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.

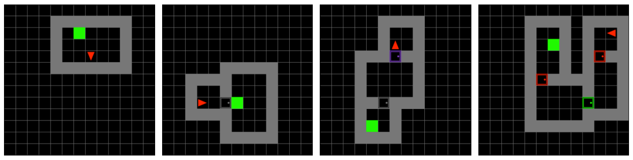

Figure 5. Example levels of each of the four difficulty levels of MultiRoom-N4-Random, in order of increasing difficulty from left to right. The agent (red triangle) must reach the goal (green square).

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.

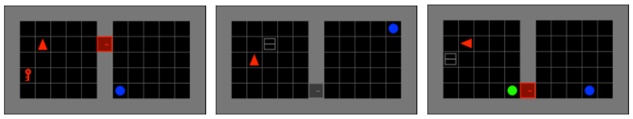

Figure 6. Example levels of each of the three difficulty levels of ObstructedMazeGamut-Easy, in order of increasing difficulty from left to right. The agent must find the key, which may be hidden under a box, to unlock a door, which may be blocked by an obstacle, to reach the goal object (blue circle).

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.

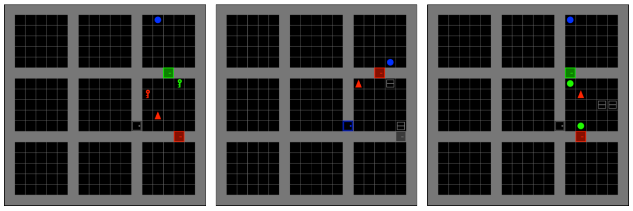

Figure 7. Example levels in increasing difficulty from left to right of each additional difficulty setting introduced by ObstructedMazeGamut-Hard in addition to those in ObstructedMazeGamut-Easy.

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.

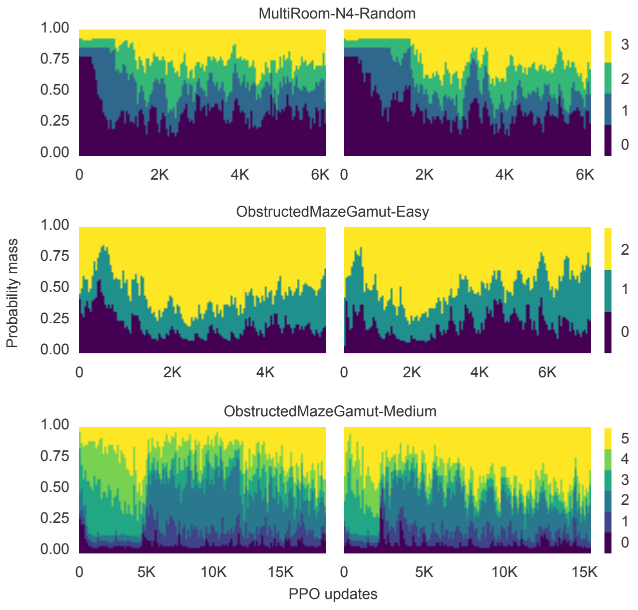

Figure 8. PLR consistently induces emergent curricula from easier to harder levels during training. Left and right correspond to two additional training runs independent of that in Figure 4.

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.

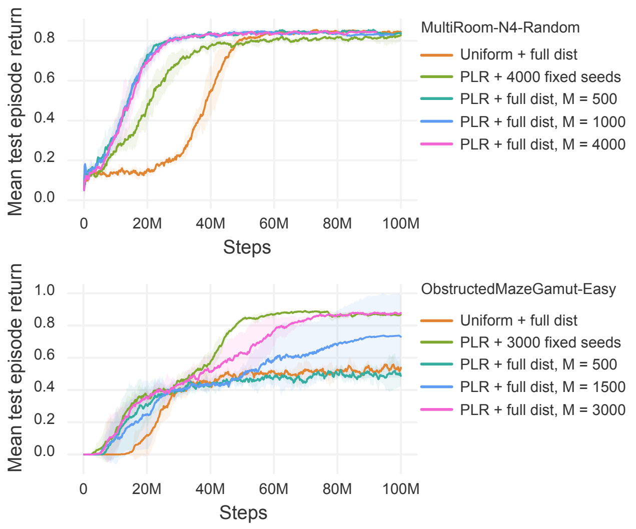

Figure 9. Mean test episodic returns on MultiRoom-N4-Random (top) and ObstructedMazeGamut-Easy (bottom) with access to the full level distribution at training. Plots are averaged over 3 runs. We set P_D to a Bernoulli parameterized as $p = 0.5$ for MultiRoom-N4-Random and $p = 0.95$ for ObstructedMazeGamut-Easy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

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	$\mathbf{182.9 \pm 8.2}$

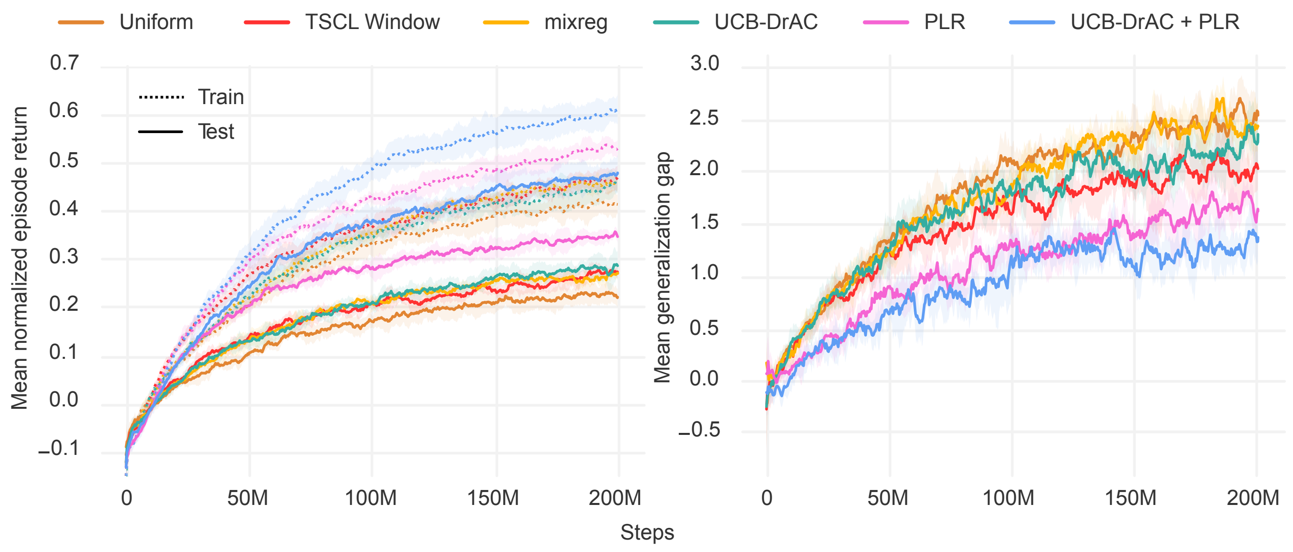

Figure 10. Left: Mean normalized train and test episode returns on Procgen Benchmark (hard). Right: Corresponding generalization gaps during training. All curves are averaged across all environments over 5 runs. The shaded area indicates one standard deviation around the mean. PLR-based methods statistically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

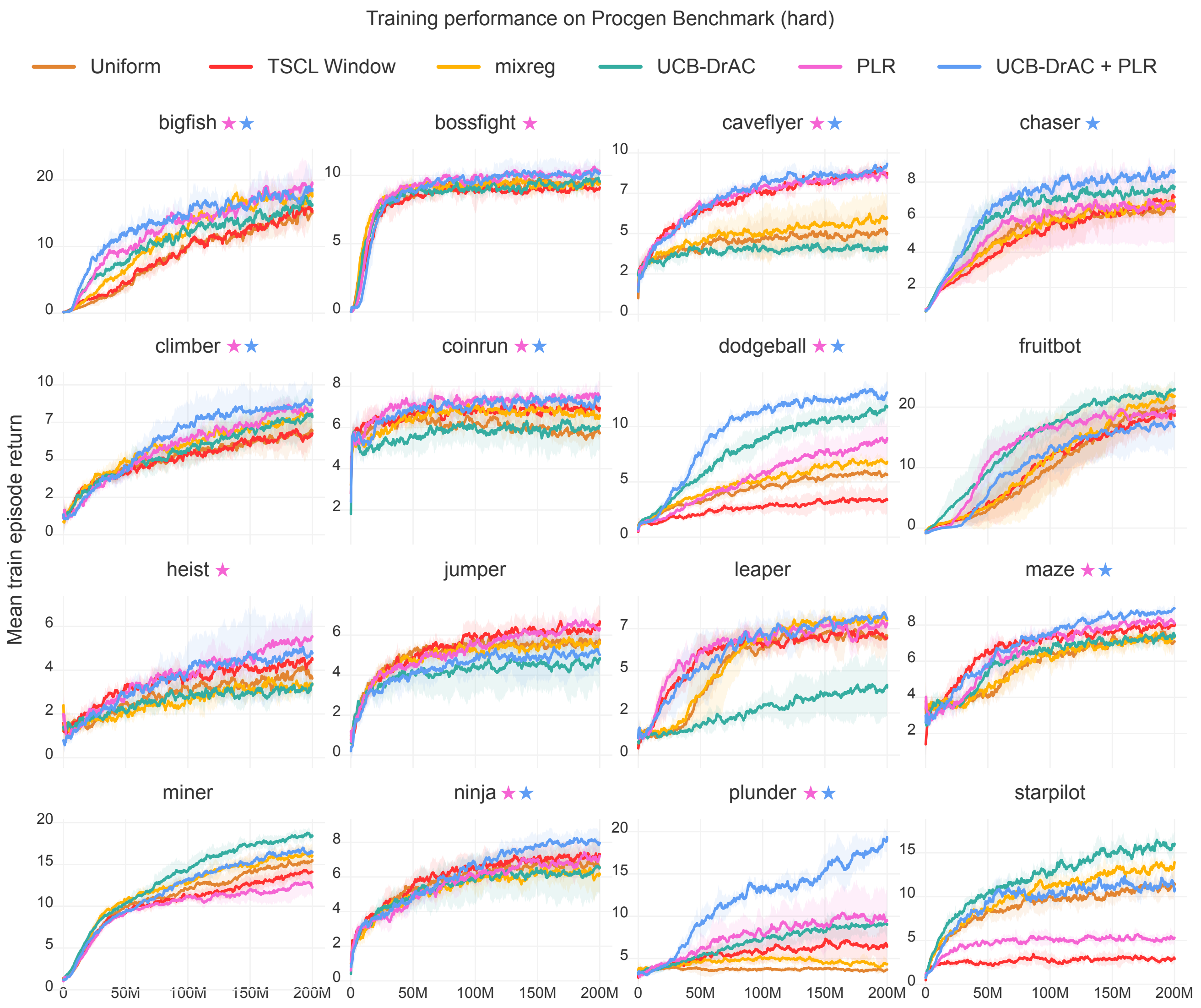

Figure 11. Mean train episode returns (5 runs) on Procgen Benchmark (hard), using the best hyperparameters found on the easy setting. The shaded area indicates one standard deviation around the mean. A ★ indicates statistically significant improvement over the uniform-sampling baseline by the PLR-based 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.

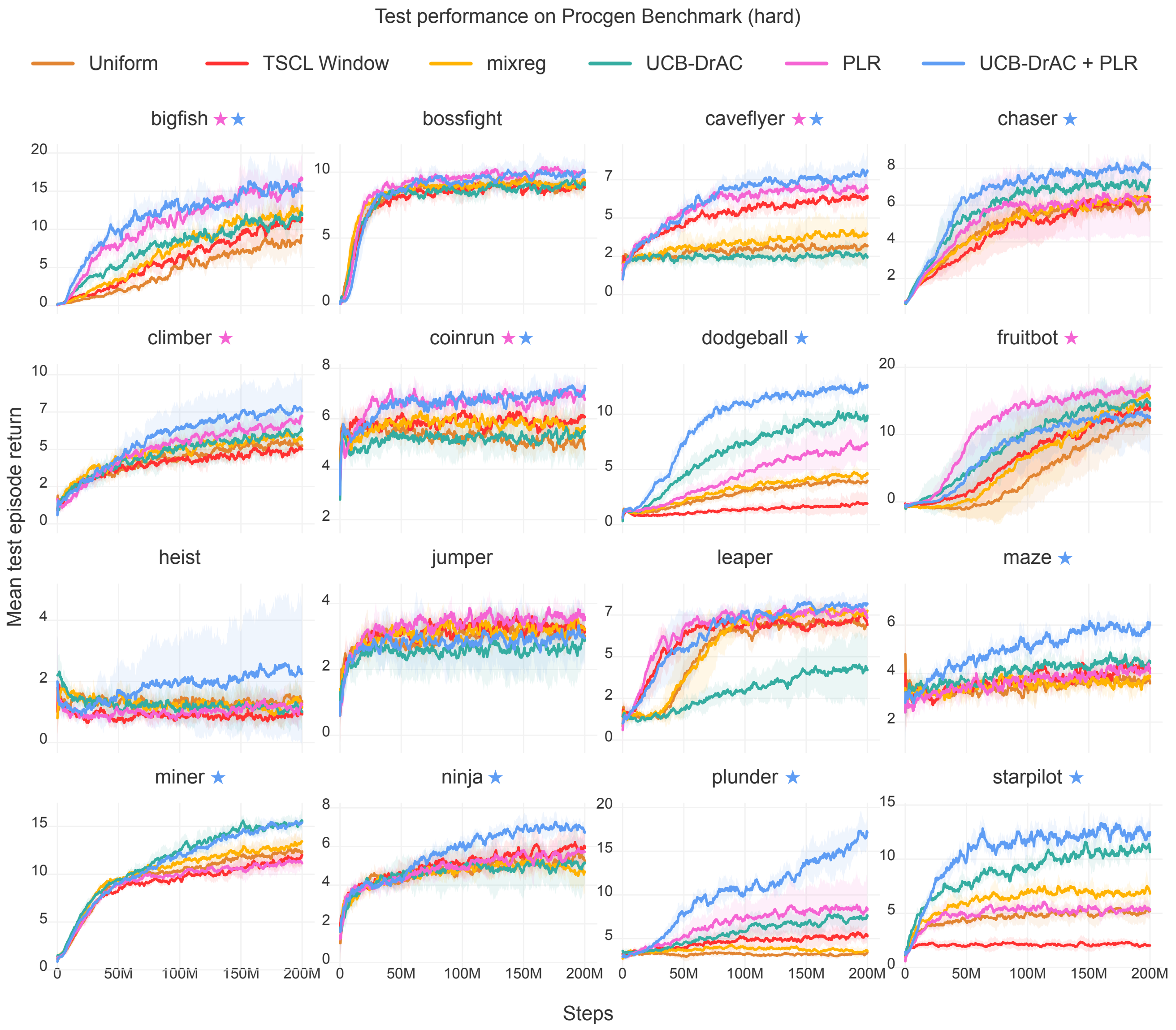

Figure 12. Mean test episode returns (5 runs) on Procgen Benchmark (hard), using best hyperparameters found on the easy setting. The shaded area indicates one standard deviation around the mean. A ★ indicates statistically significant improvement over the uniform-sampling baseline by the PLR-based method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

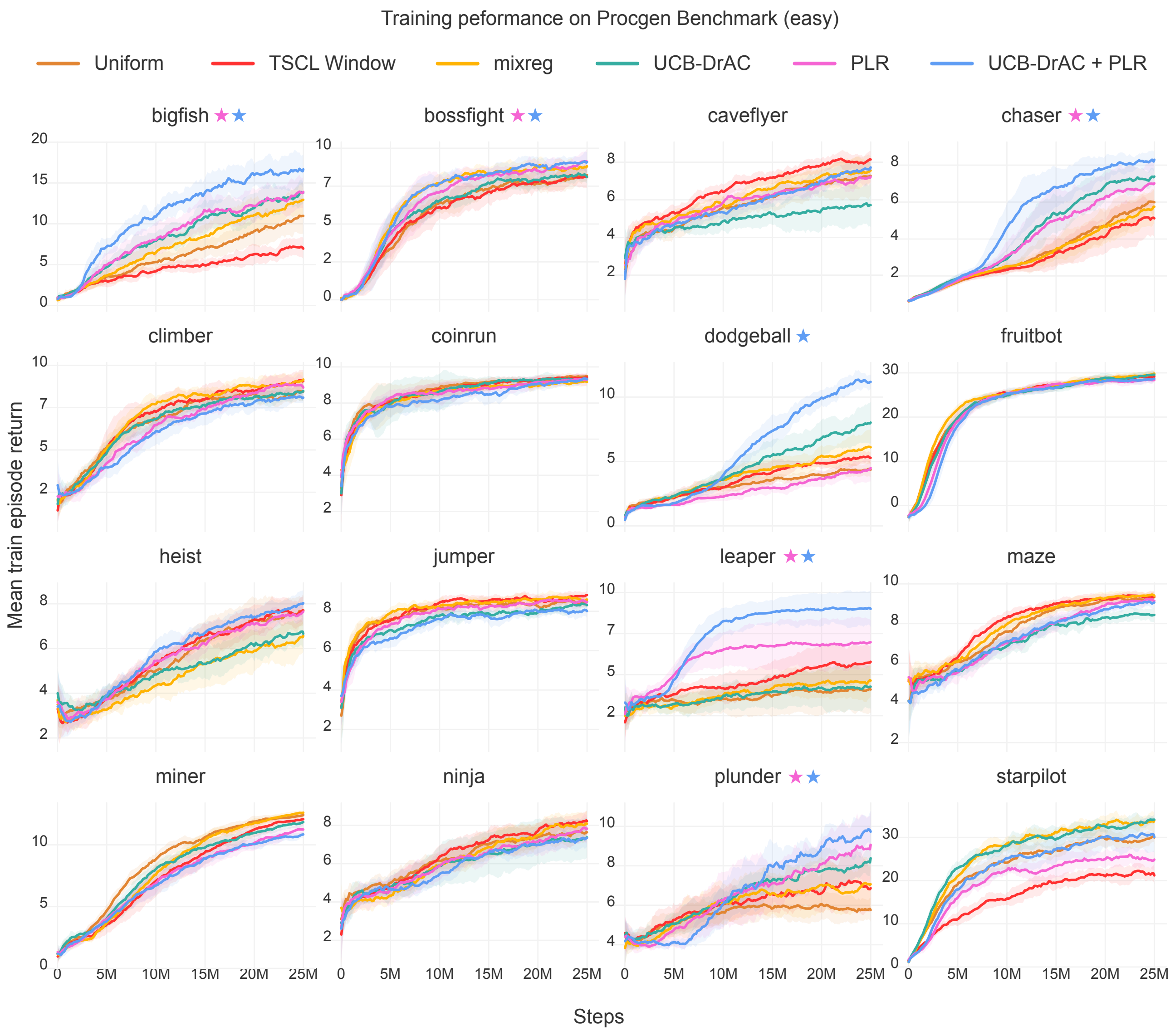

Figure 13. Mean train episode returns (5 runs) on Procgen Benchmark (easy). The shaded area indicates one standard deviation around the mean. A ★ indicates statistically significant improvement over the uniform-sampling baseline by the PLR-based method of the matching color ($p < 0.05$). PLR tends to 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.

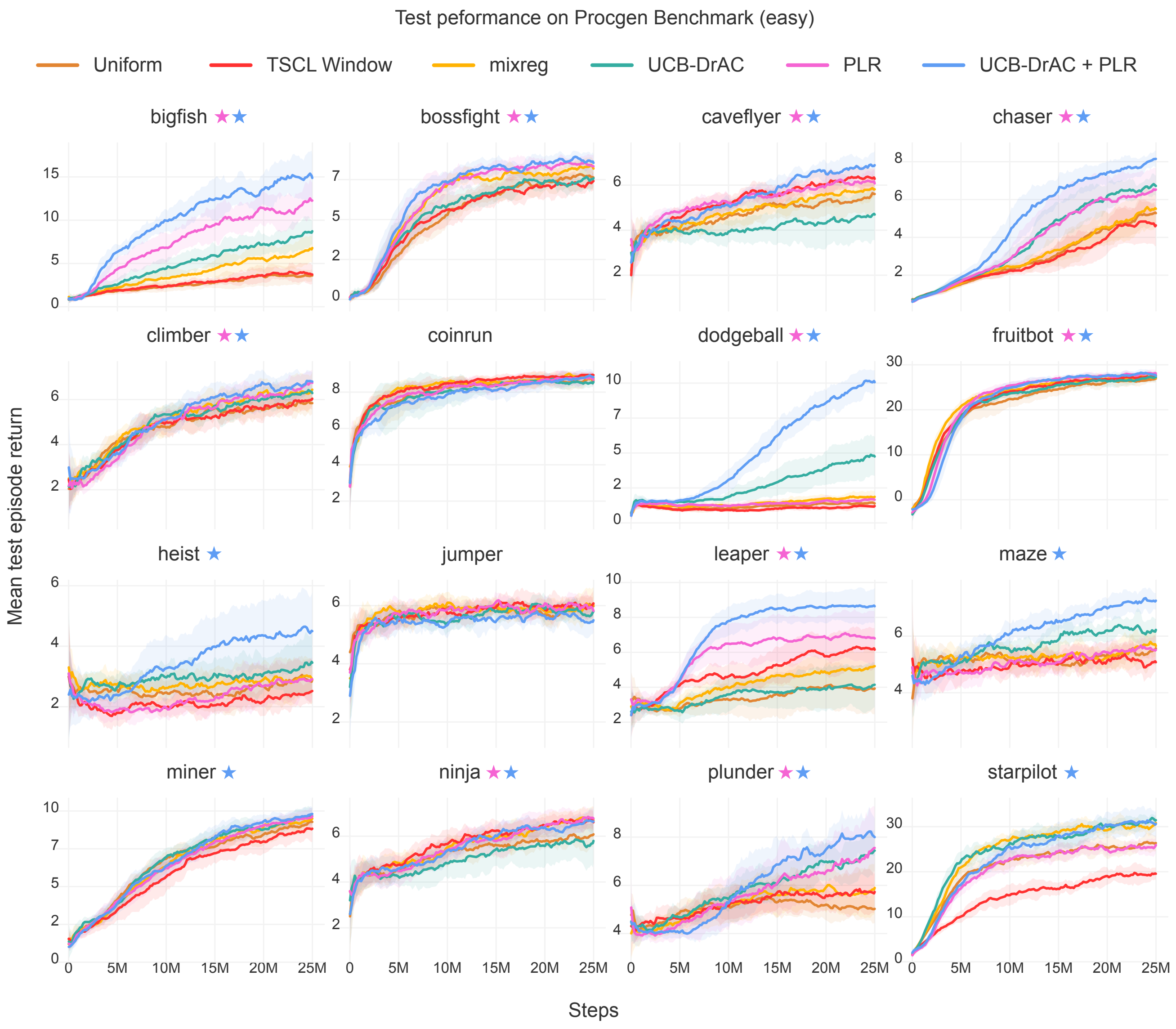

Figure 14. Mean test episode return (10 runs) on each Procgen Benchmark game (easy). The shaded area indicates one standard deviation around the mean. PLR-based methods consistently match or outperform uniform sampling with statistically significance ($p < 0.05$), indicated by a ★ of the corresponding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

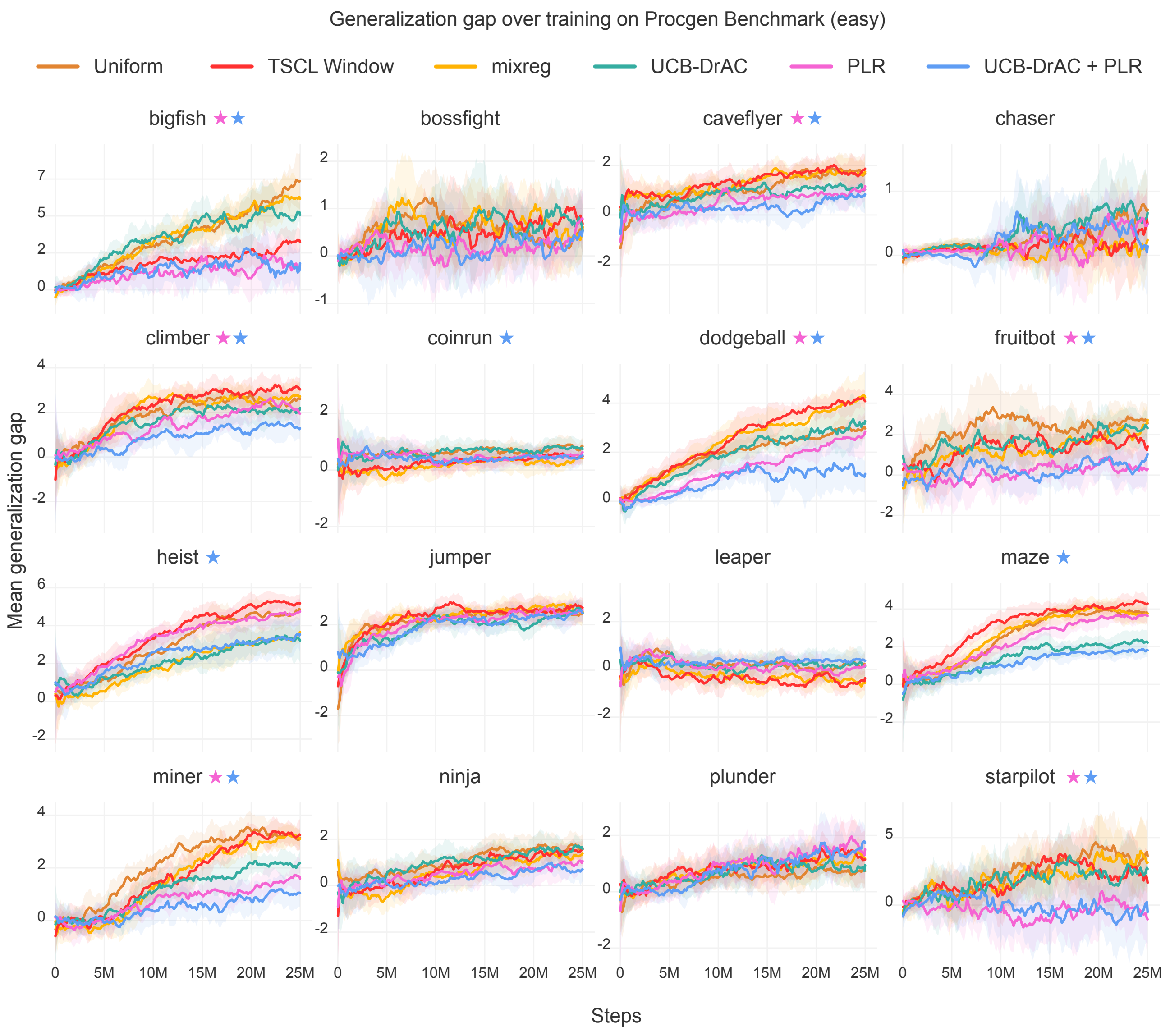

Figure 15. Mean generalization gaps throughout training (10 runs) on each Procgen Benchmark game (easy). The shaded area indicates one standard deviation around the mean. A ★ indicates the method of matching color results in a statistically significant ($p < 0.05$) reduction in generalization gap compared 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.

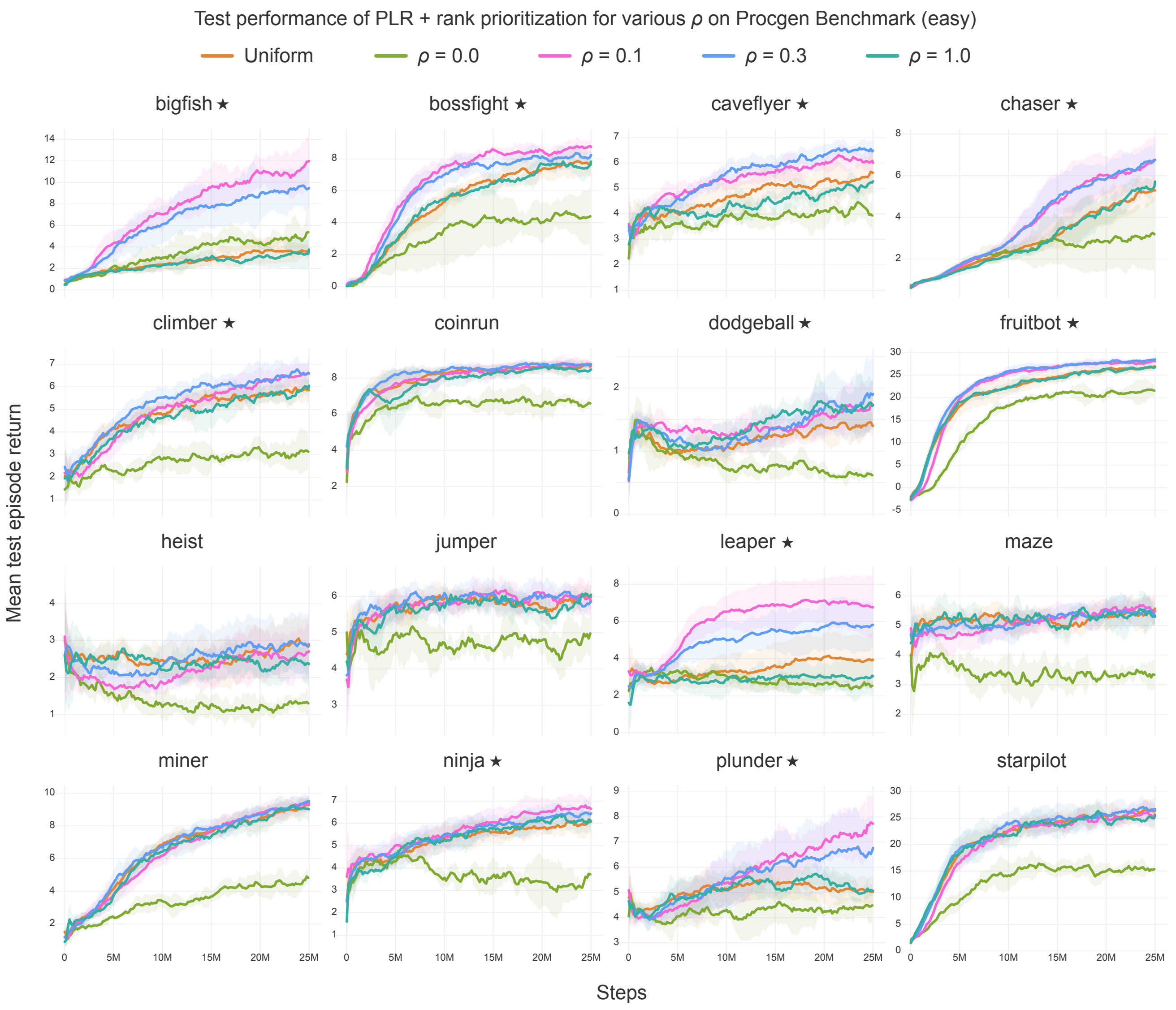

Figure 16. Mean test episode returns (10 runs) on the Procgen Benchmark (easy) for PLR with rank prioritization and $\beta = 0.1$ across a range of staleness coefficient values, ρ. The replay distribution must consider both the L1 value-loss and staleness values to realize improvements to generalization 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.

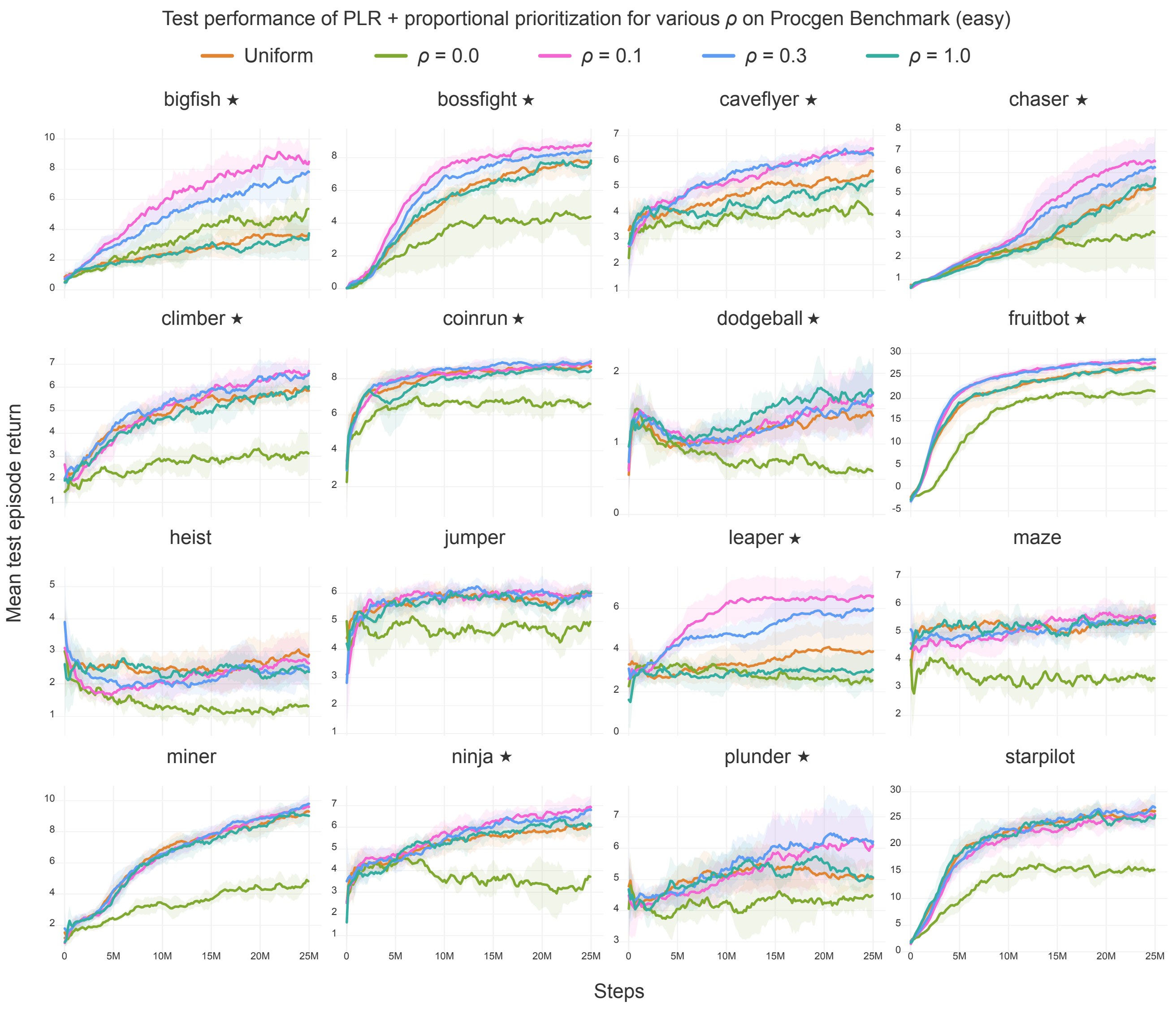

Figure 17. Mean test episode returns (10 runs) on the Procgen Benchmark (easy) for PLR with proportional prioritization and $\beta = 0.1$ across a range of values of ρ. As in the case of rank prioritization, the replay distribution must consider both the L1 value loss score and staleness values in order 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.

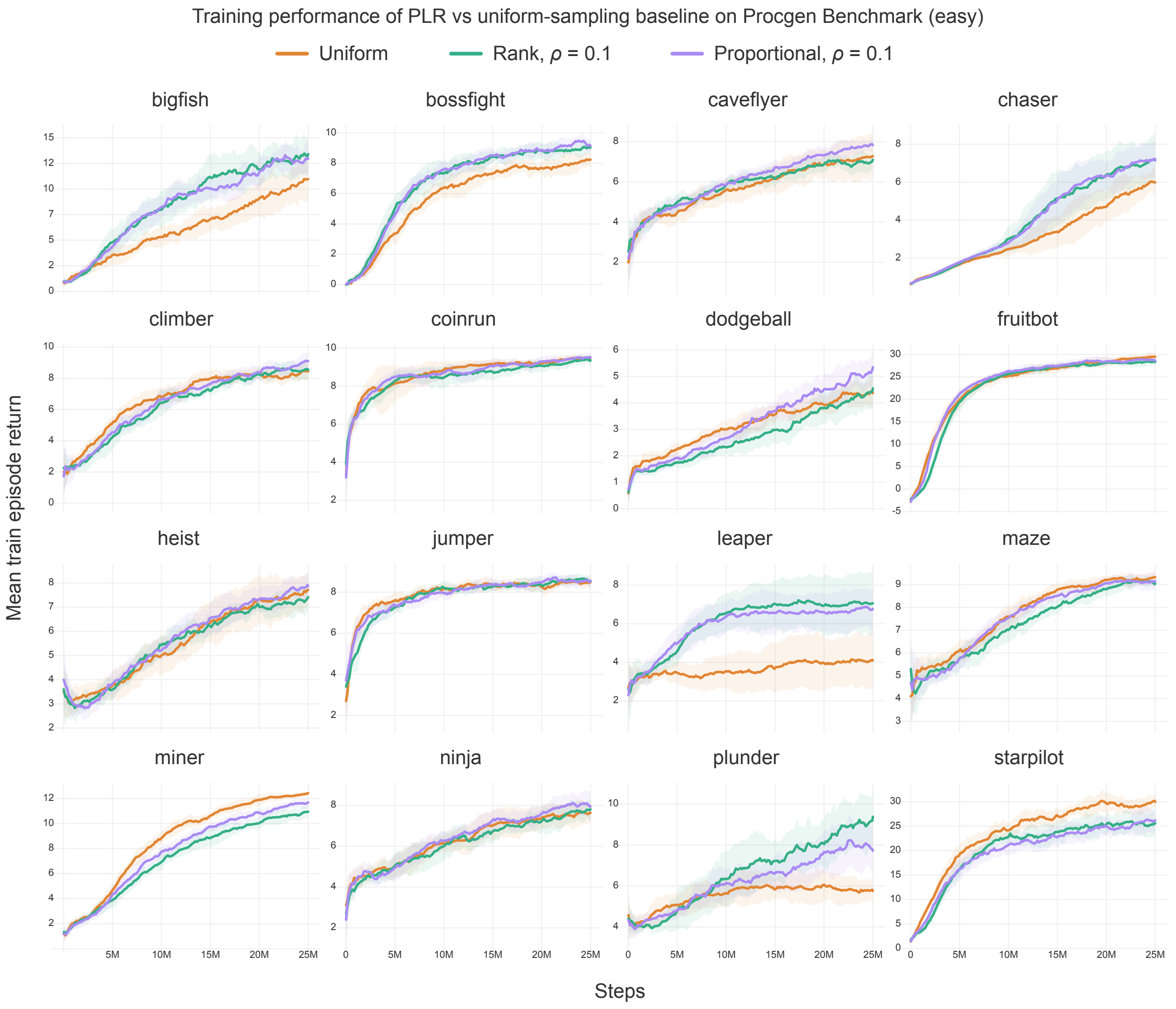

Figure 18. Mean training episode returns (10 runs) on the Procgen Benchmark for (easy) PLR with $\beta = 0.1$, $\rho = 0.1$, and each of rank and proportional prioritization. On some games, PLR improves both training sample efficiency and generalization performance (e.g. BigFish and Chaser), while on 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
    $\tilde{S}_i \leftarrow (\mathbf{score}(\tau, \pi_\theta), |\tau|)$          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
```