environment batch size, filling the first half with newly-sampled environment instances and the second, with replay levels. PLR$^{\parallel}$ then evaluates and updates its level buffer with this full batch. ACCEL$^{\parallel}$ follows exactly the same approach with a tripled environment batch size. The first two thirds of the buffer are evenly split between newly sampled and replay levels, while the last third consists of mutations of replay levels from the same batch (see Figure 5). We find that PLR$^{\parallel}$ and ACCEL$^{\parallel}$ show wall time speedups compared to their standard counterparts of 38% and 33% respectively (see Table 2). This boost results in these replay-based UED matching simple DR in wall time. Figure 4 shows that the parallel variants also ratchet up the shortest path length as training progresses, with PLR$^{\parallel}$ notably overtaking PLR$^{\perp}$.

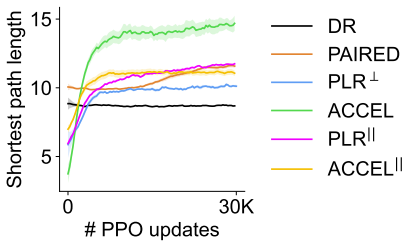

Figure 4: Shortest path lengths of training mazes per method (mean and std of 10 runs).

S5 policies We replace the LSTM in the student policy with a stack of S5 layers [55]. S5 is a variant of the structured state-space sequence model [15], which is $O(\log L)$ in the backward pass for a sequence of length L. S5 has been shown to significantly outperform LSTMs in both wall-time efficiency during training and performance on RL tasks requiring conditioning on a long history [31]. We report the wall-time speedups and performance on the full OOD maze benchmark in Figure 1

Table 4: Comparison of wall time and task performance (mean and std of 10 runs) between `minimax` and `dcd` with S5 policies, based on training for 30k PPO updates. Corresponding runs use equal PLR replay rate and PPO minibatch and epoch settings. PLR$^{\parallel}$+S5 and ACCEL$^{\parallel}$+S5 compare to PLR$^{\perp}$ and ACCEL in `dcd` respectively.

	DR	**PAIRED**	**PLR$^{\perp}$**	**PLR$^{\parallel}$**	**ACCEL**	**ACCEL$^{\parallel}$**
Wall time (hours)						
`dcd`	63 ± 2	426 ± 47	119 ± 0	–	104 ± 8	–
`minimax` + S5 policy	2 ± 0	3 ± 0	4 ± 0	3 ± 0	4 ± 0	3 ± 0
Speedup	$28\times$	$125\times$	$27\times$	$47\times$	$25\times$	$40\times$
Solved rate						
`dcd`	0.62 ± 0.05	0.52 ± 0.13	0.71 ± 0.04	–	0.75 ± 0.03	–
`minimax` + S5 policy	0.58 ± 0.05	0.58 ± 0.06	0.66 ± 0.04	0.73×0.04	0.72 ± 0.06	0.74 ± 0.04
Relative solved rate	$0.94\times$	$1.12\times$	$0.93\times$	$1.02\times$	$0.98\times$	$0.99\times$

and Table 4. The performance of `minimax` baselines, for both S5 and LSTM policies, relative to the corresponding LSTM-based PyTorch reference implementation in `dcd`, is shown in Figure 7.

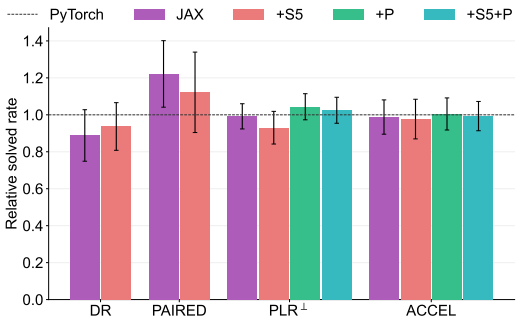

Figure 7: Relative solved rates across `AMaze` test mazes compared to PyTorch `dcd` (mean and std of 10 runs). The label +P indicates parallel DCD.

While S5 policies match the test performance of LSTM policies, they tend to exhibit greater sensitivity to hyperparameters, like learning rate, and higher variance across training runs. We find that, unlike in Lu et al, 2022 [31], which focuses on continuous-control tasks, applying layer normalization [1] to either the input or output of each S5 layer is essential for matching LSTM test performance in the maze domain. (Maze navigation episodes are limited 250 time steps, which can explain why S5 fails to outperform LSTM in test performance.) Likely, other design choices in the S5 architecture will lead to further improvements in generalization performance and reduced sensitivity to hyperparameters. Given the wall-time speedups at training when switching to policies based on structured state-space models, principled modifications of this architectural family for the RL setting is a promising area for future research.

Multi-device training All training runners in `minimax` support sharding environment rollouts and gradient computation across multiple devices. A high-level diagram of how each step of the training runner's update cycle is sharded along the environment batch dimension across devices is shown in Figure 2. Notably, the standard implementations of PLR-based methods like PLR$^{\perp}$ and ACCEL entail an `all_gather` bottleneck when updating the PLR buffer. We avoid this issue by introducing a new variant of these methods, whereby separate PLR buffers are maintained per device throughout training, effectively sharding the buffer across devices, such that the sum of all individual buffer sizes equals the original, unsharded buffer size. For D devices and an unsharded PLR buffer of size B, this approach results in D independent PLR buffers, each of size B/D and updated with only the environment instances run on its own shard. Figure 6 shows that these *synchronous data-parallel* (SDP) variants and the standard PLR-based methods result in comparable zero-shot transfer performance on the full OOD maze benchmark.

Batch-size scaling The multi-device baselines in `minimax` allow straightforward scaling to much larger environment batch sizes via the `shmap` (shard map) transform in JAX. We investigate the zero-shot transfer performance of DR, PAIRED, PLR$^{\parallel}$, and ACCEL$^{\parallel}$ on OOD test mazes with increasing training batch sizes. Figure 6 shows that increasing the environment batch size to 2048 parallel environments leads to significant improvements in zero-shot solved rates over the full OOD maze benchmark. This phenomenon aligns with previous theoretical and empirical observations of how increasing training batch size results in improved signal-to-noise ratio in the stochastic gradient estimates [33]. This results in a general trend in which larger batch sizes tend to require fewer updates to achieve the same degree of performance, while requiring more total number of samples (in this

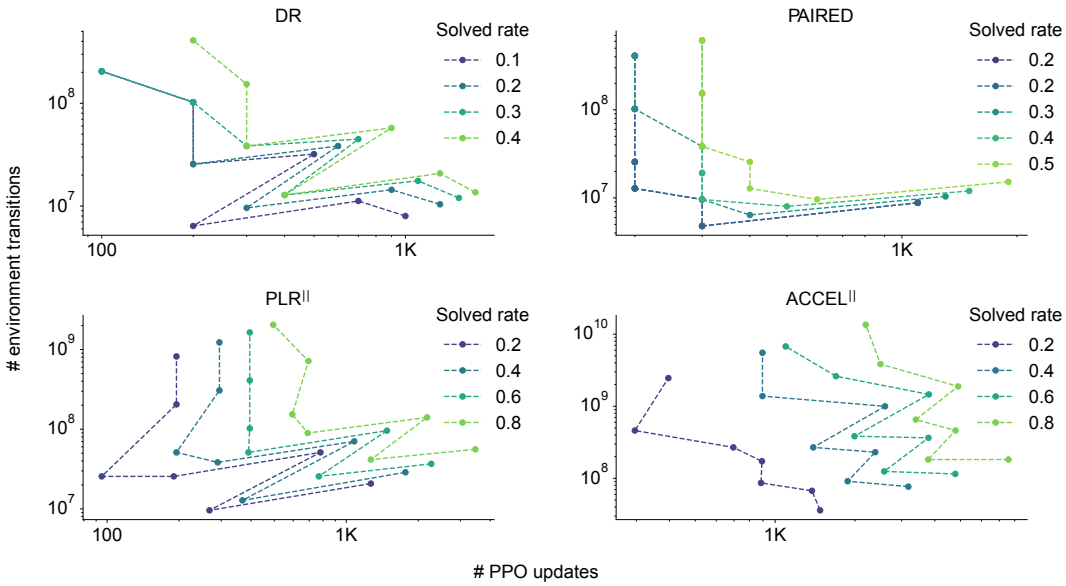

Figure 8: Minimum number of environment transitions and PPO updates needed to achieve a specific mean solved rate on validation mazes. All axes are in log scale. Each point is a mean across 10 training runs.

case, environment transitions). Figure 8 visualizes this trade-off by plotting the empirical relationship between the minimum number of environment transitions and minimum number of PPO updates required to reach a fixed degree of performance on three fixed OOD validation mazes, based on training with 32, 64, 128, 256, 512, 1024, and 2048 parallel environments per rollout. Here, each fixed performance curve tends to move from upper left to lower right, reflecting how larger batch sizes tend to be more update efficient, but less sample efficient. These results show that, independent of more sophisticated methodological changes, simply scaling the training batch size can result in significant improvements in OOD test performance when training with autocurriculum methods.

5 Related Work

Many recent works implement fast, GPU-accelerated RL algorithms and environments in JAX [6]. They include Brax [13], a differentiable physics environment; Jumanji [5], a collection of combinatorial optimization problems; gymnax [25], a JAX library for single-agent RL training with ports of the popular MinAtar suite [62] and BSuite [35]; JaxMARL [44], a collection of JAX-based multi-agent RL environments; `evosax` [26], a collection of evolutionary optimization methods; and PureJAXRL [29, 30], a minimalist framework for both evolutionary optimization and RL. Several previous works have also considered optimizing the speed of experience collection by implementing fast, asynchronous rollout workers [56, 4, 11, 23, 16]. This approach has been combined with highly GPU-optimized environment implementations to achieve impressive throughput [3, 38, 53].

6 Discussion

We introduced `minimax`, a fast JAX-based library that enables rapid experimentation in autocurriculum research for RL. By driving down computational costs, we hope our work can accelerate further progress in this exciting field. We highlighted key features of `minimax`, namely its modular structure and its associated experimental playground for rapid iteration—a fully-tensorized, hardware-accelerated procedural maze environment. Building on these components, our JAX-based autocurriculum baselines, including new parallelized and multi-device variants, resulted in training runs that were up to $120\times$ faster in wall time than the PyTorch reference implementations under the same training batch size. Experiments that once required hundreds of hours of compute can now finish in just a couple of hours on a single GPU. Crucially, our evaluations demonstrated that the `minimax` baselines either match or exceed the test performance of previous reference implementations, while drastically shrinking the timescale of autocurriculum research.

Author Contributions

MJ led the project from conception. He designed and implemented the library, formulated the new parallel variants of PLR$^{\perp}$ and ACCEL, conducted the experiments, and led paper writing. MD, EG, and TR provided invaluable feedback and suggestions. MD also assisted with the design of the example notebooks. EG first broached the idea of parallelizing the new and replay level evaluation branches in PLR. TR strongly encouraged writing this manuscript to accompany the code release.

Acknowledgments and Disclosure of Funding

We thank Chris Lu and Robert Lange for their work on scaling up JAX-based RL, which greatly influenced `minimax`. We further thank Chris Lu and Ben Ellis for conversations that benefited this work. Releasing `minimax` under the Apache 2.0 license would not be possible without the support of Leon Bottou, Naila Murray, and Kerry Andken at Meta AI. MJ is funded by Meta AI.

References

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch. Emergent tool use from multi-agent autocurricula. In *International Conference on Learning Representations*, 2019.

[3] C. Bamford. Griddly: A platform for ai research in games. *Software Impacts*, 8:100066, 2021.

[4] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[5] C. Bonnet, D. Luo, D. Byrne, S. Surana, V. Coyette, P. Duckworth, L. I. Midgley, T. Kalloniatis, S. Abramowitz, C. N. Waters, et al. Jumanji: a diverse suite of scalable reinforcement learning environments in jax. *arXiv preprint arXiv:2306.09884*, 2023.

[6] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, et al. Jax: Autograd and xla. *Astrophysics Source Code Library*, pages ascl–2111, 2021.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[8] A. Campero, R. Raileanu, H. Kuttler, J. B. Tenenbaum, T. Rocktäschel, and E. Grefenstette. Learning with AMIGo: Adversarially motivated intrinsic goals. In *International Conference on Learning Representations*, 2021.

[9] M. Chevalier-Boisvert, L. Willems, and S. Pal. Minimalistic gridworld environment for openai gym (2018). *URL https://github. com/maximecb/gym-minigrid*, 6, 2021.

[10] M. Dennis, N. Jaques, E. Vinitsky, A. Bayen, S. Russell, A. Critch, and S. Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. *Advances in neural information processing systems*, 33:13049–13061, 2020.

[11] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

[12] S. Forestier, R. Portelas, Y. Mollard, and P.-Y. Oudeyer. Intrinsically motivated goal exploration processes with automatic curriculum learning. *The Journal of Machine Learning Research*, 23(1):6818–6858, 2022.

[13] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem. Brax–a differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*, 2021.

[14] S. Frey, K. Li, P. Nagy, S. Sapora, C. Lu, S. Zohren, J. Foerster, and A. Calinescu. Jax-lob: A gpu-accelerated limit order book simulator to unlock large scale reinforcement learning for trading. *arXiv preprint arXiv:2308.13289*, 2023.

[15] A. Gu, K. Goel, and C. Re. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2021.

[16] M. Hessel, M. Kroiss, A. Clark, I. Kemaev, J. Quan, T. Keck, F. Viola, and H. van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

[17] H. Hu, A. Lerer, B. Cui, L. Pineda, N. Brown, and J. Foerster. Off-belief learning. In *International Conference on Machine Learning*, pages 4369–4379. PMLR, 2021.

[18] M. Igl, K. Ciosek, Y. Li, S. Tschiatschek, C. Zhang, S. Devlin, and K. Hofmann. Generalization in reinforcement learning with selective noise injection and information bottleneck. *Advances in neural information processing systems*, 32, 2019.

[19] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017.

[20] M. Jiang, M. Dennis, J. Parker-Holder, J. Foerster, E. Grefenstette, and T. Rocktäschel. Replay-guided adversarial environment design. *Advances in Neural Information Processing Systems*, 34:1884–1897, 2021.

[21] M. Jiang, M. Dennis, J. Parker-Holder, A. Lupu, H. Küttler, E. Grefenstette, T. Rocktäschel, and J. Foerster. Grounding aleatoric uncertainty for unsupervised environment design. *Advances in Neural Information Processing Systems*, 35:32868–32881, 2022.

[22] M. Jiang, E. Grefenstette, and T. Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, pages 4940–4950. PMLR, 2021.

[23] H. Küttler, N. Nardelli, T. Lavril, M. Selvatici, V. Sivakumar, T. Rocktäschel, and E. Grefenstette. Torchbeast: A pytorch platform for distributed rl. *arXiv preprint arXiv:1910.03552*, 2019.

[24] M. Lanctot, V. Zambaldi, A. Gruslys, A. Lazaridou, K. Tuyls, J. Pérolat, D. Silver, and T. Graepel. A unified game-theoretic approach to multiagent reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

[25] R. T. Lange. Reinforcement learning environments in jax. https://github.com/RobertTLange/gymnax, 2022.

[26] R. T. Lange. evosax: Jax-based evolution strategies. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, pages 659–662, 2023.

[27] C. Leary and T. Wang. Xla: Tensorflow, compiled. *TensorFlow Dev Summit*, 2(3), 2017.

[28] J. Z. Leibo, E. Hughes, M. Lanctot, and T. Graepel. Autocurricula and the emergence of innovation from social interaction: A manifesto for multi-agent intelligence research. *arXiv preprint arXiv:1903.00742*, 2019.

[29] C. Lu. Achieving 4000x speedups and meta-evolving discoveries with purejaxrl. https://chrislu.page/blog/meta-disco/, 2023.

[30] C. Lu, J. Kuba, A. Letcher, L. Metz, C. Schroeder de Witt, and J. Foerster. Discovered policy optimisation. *Advances in Neural Information Processing Systems*, 35:16455–16468, 2022.

[31] C. Lu, Y. Schroecker, A. Gu, E. Parisotto, J. Foerster, S. Singh, and F. Behbahani. Structured state space models for in-context reinforcement learning. *arXiv preprint arXiv:2303.03982*, 2023.

[32] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman. Teacher–student curriculum learning. *IEEE transactions on neural networks and learning systems*, 31(9):3732–3740, 2019.

[33] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[34] I. Mediratta, M. Jiang, J. Parker-Holder, M. Dennis, E. Vinitsky, and T. Rocktäschel. Stabilizing unsupervised environment design with a learned adversary. *arXiv preprint arXiv:2308.10797*, 2023.

[35] I. Osband, Y. Doron, M. Hessel, J. Aslanides, E. Sezener, A. Saraiva, K. McKinney, T. Lattimore, C. Szepesvari, S. Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

[36] J. Parker-Holder, M. Jiang, M. Dennis, M. Samvelyan, J. Foerster, E. Grefenstette, and T. Rocktäschel. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning*, pages 17473–17498. PMLR, 2022.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[38] A. Petrenko, E. Wijmans, B. Shacklett, and V. Koltun. Megaverse: Simulating embodied agents at one million experiences per second. In *International Conference on Machine Learning*, pages 8556–8566. PMLR, 2021.

[39] R. Portelas, C. Colas, K. Hofmann, and P.-Y. Oudeyer. Teacher algorithms for curriculum learning of deep rl in continuously parameterized environments. In *Conference on Robot Learning*, pages 835–853. PMLR, 2020.

[40] R. Portelas, C. Colas, L. Weng, K. Hofmann, and P.-Y. Oudeyer. Automatic curriculum learning for deep rl: a short survey. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 4819–4825, 2021.

[41] M. L. Puterman. Markov decision processes: Discrete stochastic dynamic programming, 1994.

[42] R. Raileanu and T. Rocktäschel. RIDE: rewarding impact-driven exploration for procedurally-generated environments. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[43] M. Randić and D. Klein. Resistance distance. *J. Math. Chem*, 12:81–95, 1993.

[44] A. Rutherford, B. Ellis, M. Gallici, J. Cook, A. Lupu, G. Ingvarsson, T. Willi, A. Khan, C. S. de Witt, A. Souly, S. Bandyopadhyay, M. Samvelyan, M. Jiang, R. T. Lange, S. Whiteson, B. Lacerda, N. Hawes, T. Rocktaschel, C. Lu, and J. N. Foerster. JaxMARL: Multi-Agent RL Environments in JAX. In *Agent Learning in Open-Endedness Workshop at NeurIPS*, 2023.

[45] F. Sadeghi and S. Levine. CAD2RL: real single-image flight without a single real image. In N. M. Amato, S. S. Srinivasa, N. Ayanian, and S. Kuindersma, editors, *Robotics: Science and Systems XIII, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, July 12-16, 2017*, 2017.

[46] M. Samvelyan, A. Khan, M. Dennis, M. Jiang, J. Parker-Holder, J. Foerster, R. Raileanu, and T. Rocktäschel. Maestro: Open-ended environment design for multi-agent reinforcement learning. *arXiv preprint arXiv:2303.03376*, 2023.

[47] M. Samvelyan, A. Khan, M. D. Dennis, M. Jiang, J. Parker-Holder, J. N. Foerster, R. Raileanu, and T. Rocktäschel. Maestro: Open-ended environment design for multi-agent reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2022.

[48] L. J. Savage. The theory of statistical decision. *Journal of the American Statistical association*, 1951.

[49] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[50] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[51] M. Seemann. Dependency injection is loose coupling. *Ploeh blog [online]*, 7, 2010.

[52] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences*, 51(3):400–403, 1995.

[53] B. Shacklett, L. G. Rosenzweig, Z. Xie, B. Sarkar, A. Szot, E. Wijmans, V. Koltun, D. Batra, and K. Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Trans. Graph.*, 42(4), 2023.

[54] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[55] J. T. Smith, A. Warrington, and S. W. Linderman. Simplified state space layers for sequence modeling. *arXiv preprint arXiv:2208.04933*, 2022.

[56] A. Stooke and P. Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

[57] A. A. Team, J. Bauer, K. Baumli, S. Baveja, F. Behbahani, A. Bhoopchand, N. Bradley-Schmieg, M. Chang, N. Clay, A. Collister, et al. Human-timescale adaptation in an open-ended task space. 2023.

[58] O. E. L. Team, A. Stooke, A. Mahajan, C. Barros, C. Deck, J. Bauer, J. Sygnowski, M. Trebacz, M. Jaderberg, M. Mathieu, et al. Open-ended learning leads to generally capable agents. *arXiv preprint arXiv:2107.12808*, 2021.

[59] O. E. L. Team, A. Stooke, A. Mahajan, C. Barros, C. Deck, J. Bauer, J. Sygnowski, M. Trebacz, M. Jaderberg, M. Mathieu, et al. Open-ended learning leads to generally capable agents. *arXiv preprint arXiv:2107.12808*, 2021.

[60] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

[61] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[62] K. Young and T. Tian. Minatar: An atari-inspired testbed for thorough and reproducible reinforcement learning experiments. *arXiv preprint arXiv:1903.03176*, 2019.

[63] T. Zhang, H. Xu, X. Wang, Y. Wu, K. Keutzer, J. E. Gonzalez, and Y. Tian. Bebold: Exploration beyond the boundary of explored regions. *CoRR*, abs/2012.08621, 2020.

A Additional Experiment Results

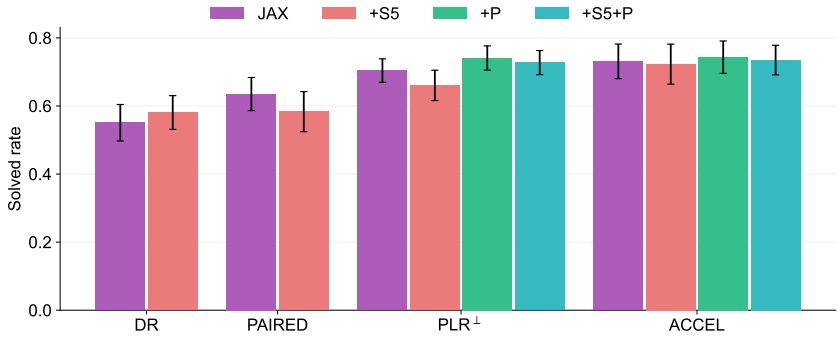

Figure 9: Zero-shot transfer solved rate of agents trained via each `minimax` baseline method averaged across all OOD test mazes (mean and std of 10 runs). This plot shows the absolute performances corresponding to the relative performances reported in Figure 7.

B Choice of Architecture and Hyperparameters

For all experiments, we use PPO [50] with generalized advantage estimation [GAE; 49] as the base RL optimization algorithm. Our student and teacher policies use the same architecture as described in Jiang et al, 2021 [20]. We report the best hyperparameter settings found for each method with an LSTM policy in Table 5 and with an S5 policy in Table 6. Unless otherwise specified, the teacher and student hyperparameters are equal. For each method we swept subsets of the following hyperparameter values (where applicable) using $5\times$ runs per inspected setting and selected the configuration with the highest mean solved rate over a set of three validation mazes (`SixteenRooms`, `Labyrinth`, and `StandardMaze`—the same used in prior works [20, 36]) after 30k PPO updates:

- learning rate in {5e-5, 1e-5, 1e-4, 3e-4}
- discount factor γ in $\{0.999, 0.995, 0.9\}$
- GAE λ discount factor in $\{0.95, 0.98, 0.99\}$
- student entropy coefficient in $\{0, 0.0001, 0.001, 0.01\}$
- teacher entropy coefficient in $\{0, 0.001, 0.01, 0.05\}$
- replay rate in $\{0.5, 0.8, 0.9\}$
- PLR scoring function in {MaxMC, PVL}
- PLR prioritization in {proportional, rank}
- PLR staleness coefficient in $\{0.3, 0.5\}$
- PLR temperature in $\{0.1, 0.3, 0.5\}$
- ACCEL number of mutations in $\{5, 10, 20\}$

Here, MaxMC refers to the maximum Monte Carlo regret estimator, and PVL, the positive value loss regret estimator [20].

S5 experiments For S5 policies, we additionally swept over the number of S5 blocks in $\{1, 2, 4\}$. We set the number of S5 layers to 2, which results in an approximately equal number of model parameters as the LSTM policy used in prior works. Table 6 reports the best hyperparameters found in combination with an S5 policy.

Batch-size experiments For batch size, we swept over the learning rate in {1e-5, 3e-5, 1e-4} while setting the remaining hyperparameters to the best-performing values for 32 parallel environments.

Table 5: Hyperparameters used for training each method with an LSTM policy.

Parameter	DR	PAIRED	PLR$^{\perp}$	PLR$^{\parallel}$	ACCEL	ACCEL$^{\parallel}$
PPO						
γ	0.995	0.995	0.995	0.999	0.995	0.995
λ_{GAE}	0.98					
PPO rollout length	256					
PPO epochs	5					
PPO minibatches per epoch	1					
PPO clip range	0.2					
PPO # parallel environments	32					
Adam learning rate	1e-4	1e-4	5e-5	1e-4	1e-4	1e-4
Adam ϵ	1e-5					
PPO max gradient norm	0.5					
PPO value clipping	yes					
return normalization	no					
value loss coefficient	0.5					
student entropy coefficient	1e-3	1e-3	0.0	0.0	0.0	0.0
generator entropy coefficient	–	0.05	–	–	–	–
PLR$^{\perp}$						
Replay rate, p	–	–	0.5	0.5	0.8	0.8
Buffer size, K	–	–	4000	4000	4000	4000
Scoring function	–	–	MaxMC	MaxMC	MaxMC	MaxMC
Prioritization	–	–	rank	rank	rank	rank
Temperature, β	–	–	0.1	0.3	0.1	0.1
Staleness coefficient	–	–	0.3	0.3	0.3	0.3
ACCEL						
Mutation subsample size, q	–	–	–	–	4	4
Mutation selection	–	–	–	–	batch	batch
# of mutations	–	–	–	–	20	10

Table 6: Hyperparameters used for training each method with an S5 policy.

Parameter	DR	PAIRED	PLR$^{\perp}$	PLR$^{\parallel}$	ACCEL	ACCEL$^{\parallel}$
PPO						
γ	0.995	0.995	0.999	0.995	0.999	0.999
λ_{GAE}	0.98					
PPO rollout length	256					
PPO epochs	5					
PPO minibatches per epoch	1					
PPO clip range	0.2					
PPO # parallel environments	32					
Adam learning rate	3e-5	1e-4	3e-5	3e-5	3e-5	1e-5
Adam ϵ	1e-5					
PPO max gradient norm	0.5					
PPO value clipping	yes					
return normalization	no					
value loss coefficient	0.5					
student entropy coefficient	1e-3	1e-3	1e-3	1e-3	1e-3	0.0
generator entropy coefficient	–	1e-3	–	–	–	–
PLR$^{\perp}$						
Replay rate, p	–	–	0.5	0.5	0.8	0.8
Buffer size, K	–	–	4000	4000	4000	4000
Scoring function	–	–	MaxMC	MaxMC	MaxMC	MaxMC
Prioritization	–	–	rank	rank	rank	rank
Temperature, β	–	–	0.3	0.3	0.3	0.3
Staleness coefficient	–	–	0.3	0.3	0.3	0.3
ACCEL						
Mutation subsample size, q	–	–	–	–	4	4
Mutation selection	–	–	–	–	batch	batch
# of mutations	–	–	–	–	10	20
S5						
# layers	2					
# blocks, p	2					
LayerNorm position	post	post	pre	post	post	post