GAGE: TRAINING GENERAL AGENTS WITH GO-EXPLORE DEMONSTRATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Go-Explore is a powerful family of methods for solving hard-exploration problems. These approaches learn by first returning to and then exploring from promising states discovered during training. However, despite dramatically outperforming other exploration techniques in video game, robotics, and text-based reasoning tasks, prior work has not used Go-Explore to train policies that generalize across problem instances. We introduce General Agents via Go-Explore (GAGE), a new method for training general policies in highly variable, sparse-reward environments. GAGE searches for successful trajectories on individual task instances with Go-Explore, distills these demonstrated behaviors into a policy with supervised learning, and then fine-tunes the policy with reinforcement learning. Our experiments on Procgen and MiniHack, two procedurally generated reinforcement learning benchmarks, demonstrate that GAGE generalizes to unseen problem instances while standard reinforcement learning methods struggle. We show that GAGE can match or exceed the sample efficiency of popular RL methods, that its performance reliably improves with additional samples, and that GAGE is robust to increasing reward sparsity. Finally, we show that GAGE can train a single multi-task policy that performs well across several environments. Our work suggests that combining Go-Explore with supervised learning is a promising and underexplored paradigm for training general agents in hard-exploration domains.

1 INTRODUCTION

Exploration is a fundamental challenge in reinforcement learning (RL). Online RL agents generate their own training data, so they must discover rewarding behavior before they can learn to reproduce it reliably. When rewards are dense, even weak policies receive frequent feedback that can guide learning. In sparse-reward environments, however, successful trajectories are extremely rare. Rewards may require long sequences of coordinated actions, occur only after completing a task, or appear long after a task is complete. Many real-world tasks have sparse rewards: a web agent may receive a success signal only after completing a long checkout flow, while a robot may be rewarded only after fully assembling an object. In sparse-reward settings, RL agents can struggle to learn the relationship between rewards and the behaviors that enabled them, or may never experience rewards at all.

Demonstrations of successful behavior provide a natural way around this data-generation bottleneck, enabling imitation learning to train a policy without requiring online exploration. (Pomerleau, 1988; Bain & Sammut, 1999). With sufficiently broad demonstrations, imitation learning can produce policies that generalize across tasks or problem instances (Pomerleau, 1988; Reed et al., 2022), and reinforcement learning can further improve these policies through finetuning (Pomerleau, 1988; Silver et al., 2016). This general recipe—learning from demonstrations and then improving through RL—has proven effective in domains ranging from game playing (Silver et al., 2016; Vinyals et al., 2019) to language-model alignment (Stiennon et al., 2020; Ouyang et al., 2022). Its applicability, however, depends on access to sufficiently broad and high-quality demonstrations. Collecting enough demonstrations to cover a diverse task distribution becomes increasingly difficult as the task space grows, and expert data can be expensive; the AI industry spends billions of dollars annually on human data annotation to train large language models (LLMs) (Douzenis et al., 2025). This raises a natural question: can successful demonstrations for hard-exploration problems be generated automatically?

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

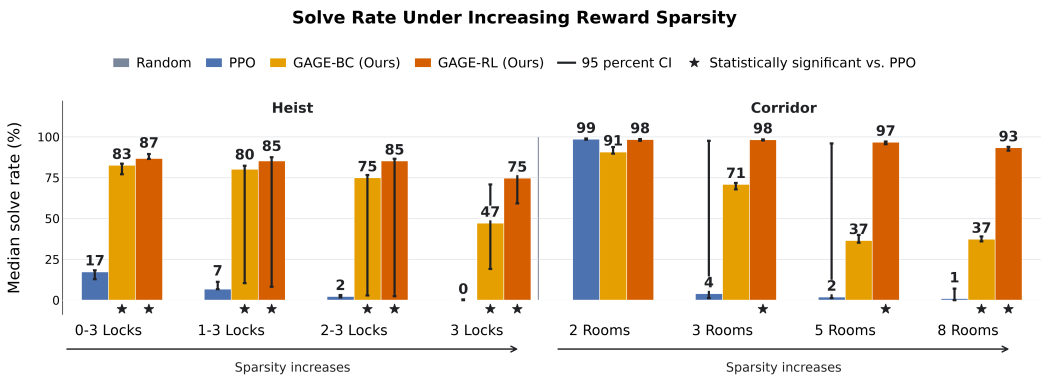


Figure 1: Solve rate of each method on environments with increasingly sparse rewards. GAGE is extremely robust to increasing sparsity, while PPO struggles with moderately sparse rewards. Solve rates are the median over 5 training runs evaluated on 1000 unseen test seeds. Statistically significant improvements over PPO are starred ($p < 0.036$).

Go-Explore offers a promising mechanism for generating such demonstrations. Go-Explore is a family of exploration methods that maintains an archive of promising states and repeatedly returns to them to explore further (Ecoffet et al., 2021). This strategy has achieved strong results on difficult sparse-reward problems, demonstrating that targeted search can discover successful trajectories even when conventional online RL struggles (Ecoffet et al., 2021). However, prior Go-Explore methods have focused on solving individual tasks or instances, rather than learning a policy that generalizes across a task distribution. Existing approaches may optimize a small number of discovered trajectories, train goal-conditioned policies that require prior knowledge of the goal state, or forgo learning a reusable policy entirely (Ecoffet et al., 2021; Gallouédec & Dellandréa, 2023; Lu et al., 2025). It remains unclear whether Go-Explore can serve as a scalable source of training data for generalist policies at a cost comparable to standard online RL methods.

We introduce General Agents via Go-Explore (GAGE), a method for training general policies in highly variable, sparse-reward environments. GAGE applies Go-Explore independently across many training instances to discover successful trajectories, then optimizes these trajectories into high-quality demonstrations. It aggregates the resulting demonstrations into a dataset and trains a policy with imitation learning, transferring behaviors discovered by Go-Explore across the task distribution. Finally, GAGE fine-tunes this policy with reinforcement learning.

We evaluate GAGE on Procgen and MiniHack, two sparse-reward, procedurally generated benchmarks. Across seven environments, GAGE learns policies that generalize to unseen instances and substantially outperform standard RL and intrinsic-motivation baselines given. Given equal environment-interaction budgets, GAGE achieves higher solve rates than these baselines, and with additional search, its performance continues to improve even after the baselines plateau. We further show that GAGE remains effective as rewards become increasingly sparse (Figure 1) and can combine demonstrations from multiple games to train a single high-performing multitask policy. These results demonstrate that Go-Explore can serve as an automated source of demonstrations for imitation learning, providing an underexplored approach to training general agents on hard-exploration tasks.

2 Background

2.1 Go-Explore

Go-Explore was developed for sparse-reward, long-horizon environments where standard reinforcement learning methods often struggle. Ecoffet et al. (2021) identify two key failure modes: detachment and derailment. Detachment occurs when an agent fails to preserve or revisit promising states discovered earlier in training. Derailment occurs when stochastic exploration disrupts the long-horizon action sequences needed to reach distant states. Go-Explore addresses these failures by maintaining an archive of promising states and returning to archived states before exploring outward from them.

Go-Explore organizes the archive by mapping states into coarse cells. This mapping, called a cell representation, can be manually defined, automatically tuned, or learned during training (Ecoffet et al., 2021; Gallouédec & Dellandréa, 2023). A state is added to the archive if it maps to an empty cell. If the cell is already occupied, Go-Explore keeps the state with higher return, breaking ties in favor of shorter trajectories. The original work introduced two variants: policy-based Go-Explore, which learns a goal-conditioned policy for returning to archived states, and policy-free Go-Explore, which replays actions in a deterministic simulator. GAGE builds on the policy-free variant.

2.2 Environment distributions

This paper studies highly variable environments where each episode is drawn from a distribution of related problem instances. Such variability is common in real-world tasks. For example, a household robot that learns to put away objects in one kitchen should ideally generalize to kitchens with different layouts, object placements, and obstacles. Procedurally generated environments provide a controlled way to study this setting by varying the problem instance while preserving the underlying task family, dynamics, and reward structure.

We model these benchmarks as underspecified partially observable Markov decision processes (UPOMDPs), where a free parameter $\theta \in \Theta$ determines the fully specified POMDP for an episode. In a maze environment, for example, θ may be the random generation seed, and the resulting POMDP is the specific maze generated by that seed. In our experiments, θ is the level-generation seed and each sampled value of θ defines one problem instance. We train policies on a set of sampled instances and evaluate generalization on held-out values of θ not seen during training.

3 Method

General Agents via Go-Explore trains a general policy by converting per-instance search into supervised learning data. For each training instance, we run policy-free Go-Explore to discover at least one successful trajectory. We then improve the discovered trajectory with Go-Explore by exploring from states along the solution path, producing optimized demonstrations. Repeating this process across many problem instances yields a dataset of successful trajectories. GAGE-BC trains a policy on this dataset with behavior cloning, and GAGE-RL further finetunes the policy with reinforcement learning. GAGE is implemented using Syllabus (Sullivan et al., 2025) for data generation and builds on CleanRL (Huang et al., 2022) for behavior cloning, PPO baselines, and RL finetuning, both of which allow future work to easily apply and extend GAGE. Our code, collected datasets, and trained models are open source and available online. Full experiment details are available in Appendix N.

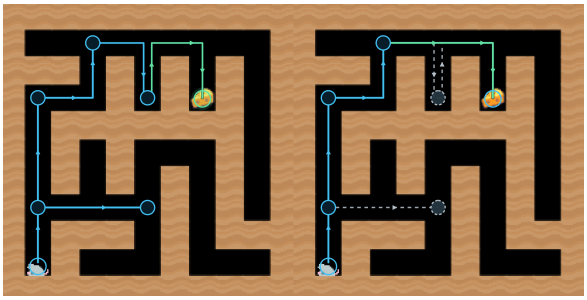
3.1 Per-instance Go-Explore

For each training instance, GAGE runs a policy-free Go-Explore search following many of the conventions of Ecoffet et al. (2021). At each exploration attempt, GAGE samples a return cell from the archive with weight $\frac{1}{\sqrt{N_{\text{seen}}+1}}$ where N_{seen} is the number of previous exploration attempts that reached the cell. Further details on cell sampling are available in Appendix C. After returning to the selected cell, GAGE explores for a fixed step budget or until the episode terminates. During exploration, actions are sampled randomly, with a high probability of repeating the previous action to encourage temporally consistent movement, as in Ecoffet et al. (2021).

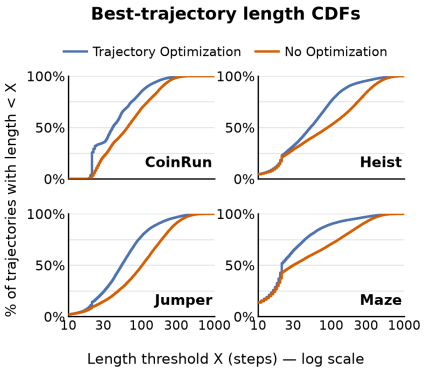
We focus on Go-Explore as the trajectory-generation method because alternatives such as human annotation, training RL agents on the full task distribution, or training separate RL agents for individual instances are substantially more expensive and are unable to solve hard problem instances in our setting. We provide supporting experiments and cost analysis in Appendix E.

3.1.1 Trajectory optimization

We introduce a trajectory optimization phase to Go-Explore to improve the quality of trajectories after the first solution is found. Before a solution is found, GAGE uses the archive-sampling procedure described in subsection 2.1. Once a successful trajectory is found, either by reaching a goal state or a prespecified target return, GAGE changes the cell sampling distribution to optimize that solution. During this phase GAGE uniformly samples only cells that lie on the trajectory to the target cell as



(a) Exploration phase (left) and Optimization Phase (right).



(b) Optimization phase ablation.

Figure 2: **a)** During exploration (**left**), GAGE samples any discovered non-terminal cell, explores from it, adds new cells to the archive, and finds shorter paths to known cells. After finding a cell that achieves the target return, GAGE samples only cells on the solution path (**right**), focusing search on finding shorter connections to the target cell. **b)** Trajectory Optimization reduces the length of the best demonstration per seed, which corresponds to more direct goal-reaching behavior.

shown in Figure 2a. This focuses exploration on improving the current solution, converting successful trajectories into shorter demonstrations with equal return. Whenever a more efficient trajectory to the target cell is found, the list of valid sampling cells is updated to reflect the new path.

Figure 2b shows that trajectory optimization improves the length of the best trajectory produced by Go-Explore in all four environments. We collected five 5000-seed datasets in each environment and compare the cumulative distribution function with and without trajectory optimization. Across the four environments, trajectory optimization produced shorter best trajectories (median = 40 steps, IQR = 20–79) than no optimization (median = 70 steps, IQR = 25–166). In our experiments, we save multiple successful trajectories with at least a 10% difference in episode length, resulting in additional demonstrations for behavior cloning. This procedure focuses search on producing a single high-quality solution per instance, but diverse strategies may still be represented in the aggregate dataset. We study the performance of alternative dataset constructions in Appendix H. We additionally ablate trajectory optimization in Appendix F. We find that it improves behavior cloning performance in several environments, but has no consistent effect after RL fine-tuning.

3.2 Behavior Cloning

GAGE-BC trains a policy via behavior cloning (Bain & Sammut, 1999; Pomerleau, 1988) trajectories generated by Go-Explore. Each training example pairs an observation o_t with the recorded action a_t and we minimize the cross-entropy loss between the policy logits and the recorded action. Behavior cloning is prone to overfitting, so we select the GAGE-BC checkpoint with the highest mean solve rate on a validation set of 100 seeds, evaluated every 100 updates, then report scores for this model on a separate test set of 1000 seeds. This checkpoint is also used as the initial policy for GAGE-RL.

3.3 Reinforcement Learning Finetuning

Reinforcement learning is commonly used to improve the performance of pretrained policies, sometimes surpassing the capabilities represented in the pretraining data (Silver et al., 2016; Baker et al., 2022). We apply this technique to GAGE by finetuning pretrained GAGE-BC policies with Proximal Policy Optimization (PPO) (Schulman et al., 2017), and call this method GAGE-RL. To prevent catastrophic forgetting during the early stages of training, a Kullback–Leibler divergence (KL) penalty is added to the loss, following precedent from prior work on RL finetuning (Stiennon et al., 2020; Ouyang et al., 2022; Baker et al., 2022). We compute this penalty using the approximate KL divergence $D_{\mathrm{KL}}(\pi_{\mathrm{old}} \,\|\, \pi_{\mathrm{new}})$ (Schulman, 2020) multiplied by a scalar coefficient that is linearly annealed to 0 over the course of training. This discourages the policy from drifting too far from GAGE-BC early in training, while allowing greater deviation as the KL coefficient anneals to zero.

4 EXPERIMENTS

In this section, we first evaluate GAGE against PPO, a strong baseline in our environments, and Random Network Distillation (RND), a popular intrinsic motivation method (Chentanez et al., 2004) that rewards an agent for reaching novel states (Burda et al., 2019). We compare GAGE-BC and GAGE-RL against these baselines on held-out environment instances to evaluate generalization, finding that GAGE dramatically outperforms PPO and RND or matches its performance across Procgen and MiniHack environments (subsection 4.1). Next, we study the impact of reward sparsity on PPO and GAGE (subsection 4.2), demonstrating that while PPO's performance is significantly affected by reward sparsity, GAGE's solve rate remains high in every setting. In the sparsest environment, GAGE reaches a 93% solve rate compared to PPO's 1%, showcasing an ability to generalize even in hard-exploration domains (Figure 1). Finally, we show that GAGE can be used to train a single general policy that plays four Procgen games and matches the performance of single-game specialist GAGE policies (subsection 4.3).

In the Procgen environments, all methods train the architecture introduced by (Cobbe et al., 2020) and in all MiniHack experiments we train the NetHack architecture introduced in (Hambro et al., 2022) for 200M timesteps. These models are standard strong baselines for each benchmark. Model architectures and hyperparameters are described in Appendix N. Full descriptions of the environments and cell representations are in Appendix A and Appendix B respectively. Unless otherwise stated, we evaluate all claims using a Mann-Whitney U-test and provide full training curves in Appendix I.

4.1 TRAINING POLICIES THAT GENERALIZE TO UNSEEN INSTANCES

We first evaluate whether trajectories generated by Go-Explore can train policies that generalize to unseen instances of sparse-reward tasks. We consider two settings. First, we give GAGE and the RL baselines the same environment-interaction budget to compare sample efficiency. Second, we give GAGE a much larger budget to test how performance scales with additional search.

We evaluate GAGE on procedurally generated benchmarks with substantial variation across instances. These environments test whether trajectories discovered on training instances can be distilled into policies that generalize to unseen levels. In each environment, the agent receives a single reward for reaching the terminal goal state, so we report solve rates for all results.

4.1.1 SAMPLE EFFICIENCY

For the equal-budget comparison, all methods train on 20,000 instances using 1B environment steps in Procgen and 200M steps in MiniHack. These are substantial training budgets: 1B steps is five times the standard 200M-step Procgen protocol (Cobbe et al., 2020), while 200M steps is the largest budget used in the original MiniHack paper's experiments (Samvelyan et al., 2021). We choose these large budgets to give PPO and RND ample opportunity to discover and learn from rare successful trajectories. GAGE allocates 800M Procgen steps to Go-Explore and 200M to RL fine-tuning. In MiniHack, it allocates 150M and 50M steps, respectively. We reserve 200M and 50M steps for RL fine-tuning because these budgets are sufficient to train strong RL policies in dense-reward Procgen and MiniHack tasks, leaving the remaining interactions for Go-Explore. We include the environment steps used by Go-Explore to return to the selected cell in the training interaction budget.

Figure 3 shows that GAGE-RL substantially improves generalization under an equal interaction budget (exact paired permutation $p = 0.031$). Its mean of per-environment median solve rates is 58%, compared with 28% for PPO and 31% for RND. GAGE-RL outperforms both baselines in five of seven environments ($p < 0.032$) and performs comparably on Maze. GAGE-BC also outperforms PPO and RND on Heist, Corridor, and River ($p < 0.012$), but underperforms on Jumper and Maze.

Maze appears to be the easiest exploration problem in our benchmark. As shown in Figure 2b, its median optimized trajectory length is fewer than 20 steps, compared with 39 steps in Coinrun, the next shortest environment. These shorter trajectories suggest a denser effective reward signal, consistent with PPO and RND performing comparably to GAGE-RL. In contrast, GAGE-RL achieves its largest Procgen gain on Heist, the sparsest Procgen task, reaching a 76% solve rate compared with 15% for PPO and 20% for RND. Finally, none of the methods make meaningful progress on Monster-Lava within the 200M-step interaction budget. Although absolute solve rates remain low, GAGE-BC and GAGE-RL significantly outperform PPO ($p < 0.0073$).

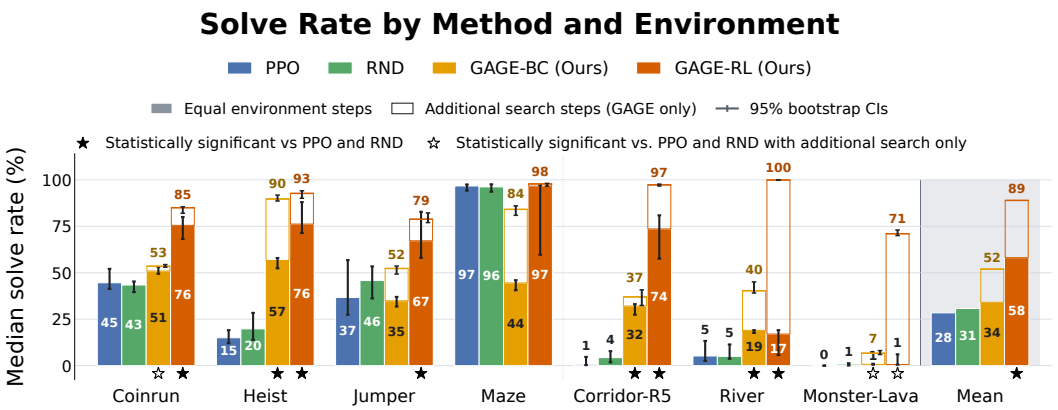
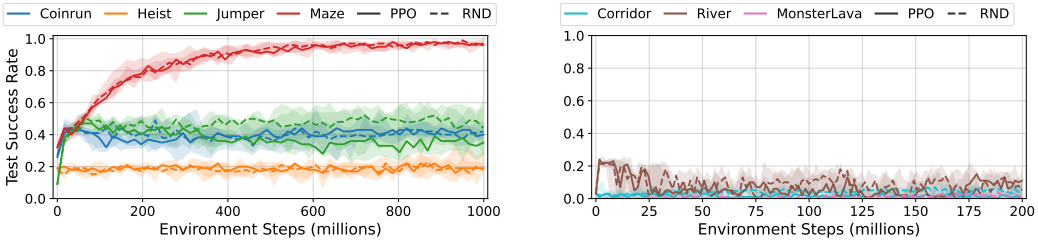


Figure 3: Given equal environment interaction budgets, GAGE-RL outperforms PPO and RND in 5 of 7 environments, and matches them in 2. We report median solve rates for PPO, RND, GAGE-BC, and GAGE-RL across five independent training runs, each evaluated on 1,000 episodes per environment. In all environments $p < 0.032$ for equal-budgets and $p < 0.019$ for additional search.

4.1.2 Performance with Additional Search

In this section, we use a larger interaction budget to test whether GAGE continues to benefit from additional search after the online RL baselines have largely stopped improving. In Procgen, PPO and RND show little or no sustained improvement beyond 200M steps in Coinrun, Heist, and Jumper (Figure 4). In MiniHack, both methods remain near their initial performance throughout the full 200M-step training run. Together, these results suggest that running the RL baselines longer would not improve their performance further.



(a) Procgen (b) MiniHack

Figure 4: Test performance of PPO and RND throughout training. In Procgen, the baselines show little or no sustained improvement in Coinrun, Heist, and Jumper, while performance continues to improve in Maze. In MiniHack, both methods remain near their initial performance throughout the training run. Each line shows median test performance across seeds, with shaded 95% bootstrap confidence intervals.

In subsubsection 4.1.1, Go-Explore solves only a fraction of the 20,000 available problem instances within the matched interaction budget. We therefore ask how GAGE performs when Go-Explore is allowed enough samples to solve a much larger portion of the training set. This is practical because Go-Explore interactions are substantially cheaper than online RL interactions: in our implementation, a Go-Explore search interaction is approximately $5\times$ faster than a PPO interaction because search requires neither policy inference nor gradient updates (see Appendix L). Thus, 5B Go-Explore search interactions take roughly the same amount of time as 1B PPO interactions, excluding subsequent BC and RL fine-tuning for GAGE. Since the online RL baselines show strong diminishing returns with additional training (Figure 4), we use this larger budget to ask whether GAGE can continue converting additional search into improved policy performance.

With additional search, GAGE-RL's mean per-environment median solve rate increases from 58% to 89% (Figure 3). GAGE-RL now matches or exceeds PPO and RND in all seven environments ($p <$

0.012) with a mean of per-environment median solve rates of 89% vs. 28% and 31% for PPO and RND respectively (exact paired permutation $p = 0.016$) The largest gains occur in MiniHack, where additional search substantially improves performance across all three environments. In Monster-Lava, the most challenging environment, GAGE-RL increases its solve-rate from approximately 1% under a 200M step budget to 71% with additional search. Overall, we find that GAGE continues to benefit from additional exploration even on tasks where the online RL baselines have plateaued.

These results also highlight the importance of RL fine-tuning. Although GAGE-BC already learns strong policies, RL fine-tuning improves performance across every environment and yields the strongest overall results. Additional experiments in Appendix J similarly show that GAGE improves reliably as the number of training instances increases. Together, these results suggest that GAGE can continue converting additional search and training data into improved policy performance after standard online RL methods have reached diminishing returns.

4.2 Robustness to Sparsity

In this section we evaluate how GAGE performs as rewards become increasingly sparse. Two of the environments, Procgen Heist and MiniHack Corridor, can be modified to increase the number of steps required to reach rewards. Heist is a Maze environment in which each instance contains up to three key-lock pairs that must be unlocked to reach the goal. Instances with fewer locks are generally easier and can often be solved through random exploration. We construct four variants containing 0–3, 1–3, 2–3, or exactly 3 locks. In MiniHack Corridor, we use 2, 3, 5, and 8-room variants, for which the average distance to the goal increases with the number of rooms. These modifications increase reward sparsity while preserving the underlying task objective.

For each setting, we train GAGE on 10,000 problem instances and allow Go-Explore up to 2M steps in Heist and 10M steps in Corridor, isolating the effect of reward sparsity on asymptotic performance. We compare against PPO trained from scratch on the same 10,000 instances for 1B steps in Procgen and 200M steps in Heist. As in subsection 4.1, the PPO agents trained here converge early in training, and in the sparser settings studied here rarely experience any rewards.

Figure 1 shows that PPO performance degrades sharply as rewards become sparser, whereas GAGE-RL maintains high solve rates across all settings. As the number of locks in Heist increases, GAGE-RL's median solve rate decreases from 87% to 75%, while PPO's drops from 17% to 0%. Similarly in Corridor, GAGE-RL performance remains above 93% in all settings, while PPO's solve rate drops from 99% to 1%. GAGE-BC is more sensitive to sparsity: its solve rate falls from 83% to 47% in Heist and from 91% to 37% in Corridor as sparsity increases. However, GAGE-BC remains a strong initialization for RL finetuning. In 8-room Corridor, for example, RL finetuning improves the GAGE-BC policy from a 37% solve rate to 93%. Together, these results show that GAGE-RL remains effective even as rewards become substantially sparser.

4.3 Generalizing across games

A key motivation for GAGE is that demonstrations generated independently across tasks can be combined into a single dataset to train multi-task policies. We test this by training a single policy across all four Procgen environments studied in this paper. We are interested in whether it is possible to train a generalist agent using GAGE, so we tune hyperparameters specifically for the generalist setting. For GAGE-BC, we combine the demonstration datasets from all four games, decrease the learning rate by a factor of 10 and increase the training time by a factor of 10 to improve stability, and otherwise use the same BC procedure. For multi-game GAGE-RL, we allocate one quarter of PPO's environment workers to each game, normalize advantages separately by task, and increase the batch size by a factor of four to reduce variance during multi-task training. Precise values and the methodology for how they were chosen can be found in Appendix N. These hyperparameters affect the computational cost of training, and we therefore do not compare the efficiency of specialist and generalist training. We refer to policies trained on one game as specialists and policies trained jointly on all four games as generalist policies.

Table 1 shows that combining demonstrations across games during behavior cloning incurs almost no loss in performance: the generalist GAGE-BC policy achieves a mean per-environment median solve rate of 68.7%, compared with 69.9% for separately trained specialist policies. Moreover, the

Table 1: Median performance across Procgen environments for specialist and general BC and RL configurations. A single generalist GAGE-BC policy matches the performance of individual specialist GAGE-BC policies in all four Procgen environments.

	Specialist BC		Generalist BC		
Environment	BC Only	+ Specialist RL	BC Only	+ Specialist RL	+ Generalist RL
Coinrun	0.535	0.850	0.513	0.846	0.778
Heist	0.897	0.927	0.898	0.961	0.844
Jumper	0.523	0.788	0.483	0.827	0.637
Maze	0.841	0.978	0.852	0.979	0.891
Mean	0.699	0.886	0.687	0.903	0.788

generalist BC policy provides a strong initialization for subsequent specialist RL. Fine-tuning a separate RL policy for each game from the multi-game BC checkpoint improves performance in all four environments ($p < 0.012$) and reaches a mean solve rate of 90.3%, which is comparable to fine-tuning from specialist BC checkpoints. This suggests that joint imitation learning produces representations that transfer beneficially across these games.

Naive multi-game RL also improves the policy's performance in every environment, but is less effective than single-game RL. Fine-tuning a single policy across all four games improves the multi-game BC policy solve rate from 68.7% to 78.8%, demonstrating that RL can improve a shared multi-task policy. This improvement is statistically significant in all environments except Maze ($p < 0.012$). However, it falls short of the 91.7% achieved by independently fine-tuning each game ($p < 0.012$, except in Maze). These preliminary results suggest that GAGE can train a single policy across heterogeneous environments without sacrificing BC performance, while effective joint RL remains an important direction for future work.

5 Related Work

Exploration has long been recognized as a central challenge in reinforcement learning. Intrinsic-motivation methods address this challenge by augmenting the reward with bonuses for visiting novel, uncertain, or controllable states (Pathak et al., 2017; Burda et al., 2019; Machado et al., 2020). Several such methods have been adapted to procedurally generated environments, for example by rewarding changes in learned state representations (Raileanu & Rocktäschel, 2020) or designing bonuses that remain informative under high environment variability (Henaff et al., 2022). Curriculum learning (Portelas et al., 2020) and Unsupervised Environment Design (Dennis et al., 2020) instead modify the distribution of training tasks to prioritize instances with high estimated learning potential (Portelas et al., 2021; Jiang et al., 2021; Kanitscheider et al., 2021; Parker-Holder et al., 2022; Rutherford et al., 2024). These approaches improve online exploration by modifying either the reward signal or the training distribution. GAGE instead separates exploration from policy learning: it uses Go-Explore to solve individual instances, then distills the resulting trajectories into a policy that generalizes across instances. These approaches are complementary, and intrinsic-motivation or curriculum-based methods could also be incorporated into GAGE's RL fine-tuning stage.

Planning methods use a predefined or learned model to evaluate possible futures before acting. Go-Explore can be viewed as a planning method that uses a simulator as a model to search from archived states rather than from the current policy state. Prior work has used planning as a policy-improvement operator, including AlphaGo Zero, Expert Iteration, MuZero, and Guided Policy Search (Silver et al., 2017; Anthony et al., 2017; Schrittwieser et al., 2020; Levine & Koltun, 2013). This framework has also been extended to achieve superhuman performance in imperfect information games including Poker and Scotland Yard (Brown & Sandholm, 2018; Brown et al., 2020; Schmid et al., 2023). However, many planning methods operate over fine-grained states and actions. GAGE instead searches the coarser cell space defined by Go-Explore, allowing it to generate successful demonstrations for long-horizon tasks without intermediate feedback.

Our work is also related to prior extensions of Go-Explore. Lu et al. (2025) introduced Intelligent Go-Explore, which uses foundation models to replace hand-designed heuristics for selecting, evaluating, and expanding promising states. They demonstrated that LLMs can approximate an archive in context to identify promising states in text-based reasoning tasks. The closest prior method is policy-based Go-Explore, introduced by Ecoffet et al. (2021), which trains a goal-conditioned policy to return to promising states in sparse-reward singleton environments. However, none of these approaches address the setting studied here: training a single generalist policy in a highly variable distribution of sparse-reward tasks. We provide a detailed comparison to prior Go-Explore methods in Appendix M.

6 Limitations

GAGE relies on domain-specific cell representations to identify promising states for further exploration. These representations must preserve distinctions that matter for reaching useful new states, but keeping all states can inflate the archive or prioritize irrelevant novelty, an instance of the noisy-TV problem (Schmidhuber, 2010; Burda et al., 2019). Prior work has addressed this challenge by automatically tuning cell-representation parameters (Ecoffet et al., 2021), deriving latent cell representations from learned policy networks (Gallouédec & Dellandréa, 2023), or querying foundation models to identify promising states (Lu et al., 2025). These approaches could be integrated with GAGE, but general and efficient cell representations remain an open problem. We therefore do not claim to solve arbitrary RL problems without tuning; rather, we show that with suitable cell representations, GAGE can train generalist policies in hard exploration environments. Lastly, our experiments assume access to a deterministic simulator. Policy-based Go-Explore suggests a path toward addressing this limitation by using a goal-conditioned return policy to revisit archived states without relying on deterministic replay (Ecoffet et al., 2021). Integrating these ideas into GAGE and adapting them to broad environment distributions is an important direction for future work.

7 Discussion and Conclusion

GAGE shows that Go-Explore can be used not only to solve individual problems, but also to train policies that generalize across environment distributions. This paper focuses on generalization within task families rather than across unrelated domains. However, prior work in imitation learning suggests that, with sufficient data, policies can generalize across diverse environments, dynamics, control schemes, and observation modalities (Reed et al., 2022). GAGE addresses one bottleneck in this direction by generating successful demonstrations in sparse-reward domains, and could be applied to train general policies in domains where expert data is unavailable.

More broadly, GAGE suggests a path toward using sparse feedback in open-ended task spaces. Open-ended environments contain many possible goals, strategies, and behaviors rather than a single fixed task (Stanley et al., 2017). Agents cannot learn everything in these settings, so they must prioritize learning useful or interesting skills (Clune, 2019; Zhang et al., 2024). When agents choose their own goals, sparse objectives are often easier to design and evaluate than dense reward functions. An agent may recognize an outcome worth reaching without knowing how to shape learning toward that objective. GAGE offers a way to exploit these sparse success signals to learn a general policy. In broader open-ended systems, goal-generation algorithms (Forestier et al., 2022; Colas et al., 2023; Faldor et al., 2025) could propose objectives, while GAGE-like methods search for solutions to create a growing demonstration dataset, training an agent with an expanding repertoire of skills. By turning sparse, self-generated goals into reusable demonstrations, GAGE could become a powerful catalyst for open-ended learning.

GAGE combines Go-Explore's strength in hard-exploration environments with imitation learning and reinforcement learning to train policies that generalize across sparse-reward task distributions. By searching each problem instance independently, GAGE separates exploration from policy learning, allowing it to generate successful demonstrations in settings where standard reinforcement learning methods struggle to discover reward. Across Procgen and MiniHack environments GAGE matches or outperforms PPO and RND, with the largest gains appearing as reward sparsity increases. These results suggest that GAGE could become a standard approach to generalization in sparse-reward domains, transforming intelligent exploration into the demonstration data needed to train generalist agents.

AI USE STATEMENT

In this work, we use generative AI tools to assist with refactoring and optimize training code, to produce code for figures, and to check for writing errors. We have not used generative AI tools for any other tasks with required disclosure. Also, tasks such as formulating or providing key ingredients for mathematical claims, proof writing, translation, and dataset formatting or generation are not applicable to this work. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies exploration in simulated reinforcement learning environments. It does not involve human subjects, personal data, or deployed systems, and we do not foresee direct ethical concerns beyond those general to reinforcement learning research.

REPRODUCIBILITY STATEMENT

We release open-source implementations of all methods and infrastructure used in the paper. Hyperparameter values and sweep ranges for every method and environment are reported in Appendix N. The evaluation protocol is described in section 4 and compute requirements are listed in Appendix L. We are confident the method is fully reproducible from the paper and the code.

REFERENCES

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017.

Michael Bain and Claude Sammut. A framework for behavioural cloning. In *Machine Intelligence 15, Intelligent Agents [St. Catherine's College, Oxford, July 1995]*, pp. 103–129, GBR, 1999. Oxford University. ISBN 0198538677.

Bowen Baker, Ilge Akkaya, Peter Zhokov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (vpt): Learning to act by watching unlabeled online videos. *Advances in Neural Information Processing Systems*, 35:24639–24654, 2022.

Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018. doi: 10.1126/science.aao1733. URL https://www.science.org/doi/abs/10.1126/science.aao1733.

Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. *Advances in neural information processing systems*, 33:17057–17069, 2020.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1lJJnR5Ym.

Nuttapong Chentanez, Andrew Barto, and Satinder Singh. Intrinsically motivated reinforcement learning. In L. Saul, Y. Weiss, and L. Bottou (eds.), *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2004. URL https://proceedings.neurips.cc/paper_files/paper/2004/file/4be5a36cbaca8ab9d2066debfe4e65c1-Paper.pdf.

Jeff Clune. AI-GAs: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 2048–2056. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/cobbe20a.html.

Cobbe, Karl and Hesse, Chris and Hilton, Jacob and Schulman, John. Procgen benchmark, 2019. URL https://github.com/openai/procgen. GitHub repository, accessed 2026-04-24.

Cédric Colas, Laetitia Teodorescu, Pierre-Yves Oudeyer, Xingdi Yuan, and Marc-Alexandre Côté. Augmenting autotelic agents with large language models. In *Conference on Lifelong Learning Agents*, pp. 205–226. PMLR, 2023.

Jeff Da, Clinton Wang, Xiang Deng, Yuntao Ma, Nikhil Barhate, and Sean Hendryx. Agent-rlvr: Training software engineering agents via guidance and environment rewards. *arXiv preprint arXiv:2506.11425*, 2025.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. *Advances in neural information processing systems*, 33:13049–13061, 2020.

Vasilis Douzenis, Laurence Wilse-Samson, Avantika Tandon, Hwee Shi Yong, and Kevin Wee. The Economic Impact of the Data Annotation Industry. Technical report, Oxford Economics, December 2025. URL https://www.oxfordeconomics.com/wp-content/uploads/2025/12/OxfordEconomics_DataAnnotationImpacts.pdf. Commissioned by Scale AI, Inc.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, February 2021. ISSN 0028-0836, 1476-4687. doi: 10.1038/s41586-020-03157-9. URL http://arxiv.org/abs/2004.12919. arXiv:2004.12919 [cs].

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.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. OMNI-EPIC: Open-endedness via models of human notions of interestingness with environments programmed in code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Y1XkzMJpPd.

Sébastien Forestier, Rémy Portelas, Yoan Mollard, and Pierre-Yves Oudeyer. Intrinsically motivated goal exploration processes with automatic curriculum learning. *Journal of Machine Learning Research*, 23(152):1–41, 2022.

Quentin Gallouédec and Emmanuel Dellandréa. Cell-free latent go-explore. In *International Conference on Machine Learning*, pp. 10571–10586. PMLR, 2023.

Cole Gulino, Justin Fu, Wenjie Luo, George Tucker, Eli Bronstein, Yiren Lu, Jean Harb, Xinlei Pan, Yan Wang, Xiangyu Chen, et al. Waymax: An accelerated, data-driven simulator for large-scale autonomous driving research. *Advances in Neural Information Processing Systems*, 36:7730–7742, 2023.

Eric Hambro, Sharada Mohanty, Dmitrii Babaev, Minwoo Byeon, Dipam Chakraborty, Edward Grefenstette, Minqi Jiang, Jo Daejin, Anssi Kanervisto, Jongmin Kim, et al. Insights from the neurips 2021 nethack challenge. In *NeurIPS 2021 Competitions and Demonstrations Track*, pp. 41–52. PMLR, 2022.

Mikael Henaff, Roberta Raileanu, Minqi Jiang, and Tim Rocktäschel. Exploration via elliptical episodic bonuses. *Advances in Neural Information Processing Systems*, 35:37631–37646, 2022.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G. M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

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

Ingmar Kanitscheider, Joost Huizinga, David Farhi, William Hebgen Guss, Brandon Houghton, Raul Sampedro, Peter Zhokhov, Bowen Baker, Adrien Ecoffet, Jie Tang, et al. Multi-task curriculum learning in a complex, visual, hard-exploration domain: Minecraft. *arXiv preprint arXiv:2106.14876*, 2021.

Sergey Levine and Vladlen Koltun. Guided policy search. In Sanjoy Dasgupta and David McAllester (eds.), *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pp. 1–9, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL https://proceedings.mlr.press/v28/levine13.html.

Cong Lu, Shengran Hu, and Jeff Clune. Intelligent go-explore: Standing on the shoulders of giant foundation models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=apErWGzCAA.

Marlos C. Machado, Marc G. Bellemare, and Michael Bowling. Count-based exploration with the successor representation. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*, 2020. URL https://arxiv.org/abs/1807.11622.

Sharada Mohanty, Jyotish Poonganam, Adrien Gaidon, Andrey Kolobov, Blake Wulfe, Dipam Chakraborty, Gražvydas Šemetulskis, João Schapke, Jonas Kubilius, Jurgis Pašukonis, et al. Measuring sample efficiency and generalization in reinforcement learning benchmarks: Neurips 2020 procgen benchmark. *arXiv preprint arXiv:2103.15332*, 2021.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning*, pp. 17473–17498. PMLR, 2022.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pp. 2778–2787. PMLR, 2017.

Ulyana Piterbarg, Lerrel Pinto, and Rob Fergus. Nethack is hard to hack. *Advances in Neural Information Processing Systems*, 36:37540–37566, 2023.

Dean A. Pomerleau. Alvinn: an autonomous land vehicle in a neural network. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems*, NIPS'88, pp. 305–313, Cambridge, MA, USA, 1988. MIT Press.

Rémy Portelas, Cédric Colas, Katja Hofmann, and Pierre-Yves Oudeyer. Teacher algorithms for curriculum learning of deep rl in continuously parameterized environments. In *Conference on Robot Learning*, pp. 835–853. PMLR, 2020.

Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic curriculum learning for deep rl: A short survey. In *IJCAI 2020-International Joint Conference on Artificial Intelligence*, 2021.

Roberta Raileanu and Tim Rocktäschel. Ride: Rewarding impact-driven exploration for procedurally-generated environments. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkg-TJBFPB.

Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.

Alex Rutherford, Michael Beukman, Timon Willi, Bruno Lacerda, Nick Hawes, and Jakob Foerster. No regrets: Investigating and improving regret approximations for curriculum discovery. *Advances in Neural Information Processing Systems*, 37:16071–16101, 2024.

Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Kuttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

Martin Schmid, Matej Moravčík, Neil Burch, Rudolf Kadlec, Josh Davidson, Kevin Waugh, Nolan Bard, Finbarr Timbers, Marc Lanctot, G Zacharias Holland, et al. Student of games: A unified learning algorithm for both perfect and imperfect information games. *Science Advances*, 9(46): eadg3256, 2023.

Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–247, 2010. doi: 10.1109/TAMD.2010.2056368.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

John Schulman. Approximating kl divergence. `https://joschu.net/blog/kl-approx.html`, March 2020. Blog post, posted 2020-03-07.

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

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL `https://doi.org/10.1038/nature16961`.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, L. Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017. URL `https://api.semanticscholar.org/CorpusID:205261034`.

Kenneth O. Stanley, Joel Lehman, and Lisa Soros. Open-endedness: The last grand challenge you've never heard of, December 2017. URL `https://www.oreilly.com/radar/open-endedness-the-last-grand-challenge-youve-never-heard-of/`. O'Reilly Radar.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in neural information processing systems*, 33:3008–3021, 2020.

Ryan Sullivan, Ryan Pégoud, Ameen Ur Rehman, Xinchen Yang, Junyun Huang, Aayush Verma, Nistha Mitra, and John P Dickerson. Syllabus: Portable curricula for reinforcement learning agents. In *Reinforcement Learning Journal*, 2025. URL `https://openreview.net/forum?id=0LFJnnMKeT`.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Joseph Dudzik, Junyoung Chung, David Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575:350 – 354, 2019. URL `https://api.semanticscholar.org/CorpusID:204972004`.

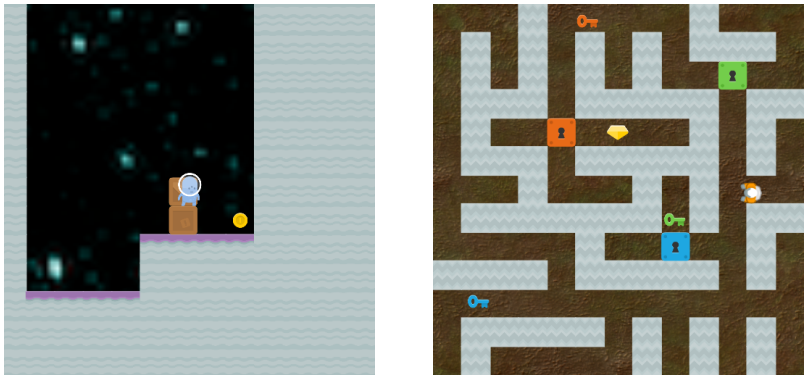
Niannian Wu, Zongyu Yang, Rongpeng Li, Ning Wei, Yihang Chen, Qianyun Dong, Jiyuan Li, Guohui Zheng, Xinwen Gong, Feng Gao, Bo Li, Min Xu, Zhifeng Zhao, and Wulyu Zhong. High-fidelity data-driven dynamics model for reinforcement learning-based control in HL-3 tokamak. *Communications Physics*, 8(1):393, October 2025. ISSN 2399-3650. doi: 10.1038/s42005-025-02302-y. URL https://doi.org/10.1038/s42005-025-02302-y.

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. OMNI: Open-endedness via models of human notions of interestingness. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=AgM3MzT99c.

Tianyu Zheng, Tianshun Xing, Qingshui Gu, Taoran Liang, Xingwei Qu, Xin Zhou, Yizhi Li, Zhoufutu Wen, Chenghua Lin, Wenhao Huang, et al. First return, entropy-eliciting explore. *arXiv preprint arXiv:2507.07017*, 2025.

A ENVIRONMENTS

A.1 PROCGEN BENCHMARK



(a) Coinrun (b) Heist

(c) Jumper (d) Maze

Figure 5: Procgen environments.

The Procgen Benchmark is a collection of 16 procedurally generated video-game environments designed to test generalization for RL agents (Cobbe et al., 2020). Agents observe an 64x64 RGB image of the environment and select from up to 16 discrete actions. These actions vary between games, but typically include movement in 2-8 cardinal directions, jumping, and shooting a weapon. We evaluate our method on Coinrun, Heist, Jumper, and Maze because they provide sparse goal rewards and pose hard-exploration challenges for standard PPO baselines, especially on the hard distribution. In all four environments the agent receives a reward only upon reaching the goal, with no intermediate shaping rewards, and the episode terminates upon reaching the goal or after 1000 steps.

Coinrun - A simple platformer in which the agent must collect the coin at the far right of the level to receive a reward of 10. The agent must jump over obstacles, avoid stationary traps and pitfalls, and dodge moving enemies to safely reach the coin.

Heist - A maze environment in which the agent must navigate corridors, collect keys to unlock doors, and reach the gem at the center of the level, which gives a reward of 10.

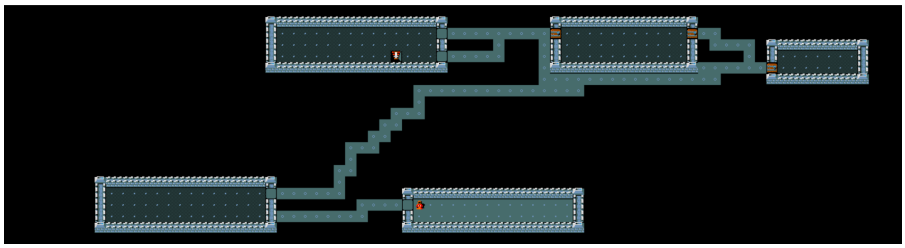
Jumper - A platformer in which the agent must navigate through the world to find the carrot which gives a reward of 10. The agent can double jump to navigate around spike obstacles and reach high platforms.

Maze - A standard maze environment in which the agent must navigate to a randomly placed goal state to earn a reward of 10.

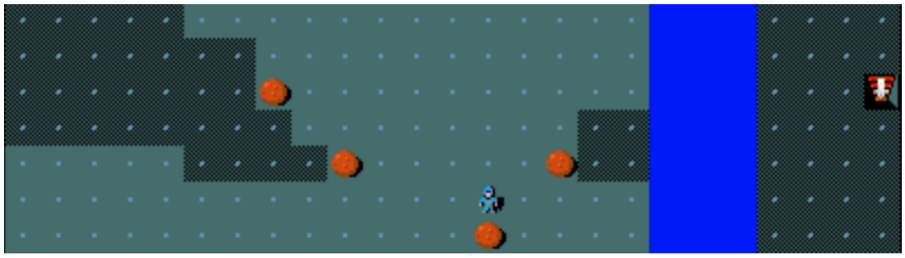
All experiments use the “hard” setting which features larger and more challenging environments than the“easy” distribution that is more commonly studied in the literature. However, the hard distribution

still includes some small or easy instances. This allows random agents to achieve moderate solve rates, and with a sufficient number of seeds, forms a natural curriculum (Cobbe et al., 2020). Of the environments we study, PPO is only able to solve Maze, which can be attributed to this curriculum of levels. Additionally, roughly 7% of Jumper levels are unsolvable due to the agent spawning on an obstacle, so we filter these seeds out during data collection.

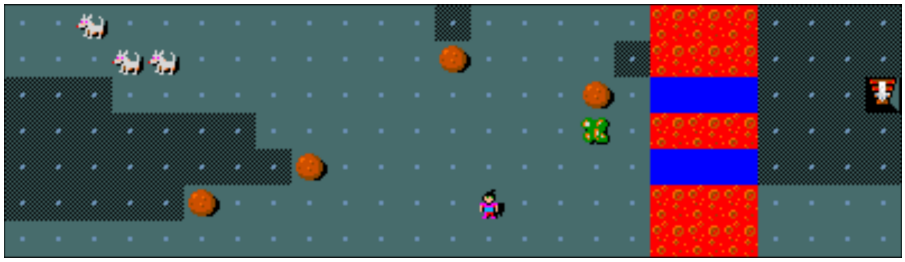
A.2 MiniHack



(a) Corridor



(b) River



(c) Monster Lava

Figure 6: MiniHack environments.

MiniHack is a sandbox for designing small RL scenarios using the mechanics of NetHack, a procedurally generated rogue-like game with complex item interactions, monsters, traps, and partial observability. MiniHack exposes this complexity through smaller, configurable tasks that are more tractable for current RL methods. MiniHack was released with several prebuilt environments that test a variety of agent capabilities, including skill acquisition, exploration, planning, and memory (Samvelyan et al., 2021).

Agents receive complex observations including the ASCII glyphs currently on the screen, the agent's current attributes such as health and strength, any message resulting from the agent's previous action, and many more features. The action space includes between 8 to 75 discrete actions selected from the full space of actions available in NetHack. More information about these environments can be found in (Samvelyan et al., 2021). We study the Corridor and River environments because they have sparse rewards and present hard exploration challenges. In both environments, the agents receive a single reward of 1.0 for reaching a goal state (the staircase to the next dungeon level). MiniHack environments typically include a step penalty to encourage faster progress, but we disable the penalty

in this paper. This is standard practice for our MiniHack PPO baseline, described in Appendix N. Episodes terminate when the agent reaches the staircase goal state or after a fixed number of steps.

Corridor (MiniHack-Corridor-R5-v0) — A navigation environment in which the agent must reach a staircase located in one of several randomly generated rooms. As in NetHack, rooms are connected by long corridors that are initially obscured by fog of war. Progress may be blocked by boulders, locked doors, or hidden corridors. Boulders must be pushed aside or navigated around, while locked doors can be opened or kicked down with a small probability of success. Hidden corridors must be revealed using the search action, which uncovers nearby tiles probabilistically and may require repeated attempts. Together, these mechanics make Corridor substantially more challenging than standard navigation. The environment comes in 2-, 3-, and 5-room variants; we introduce an additional 8-room variant in subsection 4.2. The action space consists of eight movement actions, search, open, and kick, and episodes terminate after 1,000 steps.

River (MiniHack-River-v0) — A puzzle environment in which a three-tile-wide river separates the agent from the staircase. Entering the river forces the agent back to land and carries a small risk of drowning, so the agent must push nearby boulders into the water to construct a bridge. Two boulders are often sufficient, allowing the agent to swim across the final tile. The puzzle contains many irreversible failure states, such as pushing boulders into corners where they can no longer be moved or placing them in positions that cannot form a bridge. The action space consists of eight movement actions, and episodes terminate after 350 steps.

Monster Lava (MiniHack-River-MonsterLava-v0) — A harder variant of River that replaces the river with lava and adds hostile monsters. Lava kills the agent immediately, while monsters can kill the agent by reducing the agent's health through attacks. The agent can fight and kill monsters while simultaneously solving the boulder-placement puzzle. The action space consists of eight movement actions, search, open, and kick, and episodes terminate after 350 steps.

B CELL REPRESENTATIONS

Procgen environments have similar observations and dynamics to the Atari games studied by Ecoffet et al. (2021), so we adapt their cell representations in these environments. For Procgen, we first convert observations to grayscale, apply maximum pooling to downsample the image to the target cell width and height, and quantize grayscale values into the number of bins. The exact values for these operations are listed in Table 2. In the Heist environment, we also added the number of collected keys to the cell representation. Although the number of keys is visible in the observation and can be identified from higher resolution representations (i.e 32x32 images), this dramatically increases the number of cells in the archive and decreases environment sample throughput. It was not difficult to find cell representations that allow Go-Explore to solve these environments, but unoptimized values had a significant impact on the speed of experiments and therefore the computational costs. Cell representation values were manually tuned to accelerate data collection and reduce experimental costs. Appendix K describes our implementation of Go-Explore and the interaction between cell representations and data collection throughput.

The MiniHack environments all use the same cell representation. It contains the x and y position of the agent, each downsampled into bins of size 2, the agent's total experience, and the glyphs in a 3 by 3 window centered on the agent. This allows the cell representation to capture the agent's approximate location and nearby objects, which are necessary for many MiniHack tasks. The x and y bin values were manually tuned on hard instances of each problem to allow Go-Explore to find a successful trajectory within 10,000,000 environment steps, and to increase throughput. All cell representations for Procgen and MiniHack are also hashed to reduce the latency of memory costs of transferring and storing images.

C ADDITIONAL CELL SAMPLING DETAILS

For each training instance, GAGE runs a policy-free Go-Explore search following many of the conventions of Ecoffet et al. (2021). At each exploration attempt, GAGE samples a return cell from the archive with weight $\frac{1}{\sqrt{N_{\text{seen}}+1}}$ where N_{seen} is the number of previous exploration attempts in which that cell was visited. If the archive is empty, GAGE returns to the initial state. Terminal cells

Table 2: Cell representation hyperparameters for Procgen environments. The Heist cell representation also includes the number of collected keys.

Environment	Cell Width	Cell Height	Grayscale Bins
Coinrun	8	8	4
Heist	8	8	4
Jumper	4	4	4
Maze	16	16	4

are excluded from sampling, as are cells that have failed to produce a new cell over 25 exploration attempts. These stale cells are marked for replacement the next time they are encountered. If this filtering were to remove every cell from the sampling distribution, GAGE would instead sample uniformly from the archive, although this case never occurred in our experiments.

After returning to the selected cell, GAGE explores for a fixed step budget or until the episode terminates. During exploration, actions are sampled randomly, with a high probability of repeating the previous action to encourage temporally consistent movement, as in Ecoffet et al. (2021). However, in MiniHack, these repeat actions are disabled in 50% of episodes to allow Go-Explore to discover both precise and long-ranging behaviors. NOOP actions which cause the agent to stand still are not repeated, because they are not strategically necessary in any of the environments we study.

D Budget Constrained Early Stopping

One challenge of searching for expert trajectories is that we do not know what the best trajectory length for a problem instance is a priori. It is therefore possible to waste significant compute attempting to optimize a trajectory that cannot be improved further. For the budget constrained experiments in subsection 4.1 we introduce several early stopping mechanisms to avoid spending excessive compute on improving near-optimal trajectories. Search is terminated for a seed if any of the following conditions are met:

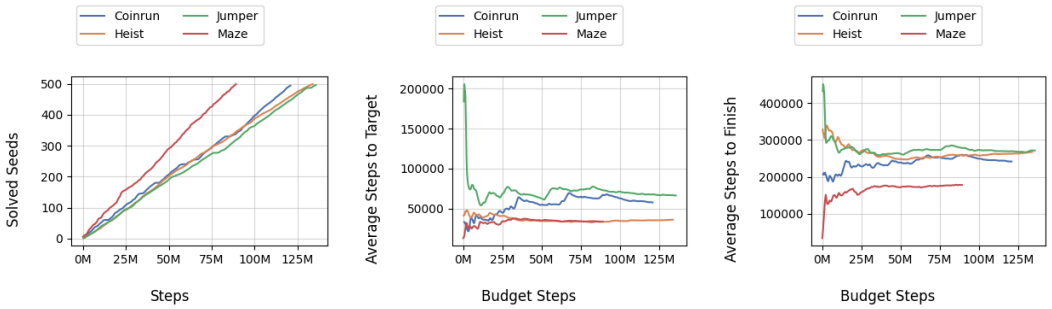
1. More than T steps have been spent on the seed.
2. A successful trajectory shorter than 20 steps has been found.
3. A successful trajectory has been found, and there has been less than a 5% decrease in length in the last 300,000 steps.

These conditions were chosen based on observing search progress on a small number of seeds. T is set to 200,000 in Procgen and 2,000,000 in MiniHack. It may also be possible to adjust these values adaptively, such as by increasing the number of steps between improvement checks when search is spending less of the budget than anticipated. We also note that roughly 7% of seeds for the Jumper environment are unsolvable, because the agent spawns directly on top of an obstacle, resulting in a length 0 episode with 0.0 return. This is a known issue reported in the GitHub repository for Procgen (Cobbe, Karl and Hesse, Chris and Hilton, Jacob and Schulman, John, 2019). When we encounter this issue, we discard the seed, do not count any steps taken in that instance, and sample a new seed.

E Demonstration Generator Solve Rates and Costs

To our knowledge, there are no search-based methods besides Go-Explore that can solve sparse-reward RL environments end-to-end. However, any agent can be used to generate demonstrations to some degree. We test a trained RND agent on the full distribution of a Procgen environment and per-instance RND agents trained on individual Procgen instances as trajectory generators. We present the solve rates in Table 3 and the costs of these methods as well as human data labeling in Table 4.

Not only does Go-Explore solve more instances, but it costs orders of magnitude less to solve each instance. We present the cost of each trajectory generation method in Table 2 below, plus an estimate of the human data labeling cost based on Mechanical Turk rates. At the observed 10% solve rate



(a) Seeds solved across search. (b) Mean steps until goal is found. (c) Mean steps used per seed.

Figure 7: Progression of search during budget constrained experiments.

for per-instance RND, obtaining 100 successful demonstrations would require approximately 1,000 attempted instances, yielding an estimated cost of $88 USD, or over 300× the cost of obtaining 100 successful demonstrations with Go-Explore. Human data labeling could cost up to $100 USD, while Go-Explore costs only $0.27 USD in expectation.

Table 3: Solve rates across environments for Frozen RND, Per-Instance RND, and Go-Explore.

Environment	Frozen RND	Per-Instance RND	Go-Explore
Coinrun	64%	61%	100%
Heist	52%	49%	100%
Heist (3 Key)	20%	10%	100%
Jumper*	54%	46%	98%
Maze	100%	85%	100%

Table 4: Estimated costs of each data collection method to attempt solving 100 seeds of 1 Procgen game. Automatic methods are priced against a g2-standard-8 instance and e2-standard-8 on Google Cloud Platform. Human demonstration prices are estimated from Mechanical Turk pricing assuming 1 minute per instance. Costs are roughly estimated from Procgen experiment logs and averaged across environments. The frozen RND agent has an additional one-time pretraining cost that can be amortized across demonstrations.

Unit	Frozen RND	Per-Instance RND	Go-Explore	Human Label
GPU hours	120 (one time) + 3	10	0	–
CPU hours	960 (one time) + 24	83	8	–
Cost (USD)	$102.40 (one-time) + $2.56	$8.80	$0.27	$50–$100

F Trajectory Optimization Ablation

We ablate the importance of trajectory optimization on GAGE-BC and GAGE-RL training. We use the same methodology as subsubsection 4.1.1, except we limit Go-Explore to 5 trajectories in each problem instance to control for the dataset size produced by each method. We run the experiment with and without trajectory optimization enabled for 5 seeds of each Procgen game, and evaluate the median score of the resulting policies on 1000 held-out test instances.

The results are presented in Table 5. We find that GAGE-BC agents trained with trajectory optimization match or exceeds the performance of agents trained without trajectory optimization, producing

significant improvements in Heist ($p = 0.0119$) and Jumper (0.0079. However, we find that there is no significant difference between the performance of GAGE-RL agents with and without trajectory optimization. As such, we cannot recommend trajectory optimization for broader GAGE-RL training. However, trajectory optimization is beneficial for GAGE-BC, and produces the largest (non-significant) improvement for GAGE-RL in Heist, the sparsest Procgen setting. As such, we feel it is a worth considering in work future extending GAGE to increasingly challenging domains.

Table 5: Median checkpoint-level mean episode returns and per-environment Mann–Whitney U tests for trajectory-optimization ablations.

	BC			RLFT		
Environment	Opt	No-Opt	p (raw)	Opt	No-Opt	p (raw)
Coinrun	5.26	5.04	0.0556	7.81	7.60	0.9048
Heist	5.74	5.03	0.0119	8.11	6.93	0.0556
Jumper	3.58	3.24	0.0079	5.94	7.43	0.2222
Maze	4.18	4.42	0.5296	6.00	9.02	0.6905

G Additional Procgen Results

The main paper demonstrated how GAGE can achieve high scores in sparse-reward RL environments with a relativly high number of environment instances and environment interactions. Here we evaluate whether GAGE remains competitive with PPO under the much lower instance and interaction budget proposed by Cobbe et al. (2020). In subsection 4.1 PPO plateaued at moderate solve rates, while GAGE performed well with an uncapped budget for data collection, but this does not show how each method would perform in a sample-constrained setting.

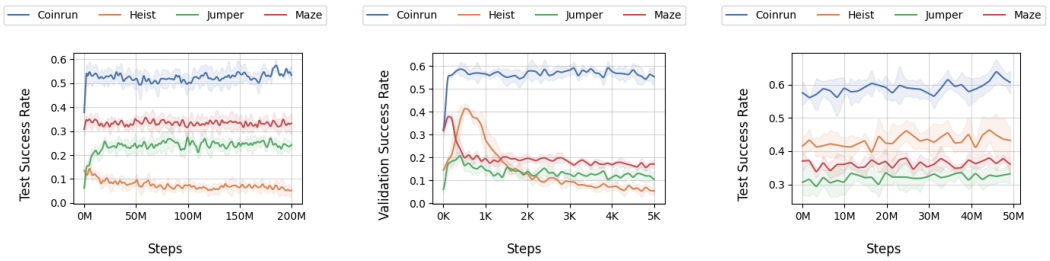
Following the evaluation protocol suggested by Cobbe et al. (2020), we evaluate PPO and GAGE on the Procgen environments with a fixed budget of 500 training seeds and 200M environment steps. We allocate up to 150M steps to Go-Explore to generate trajectories across the same 500 seeds and reserve 50M steps for RL finetuning to train GAGE-RL. Both the steps required to return to a state and the exploration steps are deducted from this budget. GAGE-BC is trained only on the trajectories collected during the 150M environment steps. During RL finetuning, we freeze all weights except the critic for the first 500 gradient updates, following prior work that uses a critic-only warmup phase to stabilize RL finetuning of behavior-cloned policies (Baker et al., 2022). To avoid wasting environment interactions on easy problem instances, we apply per-seed early stopping mechanisms to Go-Explore when a sufficiently short trajectory is found, and when there has been no recent improvement in trajectory length. More details for these conditions can be found in Appendix D.

Table 6: Median performance of each method over 5 training seeds with a small, fixed environment interaction budget. Statistically significant improvements over PPO are bolded.

Environment	Random	PPO	RND	GAGE-BC (Ours)	GAGE-RL (Ours)
Coinrun	0.255	0.481	0.525	0.537	0.565
Heist	0.178	0.086	0.072	**0.297**	**0.403**
Jumper	0.085	0.301	0.335	0.288	0.353
Maze	0.367	0.399	0.417	0.430	0.419
Mean	0.224	0.290	0.33725	**0.415**	**0.437**

Figure 8 shows the training curves for each environment and Table 6 shows that GAGE-RL matches or exceeds PPO in all environments under the same environment-interaction budget. Both GAGE-BC and GAGE-RL outperform PPO on Coinrun and Heist ($p < 0.012$) and match PPO on Jumper and Maze. We also examine how Go-Explore uses the data-collection budget under our early stopping criteria. Across environments, Go-Explore requires fewer than 50,000 steps on average to solve a seed

and fewer than 250,000 steps to converge to an optimized trajectory. The data-collection budget is an upper bound rather than a target, so GAGE does not use the full budget in any environment, consuming between 59.3% of the budget in Maze and 90.4% in Jumper. This suggests that performance could be improved by adapting early stopping thresholds based on the projected sample budget usage or reallocating unused data-collection steps to RL finetuning. Improving GAGE's sample efficiency therefore remains an important direction for future work.



(a) PPO (b) GAGE-BC (c) GAGE-RL

Figure 8: Training curves for the results in Appendix G. Each method is trained on 500 seeds with a fixed budget of 200M environment steps.

H Behavior Cloning Data Quality

In order to better understand how Go-Explore's data collection interacts with behavior cloning and finetuning, we construct alternative datasets based on the Procgen datasets used in subsubsection 4.1.2 and compare the performance of GAGE-BC and GAGE-RL agents trained on them. The original datasets include all successful demonstrations produced by Go-Explore. We create a "Shortest" dataset by taking only the shortest trajectory from this set, and a "Longest" dataset which includes only the longest demonstration for each problem instance.

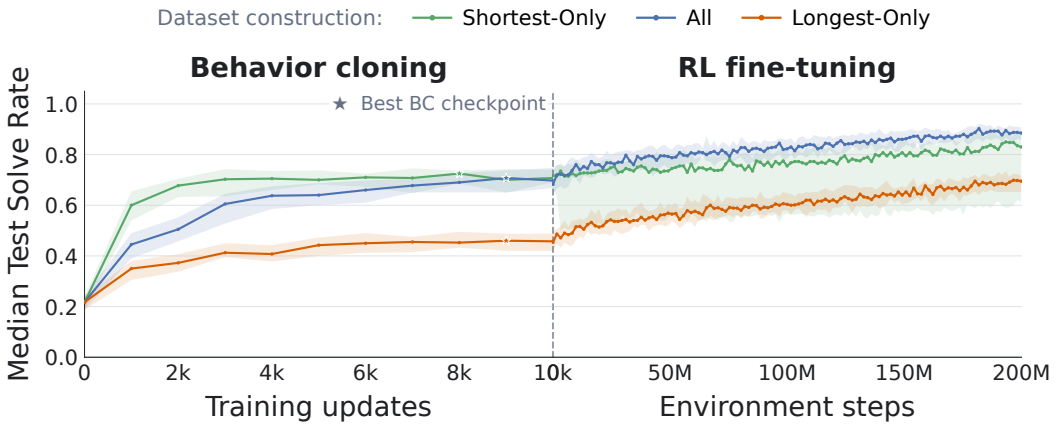
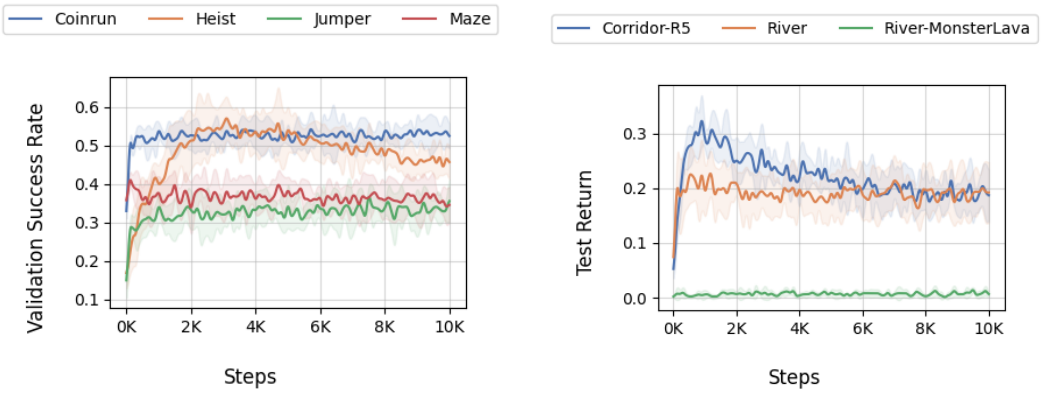


Figure 9: Training curves averaged across Coinrun, Heist, Jumper, and Maze for the Shortest, All, and Longest dataset constructions. Curves show mean test solve rate across seeds with min–max shading. The left panel shows behavior cloning and the right panel shows RL fine-tuning. RL finetuning starts from the best BC checkpoint, not the last.

Figure 9 shows the results of the experiment. We find that using all data performs the best for both BC and RL finetuning. The Shortest-Only dataset performs better than the Longest-Only dataset,

I Training Curves

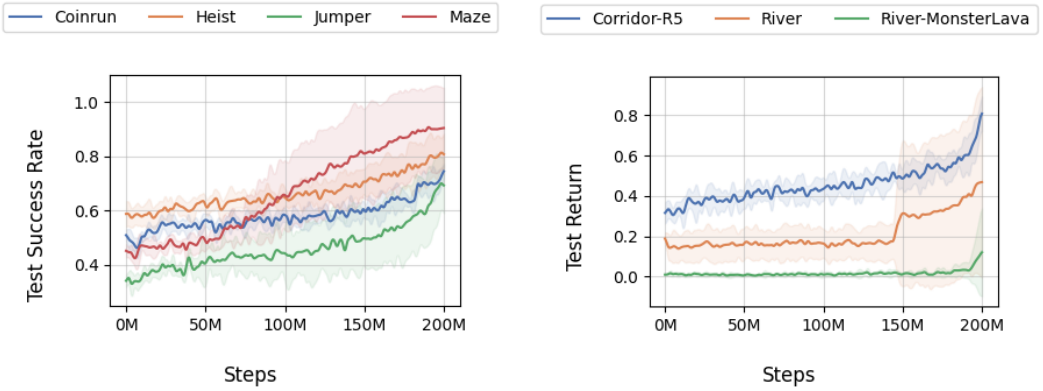
This section contains training curves for all of the learning experiments in our paper. In every plot, agents are evaluated every 100 updates on a held out set of 100 seeds. For BC experiments, this is the set used to select the best checkpoint for GAGE-BC and to initialize GAGE-RL. Figure 12, Figure 13, and Figure 14 contain all of the training curves for the experiments in subsection 4.1. Figure 8a, Figure 8b, and Figure 8c contain all of the training curves for the experiments in Appendix G. Figure 15, Figure 16, and Figure 17 contain all of the training curves for the experiments in subsection 4.2.



(a) Procgen BC

(b) MiniHack BC

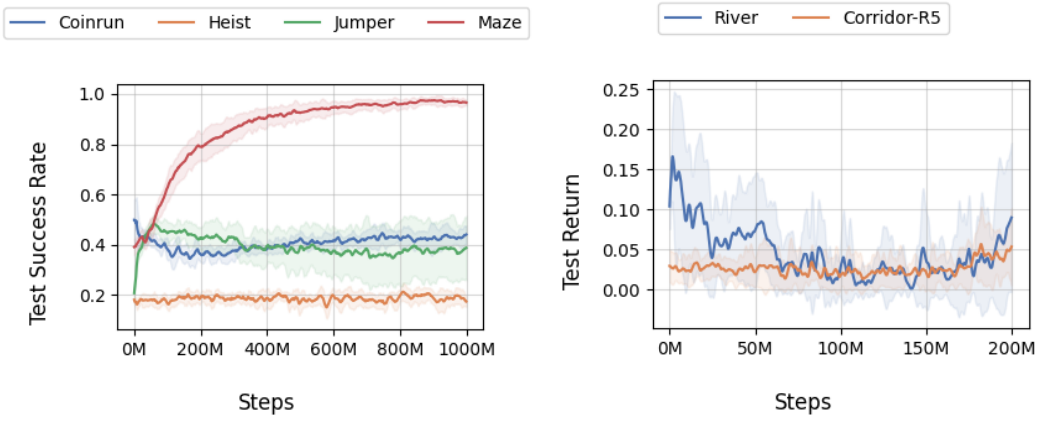
Figure 10: Training curves for GAGE-BC in subsubsection 4.1.1. Each method is trained on 20,000 seeds.



(a) Procgen RL fine-tuning

(b) MiniHack RL fine-tuning

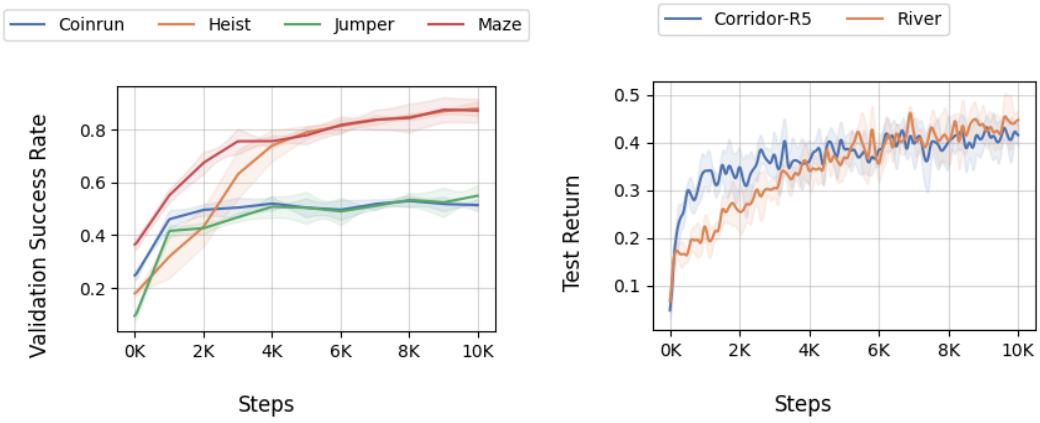
Figure 11: Training curves for GAGE-RL in subsubsection 4.1.1. Each method is trained on 20,000 seeds.



(a) Procgen PPO baselines

(b) MiniHack PPO baselines

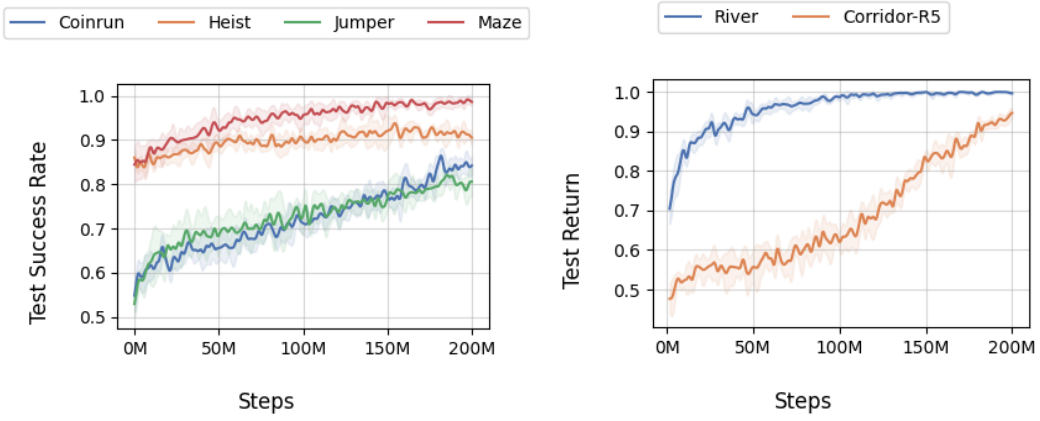
Figure 12: Training curves for PPO in subsubsection 4.1.2. Each method is trained on 20,000 seeds.



(a) Procgen GAGE-BC

(b) MiniHack GAGE-BC

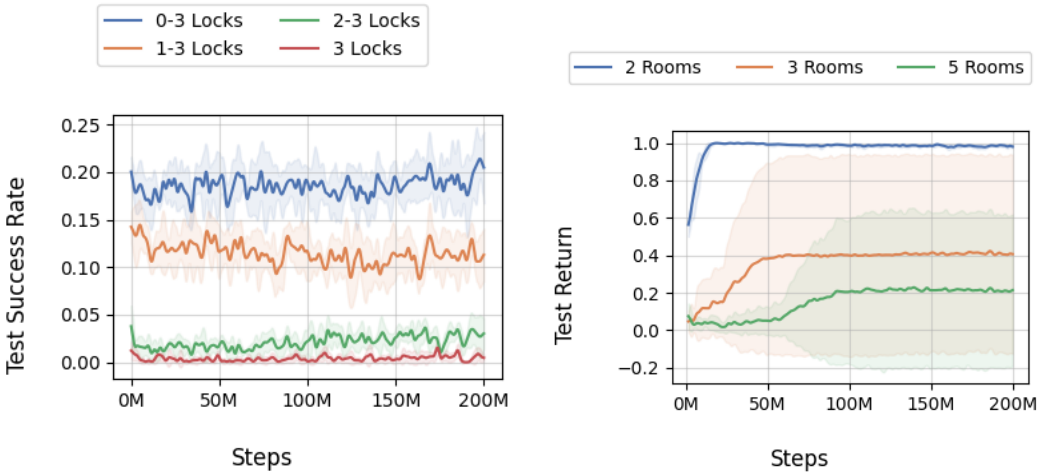
Figure 13: Training curves for GAGE-BC in subsubsection 4.1.2. Each method is trained on 20,000 seeds.



(a) Procgen GAGE-RL fine-tuning

(b) MiniHack GAGE-RL fine-tuning

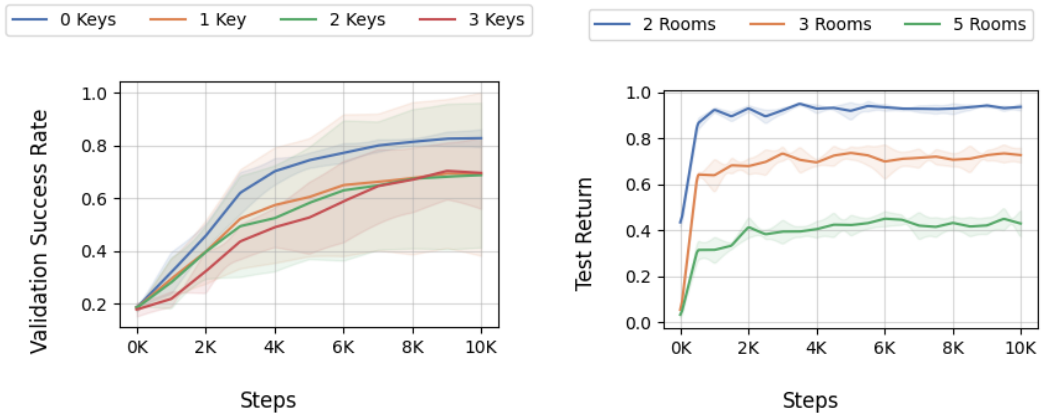
Figure 14: Training curves for GAGE-RL in subsubsection 4.1.2. Each method is trained on 20,000 seeds.



(a) Procgen PPO baselines

(b) MiniHack PPO baselines

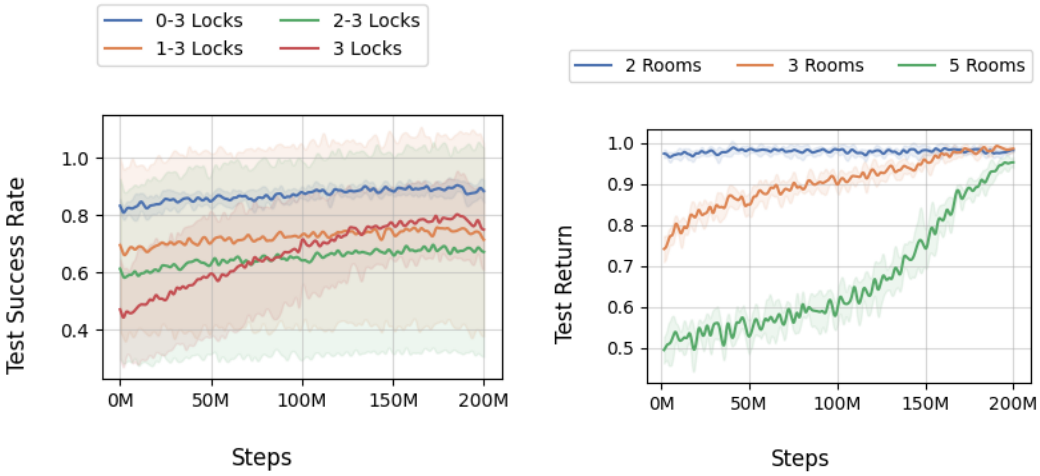
Figure 15: Training curves for PPO in subsection 4.2.



(a) Procgen GAGE-BC

(b) MiniHack GAGE-BC.

Figure 16: Training curves for GAGE-BC in subsection 4.2.



(a) Procgen GAGE-RL fine-tuning

(b) MiniHack GAGE-RL fine-tuning

Figure 17: Training curves for GAGE-RL in subsection 4.2.

J Scaling Problem Instances

In this section we study how the performance of GAGE scales with the number of training problem instances. We train GAGE on sets of Procgen instances of varying sizes, allowing Go-Explore to use up to 2M samples to solve each instance. Unlike standard behavior cloning, where dataset size is typically measured in trajectories, we scale the number of distinct problem instances, each of which may yield multiple demonstrations. This distinction is particularly relevant for data-driven simulators, which can generate many samples per scenario but may provide only a limited number of distinct scenarios. Such simulators are increasingly used to train agents for real-world settings such as autonomous driving and computer use (Wu et al., 2025; Gulino et al., 2023; Da et al., 2025). We study the scaling properties of GAGE when instances are limited but environment interactions are cheap.

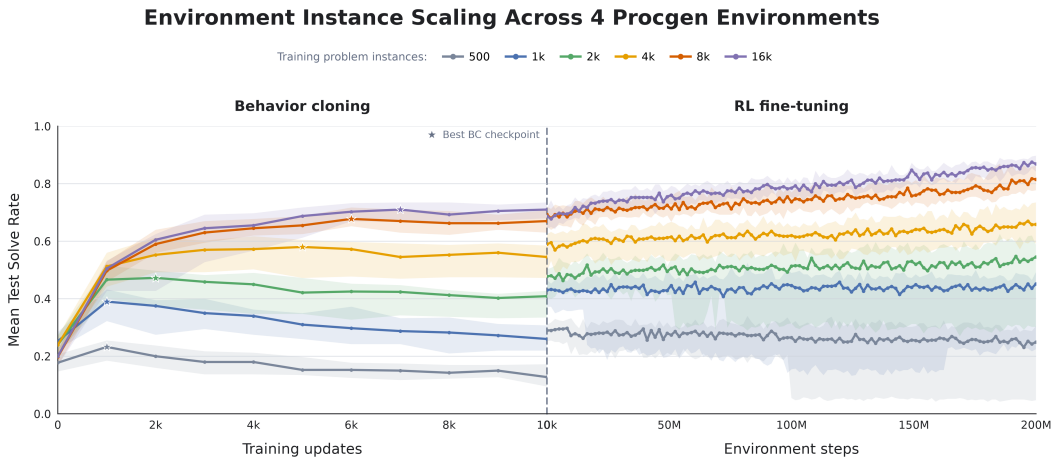


Figure 18: Normalized mean solve rate across four Procgen environments (five seeds each), evaluated on 1,000 unseen seeds. RL finetuning starts from the best BC checkpoint for each dataset size.

Figure 18 shows BC training followed by RL finetuning for each dataset size. We find that increasing the number of training instances consistently improves BC test performance. RLFT can further improve the converged GAGE-BC policy, but its effectiveness depends strongly on dataset size. With fewer than 1,000 training instances, policies collapse early in RLFT and do not recover, and below 4,000 instances RLFT provides little benefit.

At 4,000 instances and above, RLFT produces clear and sustained gains, with faster improvement as the dataset grows. For example, with 4,000 instances, mean performance increases from 58% to 68%, but with 16,000 instances it increases from 62% to 85%. Moreover, the larger-instance GAGE-RL runs have not converged after 200M environment steps, suggesting that additional RL finetuning may yield further gains. Overall, GAGE improves substantially as the number of training problem instances increases, both before and during RL finetuning.

K System Architecture

We implement Go-Explore using Syllabus, based on the original open-source implementation at `https://github.com/uber-research/go-explore`. We use Syllabus because it provides an API for portable curriculum learning methods, allowing curricula to be applied to existing RL codebases without modification (Sullivan et al., 2025). Syllabus also handles synchronization across environments running in separate processes, which lets us maintain a single global archive while running many environments in parallel. This design makes it easier for future work to apply our method to new environments without additional engineering effort.

Our implementation has two main components: a global Go-Explore curriculum and a wrapper applied to individual environments. The curriculum maintains the archive of promising discovered states and samples cells for workers to return to. Each environment wrapper computes the cell representation for every state and periodically sends newly discovered states to the curriculum.

Each environment wrapper also maintains a local archive with partial information about the current global archive. This local archive stores the cell representations for all encountered cells, along with the trajectory length and return of the current occupant of each cell. Local archives are periodically updated with recent changes from the global curriculum. Rather than sharing the full archive, the curriculum sends only a small, versioned delta. Each worker stores the most recent version number it has received, and the global curriculum uses this version number to compute the next archive delta.

The local archive lets each worker decide whether a state should be proposed for the global archive. A state is sent to the curriculum if it occupies an empty cell, has a higher return than the current cell occupant, or has the same return with a shorter trajectory length. When a state passes these checks, the worker sends a larger update containing the information needed to return to that state. This includes the simulator state, the list of trajectory segments used to reach the cell, the trajectory of cells visited during the exploration attempt, the actions taken during the exploration attempt, and a copy of the final observation and info, which are needed for the first step of the next exploration attempt. These larger updates are sent only for promising states, so the amount of data transferred from workers to the curriculum is substantially reduced. This allows our implementation to reach speeds of 20,000 steps per second in Procgen and MiniHack across 16 workers.

Saving trajectories for every cell in the archive can require substantial memory, especially in long-horizon environments with long episodes. In addition, whenever a cell is sampled, the trajectory leading to that cell must be saved separately. This prevents race conditions in which two workers update the same cell at the same time, causing a later update to produce an invalid trajectory. Repeatedly copying these trajectories for each sampled cell can reduce throughput.

We reduce this cost by exploiting the structure of Go-Explore. Go-Explore reliably returns to states before exploring from them, so all cells discovered during an exploration attempt share a trajectory prefix with the sampled return cell. We therefore store trajectories as an array of segments, where each segment contains the actions from a single exploration attempt, and store segment indices instead of full trajectories. In practice, we send the relevant trajectory segments to each worker; these segments could instead be versioned on the curriculum side to avoid repeatedly sending the same data across processes, but we found the cost of sending trajectory segments to be negligible.

We return to states either by restoring simulator save states, in Procgen, or by replaying a saved action sequence in a seeded, deterministic environment, in MiniHack. When simulator states are used, we save them to temporary files and send the file paths to the global curriculum. This occurs only when a promising state is discovered according to the local archive. For action-sequence replay, we reuse the saved trajectory segments.

Together, these optimizations substantially improve the compute efficiency of our implementation. Go-Explore consumes environment samples up to 10 times faster than our PPO training code. As a result, for the same environment interaction budget, our approach is not only more sample efficient (subsection 4.1) but also more time efficient (Appendix L).

L Computational Costs

The computational requirements of experiments in each section, as well as the total computational hours for the results presented in this paper, are displayed in Table 7. These values are taken directly from experiment logs, except for a small number of values marked as estimates. The development of our method and hyperparameter tuning required additional compute not listed here. Experiments were conducted on a large variety of CPUs and GPUs, so the numbers shown are approximate averages. The majority of experiments were run on NVIDIA 1080Ti, NVIDIA RTX 2080 Ti, NVIDIA RTX 5080, NVIDIA RTX 5090, and NVIDIA L40S GPUs. As a result, it is not possible to directly compare the computational requirements of each method in general from our experiments. These values represent the actual costs incurred to perform the experiments, rather than the costs one would expect from reproducing the results on different hardware. We present these values instead for transparency and as a reference for practitioners.

Table 7: Computational cost of the GAGE experiments, aggregated by paper section.

Section	Resource	GAGE			
		Data	BC	RL	Total
Equal-budget (subsubsection 4.1.1)	GPU-hours	0	51	275	326
	CPU-hours	4,309	408	2,201	6,917
Additional Search (subsubsection 4.1.2)	GPU-hours	0	146	733	879
	CPU-hours	38,400[1]	1,164	7,326	46,890
Sparsity (subsection 4.2)	GPU-hours	0	123	703	827
	CPU-hours	23,040[1]	986	6,056	30,082
Generalization (subsection 4.3)	GPU-hours	0[2]	54	290	344
	CPU-hours	–[2]	433	4,633	–
Trajectory optimization ablation (Appendix F)	GPU-hours	0	184	585	769
	CPU-hours	7,763	1,472	5,021	14,256
Low-Data (Appendix G)	GPU-hours	0	22	136	158
	CPU-hours	260	174	1,630	2,064
Instance scaling (Appendix J)	GPU-hours	0[2]	205	1,620	1,826
	CPU-hours	–[2]	1,642	12,963	–
Overall total	**GPU-hours**	**0**	**785**	**4,342**	**5,127**
	CPU-hours	**73,772**	**6,279**	**39,829**	**119,880**[3]

M Prior Extensions to Go-Explore

This section compares GAGE with the original Go-Explore variants and subsequent extensions. GAGE builds on policy-free Go-Explore, but uses it as a data-generation procedure across many task instances rather than as a solver for a single environment.

Policy-free Go-Explore (Ecoffet et al., 2021): Policy-free Go-Explore returns to archived states by restoring simulator states or replaying action sequences in deterministic environments. GAGE uses this mechanism to discover successful trajectories in individual instances of a task distribution. However, GAGE adds a trajectory optimization phase to improve these solutions, aggregates trajectories across many instances, and distills the resulting dataset into a generalist policy.

In the original Go-Explore work, successful trajectories discovered by policy-free search are converted into policies through a robustification phase. This phase trains a policy from a demonstration trajectory

[1] Values are estimated from a small sample of experiments.

[2] Reused data from prior experiments.

Table 8: Computational cost of the PPO and RND baselines, aggregated by paper section.

Section	Resource	Baseline	
		PPO	RND
Equal-budget (subsubsection 4.1.1)	GPU-hours	1,713.56	1,806.61
	CPU-hours	19,220.80	44,314.49
Sparsity (subsection 4.2)	GPU-hours	559.88	0
	CPU-hours	4,479.07	0
Low-Data (Appendix G)	GPU-hours	452.08	224.84
	CPU-hours	5,424.91	3,426.27
Overall total	**GPU-hours**	**2,725.52**	**2,031.45**
	CPU-hours	**29,124.79**	**47,740.76**

using a reverse curriculum: the agent starts near the end of the trajectory and the initial state is gradually moved earlier as the agent learns to reliably reach the goal. This produces policies that are robust to stochasticity in single environments, but it is unclear whether the same procedure scales to broad task distributions. GAGE instead trains from many optimized trajectories using imitation learning followed by reinforcement learning, enabling generalization to unseen instances.

Policy-based Go-Explore (Ecoffet et al., 2021): Policy-based Go-Explore trains a goal-conditioned policy to return to archived states, reducing the need for a deterministic or restorable simulator. To return to a target state, the policy is conditioned on cells along the trajectory to that target. During exploration, actions are selected both randomly and from the trained return policy.

Applying this approach directly to broad task distributions introduces new challenges. In a single environment, archived cells correspond to states with known reachability and exploration value. Across many instances, the same cell representation may correspond to states with different reachability, return paths, or future exploration potential. One could maintain a separate archive for each instance and condition the return policy on both the instance and target cell, but this would reduce the opportunity for generalization across instances. Adapting policy-based Go-Explore to broad task distributions is therefore a promising direction for future work, but requires empirical validation.

Cell-Free Latent Go-Explore (Gallouédec & Dellandréa, 2023): Cell-Free Latent Go-Explore builds on policy-based Go-Explore by replacing manually engineered cell representations with latent encodings derived from the policy's observation encoder. This work addresses how to define useful cells without manual tuning, which is complementary to GAGE. GAGE instead studies how per-instance Go-Explore solutions can be aggregated and distilled into policies that generalize across sparse-reward task distributions.

First-Return, Entropy-Eliciting Explore (FR3E) (Zheng et al., 2025): FR3E is conceptually related to Go-Explore, but applies search-like ideas to text-based reasoning. Rather than maintaining an explicit archive of promising states to revisit, FR3E performs autoregressive rollouts until it reaches high-entropy decision points, then branches into multiple completions to identify stronger continuations. FR3E trains a general policy across text-based tasks, but it differs from GAGE in both mechanism and setting: it does not use a persistent archive or return mechanism, and it studies text reasoning rather than procedurally generated sparse-reward environments.

N HYPERPARAMETERS

The Procgen benchmark used in all experiments is the PPO model architecture from Cobbe et al. (2020). The feature extractor is a ResNet CNN architecture from IMPALA (Espeholt et al., 2018). MiniHack experiments train the CDGPT5 architecture introduced by Hambro et al. (2022). We chose this architecture over the one introduced with MiniHack because the CDGPT5 model outperforms the MiniHack model when both are trained and evaluated on NetHack (Hambro et al., 2022). We also found that it performs better on dense reward MiniHack environments in preliminary testing

and on the Corridor-R3 environment presented in subsection 4.2, making it a stronger baseline for comparison.

The key hyperparameters for generating data with Go-Explore are the probability of loading a state from the archive instead of starting from the initial state, exploration timestep limit, whether to use action repeat, and the cell representations arguments. We describe the cell representations and their hyperparameters in Appendix B. For Procgen, most hyperparameter choices resulted in a near 100% solve rate, so hyperparameters were tuned manually to improve environment interaction throughput and sample efficiency. MiniHack hyperparameters were found through a small grid search on 10 seeds over the archive load probability, exploration timestep limit, and whether to use action repeat or a 50% chance per episode of using action repeat. This search was conducted on Corridor-R5 and River, and settings that maximized solve rate and minimized final trajectory length were chosen.

Behavior cloning hyperparameters were found through a grid search over the batch size, the number of minibatches, and the learning rate. We run 3 seeds each of Heist and Maze for Procgen and 3 seeds of Corridor-R5 for MiniHack. These hyperparameters are reported in Table 12.

For reinforcement learning, we use tuned baselines for the environments with shared hyperparameters across all environments. The Procgen hyperparameters were tuned by (Cobbe et al., 2020) and the MiniHack hyperparameters were reported in (Piterbarg et al., 2023) which similarly studied imitation learning and reinforcement learning on NetHack. These hyperparameters are reported in Table 9. Our RND implementation follows the recommendations from the 2020 Procgen NeurIPS Competition (Mohanty et al., 2021). Specifically, we treat RND as a lifelong novelty bonus and do not clip it at episode boundaries. We perform a grid search of RND's reward coefficient 0.1, 0.5, 1.0, 10.0, 100.0, the RND module's learning rate 0.001, 0.0001, 0.0005, the RND module's hidden size 128, 256, and the RND gamma value 0.99, 0.999, on Heist and Maze. However, none of the values improved the performance of RND on Heist, and there was little variation in Maze, so we chose to keep the tuned values used in the original RND paper (Burda et al., 2019) and the Procgen competition.

Table 9: PPO hyperparameters for the Procgen and MiniHack experiments.

Hyperparameter	Procgen	MiniHack
Learning rate	$5e{-4}$	$1e{-4}$
Learning-rate annealing	False	False
Adam epsilon	$1e{-5}$	$1e{-7}$
Parallel environments	64	64
Rollout / sequence length	256	256
Batch size	16384	16384
Minibatches	8	8
PPO epochs	3	3
Discount factor γ	0.999	0.995
GAE λ	0.95	0.95
Clip coefficient ϵ	0.2	0.1
Entropy coefficient	0.01	0.001
Value loss coefficient	0.5	1.0
Max grad norm	0.5	0.5

Table 10: PPO and RND hyperparameters for Procgen and MiniHack experiments.

Hyperparameter	Procgen	MiniHack
Learning rate	$5e{-4}$	$1e{-4}$
Learning-rate annealing	False	False
Adam epsilon	$1e{-5}$	$1e{-7}$
Parallel environments	64	64
Rollout / sequence length	256	256
Batch size	16384	16384
Minibatches	8	8
PPO epochs	3	3
Discount factor γ	0.999	0.995
GAE λ	0.95	0.95
Clip coefficient ϵ	0.2	0.1
Entropy coefficient	0.01	0.001
Value loss coefficient	0.5	1.0
Max grad norm	0.5	0.5
RND hyperparameters		
RND predictor hidden size	128	128
RND reward coefficient	0.5	0.5
RND discount factor γ_{RND}	0.99	0.99
RND observation-norm clip	5.0	5.0
RND observation-norm initialization updates	50	50
Intrinsic predictor learning rate	$5e{-4}$	$1e{-4}$
Intrinsic loss coefficient	1.0	1.0
RND update proportion	0.5	0.5

Table 11: GAGE data generation hyperparameters for the Procgen and MiniHack experiments.

Hyperparameter	Procgen	MiniHack
Number of workers	15	15
Exploration steps	100	200
Action repeat	95%	90%
Action repeat dropout	0%	50%
Archive load probability	75%	99%
Timesteps per seed	3e6	2e6
Early stopping window	1e5	1e5

Table 12: GAGE-BC hyperparameters for the Procgen and MiniHack experiments.

Hyperparameter	Procgen	MiniHack
Gradient updates	10000	10000
Learning rate	$1e{-3}$	$2.5e{-4}$
Learning-rate annealing	True	True
Rollout / sequence length	32	32
Batch size	8192	8192
Minibatches	4	4
Max grad norm	0.5	0.5

Table 13: GAGE-RL hyperparameters for the Procgen and MiniHack experiments.

Hyperparameter	Procgen	MiniHack
Learning rate	5e-4	1e-4
Learning-rate annealing	False	False
Adam epsilon	1e-5	1e-7
Parallel environments	64	64
Rollout / sequence length	256	256
Batch size	16384	16384
Minibatches	8	8
PPO epochs	3	3
Discount factor γ	0.999	0.995
GAE λ	0.95	0.95
Clip coefficient ϵ	0.2	0.1
Entropy coefficient	0.01	0.001
Value loss coefficient	0.5	1.0
KL coefficient	0.5	0.5
Max grad norm	0.5	0.5