Prioritized Level Replay

Minqi Jiang [1 2] Edward Grefenstette [1 2] Tim Rocktäschel [1 2]

Abstract

Environments with procedurally generated content serve as important benchmarks for testing systematic generalization in deep reinforcement learning. In this setting, each *level* is an algorithmically created environment instance with a unique configuration of its factors of variation. Training on a prespecified subset of levels allows for testing generalization to unseen levels. What can be learned from a level depends on the current policy, yet prior work defaults to uniform sampling of training levels independently of the policy. We introduce *Prioritized Level Replay* (PLR), a general framework for selectively sampling the next training level by prioritizing those with higher estimated learning potential when revisited in the future. We show TD-errors effectively estimate a level's future learning potential and, when used to guide the sampling procedure, induce an emergent curriculum of increasingly difficult levels. By adapting the sampling of training levels, PLR significantly improves sample efficiency and generalization on Procgen Benchmark—matching the previous state-of-the-art in test return—and readily combines with other methods. Combined with the previous leading method, PLR raises the state-of-the-art to over 76% improvement in test return relative to standard RL baselines.

1. Introduction

Deep reinforcement learning (RL) easily overfits to training experiences, making generalization a key open challenge to widespread deployment of these methods. Procedural content generation (PCG) environments have emerged as a promising class of problems with which to probe and address this core weakness (Risi & Togelius, 2020; Chevalier-Boisvert et al., 2018; Cobbe et al., 2020a; Juliani et al., 2019; Zhong et al., 2020; Küttler et al., 2020). Unlike singleton environments, like the Arcade Learning Environment games (Bellemare et al., 2013), PCG environments take on algorithmically created configurations at the start of each training episode, potentially varying the layout, asset appearances, or even game dynamics. Each environment instance generated this way is called a *level*. In mapping an identifier, such as a random seed, to each level, PCG environments allow us to measure a policy's generalization to held-out test levels. In this paper, we assume only a blackbox generation process that returns a level given an identifier. We avoid the strong assumption of control over the generation procedure itself, explored by several prior works (see Section 6). Further, we assume levels follow common latent dynamics, so that in aggregate, experiences collected in individual levels reveal general rules governing the entire set of levels.

Despite its humble origin in games, the PCG abstraction proves far-reaching: Many control problems, such as teaching a robotic arm to stack blocks in a specific formation, easily conform to PCG. Here, each level may consist of a unique combination of initial block positions, arm state, and target formation. In a vastly different domain, Hanabi (Bard et al., 2020) also conforms to PCG, where levels map to initial deck orderings. These examples illustrate the generality of the PCG abstraction: Most challenging RL problems entail generalizing across instances (or levels) differing along some underlying factors of variation and thereby can be aptly framed as PCG. This highlights the importance of effective deep RL methods for PCG environments.

Many techniques have been proposed to improve generalization in the PCG setting (see Section 6), requiring changes to the model architecture, learning algorithm, observation space, or environment structure. Notably, these approaches default to uniform sampling of training levels. We instead hypothesize that the variation across levels implies that at each point of training, each level likely holds different potential for an agent to learn about the structure shared across levels to improve generalization. Inspired by this insight and selective sampling methods from active learning, we investigate whether sampling the next training level weighed by its learning potential can improve generalization.

We introduce Prioritized Level Replay (PLR), illustrated in Figure 1, a method for sampling training levels that exploits

[1]Facebook AI Research, London, United Kingdom [2]University College London, London, United Kingdom. Correspondence to: Minqi Jiang <msj@fb.com>.

Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

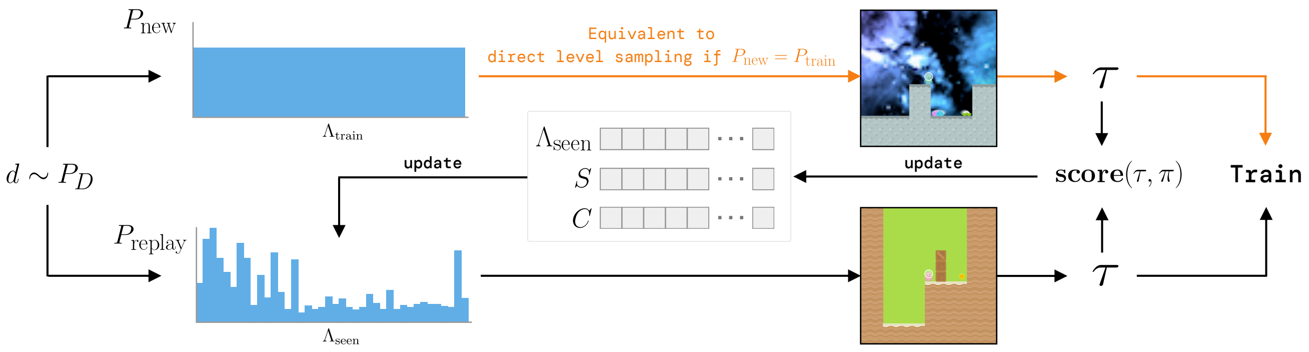

Figure 1. Overview of Prioritized Level Replay. The next level is either sampled from a distribution with support over unseen levels (top), which could be the environment's (perhaps implicit) full training-level distribution, or alternatively, sampled from the replay distribution, which prioritizes levels based on future learning potential (bottom). In either case, a trajectory τ is sampled from the next level and used to update the replay distribution. This update depends on the lists of previously seen levels Λ_{seen}, their latest estimated learning potentials S, and last sampled timestamps C.

the differences in learning potential among levels to improve both sample efficiency and generalization. PLR selectively samples the next training level based on an estimated learning potential of replaying each level anew. During training, our method updates scores estimating each level's learning potential as a function of the agent's policy and temporal-difference (TD) errors collected along the last trajectory sampled on that level. Our method then samples the next training level from a distribution derived from a normalization procedure over these level scores. PLR makes no assumptions about how the policy is updated, so it can be used in tandem with any RL algorithm and combined with many other methods such as data augmentation. Our method also does not assume any external, predefined ordering of levels by difficulty or other criteria, but instead derives level scores dynamically during training based on how the policy interacts with the environment. The only requirements are as follows—satisfied by almost any problem that can be framed as PCG, including RL environments implemented as seeded simulators: (i) Some notion of "level" exists, such that levels follow common latent dynamics; (ii) such levels can be sampled from the environment in an identifiable way; and (iii) given a level identifier, the environment can be set to that level to collect new experiences from it.

While previous works in off-policy RL devised effective methods to directly reuse *past* experiences for training (Schaul et al., 2016; Andrychowicz et al., 2017), PLR uses past experiences to inform the collection of *future* experiences by estimating how much replaying each level anew will benefit learning. Our method can thus be seen as a forward-view variation of prioritized experience replay, and an online counterpart to this off-policy method.

This paper makes the following contributions[1]: (i) We introduce a computationally-efficient algorithm for sampling levels during training based on an estimate of the future learning potential of collecting new experiences from each level; (ii) we show our method significantly improves generalization on 10 of 16 environments in Procgen Benchmark and two challenging MiniGrid environments; (iii) we demonstrate our method combines with the previous leading method to set a new state-of-the-art on Procgen Benchmark; and (iv) we show our method induces an implicit curriculum over training levels in sparse reward settings.

[1]Our code is available at `https://github.com/facebookresearch/level-replay`.

2. Background

In this paper, we refer to a *PCG environment* as any computational process that, given a level identifier (e.g. an index or a random seed), generates a *level*, defined as an environment instance exhibiting a unique configuration of its underlying factors of variation, such as layout, asset appearances, or specific environment dynamics (Risi & Togelius, 2020). For example, MiniGrid's MultiRoom environment instantiates mazes with varying numbers of rooms based on the seed (Chevalier-Boisvert et al., 2018). We refer to sampling a new trajectory generated from the agent's latest policy acting on a given level l as *replaying* that level l.

The level diversity of PCG environments makes them useful testbeds for studying the robustness and generalization ability of RL agents, measured by agent performance on unseen test levels. The standard test evaluation protocol for PCG environments consists of training the agent on a finite number of training levels Λ_{train}, and evaluating performance on unseen test levels Λ_{test}, drawn from the set of all levels. Training levels are sampled from an arbitrary distribution $P_{\text{train}}(l|\Lambda_{\text{train}})$. We call this training process *direct level sampling*. A common variation of this protocol sets Λ_{train} to the set of all levels, though in practice, the agent will still only effectively see a finite set of levels after training for a finite number of steps. In the case of a finite training set, typically $P_{\text{train}}(l|\Lambda_{\text{train}}) = \mathbf{Uniform}(l;\Lambda_{\text{train}})$. See Appendix C for the pseudocode outlining this procedure.

PCG environments naturally lend themselves to curriculum learning. Prior works have shown that directly altering levels to match their difficulty to the agent's abilities can improve generalization (Justesen et al., 2018; Dennis et al., 2020; Chevalier-Boisvert et al., 2018; Zhong et al., 2020). These findings further suggest the levels most useful for improving an agent's policy vary throughout the course of training. In this work, we consider how to automatically discover a curriculum that improves generalization for a general black-box PCG environment—crucially, without assuming any knowledge or control of how levels are generated (beyond providing the random seed or other indicial level identifier).

3. Prioritized Level Replay

In this section, we present *Prioritized Level Replay* (PLR), an algorithm for selectively sampling the next training level given the current policy, by prioritizing levels with higher estimated learning potential when replayed (that is, revisited). PLR is a drop-in replacement for the experience-collection process used in a wide range of RL algorithms. Algorithm 1 shows how it is straightforward to incorporate PLR into a generic policy-gradient training loop. For clarity, we focus on training on batches of complete trajectories (see Appendix C for pseudocode of PLR with T-step rollouts).

Our method, illustrated in Figure 1 and fully specified in Algorithm 2, induces a dynamic, nonparametric sampling distribution $P_{\text{replay}}(l|\Lambda_{\text{seen}})$ over previously visited training levels Λ_{seen} that prioritizes visited levels with higher learning potential based on properties of the agent's past trajectories. We refer to $P_{\text{replay}}(l|\Lambda_{\text{seen}})$ as the *replay distribution*. Throughout training, our method updates this replay distribution according to a heuristic score, assigning greater weight to visited levels with higher *future* learning potential. Using dynamic arrays S and C of equal length to Λ_{seen}, PLR tracks level scores $S_i \in S$ for each visited training level l_i based on the latest episode trajectory on l_i, as well as the episode count $C_i \in C$ at which each level $l_i \in \Lambda_{\text{seen}}$ was last sampled. Our method updates P_{replay} after each terminated episode by computing a mixture of two distributions, P_S, based on the level scores, and P_C, based on how long ago each level was last sampled:

$$P_{\text{replay}} = (1-\rho)\cdot P_S + \rho\cdot P_C, \qquad (1)$$

where the staleness coefficient $\rho \in [0,1]$ is a hyperparameter. We discuss how we compute level scores S_i, parameterizing the scoring distribution P_S, and the staleness distribution P_C, in Sections 3.1 and 3.2, respectively.

PLR chooses the next level at the start of every training episode by first sampling a replay-decision from a Bernoulli (or similar) distribution $P_D(d)$ to determine whether to replay a level sampled from the replay distribution $P_{\text{replay}}(l|\Lambda_{\text{seen}})$ or to experience a new, unseen level from Λ_{train}, according to some distribution $P_{\text{new}}(l|\Lambda_{\text{train}} \setminus \Lambda_{\text{seen}})$. In practice, for the case of a finite number of training levels, we implement P_{new} as a uniform distribution over the remaining unseen levels. For the case of a countably infinite number of training levels, we simulate P_{new} by sampling levels from P_{train} until encountering an unseen level. In our experiments based on a finite number of training levels, we opt to naturally anneal $P_D(d=1)$ as $|\Lambda_{\text{seen}}|/|\Lambda_{\text{train}}|$, so replay occurs more often as more training levels are visited.

Algorithm 1 Policy-gradient training loop with PLR

```
Input: Training levels $\Lambda_{\text{train}}$, policy $\pi_\theta$,
  policy update function $\mathcal{U}(\mathcal{B},\theta) \to \theta'$, and batch size $N_b$.
  Initialize level scores $S$ and level timestamps $C$
  Initialize global episode counter $c \leftarrow 0$
  Initialize the ordered set of visited levels $\Lambda_{\text{seen}} = \varnothing$
  Initialize experience buffer $\mathcal{B} = \varnothing$
  while training do
    $\mathcal{B} \leftarrow \varnothing$
    while collecting experiences do
      $\mathcal{B} \leftarrow \mathcal{B} \cup$ collect_experiences($\Lambda_{\text{train}}, \Lambda_{\text{seen}}, \pi_\theta, S, C, c$)
                                                    Using Algorithm 2
    end while
    $\theta \leftarrow \mathcal{U}(\mathcal{B},\theta)$      Update policy using collected experiences
  end while
```

Algorithm 2 Experience collection with PLR

```
Input: Training levels $\Lambda_{\text{train}}$, visited levels $\Lambda_{\text{seen}}$, policy $\pi$,
  global level scores $S$, global level timestamps $C$, and
  global episode counter $c$.
Output: A sampled trajectory $\tau$
  $c \leftarrow c + 1$
  Sample replay decision $d \sim P_D(d)$
  if $d = 0$ and $|\Lambda_{\text{train}} \setminus \Lambda_{\text{seen}}| > 0$ then
    Define new index $i \leftarrow |S| + 1$
    Sample $l_i \sim P_{\text{new}}(l|\Lambda_{\text{train}}, \Lambda_{\text{seen}})$      Sample an unseen level
    Add $l_i$ to $\Lambda_{\text{seen}}$
    Add initial value $S_i = 0$ to $S$ and $C_i = 0$ to $C$
  else
    Sample $l_i \sim (1-\rho)\cdot P_S(l|\Lambda_{\text{seen}}, S) + \rho\cdot P_C(l|\Lambda_{\text{seen}}, C, c)$
                                                    Sample a level for replay
  end if
  Sample $\tau \sim P_\pi(\tau|l_i)$
  Update score $S_i \leftarrow$ score($\tau, \pi$) and timestamp $C_i \leftarrow c$
```

The following sections describes how Prioritized Level Replay updates the replay distribution $P_{\text{replay}}(l|\Lambda_{\text{seen}})$, namely through level scoring and staleness-aware prioritization.

3.1. Scoring Levels for Learning Potential

After collecting each complete episode trajectory τ on level l_i using policy π, our method assigns l_i a score $S_i = \mathbf{score}(\tau,\pi)$ measuring the learning potential of replaying l_i in the future. We employ a function of the TD-error at timestep t, $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, as a proxy for this learning potential. The expectation of the TD-error over next states is equivalent to the advantage estimate, and therefore higher-magnitude TD-errors imply greater discrepancy between expected and actual returns, making δ_t a useful measure of the learning potential in revisiting a particular state transition. To prioritize the learning potential of future experiences resulting from replaying a level, we use a scoring function based on the *average magnitude* of the Generalized Advantage Estimate (GAE; Schulman et al., 2016) over each of T time steps in the latest trajectory τ from that level:

$$S_i = \mathbf{score}(\tau, \pi) = \frac{1}{T}\sum_{t=0}^{T}\left|\sum_{k=t}^{T}(\gamma\lambda)^{k-t}\delta_k\right|. \tag{2}$$

While the GAE at time t is most commonly expressed as the discounted sum of all 1-step TD-errors starting at t as in Equation 2, it is equivalent to an exponentially-discounted sum of all k-step TD-errors from t, with discount factor λ. By considering all k-step TD-errors, the GAE mitigates the bias introduced by the bootstrap term in 1-step TD-errors. The discount factor λ then controls the trade-off between bias and variance. Our scoring function considers the absolute value of the GAE, as we assume the learning potential grows with the magnitude of the TD-error irrespective of its sign. This also avoids opposite signed errors canceling out.

Another useful interpretation of Equation 2 comes from observing that the GAE magnitude at t is equivalent to the L1 value loss $|\hat{V}_t - V_t|$ under a policy-gradient algorithm that uses GAE for its own advantage estimates (and therefore value targets $\hat{V}_t$), as done in state-of-the-art implementations of PPO (Schulman et al., 2017) used in our experiments. Unless otherwise indicated, PLR refers to the instantiation of our algorithm with L1 value loss as the scoring function.

We further formulate the *Value Correction Hypothesis* to motivate our approach: In sparse reward settings, prioritizing the sampling of training levels with greatest average absolute value loss leads to a curriculum that improves both sample efficiency and generalization. We reason that on threshold levels (i.e. those at the limit of the agent's current abilities) the agent will see non-stationary returns (or value targets)—and therefore incur relatively high value errors—until it learns to solve them consistently. In contrast, levels beyond the agent's current abilities tend to result in stationary value targets signaling failure and therefore low value errors, until the agent learns useful, transferable behaviors from threshold levels. Prioritizing levels by value loss then naturally guides the agent along the expanding threshold of its ability—without the need for any externally provided measure of difficulty. We believe that learning behaviors systematically aligned with the inherent complexities of the environment in this way may lead to better generalization, and will seek to verify this empirically in Section 5.2.

While we provide principled motivations for our specific choice of scoring function, we emphasize that in general, the scoring function can be any approximation of learning potential based on trajectory values. Note that candidate scoring functions should asymptotically decrease with frequency of level visitation to avoid mode collapse of P_{replay} to a limited set of levels and possible overfitting. In Section 4, we compare our choice of the GAE magnitude, or equivalently, the L1 value loss, to alternative TD-error-based and uncertainty-based scoring approaches, listed in Table 1.

Given level scores, we use normalized outputs of a prioritization function h evaluated over these scores and tuned using a temperature parameter β to define the score-prioritized distribution $P_S(\Lambda_{\text{train}})$ over the training levels, under which

$$P_S(l_i|\Lambda_{\text{seen}}, S) = \frac{h(S_i)^{1/\beta}}{\sum_j h(S_j)^{1/\beta}}. \tag{3}$$

The function h defines how differences in level scores translate into differences in prioritization. The temperature parameter β allows us to tune how much $h(S)$ ultimately determines the resulting distribution. We make the design choice of using rank prioritization, for which $h(S_i) = 1/\text{rank}(S_i)$, where $\text{rank}(S_i)$ is the rank of level score S_i among all scores sorted in descending order. We also experimented with proportional prioritization ($h(S_i) = S_i$) as well as greedy prioritization (the level with the highest score receives probability 1), both of which tend to perform worse.

3.2. Staleness-Aware Prioritization

As the scores used to parameterize P_S are a function of the state of the policy at the time the associated level was last played, they come to reflect a gradually more off-policy measure the longer they remain without an update through replay. We mitigate this drift towards "off-policy-ness" by explicitly mixing the sampling distribution with a staleness-prioritized distribution P_C:

$$P_C(l_i|\Lambda_{\text{seen}}, C, c) = \frac{c - C_i}{\sum_{C_j \in C} c - C_j} \tag{4}$$

which assigns probability mass to each level l_i in proportion to the level's *staleness* $c - C_i$. Here, c is the count of total episodes sampled so far in training and C_i (referred to as the level's timestamp) is the episode count at which l_i was last sampled. By pushing support to levels with staler scores, P_C ensures no score drifts too far off-policy.

Plugging Equations 3 and 4 into Equation 1 gives us a replay distribution that is calculated as

$$P_{\text{replay}}(l_i) = (1-\rho)\cdot P_S(l_i|\Lambda_{\text{seen}}, S) + \rho\cdot P_C(l_i|\Lambda_{\text{seen}}, C, c).$$

Thus, a level has a greater chance of being sampled when its score is high or it has not been sampled for a long time.

4. Experimental Setting

We evaluate PLR on several PCG environments with various combinations of scoring functions and prioritization schemes, and compare to the most common direct level sampling baseline of $P_{\text{train}}(l|\Lambda_{\text{train}}) = \mathbf{Uniform}(l; \Lambda_{\text{train}})$. We train and test on all 16 environments in the Procgen Benchmark on easy and hard difficulties, but focus discussion on the easy results, which allow direct comparison to several prior studies. We compare to UCB-DrAC (Raileanu et al., 2021), the state-of-the-art image augmentation method on this benchmark, and mixreg (Wang et al., 2020a), a recently introduced data augmentation method. We also compare to TSCL Window (Matiisen et al., 2020), which resembles PLR with an alternative scoring function using the slope of recent returns and no staleness sampling. For fair comparison, we also evaluate a custom TSCL Window variant that mixes in the staleness distribution P_C weighted by $\rho > 0$. Further, to demonstrate the ease of combining PLR with other methods, we evaluate UCB-DrAC using PLR for sampling training levels. Finally, we test the Value Correction Hypothesis on two challenging MiniGrid environments.

We measure episodic *test returns* per game throughout training, as well as the performance of the final policy over 100 unseen test levels of each game relative to PPO with uniform sampling. We also evaluate the mean normalized episodic test return and mean generalization gap, averaged over all games (10 runs each). We normalize returns according to Cobbe et al. (2019) and compute the generalization gap as train returns minus test returns. Thus, a larger gap indicates more overfitting, making it an apt measure of generalization. We assess statistical significance at $p = 0.05$, using the Welch t-test.

In line with the standard baseline for these environments, all experiments use PPO with GAE for training. For Procgen, we use the same ResBlock architecture as Cobbe et al. (2020a) and train for 25M total steps on 200 levels on the easy setting as in the original baselines. For MiniGrid, we use a 3-layer CNN architecture based on Igl et al. (2019), and provide approximately 1000 levels of each difficulty per environment during training. Detailed descriptions of the environments, architectures, and hyperparameters used in our experiments (and how they were set or obtained) can be found in Appendix A. See Table 1 for the full set of scoring functions investigated in our experiments.

Additionally, we extend PLR to support training on an unbounded number of levels by tracking a rolling, finite buffer of the top levels so far encountered by learning potential. Appendix B.3 reports the results of this extended PLR algorithm when training on the full level distribution of the MiniGrid environments featured in our main experiments.

Table 1. Scoring functions investigated in this work.

Scoring function	$\mathbf{score}(\tau, \pi)$
Policy entropy	$\frac{1}{T}\sum_{t=0}^{T}\sum_{a}\pi(a, s_t)\log\pi(a, s_t)$
Policy min-margin	$\frac{1}{T}\sum_{t=0}^{T}(\max_a \pi(a, s_t) - \max_{a \neq \max_a \pi(a,s_t)} \pi(a, s_t))$
Policy least-confidence	$\frac{1}{T}\sum_{t=0}^{T}(1 - \max_a \pi(a, s_t))$
1-step TD error	$\frac{1}{T}\sum_{t=0}^{T}\lvert\delta_t\rvert$
GAE	$\frac{1}{T}\sum_{t=0}^{T}\sum_{k=t}^{T}(\gamma\lambda)^{k-t}\delta_k$
GAE magnitude (L1 value loss)	$\frac{1}{T}\sum_{t=0}^{T}\left\lvert\sum_{k=t}^{T}(\gamma\lambda)^{k-t}\delta_k\right\rvert$

5. Results and Discussion

Our main findings are that (i) PLR significantly improves both sample efficiency and generalization, attaining the highest normalized mean test and train returns and mean reduction in generalization gap on Procgen out of all individual methods evaluated, while matching UCB-DrAC in test improvement relative to PPO; (ii) alternative scoring functions lead to inconsistent improvements across environments; (iii) PLR combined with UCB-DrAC sets a new state-of-the-art on Procgen; and (iv) PLR induces an implicit curriculum over training levels, which substantially aids training in two challenging MiniGrid environments.

5.1. Procgen Benchmark

Our results, summarized in Figure 2, show PLR with rank prioritization ($\beta = 0.1$, $\rho = 0.1$) leads to the largest statistically significant gains in mean normalized test and train returns and reduction in generalization gap compared to uniform sampling, outperforming all other methods besides UCB-DrAC + PLR. PLR combined with UCB-DrAC sees the most drastic improvements in these metrics. As reported in Table 2, UCB-DrAC + PLR yields a 76% improvement in mean test return relative to PPO with uniform sampling, and a 28% improvement relative to the previous state-of-the-art set by UCB-DrAC. While PLR with rank prioritization leads to statistically significant gains in test return on 10 of 16 environments and proportional prioritization, on 11 of 16 games, we prefer rank prioritization: While we find the two comparable in mean normalized returns, Figure 3 shows rank prioritization results in higher mean *unnormalized* test returns and a significantly lower mean generalization gap, averaged over all environments.

Further, Figure 3 shows that gains only occur when P_{replay} considers *both* level scores and staleness ($0 < \rho < 1$), highlighting the importance of staleness-based sampling in keeping scores from drifting off-policy. Lastly, we also

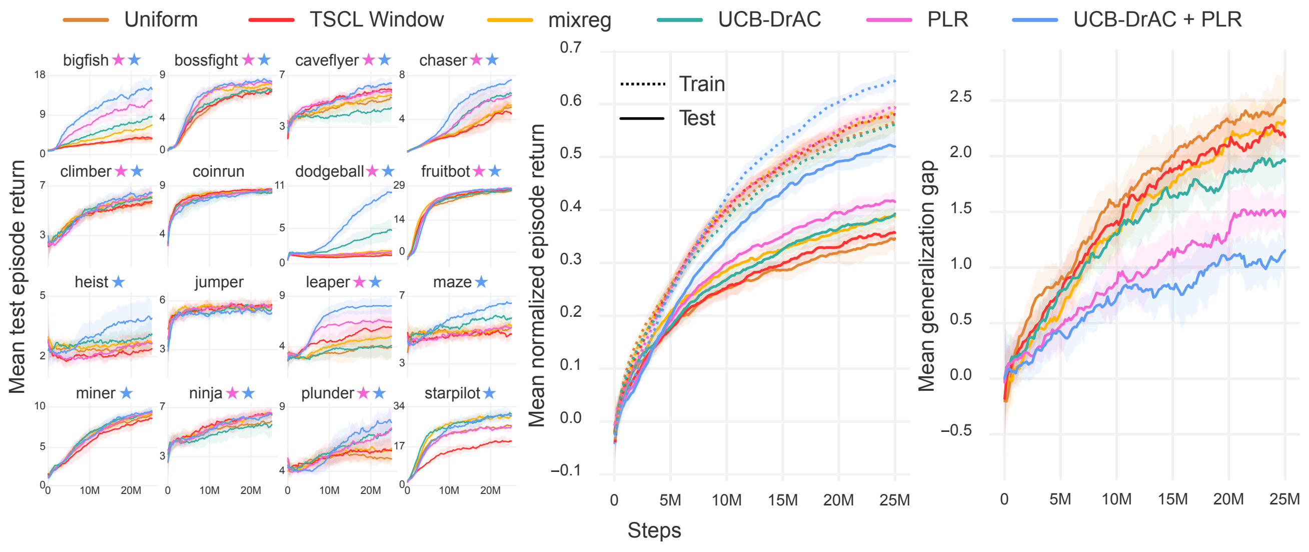

Figure 2. Left: Mean episodic test returns (10 runs) of each method. Each colored ★ indicates statistically significant ($p < 0.05$) gains in final test performance or sample complexity along the curve, relative to uniform sampling, for the PLR-based method of the same color. Center: Mean normalized train and test returns averaged across all games. Right: Mean generalization gaps averaged across all games.

benchmarked PLR on the hard setting against the same set of methods, where it again leads with 35% greater test returns relative to uniform sampling and 83% greater test returns when combined with UCB-DrAC. Figures 10–18 and Table 4 in Appendix B report additional details on these results.

The alternative scoring metrics based on TD-error and classifier uncertainty perform inconsistently across games. While certain games, such as BigFish, see improved sample-efficiency and generalization under various scoring functions, others, such as Ninja, see no improvement or worse, degraded performance. See Figure 3 for an example of this inconsistent effect across games. We find the best-performing variant of TSCL Window does not incorporate staleness information ($\rho = 0$) and similarly leads to inconsistent outcomes across games at test time, notably significantly worsening performance on StarPilot, as seen in Figure 2, and increasing the generalization gap on some environments as revealed in Figure 15 of Appendix B.

5.2. MiniGrid

We provide empirical support for the Value Correction Hypothesis (defined in Section 3) on two challenging MiniGrid environments, whose levels fall into discrete difficulties (e.g. by number of rooms to be traversed). In both, PLR with rank prioritization significantly improves sample efficiency and generalization over uniform sampling, demonstrating our method also works well in discrete state spaces. We find a staleness coefficient of $\rho = 0.3$ leads to the best test performance on MiniGrid. The top row of Figure 4 summarizes these results.

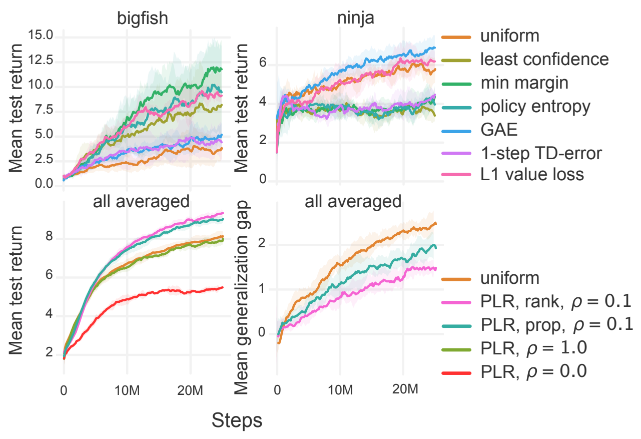

Figure 3. Top: Two example Procgen environments, between which all scoring functions except L1 value loss show inconsistent improvements to test performance (rank prioritization, $\beta = 0.1$, $\rho = 0.3$). This inconsistency holds across settings in our grid search. Bottom: Mean unnormalized episodic test returns (left) and mean generalization gap (right) for various PLR settings.

To test our hypothesis, we bin each level into its corresponding difficulty, expressed as ascending, discrete values (note that PLR does not have access to this privileged information). In the bottom row of Figure 4, we see how the expected difficulty of levels sampled using PLR changes during training for each environment. We observe that as P_{replay} is updated, levels become sampled according to an implicit curriculum over the training levels that prioritizes progressively harder levels. Of particular note, PLR seems to struggle to discover a useful curriculum for around the first 4,000 updates on ObstructedMazeGamut-Medium, at which point it discovers

Table 2. Test returns of policies trained using each method with its best hyperparameters. Following Raileanu et al. (2021), the reported mean and standard deviations per environment are computed by evaluating the final policy's average return on 100 test episodes, aggregated across multiple training runs (10 runs for Procgen Benchmark and 3 for MiniGrid, each initialized with a different training seed). Normalized test returns per run are computed by dividing the average test return per run for each environment by the corresponding average test return of the uniform-sampling baseline over all runs. We then report the means and standard deviations of normalized test returns aggregated across runs. We report the normalized return statistics for Procgen and MiniGrid environments separately. Bolded methods are not significantly different from the method with highest mean, unless all are, in which case none are bolded.

Environment	Uniform	TSCL	mixreg	UCB-DrAC	PLR	UCB-DrAC + PLR
BigFish	3.7 ± 1.2	4.3 ± 1.3	6.9 ± 1.6	8.7 ± 1.1	10.9 ± 2.8	$\mathbf{14.3 \pm 2.1}$
BossFight	7.7 ± 0.4	7.4 ± 0.8	8.1 ± 0.7	7.7 ± 0.7	$\mathbf{8.9 \pm 0.4}$	$\mathbf{8.8 \pm 0.8}$
CaveFlyer	5.4 ± 0.8	$\mathbf{6.3 \pm 0.6}$	6.0 ± 0.6	4.6 ± 0.9	$\mathbf{6.3 \pm 0.5}$	$\mathbf{6.8 \pm 0.7}$
Chaser	5.2 ± 0.7	4.9 ± 1.0	5.7 ± 1.1	6.8 ± 0.9	6.9 ± 1.2	$\mathbf{8.0 \pm 0.6}$
Climber	5.9 ± 0.6	6.0 ± 0.8	$\mathbf{6.6 \pm 0.7}$	$\mathbf{6.4 \pm 0.9}$	$\mathbf{6.3 \pm 0.8}$	$\mathbf{6.8 \pm 0.7}$
CoinRun	8.6 ± 0.4	$\mathbf{9.2 \pm 0.2}$	8.6 ± 0.3	8.6 ± 0.4	8.8 ± 0.5	$\mathbf{9.0 \pm 0.4}$
Dodgeball	1.7 ± 0.2	1.2 ± 0.4	1.8 ± 0.4	5.1 ± 1.6	1.8 ± 0.5	$\mathbf{10.3 \pm 1.4}$
FruitBot	27.3 ± 0.9	27.1 ± 1.6	27.7 ± 0.8	27.0 ± 1.3	28.0 ± 1.3	27.6 ± 1.5
Heist	2.8 ± 0.9	2.5 ± 0.6	2.7 ± 0.4	3.2 ± 0.7	2.9 ± 0.5	$\mathbf{4.9 \pm 1.3}$
Jumper	5.7 ± 0.4	6.1 ± 0.6	6.1 ± 0.3	5.6 ± 0.5	5.8 ± 0.5	5.9 ± 0.3
Leaper	4.2 ± 1.3	6.4 ± 1.2	5.2 ± 1.1	4.4 ± 1.4	6.8 ± 1.2	$\mathbf{8.7 \pm 1.0}$
Maze	5.5 ± 0.4	5.0 ± 0.3	5.4 ± 0.5	6.2 ± 0.5	5.5 ± 0.8	$\mathbf{7.2 \pm 0.8}$
Miner	8.7 ± 0.7	8.9 ± 0.6	9.5 ± 0.4	$\mathbf{10.1 \pm 0.6}$	$\mathbf{9.6 \pm 0.6}$	$\mathbf{10.0 \pm 0.5}$
Ninja	6.0 ± 0.4	$\mathbf{6.8 \pm 0.5}$	$\mathbf{6.9 \pm 0.5}$	5.8 ± 0.8	$\mathbf{7.2 \pm 0.4}$	$\mathbf{7.0 \pm 0.5}$
Plunder	5.1 ± 0.6	5.9 ± 1.1	5.7 ± 0.5	$\mathbf{7.8 \pm 0.9}$	$\mathbf{8.7 \pm 2.2}$	$\mathbf{7.7 \pm 0.9}$
StarPilot	26.8 ± 1.5	19.8 ± 3.4	$\mathbf{32.7 \pm 1.5}$	$\mathbf{31.7 \pm 2.4}$	27.9 ± 4.4	29.6 ± 2.2
Normalized test returns (%)	100.0 ± 4.5	103.0 ± 3.6	113.8 ± 2.8	129.8 ± 8.2	128.3 ± 5.8	$\mathbf{176.4 \pm 6.1}$
MultiRoom-N4-Random	0.80 ± 0.04	–	–	–	$\mathbf{0.81 \pm 0.01}$	–
ObstructedMazeGamut-Easy	0.53 ± 0.04	–	–	–	$\mathbf{0.85 \pm 0.04}$	–
ObstructedMazeGamut-Med	0.65 ± 0.01	–	–	–	$\mathbf{0.73 \pm 0.07}$	–
Normalized test returns (%)	100.0 ± 2.5	–	–	–	$\mathbf{124.3 \pm 4.7}$	–

a curriculum that gradually assigns more weight to harder levels. This curriculum enables PLR with access to only 6,000 training levels to attain even higher mean test returns than the uniform-sampling baseline with access to the full set of training levels, of which there are roughly 4 billion (so our training levels constitute 0.00015% of the total number).

We further tested an extended version of PLR that trains on the full level distribution on these two environments by tracking a buffer of levels with the highest estimated learning potential. We find it outperforms uniform sampling with access to the full level distribution. These additional results are presented in Appendix B.3.

6. Related Work

Several methods for improving generalization in deep RL adapt techniques from supervised learning, including stochastic regularization (Igl et al., 2019; Cobbe et al., 2020a), data augmentation (Kostrikov et al., 2020; Raileanu et al., 2021; Wang et al., 2020a), and feature distillation (Igl et al., 2020; Cobbe et al., 2020b). In contrast, PLR modifies only how the next training level is sampled, thereby easily combining with any model or RL algorithm.

The selective-sampling performed by PLR makes it a form of active learning (Cohn et al., 1994; Settles, 2009). Our work also echoes ideas from Graves et al. (2017), who train a multi-armed bandit to choose the next task in multi-task supervised learning, so to maximize gradient-based progress signals. Sharma et al. (2018) extend these ideas to multi-task RL, but add the additional requirement of knowing a maximum target return for each task a priori. Zhang et al. (2020b) use an ensemble of value functions for selective goal sampling in the off-policy continuous control setting, requiring prior knowledge of the environment structure to generate candidate goals. Unlike PLR, these methods assume the ability to sample tasks or levels based on their structural properties, an assumption that does not typically hold for PCG simulators. Instead, our method automatically uncovers similarly difficult levels, giving rise to a curriculum without prior knowledge of the environment.

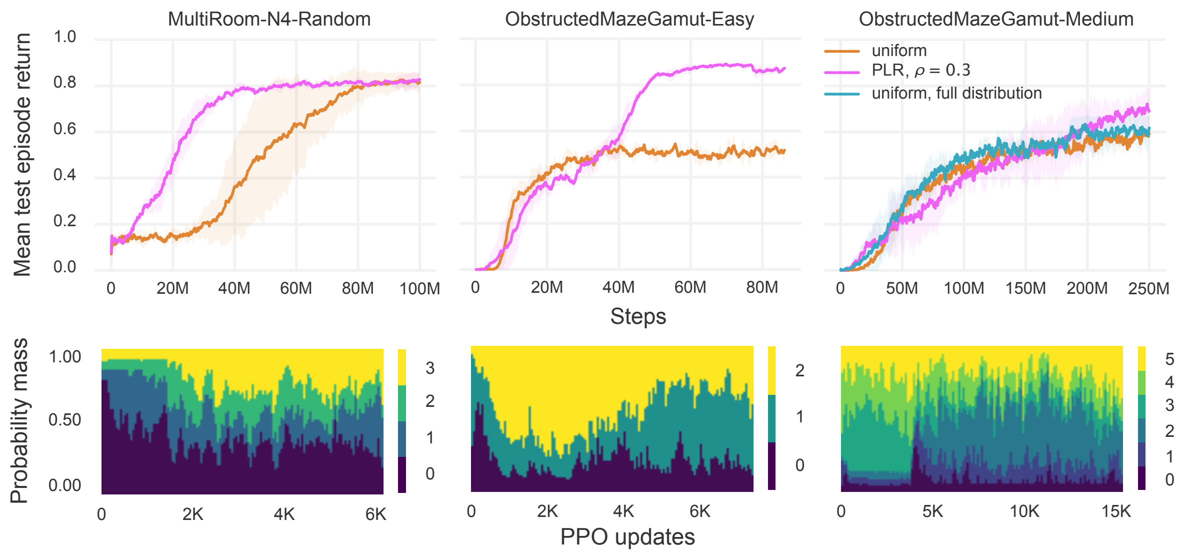

Figure 4. Top: Mean episodic test returns of PLR and the uniform-sampling baseline on MultiRoom-N4-Random (4 runs), ObstructedMazeGamut-Easy (3 runs), and ObstructedMazeGamut-Medium (3 runs). Bottom: The probability mass assigned to levels of varying difficulty over the course of training in a single, randomly selected run for the respective environment.

A recent theme in the PCG setting explores adaptively generating levels to facilitate learning (Sukhbaatar et al., 2017; Wang et al., 2019; 2020b; Khalifa et al., 2020; Akkaya et al., 2019; Campero et al., 2020; Dennis et al., 2020). Unlike these approaches, our method does not assume control over level generation, requiring only the ability to replay previously visited levels. These methods also require parameterizing level generation with additional learning modules. In contrast, our approach does not require such extensions of the environment, for example including teacher-specific action spaces in the case of Campero et al. (2020). Most similar to our method, Matiisen et al. (2020) proposes a teacher-student curriculum learning (TSCL) algorithm that samples training levels by considering the change in episodic returns per level, though they neither design nor test the method for generalization. As shown in Section 5.1, it provides inconsistent benefits at test time. Further, unlike TSCL, PLR does not assume access to all levels at the start of training, and as we show in Appendix B.3, PLR can be extended to improve sample efficiency and generalization by training on an unbounded number of training levels.

Like our method, Schaul et al. (2016) and Kapturowski et al. (2019) use TD-errors to estimate learning potential. While these methods make use of TD-errors to prioritize learning from *past* experiences, our method uses such estimates to prioritize revisiting levels for generating entirely new *future* experiences for learning.

Generalization requires sufficient exploration of environment states and dynamics. Thus, recent exploration strategies (e.g. Raileanu & Rocktäschel, 2020; Campero et al., 2020; Zhang et al., 2020a; Zha et al., 2021) shown to benefit simple PCG settings are complementary to the aims of this work. However, as these studies focus on PCG environments with low-dimensional state spaces, whether such methods can be successfully applied to more complex PCG environments like Procgen Benchmark remains to be seen. If so, they may potentially combine with PLR to yield additive improvements. We believe the interplay between such exploration methods and PLR to be a promising direction for future research.

7. Conclusion and Future Work

We introduced Prioritized Level Replay (PLR), an algorithm for selectively sampling the next training level in PCG environments based on the estimated learning potential of revisiting each level for the current policy. We showed that our method remarkably improves both the sample efficiency and generalization of deep RL agents in PCG environments, including the majority of environments in Procgen Benchmark and two challenging MiniGrid environments. We further combined PLR with the prior leading method to set a new state-of-the-art on Procgen Benchmark. Further, on MiniGrid environments, we showed PLR induces an emergent curriculum of increasingly more difficult levels.

The flexibility of the PCG abstraction makes PLR applicable to many problems of practical importance, for example, robotic object manipulation tasks, where domain randomized environment instances map to the notion of levels. We believe PLR may even be applicable to singleton environments, given a procedure for generating variations of the underlying MDP as a function of a level identifier, for example, by varying the starting positions of entities. Another natural extension of PLR is to adapt the method to operate in the goal-conditioned setting, by incorporating goals into the level parameterization.

Despite the wide applicability of PCG and consequently PLR, not all problem domains can be effectively represented in seed-based simulation. Many real world problems require transfer into domains too complex to be adequately captured by simulation, such as car driving, where realizing a completely faithful simulation would entail solving the very same control problem of interest, creating a chicken-and-egg dilemma. Further, environment resets are not universally available, such as in the continual learning setting, where the agent interacts with the environment without explicit episode boundaries—arguably, a more realistic interaction model for a learning agent deployed in the wild.

Still, pre-training in simulation with resets can nevertheless benefit such settings, where the target domain is rife with open-ended complexity and where resets are unavailable, especially as training through real-world interactions can be slow, expensive, and precarious. In fact, in practice, we almost exclusively train deep RL policies in simulation for these reasons. As PLR provides a simple method to more fully exploit the simulator for improved test-time performance, we believe PLR can also be adapted to improve learning in these settings.

We further note that while we empirically demonstrated that the L1 value loss acts as a highly effective scoring function, there likely exist even more potent choices. Directly learning such functions may reveal even better alternatives. Lastly, combining PLR with various exploration strategies may further improve test performance in hard exploration environments. We look forward to investigating each of these promising directions in future work, prioritized accordingly, by learning potential.

Acknowledgements

We thank Roberta Raileanu, Heinrich Küttler, and Jakob Foerster for useful discussions and feedback on this work, and our anonymous reviewers, for their recommendations on improving this paper.

References

Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., Schneider, J., Tezak, N., Tworek, J., Welinder, P., Weng, L., Yuan, Q., Zaremba, W., and Zhang, L. Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113, 2019. URL http://arxiv.org/abs/1910.07113.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, O. P., and Zaremba, W. Hindsight experience replay. In *Advances in neural information processing systems*, pp. 5048–5058, 2017.

Bard, N., Foerster, J. N., Chandar, S., Burch, N., Lanctot, M., Song, H. F., Parisotto, E., Dumoulin, V., Moitra, S., Hughes, E., Dunning, I., Mourad, S., Larochelle, H., Bellemare, M. G., and Bowling, M. The hanabi challenge: A new frontier for AI research. *Artif. Intell.*, 280:103216, 2020. doi: 10.1016/j.artint.2019.103216. URL https://doi.org/10.1016/j.artint.2019.103216.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013. ISSN 1076-9757. doi: 10.1613/jair.3912. URL http://dx.doi.org/10.1613/jair.3912.

Campero, A., Raileanu, R., Küttler, H., Tenenbaum, J. B., Rocktäschel, T., and Grefenstette, E. Learning with AMIGo: Adversarially Motivated Intrinsic Goals. *CoRR*, abs/2006.12122, 2020. URL https://arxiv.org/abs/2006.12122.

Chevalier-Boisvert, M., Bahdanau, D., Lahlou, S., Willems, L., Saharia, C., Nguyen, T. H., and Bengio, Y. BabyAI: First steps towards grounded language learning with a human in the loop. *CoRR*, abs/1810.08272, 2018. URL http://arxiv.org/abs/1810.08272.

Chevalier-Boisvert, M., Willems, L., and Pal, S. Minimalistic gridworld environment for OpenAI Gym. https://github.com/maximecb/gym-minigrid, 2018.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. Quantifying generalization in reinforcement learning. In *International Conference on Machine Learning*, pp. 1282–1289. PMLR, 2019.

Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. Leveraging Procedural Generation to Benchmark Reinforcement Learning. In *International Conference on Machine Learning*, pp. 2048–2056. PMLR, November 2020a. URL http://proceedings.mlr.press/v119/cobbe20a.html. ISSN: 2640-3498.

Cobbe, K., Hilton, J., Klimov, O., and Schulman, J. Phasic Policy Gradient. *CoRR*, abs/2009.04416, 2020b. URL https://arxiv.org/abs/2009.04416.

Cohn, D., Atlas, L., and Ladner, R. Improving generalization with active learning. *Machine learning*, 15(2): 201–221, 1994.

Dennis, M., Jaques, N., Vinitsky, E., Bayen, A. M., Russell, S., Critch, A., and Levine, S. Emergent complexity and zero-shot transfer via unsupervised environment design. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/985e9a46e10005356bbaf194249f6856-Abstract.html.

Graves, A., Bellemare, M. G., Menick, J., Munos, R., and Kavukcuoglu, K. Automated curriculum learning for neural networks. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1311–1320. PMLR, 2017. URL http://proceedings.mlr.press/v70/graves17a.html.

Igl, M., Ciosek, K., Li, Y., Tschiatschek, S., Zhang, C., Devlin, S., and Hofmann, K. Generalization in reinforcement learning with selective noise injection and information bottleneck. In *Advances in Neural Information Processing Systems*, pp. 13978–13990, 2019.

Igl, M., Farquhar, G., Luketina, J., Boehmer, W., and Whiteson, S. The impact of non-stationarity on generalisation in deep reinforcement learning. *CoRR*, abs/2006.05826, 2020. URL https://arxiv.org/abs/2006.05826.

Juliani, A., Khalifa, A., Berges, V.-P., Harper, J., Teng, E., Henry, H., Crespi, A., Togelius, J., and Lange, D. Obstacle tower: A generalization challenge in vision, control, and planning. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Aug 2019. doi: 10.24963/ijcai.2019/373. URL http://dx.doi.org/10.24963/ijcai.2019/373.

Justesen, N., Torrado, R. R., Bontrager, P., Khalifa, A., Togelius, J., and Risi, S. Procedural level generation improves generality of deep reinforcement learning. *CoRR*, abs/1806.10729, 2018. URL http://arxiv.org/abs/1806.10729.

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., and Dabney, W. Recurrent experience replay in distributed reinforcement learning. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=r1lyTjAqYX.

Khalifa, A., Bontrager, P., Earle, S., and Togelius, J. PCGRL: procedural content generation via reinforcement learning. *CoRR*, abs/2001.09212, 2020. URL https://arxiv.org/abs/2001.09212.

Kostrikov, I., Yarats, D., and Fergus, R. Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels. *CoRR*, abs/2004.13649, 2020. URL https://arxiv.org/abs/2004.13649.

Küttler, H., Nardelli, N., Miller, A. H., Raileanu, R., Selvatici, M., Grefenstette, E., and Rocktäschel, T. The nethack learning environment. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/569ff987c643b4bedf504efda8f786c2-Abstract.html.

Matiisen, T., Oliver, A., Cohen, T., and Schulman, J. Teacher–student curriculum learning. *IEEE Transactions on Neural Networks and Learning Systems*, 31 (9):3732–3740, Sep 2020. ISSN 2162-2388. doi: 10.1109/tnnls.2019.2934906. URL http://dx.doi.org/10.1109/TNNLS.2019.2934906.

Raileanu, R. and Rocktäschel, T. RIDE: rewarding impact-driven exploration for procedurally-generated environments. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rkg-TJBFPB.

Raileanu, R., Goldstein, M., Yarats, D., Kostrikov, I., and Fergus, R. Automatic data augmentation for generalization in reinforcement learning, 2021. URL https://openreview.net/forum?id=9l9WD4ahJgs.

Risi, S. and Togelius, J. Increasing generality in machine learning through procedural content generation. *Nature Machine Intelligence*, 2(8):428–436, Aug 2020. ISSN 2522-5839. doi: 10.1038/s42256-020-0208-z. URL http://dx.doi.org/10.1038/s42256-020-0208-z.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. In Bengio, Y. and LeCun, Y. (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05952.

Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. In Bengio, Y. and LeCun, Y. (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1506.02438.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Settles, B. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

Sharma, S., Jha, A. K., Hegde, P., and Ravindran, B. Learning to multi-task by active sampling. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=B1nZ1weCZ.

Sukhbaatar, S., Kostrikov, I., Szlam, A., and Fergus, R. Intrinsic motivation and automatic curricula via asymmetric self-play. *CoRR*, abs/1703.05407, 2017. URL http://arxiv.org/abs/1703.05407.

Wang, K., Kang, B., Shao, J., and Feng, J. Improving generalization in reinforcement learning with mixture regularization. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020a. URL https://proceedings.neurips.cc/paper/2020/hash/5a751d6a0b6ef05cfe51b86e5d1458e6-Abstract.html.

Wang, R., Lehman, J., Clune, J., and Stanley, K. O. POET: open-ended coevolution of environments and their optimized solutions. In Auger, A. and Stützle, T. (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pp. 142–151. ACM, 2019. doi: 10.1145/3321707.3321799. URL https://doi.org/10.1145/3321707.3321799.

Wang, R., Lehman, J., Rawal, A., Zhi, J., Li, Y., Clune, J., and Stanley, K. Enhanced POET: Open-ended Reinforcement Learning through Unbounded Invention of Learning Challenges and their Solutions. In *International Conference on Machine Learning*, pp. 9940–9951. PMLR, November 2020b. URL http://proceedings.mlr.press/v119/wang20l.html. ISSN: 2640-3498.

Zha, D., Ma, W., Yuan, L., Hu, X., and Liu, J. Rank the episodes: A simple approach for exploration in procedurally-generated environments. *CoRR*, abs/2101.08152, 2021. URL https://arxiv.org/abs/2101.08152.

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

Zhang, Y., Abbeel, P., and Pinto, L. Automatic curriculum learning through value disagreement. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020b. URL https://proceedings.neurips.cc/paper/2020/hash/566f0ea4f6c2e947f36795c8f58ba901-Abstract.html.

Zhong, V., Rocktäschel, T., and Grefenstette, E. RTFM: generalising to new environment dynamics via reading. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=SJgob6NKvH.

A. Experiment Details and Hyperparameters

A.1. Procgen Benchmark

Procgen Benchmark consists of 16 PCG environments of varying styles, exhibiting a diversity of gameplay similar to that of the ALE benchmark. Game levels are determined by a random seed and can vary in navigational layout, visual appearance, and starting positions of entities. All Procgen environments share the same discrete 15-dimensional action space and produce $64 \times 64 \times 3$ RGB observations. (Cobbe et al., 2020a) provides a comprehensive description of each of the 16 environments. State-of-the-art RL algorithms, like PPO, lead to significant generalization gaps between test and train performance in all games, making Procgen a useful benchmark for assessing generalization performance.

We follow the standard protocol for testing generalization performance on Procgen: We train an agent for each game on a finite number of levels, N_{train}, and sample test levels from the full distribution of levels. Normalized test returns are computed as $(R - R_{\text{min}})/(R_{\text{max}} - R_{\text{min}})$, where R is the unnormalized return and each game's minimum return, R_{min}, and maximum return, R_{max}, are provided in Cobbe et al. (2020a), which uses this same normalization.

To make the most efficient use of our computational resources, we perform hyperparameter sweeps on the easy setting. This also makes our results directly comparable to most prior works benchmarked on Procgen, which have likewise focused on the easy setting. In Procgen easy, our experiments use the recommended settings of $N_{\text{train}} = 200$ and 25M steps of training, as well as the same ResNet policy architecture and PPO hyperparameters shared across all games as in Cobbe et al. (2020a) and Raileanu et al. (2021). We find 25M steps to be sufficient for uncovering differences in generalization performance among our methods and standard baselines. Moreover, under this setup, we find Procgen training runs require much less wall-clock time than training runs on the two MiniGrid environments of interest over an equivalent number of steps needed to uncover differences in generalization performance. Therefore we survey the empirical differences across various settings of PLR on Procgen easy rather than MiniGrid.

To find the best hyperparameters for PLR, we evaluate each combination of the scoring function choices in Table 1 with both rank and proportional prioritization, performing a coarse grid search for each pair over different settings of the temperature parameter β in $\{0.1, 0.5, 1.0, 1.4, 2.0\}$ and the staleness coefficient ρ in $\{0.1, 0.3, 1.0\}$. For each setting, we run 4 trials across all 16 of games of the Procgen Benchmark, evaluating based on mean unnormalized test return across games. In our TD-error-based scoring functions, we set γ and λ equal to the same respective values used by the GAE in PPO during training. We found PLR offered the most pronounced gains at $\beta = 0.1$ and $\rho = 0.1$ on Procgen, but these benefits also held for higher values ($\beta = 0.5$ and $\rho = 0.3$), though to a lesser degree.

For UCB-DrAC, we make use of the best-reported hyperparameters on the easy setting of Procgen in Raileanu et al. (2021), listed in Table 3.

We found the default setting of mixreg's $\alpha = 0.2$, as used by Wang et al. (2020a) in the hard setting, performs poorly on the easy setting. Instead, we conducted a grid search over α in $\{0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.8, 0.2, 0.5, 0.8, 1\}$.

Since the TSCL Window algorithm was not previously evaluated on Procgen Benchmark, we perform a grid search over different settings for both Boltzmann and ϵ-greedy variants of the algorithm to determine the best hyperparameter settings for Procgen easy. We searched over window size K in $\{10, 100, 1000, 10000\}$, bandit learning rate α in $\{0.01, 0.1, 0.5, 1.0\}$, random exploration probability ϵ in $\{0.0, 0.01, 0.1, 0.5\}$ for the ϵ-greedy variant, and temperature τ in $\{0.1, 0.5, 1.0\}$ for the Boltzmann variant. Additionally, for a fairer comparison to PLR we further evaluated a variant of TSCL Window that, like PLR, incorporates the staleness distribution, by additionally searching over values of the staleness coefficient ρ in $\{0.0, 0.1, 0.3, 0.5\}$, though we ultimately found that TSCL Window performed best without staleness sampling ($\rho = 0$).

See Table 3 for a comprehensive overview of the hyperparameters used for PPO, UCB-DrAC, mixreg, and TSCL Window, shared across all Procgen environments to generate our reported results on Procgen easy.

The evaluation protocol on the hard setting entails training on 500 levels over 200M steps (Cobbe et al., 2020a), making it more compute-intensive than the easy setting. To save on computational resources, we make use of the same hyperparameters found in the easy setting for each method on Procgen hard, with one exception: As our PPO implementation does not use multi-GPU training, we were unable to quadruple our GPU actors as done in Cobbe et al. (2020a) and Wang et al. (2020a). Instead, we resorted to doubling the number of environments in our single actor to 128, resulting in mini-batch sizes half as large as used in these two prior works. As such, our baseline results on hard are not directly comparable to theirs. We found setting mixreg's $\alpha = 0.2$ as done in Wang et al. (2020a) led to poor performance under this reduced batch size. We conducted an additional grid search, finding $\alpha = 0.01$ to perform best, as on Procgen easy.

A.2. MiniGrid

The MiniGrid suite (Chevalier-Boisvert et al., 2018) features a series of highly structured environments of increasing difficulty. Each environment features a task in a grid world

Table 3. Hyperparameters used for training on Procgen Benchmark and MiniGrid environments.

Parameter	Procgen	MiniGrid
PPO		
γ	0.999	0.999
λ_{GAE}	0.95	0.95
PPO rollout length	256	256
PPO epochs	3	4
PPO minibatches per epoch	8	8
PPO clip range	0.2	0.2
PPO number of workers	64	64
Adam learning rate	5e-4	7e-4
Adam ϵ	1e-5	1e-5
return normalization	yes	yes
entropy bonus coefficient	0.01	0.01
value loss coefficient	0.5	0.5
PLR		
Prioritization	rank	rank
Temperature, β, 0.1	0.1	0.1
Staleness coefficient, ρ	0.1	0.3
UCB-DrAC		
Window size, K	10	-
Regularization coefficient, α_r	0.1	-
UCB exploration coefficient, c	0.1	-
mixreg		
Beta shape, α	0.01	-
TSCL Window		
Bandit exploration strategy	ϵ-greedy	-
Window size, K	10	-
Bandit learning rate, α	1.0	-
Exploration probability, ϵ	0.5	-

setting, and as in Procgen, environment levels are determined by a seed. Harder levels require the agent to perform longer action sequences over a combinatorially-rich set of game entities, on increasingly larger grids. The clear ordering of difficulty over subsets of MiniGrid environments allows us to track the relative difficulty of levels sampled by PLR over the course of training.

MiniGrid environments share a discrete 7-dimensional action space and produce a 3-channel integer state encoding of the 7×7 grid immediately including and in front of the agent. However, following the training setup in Igl et al. (2019), we modify the environment to produce an $N \times M \times 3$ encoding of the full grid, where N and M vary according to the maximum grid dimensions of each environment. Full observability makes generalization harder, requiring the agent to generalize across different level layouts in their entirety.

We evaluate PLR with rank prioritization on two MiniGrid environments whose levels are uniformly distributed across several difficulty settings. Training on levels of varying difficulties helps agents make use of the easier levels as stepping stones to learn useful behaviors that help the agent make progress on harder levels. However, under the uniform-sampling baseline, learning may be inefficient, as the training process does not selectively train the agent on levels of increasing difficulty, leading to wasted training steps when a difficult level is sampled early in training. On the contrary, if PLR scores levels according to the time-averaged L1 value loss of recently experienced level trajectories, the average difficulty of the sampled levels should adapt to the agent's current abilities, following the reasoning outlined in the Value Correction Hypothesis, stated in Section 3.

As in Igl et al. (2019), we parameterize the agent policy as a 3-layer CNN with 16, 32, and 32 channels, with a final hidden layer of size 64. All kernels are 2×2 and use a stride of 1. For the ObstructedMazeGamut environments, we increase the number of channels of the final CNN layer to 64. We follow the same high-level generalization evaluation protocol used for Procgen, training the agent on a fixed set of 4000 levels for MultiRoom-N4-Random, 3000 levels for ObstructedMazeGamut-Easy, and 6000 levels for ObstructedMazeGamut-Medium, and testing on the full level distribution. We chose these values for $|\Lambda_{\text{train}}|$ to ensure roughly 1000 training levels of each difficulty setting of each environment. We model our PPO parameters on those used by Igl et al. (2019) in their MiniGrid experiments. We performed a grid search to find that PLR with rank prioritization, $\beta = 0.1$, and $\rho = 0.3$ learned most quickly on the MultiRoom environment, and used this setting for all our MiniGrid experiments. Table 3 summarizes these hyperparameter choices.

The remainder of this section provides more details about the various MiniGrid environments used in this work.

MultiRoom-N4-Random This environment requires the agent to navigate through 1, 2, 3, or 4 rooms respectively to reach a goal object, resulting in a natural ordering of levels over four levels of difficulty. The agent always starts at a random position in the furthest room from the goal object, facing a random direction. The goal object is also initialized to a random position within its room. See Figure 5 for screenshots of example levels.

ObstructedMazeGamut-Easy This environment consists of levels uniformly distributed across the first three difficulty settings of the ObstructedMaze environment, in which the agent must locate and pick up the key in order to unlock the door to pick up a goal object in a second room. The agent and goal object are always initialized in random positions in different rooms separated by the locked door.

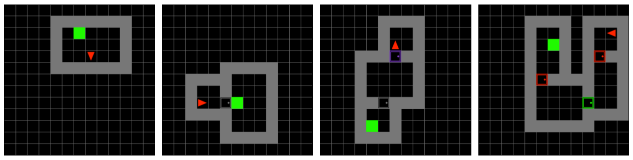

Figure 5. Example levels of each of the four difficulty levels of MultiRoom-N4-Random, in order of increasing difficulty from left to right. The agent (red triangle) must reach the goal (green square).

The second difficulty setting further requires the agent to first uncover the key from under a box before picking up the key. The third difficulty level further requires the agent to first move a ball blocking the door before entering the door. See Figure 6 for screenshots of example levels.

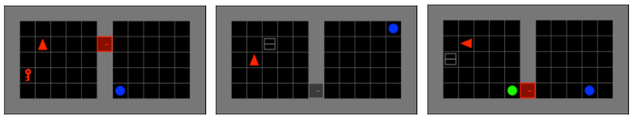

Figure 6. Example levels of each of the three difficulty levels of ObstructedMazeGamut-Easy, in order of increasing difficulty from left to right. The agent must find the key, which may be hidden under a box, to unlock a door, which may be blocked by an obstacle, to reach the goal object (blue circle).

ObstructedMazeGamut-Hard This environment consists of levels uniformly distributed across the first six difficulty levels of the ObstructedMaze environment. Harder levels corresponding to the fourth, fifth, and sixth difficulty settings include two additional rooms with no goal object to distract the agent. Each instance of these harder levels also contain two pairs of keys of different colors, each opening a door of the same color. The agent always starts one room away from the randomly positioned goal object. Each of the two keys is visible in the fourth difficulty setting and doors are unobstructed. The fifth difficulty setting hides the keys under boxes, and the sixth again places obstacles that must be removed before entering two of the doors, one of which is always the door to the goal-containing room. See Figure 7 for example screenshots.

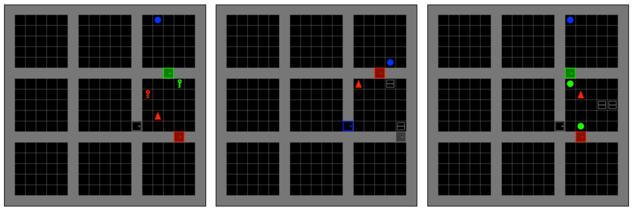

Figure 7. Example levels in increasing difficulty from left to right of each additional difficulty setting introduced by ObstructedMazeGamut-Hard in addition to those in ObstructedMazeGamut-Easy.

B. Additional Experimental Results

B.1. Extended Results on Procgen Benchmark

We present an overview of the improvements in test performance of each method across all 16 Procgen Benchmark games over 10 runs in Figure 14. For each game, Figure 15 further shows how the generalization gap changes over the course of training under each method tested. We show in Figures 16 and 17, the mean test episodic returns on the Procgen Benchmark (easy) for PLR with rank and proportional prioritization, respectively. In both of these plots, we can see that using only staleness ($\rho = 1$) or only L1 value loss scores ($\rho = 0$) is considerably worse than direct level sampling. Thus, we only observe gains compared to the baseline when both level scores and staleness are used for the sampling distribution. Comparing Figures 16 with 17 we find that PLR with proportional instead of rank prioritization provides statistically significant gains over uniform level sampling on an additional game (CoinRun), but rank prioritization leads to slightly larger mean improvements on several games.

Figure 18 shows that when PLR improves generalization performance, it also either matches or improves training sample efficiency, suggesting that when beneficial to test performance, the representations learned via the auto-curriculum induced by PLR prove similarly useful on training levels. However we see that our method reduces training sample efficiency on two games on which our method does not improve generalization performance. Since our method does not discover useful auto-curricula for these games, it is likely that uniformly sampling levels at training time allows the agent to better memorize useful behaviors on each of the training levels compared to the selective sampling performed by our method.

Finally, we also benchmarked PLR and UCB-DrAC + PLR against uniform sampling, TSCL Window, mixreg, and UCB-DrAC on Procgen hard across 5 runs per environment. Due to the high computational cost of the evaluation protocol for Procgen hard, which entails 200M training steps, we directly use the best hyperparameters found in the easy setting for each method. The results in Figure 10 show the two PLR-based methods significantly outperform all other methods in terms of normalized mean train and test episodic return, as well as reduction in mean generalization gap, attaining even greater margins of improvement than in the easy setting. As summarized by Table 4, the gains of PLR and UCB + PLR in mean normalized test return relative to uniform sampling in the hard setting are comparable to those in the easy setting. We provide plots of episodic test return over training for each individual environment in Figure 12.

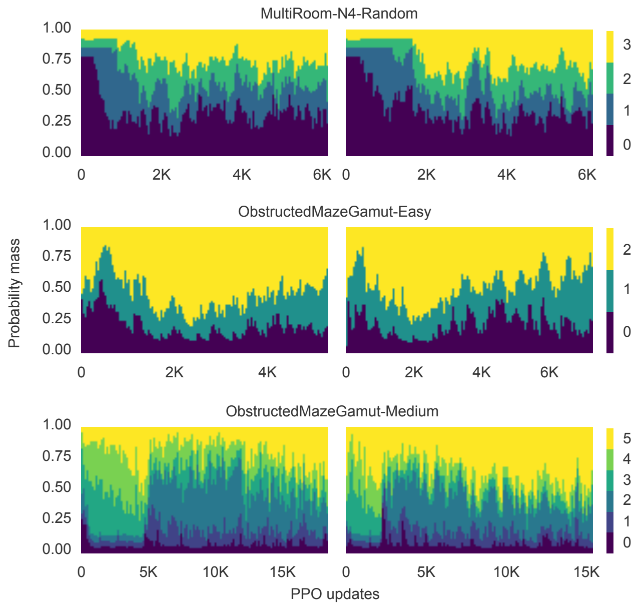

Figure 8. PLR consistently induces emergent curricula from easier to harder levels during training. Left and right correspond to two additional training runs independent of that in Figure 4.

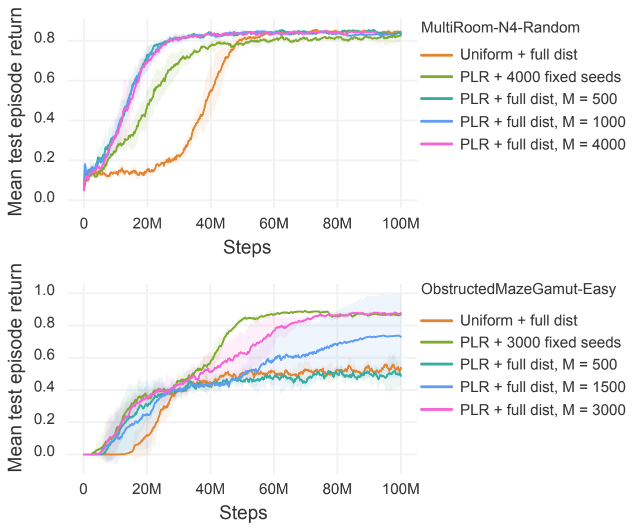

Figure 9. Mean test episodic returns on MultiRoom-N4-Random (top) and ObstructedMazeGamut-Easy (bottom) with access to the full level distribution at training. Plots are averaged over 3 runs. We set P_D to a Bernoulli parameterized as $p = 0.5$ for MultiRoom-N4-Random and $p = 0.95$ for ObstructedMazeGamut-Easy (found via grid search). As with all MiniGrid experiments using PLR, we use rank prioritization, $\beta = 0.1$, and $\rho = 0.3$.

B.2. Extended Results on Minigrid

To demonstrate that PLR consistently induces an emergent curriculum, we present plots showing the change in probability mass over different difficulty bins for additional training runs in Figure 8. Like in Figure 4, we see the probability mass assigned by P_{replay} gradually shifts from easier to harder levels over the course of training.

B.3. Training on the Full Level Distribution

While assessing generalization performance calls for using a fixed set of training levels, ideally our method can also make use of the full level distribution if given access to it. We take advantage of an unbounded number of training levels by modifying the list structures for storing scores and timestamps (see Algorithm 1 and 2) to track the top M levels by learning potential in our finite level buffer. When the lists are full, we set the next level for replacement to be

$$l_{\text{min}} = \arg\min_{l} P_{\text{replay}}(l).$$

When the outcome of the Bernoulli P_D entails sampling a new level l, the score and timestamps of l replace those of l_{min} only if the score of l_{min} is lower than that of l. In this way, PLR keeps a running buffer throughout training of the top M levels appraised to have the highest learning potential for replaying anew.

Figure 9 shows that with access to the full level distribution at training, PLR improves sample efficiency and generalization performance in both environments compared to uniform sampling on the full distribution. In MultiRoom-N4-Random, the value M makes little difference to test performance, and training with PLR on the full level distribution leads to a policy outperforming one trained with PLR on a fixed set of training levels. However, on ObstructedMazeGamut-Easy, a smaller M leads to worse test performance. Nevertheless, for all but $M = 500$, including the case of a fixed set of 3000 training levels, PLR leads to better mean test performance than uniform sampling on the full level distribution.

C. Algorithms

In this section, we provide detailed pseudocode for how PLR can be used for experience collection when using T-step rollouts. Algorithm 3 presents the extension of the generic policy-gradient training loop presented in Algorithm 1 to the case of T-step rollouts, and Algorithm 4 presents an implementation of experience collection in this setting (extending Algorithm 2). Note that when using T-step rollouts in the training loop, rollouts may start and end between episode boundaries. To compute level scores on full trajectories segmented across rollouts, we compute scores of partial episodes according to Equation 2, and record these partial scores alongside the partial episode step count in a separate buffer $\tilde{S}$. The function **score** then technically, optionally takes the additional input $\tilde{S}$ (through an abuse of notation) as an additional argument to stitch together this partial information into scores of full episodic trajectories.

Table 4. Comparison of test scores of PPO with PLR against PPO with uniform-sampling on the hard setting of Procgen Benchmark. Following (Raileanu et al., 2021), reported figures represent the mean and standard deviation of average test scores over 100 episodes aggregated across 5 runs, each initialized with a unique training seed. For each run, a normalized average return is computed by dividing the average test return for each game by the corresponding average test return of the uniform-sampling baseline over all 500 test episodes of that game, followed by averaging these normalized returns over all 16 games. The final row reports the mean and standard deviation of the normalized returns aggregated across runs. Bolded methods are not significantly different from the method with highest mean, unless all are, in which case none are bolded.

Environment	Uniform	TSCL	mixreg	UCB-DrAC	PLR	UCB-DrAC + PLR
BigFish	9.7 ± 1.8	$\mathbf{11.9 \pm 2.5}$	$\mathbf{12.0 \pm 2.5}$	10.9 ± 1.6	$\mathbf{15.3 \pm 3.6}$	$\mathbf{15.5 \pm 2.8}$
BossFight	$\mathbf{9.6 \pm 0.2}$	8.4 ± 0.7	$\mathbf{9.3 \pm 0.9}$	9.0 ± 0.2	$\mathbf{9.7 \pm 0.4}$	$\mathbf{9.5 \pm 1.1}$
CaveFlyer	3.5 ± 0.8	6.3 ± 0.6	4.0 ± 1.0	2.6 ± 0.8	6.4 ± 0.6	$\mathbf{8.0 \pm 0.9}$
Chaser	5.9 ± 0.5	6.2 ± 1.0	$\mathbf{6.5 \pm 0.8}$	$\mathbf{7.0 \pm 0.6}$	$\mathbf{6.8 \pm 2.2}$	$\mathbf{7.6 \pm 0.2}$
Climber	5.3 ± 1.1	5.2 ± 0.7	5.7 ± 0.7	6.1 ± 1.0	7.4 ± 0.6	7.6 ± 1.8
CoinRun	4.5 ± 0.4	5.8 ± 0.8	$\mathbf{6.2 \pm 1.0}$	5.2 ± 1.0	$\mathbf{6.8 \pm 0.6}$	$\mathbf{7.1 \pm 0.5}$
Dodgeball	3.9 ± 0.6	1.9 ± 0.9	4.7 ± 1.0	9.9 ± 1.2	7.4 ± 1.3	$\mathbf{12.4 \pm 0.7}$
FruitBot	$\mathbf{11.9 \pm 4.2}$	13.1 ± 2.3	$\mathbf{14.7 \pm 2.2}$	$\mathbf{15.6 \pm 3.7}$	$\mathbf{16.7 \pm 1.0}$	$\mathbf{12.9 \pm 5.1}$
Heist	1.5 ± 0.4	0.9 ± 0.3	1.2 ± 0.4	1.1 ± 0.3	1.3 ± 0.4	2.6 ± 2.2
Jumper	3.2 ± 0.3	3.2 ± 0.3	3.3 ± 0.4	2.9 ± 0.9	3.5 ± 0.5	3.3 ± 0.8
Leaper	7.1 ± 0.3	$\mathbf{7.5 \pm 0.5}$	$\mathbf{7.5 \pm 0.5}$	3.8 ± 1.6	$\mathbf{7.4 \pm 0.2}$	$\mathbf{8.2 \pm 0.7}$
Maze	3.6 ± 0.7	3.8 ± 0.6	3.9 ± 0.5	4.4 ± 0.2	4.0 ± 0.4	$\mathbf{6.2 \pm 0.4}$
Miner	12.8 ± 1.4	11.7 ± 0.9	13.3 ± 1.6	$\mathbf{16.1 \pm 0.6}$	11.3 ± 0.7	$\mathbf{15.3 \pm 0.8}$
Ninja	5.2 ± 0.1	5.9 ± 0.8	5.0 ± 1.0	5.2 ± 1.0	$\mathbf{6.1 \pm 0.6}$	$\mathbf{6.9 \pm 0.3}$
Plunder	3.2 ± 0.1	5.4 ± 1.1	3.7 ± 0.4	7.8 ± 1.1	8.6 ± 2.7	$\mathbf{17.5 \pm 1.3}$
StarPilot	5.5 ± 0.6	2.1 ± 0.4	6.9 ± 0.6	$\mathbf{11.2 \pm 1.7}$	5.4 ± 0.8	$\mathbf{12.3 \pm 1.5}$
Normalized test returns (%)	100.0 ± 2.0	103.9 ± 3.5	110.6 ± 3.9	126.6 ± 3.0	135.0 ± 6.1	$\mathbf{182.9 \pm 8.2}$

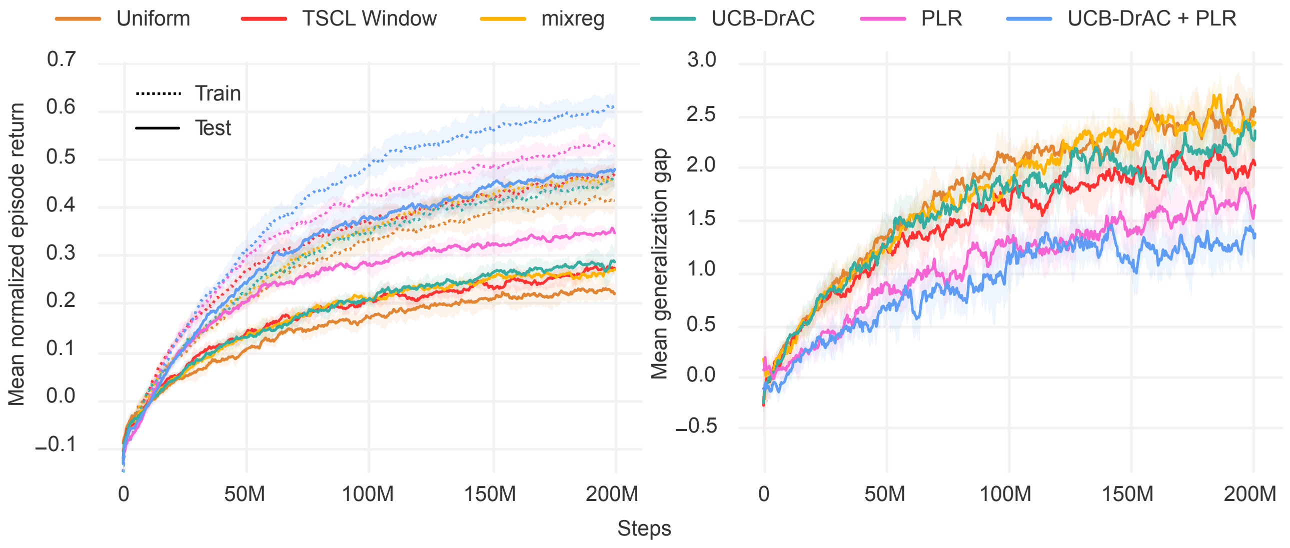

Figure 10. Left: Mean normalized train and test episode returns on Procgen Benchmark (hard). Right: Corresponding generalization gaps during training. All curves are averaged across all environments over 5 runs. The shaded area indicates one standard deviation around the mean. PLR-based methods statistically significantly outperform all others in both train and test returns. Only the PLR-based methods statistically significantly reduce the generalization gap ($p < 0.05$).

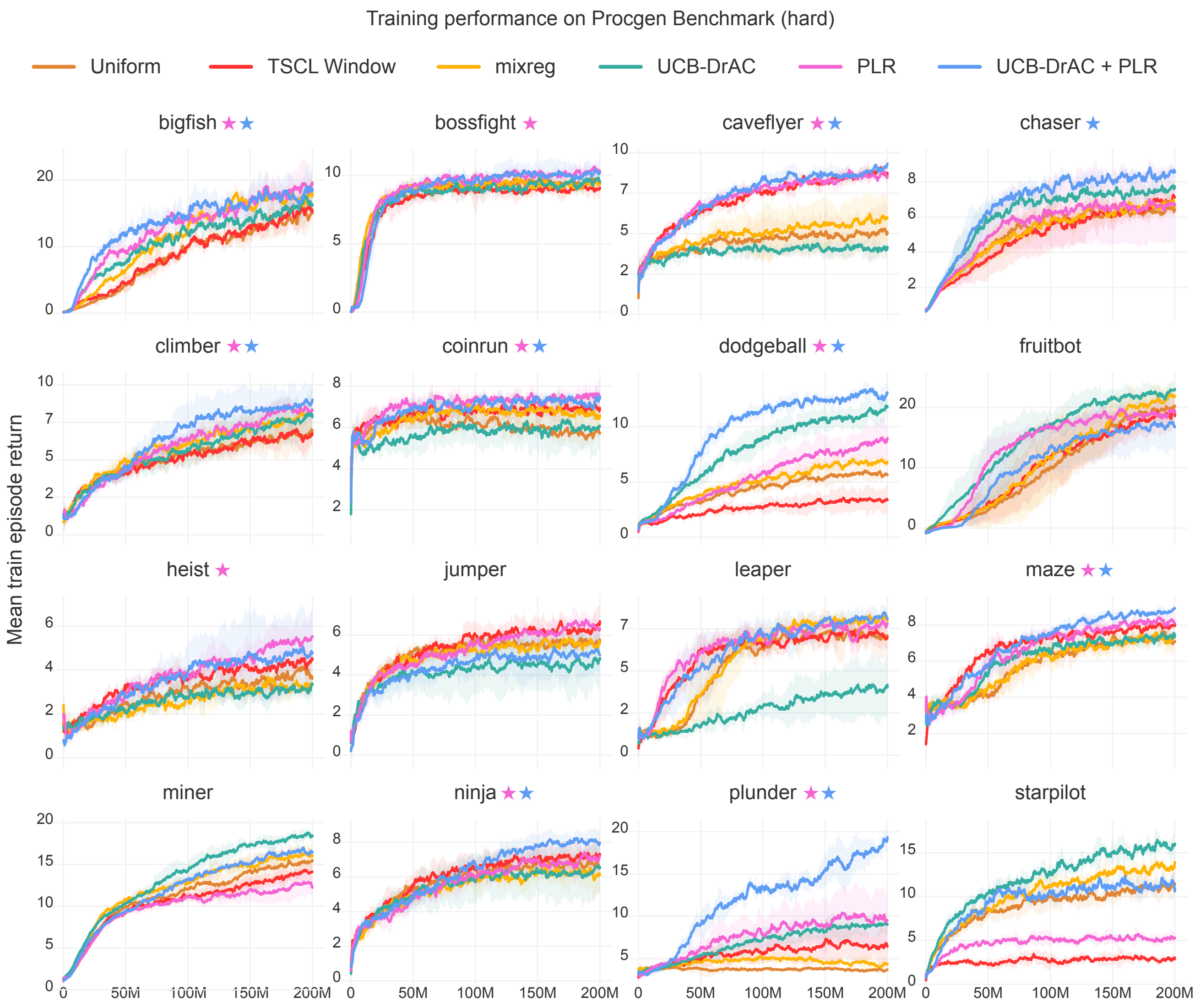

Figure 11. Mean train episode returns (5 runs) on Procgen Benchmark (hard), using the best hyperparameters found on the easy setting. The shaded area indicates one standard deviation around the mean. A ★ indicates statistically significant improvement over the uniform-sampling baseline by the PLR-based method of the matching color ($p < 0.05$). Note that while PLR reduces training performance on StarPilot, it performs comparably to the uniform-sampling baseline at test time, indicating less overfitting to training levels.

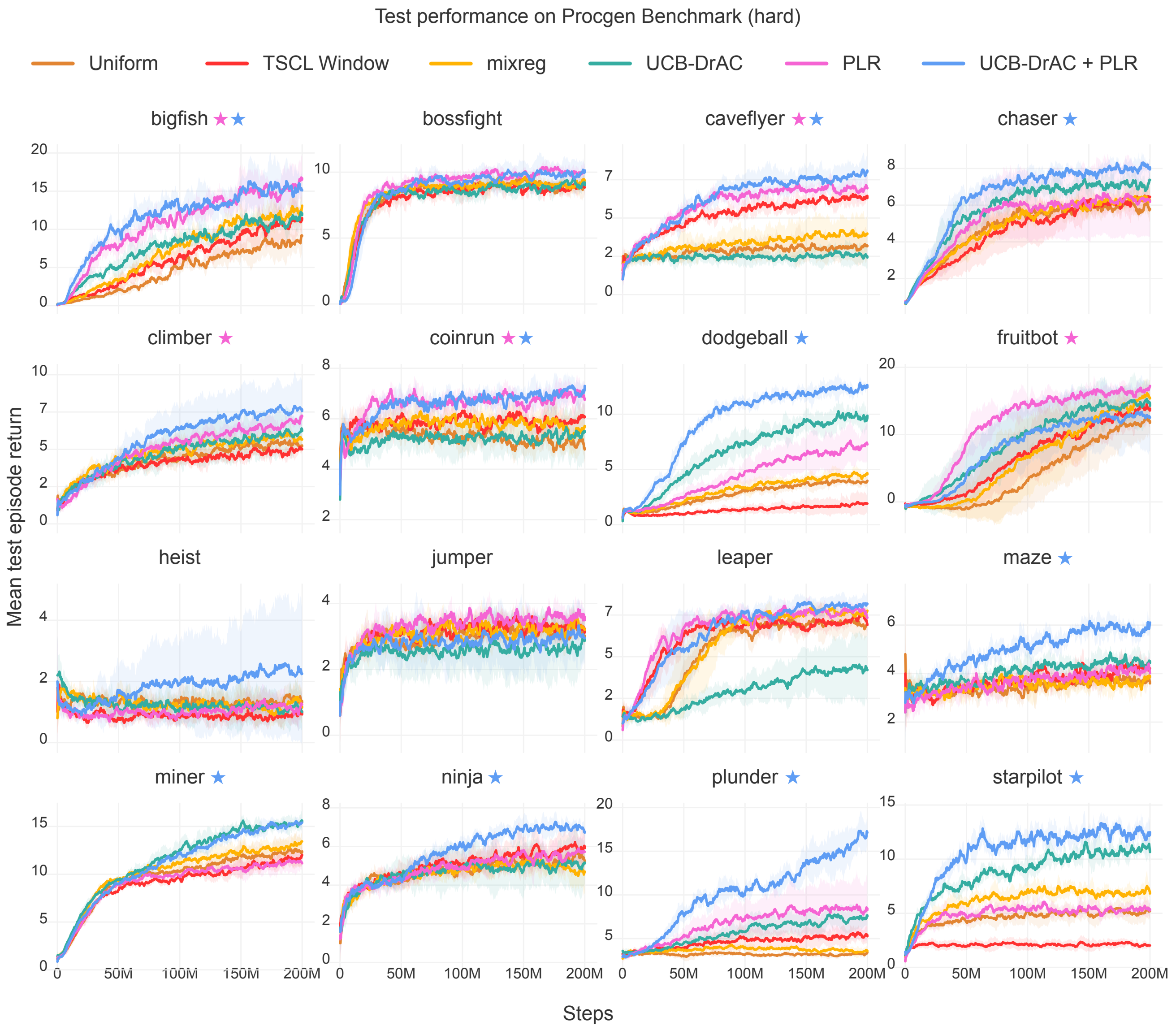

Figure 12. Mean test episode returns (5 runs) on Procgen Benchmark (hard), using best hyperparameters found on the easy setting. The shaded area indicates one standard deviation around the mean. A ★ indicates statistically significant improvement over the uniform-sampling baseline by the PLR-based method of the matching color ($p < 0.05$).

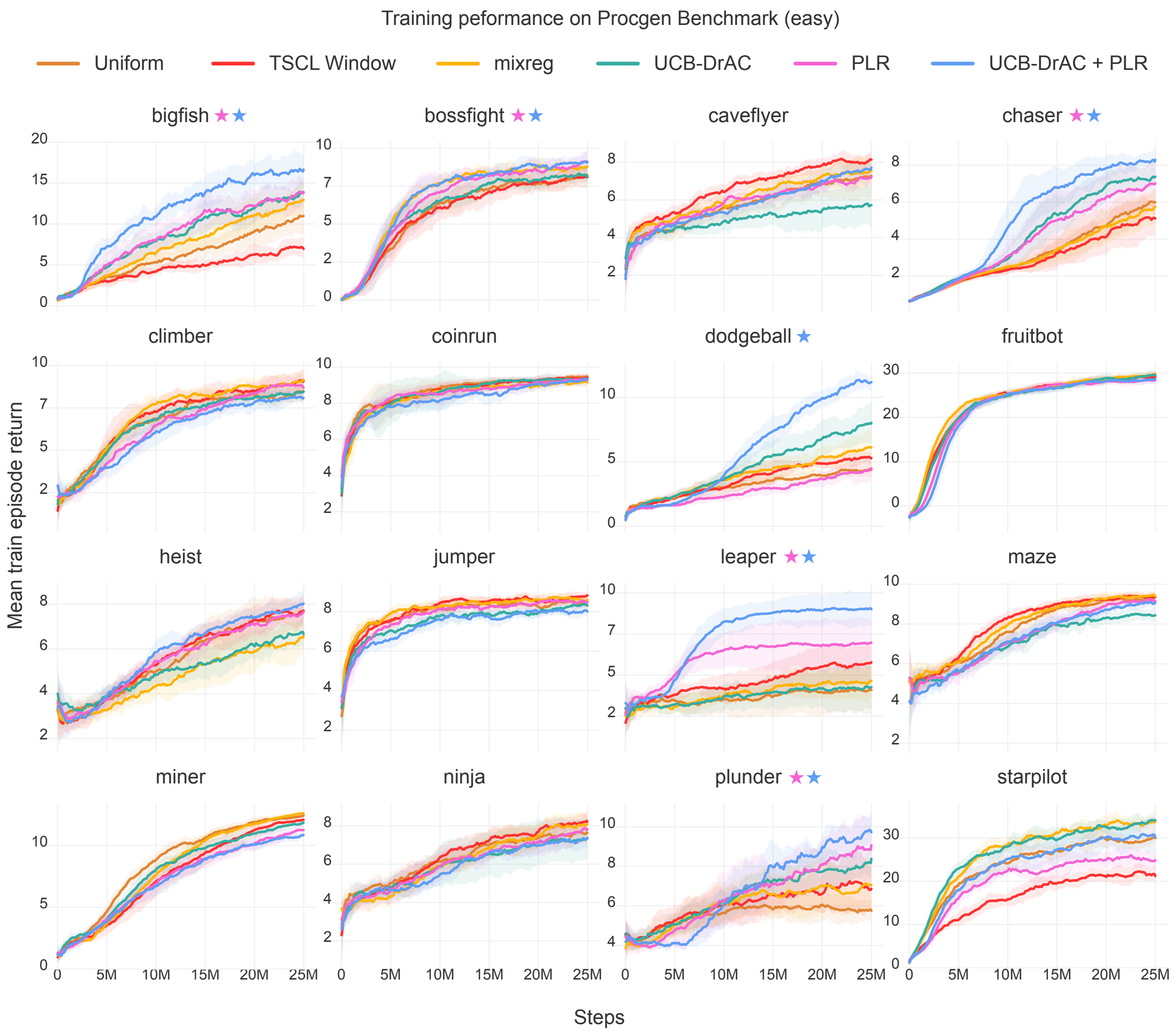

Figure 13. Mean train episode returns (5 runs) on Procgen Benchmark (easy). The shaded area indicates one standard deviation around the mean. A ★ indicates statistically significant improvement over the uniform-sampling baseline by the PLR-based method of the matching color ($p < 0.05$). PLR tends to improve or match training sample efficiency. Note that while PLR reduces training performance on StarPilot, it performs comparably to the uniform-sampling baseline at test time, indicating less overfitting to training levels.

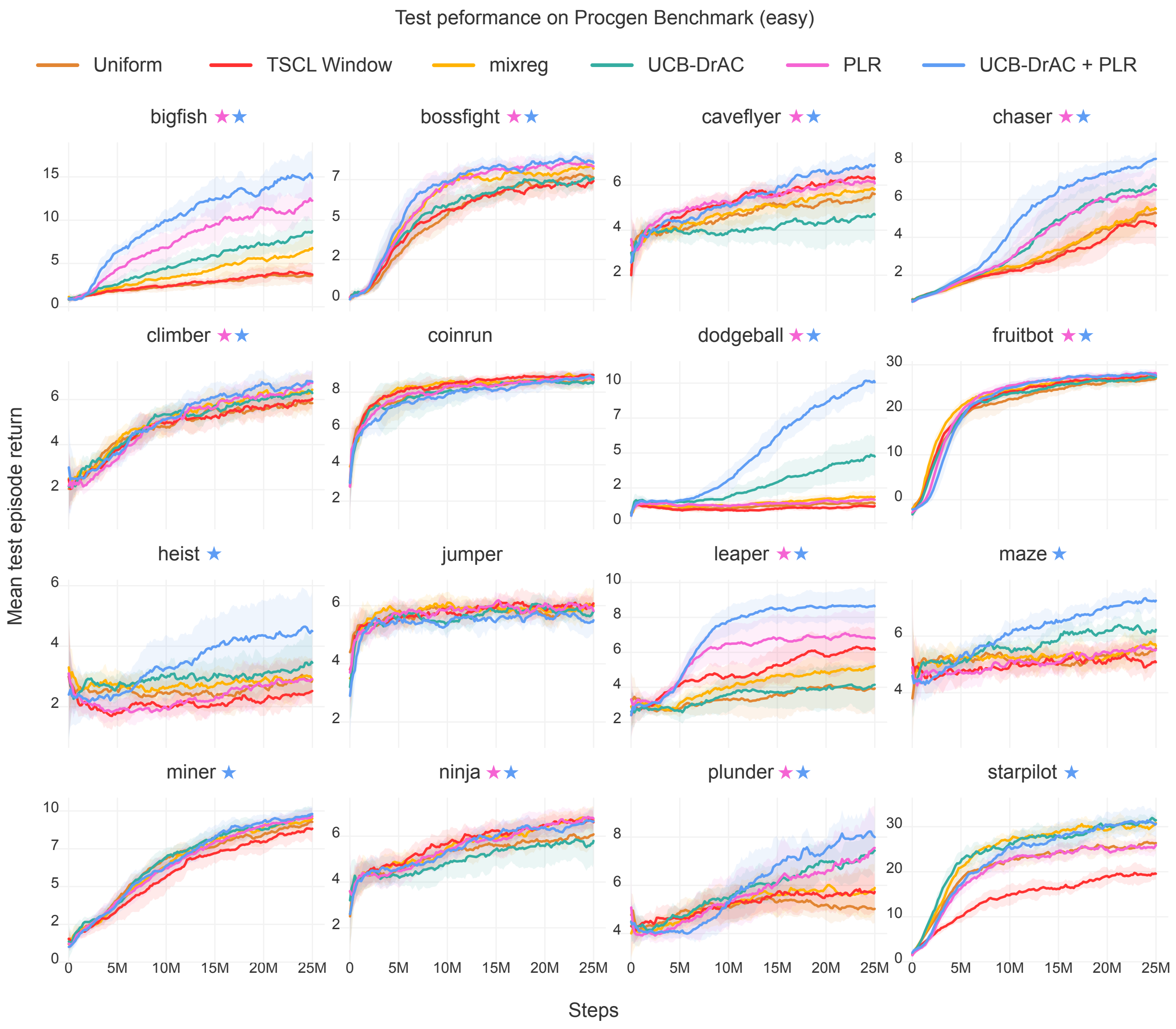

Figure 14. Mean test episode return (10 runs) on each Procgen Benchmark game (easy). The shaded area indicates one standard deviation around the mean. PLR-based methods consistently match or outperform uniform sampling with statistically significance ($p < 0.05$), indicated by a ★ of the corresponding color. We see that TSCL results in inconsistent outcomes across games, notably drastically lower test returns on StarPilot.

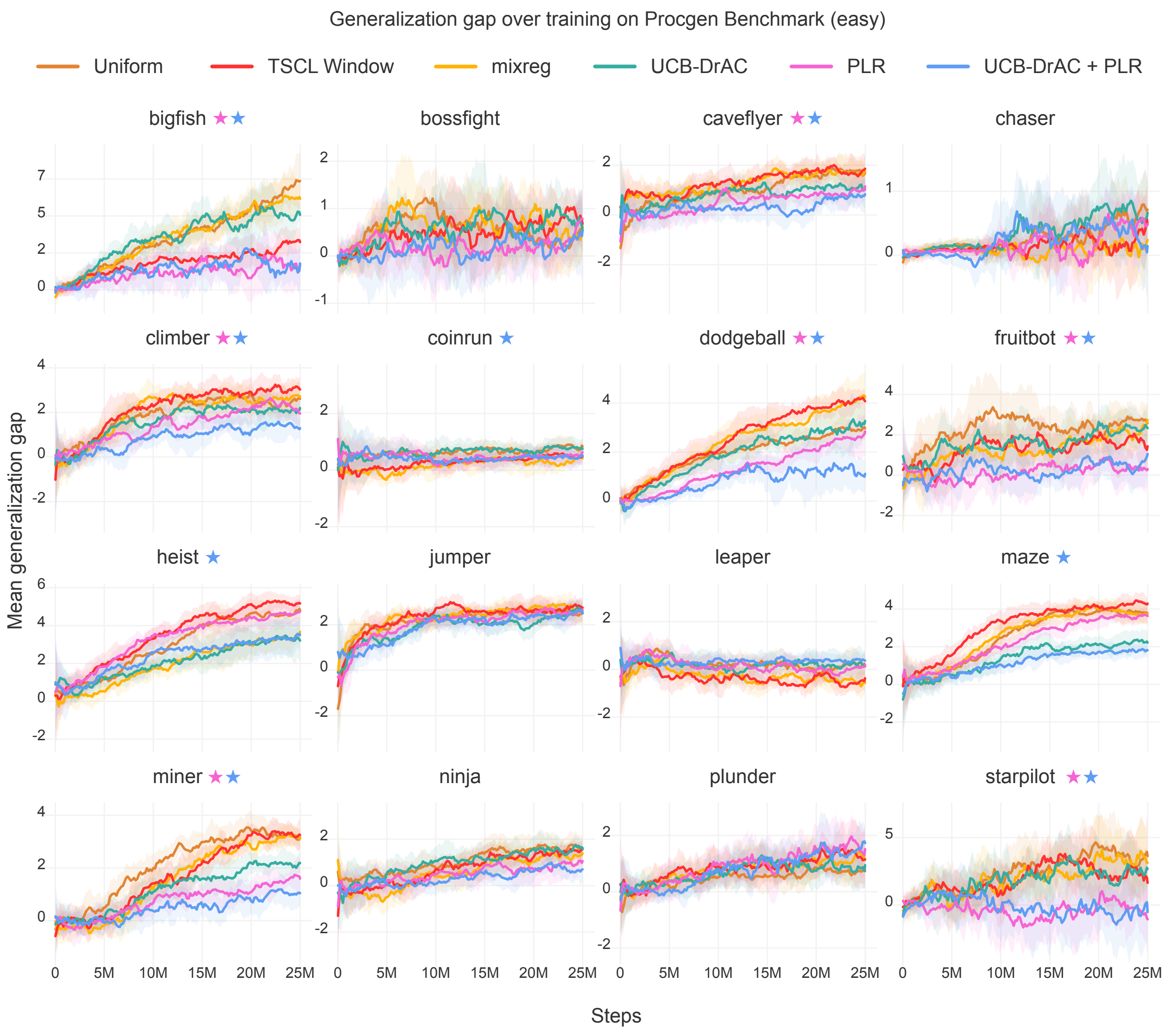

Figure 15. Mean generalization gaps throughout training (10 runs) on each Procgen Benchmark game (easy). The shaded area indicates one standard deviation around the mean. A ★ indicates the method of matching color results in a statistically significant ($p < 0.05$) reduction in generalization gap compared to the uniform-sampling baseline. By itself, PLR significantly reduces the generalization gap on 7 games, and UCB-DrAC, on 5 games. This number jumps to 10 of 16 games when these two methods are combined. TSCL only significantly reduces generalization gap on 2 of 16 games relative to uniform sampling, while increasing it on others, most notably on Dodgeball.

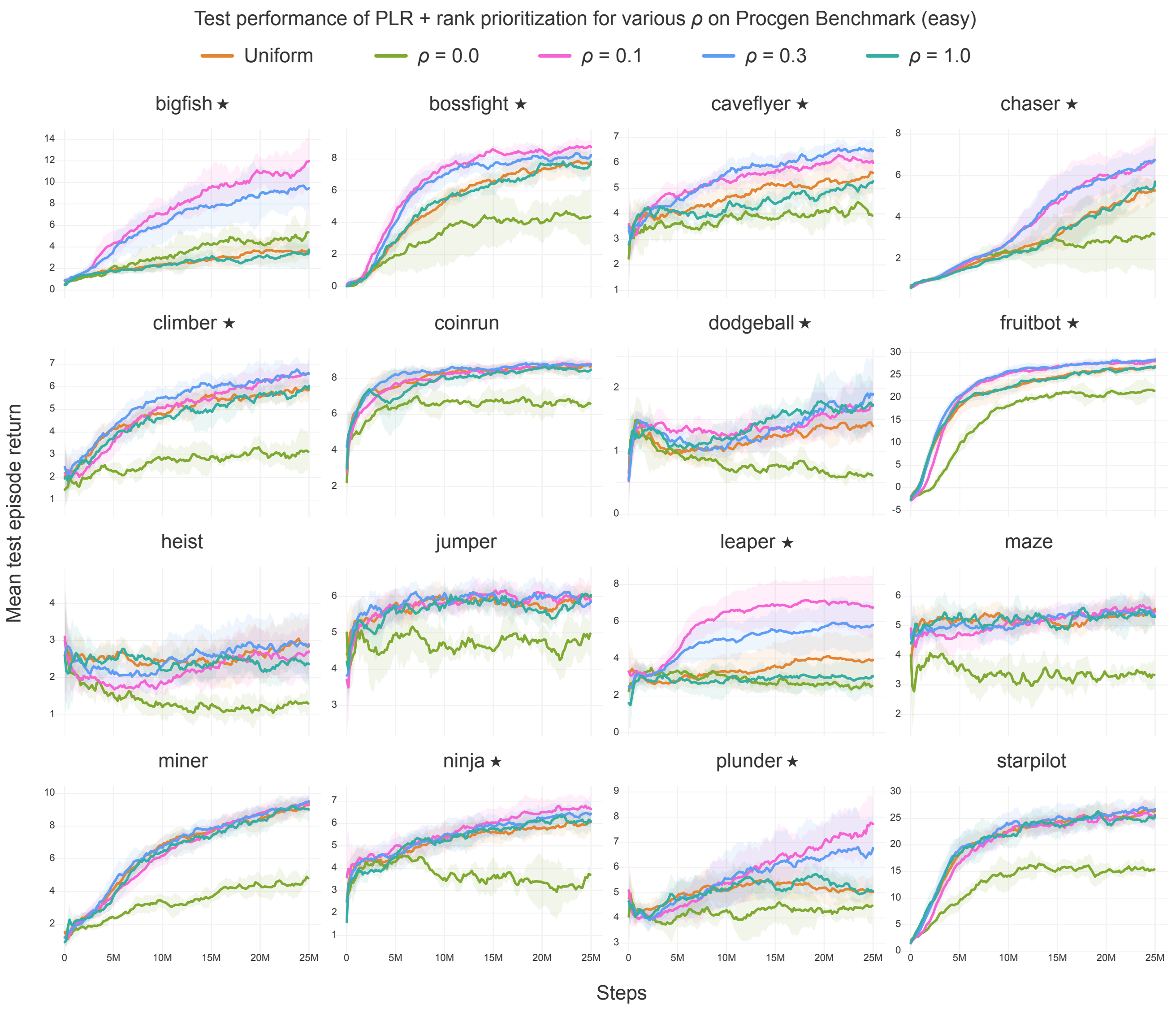

Figure 16. Mean test episode returns (10 runs) on the Procgen Benchmark (easy) for PLR with rank prioritization and $\beta = 0.1$ across a range of staleness coefficient values, ρ. The replay distribution must consider both the L1 value-loss and staleness values to realize improvements to generalization and sample efficiency. The shaded area indicates one standard deviation around the mean. A ★ next to the game name indicates that $\rho = 0.1$ exhibits statistically significantly better final test returns or sample efficiency along the test curve ($p < 0.05$), which we observe in 10 of 16 games.

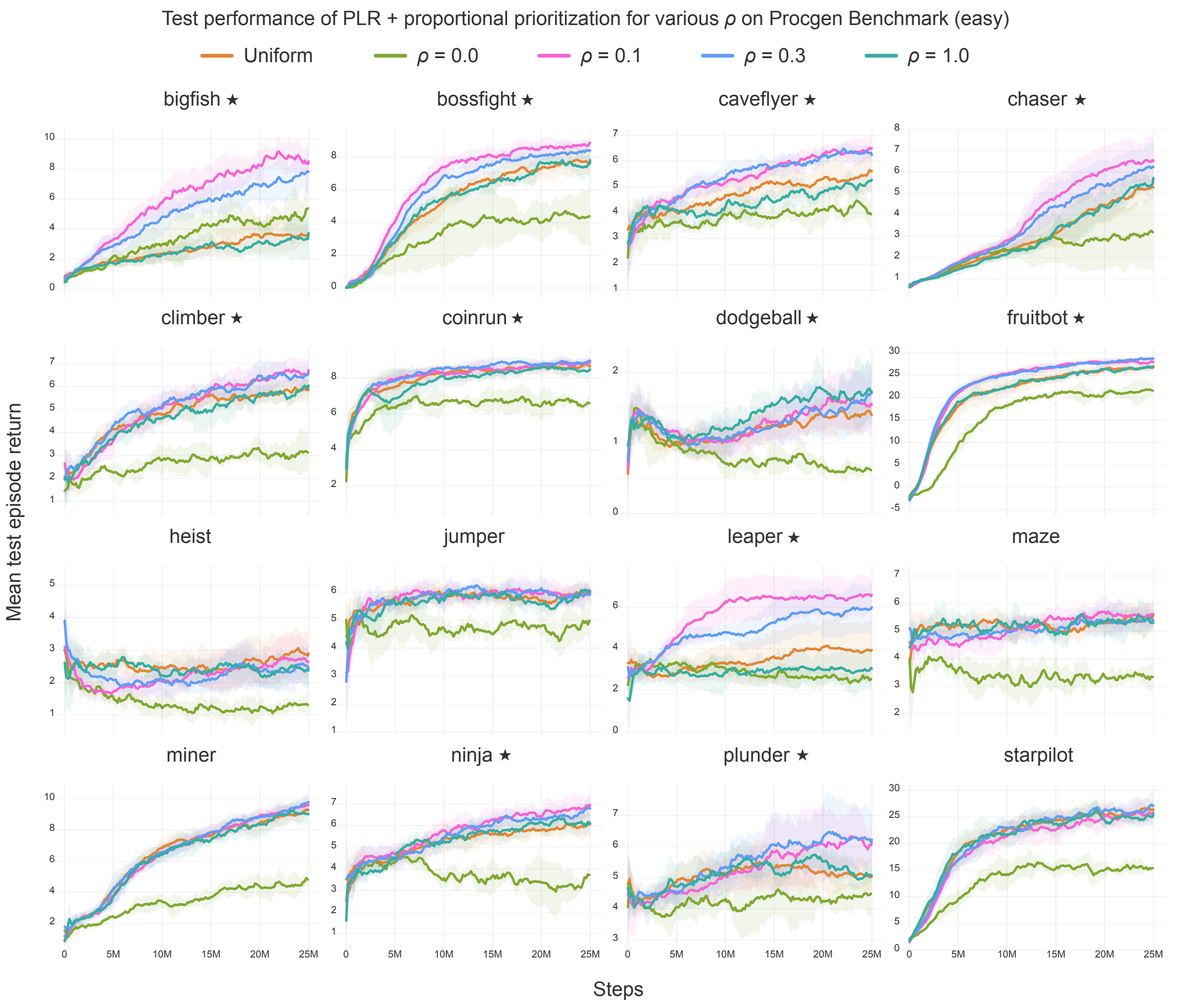

Figure 17. Mean test episode returns (10 runs) on the Procgen Benchmark (easy) for PLR with proportional prioritization and $\beta = 0.1$ across a range of values of ρ. As in the case of rank prioritization, the replay distribution must consider both the L1 value loss score and staleness values in order to realize performance improvements. The shaded area indicates one standard deviation around the mean. A ★ next to the game name indicates the condition $\rho = 0.1$ exhibits statistically significantly better final test returns or sample efficiency along the test curve ($p < 0.05$), which we observe in 11 of 16 games.

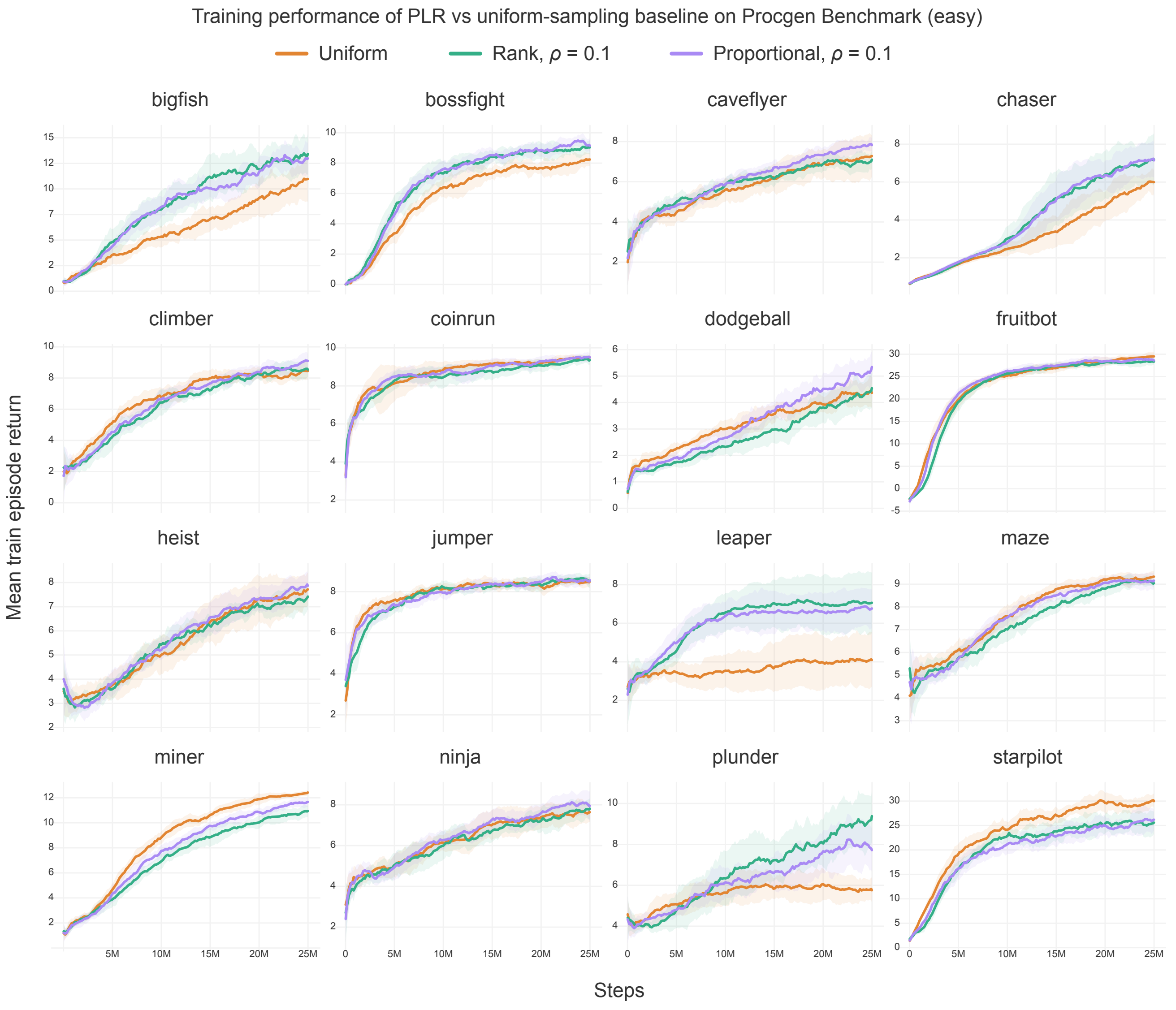

Figure 18. Mean training episode returns (10 runs) on the Procgen Benchmark for (easy) PLR with $\beta = 0.1$, $\rho = 0.1$, and each of rank and proportional prioritization. On some games, PLR improves both training sample efficiency and generalization performance (e.g. BigFish and Chaser), while on others, only generalization performance (e.g. CaveFlyer with rank prioritization). The shaded area indicates one standard deviation around the mean.

Algorithm 3 Generic T-step policy-gradient training loop with prioritized level replay

```
Input: Training levels $\Lambda_{\text{train}}$ of an environment, policy $\pi_\theta$, rollout length $T$, number of updates $N_u$, batch size $N_b$,
  policy update function $\mathcal{U}(\mathcal{B}, \theta) \rightarrow \theta'$.
  Initialize level scores $S$, partial level scores $\tilde{S}$, and level timestamps $C$
  Initialize global episode count $c \leftarrow 0$
  Initialize set of visited levels $\Lambda_{\text{seen}} = \varnothing$
  Initialize experience buffer $\mathcal{B} = \varnothing$
  Initialize $N_b$ parallel environment instances $E$, each set to a random level in $\in \Lambda_{\text{train}}$
  for $u = 1$ to $N_u$ do
    $\mathcal{B} \leftarrow \varnothing$
    for $k = 1$ to $N_b$ do
      $\mathcal{B} \leftarrow \mathcal{B} \cup$ collect_experiences$(k, E, \Lambda_{\text{train}}, \Lambda_{\text{seen}}, \pi_\theta, T, S, \tilde{S}, C, c)$    Using Algorithm 4
    end for
    $\theta \leftarrow \mathcal{U}(\mathcal{B}, \theta)$
  end for
```

Algorithm 4 Collect T-step rollouts with prioritized level replay

```
Input: Actor index $k$, batch environments $E$, training levels $\Lambda_{\text{train}}$, visited levels $\Lambda_{\text{seen}}$, current level $l$, policy $\pi_\theta$, rollout
  length $T$, scoring function $\mathbf{score}$, level scores $S$, partial scores $\tilde{S}$, staleness values $C$, and global episode count $c$.
Output: Experience buffer $\mathcal{B}$
  Initialize $\mathcal{B} = \varnothing$, and set current level $l_i = E_k$
  Observe current state $s_0$, termination flag $d_0$
  if $d_0$ then
    Define new index $i \leftarrow |S| + 1$
    Choose current level $l_i \leftarrow \mathbf{sampleNextLevel}(\Lambda_{\text{train}}, S, C, c)$ and $E_k \leftarrow l_i$
    Update level timestamp $C_i \leftarrow c$
    Observe initial state $s_0$
  end if
  Choose $a_0 \sim \pi_\theta(\cdot|s_0)$
  t = 1
  Initialize episodic trajectory buffer $\tau = \varnothing$
  while $t < T$ do
    Observe $(s_t, r_t, d_t)$
    $\mathcal{B} \leftarrow \mathcal{B} \cup (s_{t-1}, a_{t-1}, s_t, r_t, d_t, \log \pi_\theta(a))$
    $\tau \leftarrow \tau \cup (s_{t-1}, a_{t-1}, s_t, r_t, d_t, \log \pi_\theta(a))$
    if $d_t$ then
      Update level score $S_i \leftarrow \mathbf{score}(\tau, \pi_\theta, \tilde{S}_i)$ and partial score $\tilde{S}_i \leftarrow 0$
      $\tau \leftarrow \varnothing$
      Define new index $i \leftarrow |S| + 1$
      Update current level $l_i \leftarrow$ sampleNextLevel$(\Lambda_{\text{train}}, S, C, c)$ and $E_k \leftarrow l_i$
      Update level timestamp $C_i \leftarrow c$
    end if
    Choose $a_{t+1} \sim \pi_\theta(\cdot|s_t)$
    $t \leftarrow t + 1$
  end while
  if not $d_t$ then
    $\tilde{S}_i \leftarrow$ (score$(\tau, \pi_\theta), |\tau|)$          Track partial time-averaged score and $|\tau|$
  end if

  function sampleNextLevel$(\Lambda_{\text{train}}, S, C, c)$
    $c \leftarrow c + 1$
    Sample replay decision $d \sim P_D(d)$
    if $d = 0$ and $|\Lambda_{\text{train}} \setminus \Lambda_{\text{seen}}| > 0$ then
      Define new index $i \leftarrow |S| + 1$
      Sample $l_i \sim P_{\text{new}}(l|\Lambda_{\text{train}}, \Lambda_{\text{seen}})$          Sample an unseen level, if any
      Add $l_i$ to $\Lambda_{\text{seen}}$, add initial value $S_i = 0$ to $S$ and $C_i = 0$ to $C$
    else
      Sample $l_i \sim (1 - \rho) \cdot P_S(l|\Lambda_{\text{seen}}, S) + \rho \cdot P_C(l|\Lambda_{\text{seen}}, C, c)$          Sample a level for replay
    end if
    return $l_i$
  end function
```