

A PPO

Proximal Policy Optimization (PPO) Schulman et al. (2017) is an actor-critic RL algorithm that learns a policy π_θ and a value function V_θ with the goal of finding an optimal policy for a given MDP. PPO alternates between sampling data through interaction with the environment and optimizing an objective function using stochastic gradient ascent. At each iteration, PPO maximizes the following objective:

$$J_{\text{PPO}} = J_\pi - \alpha_1 J_V + \alpha_2 S_{\pi_\theta}, \quad (7)$$

where α_1, α_2 are weights for the different loss terms, S_{π_θ} is the entropy bonus for aiding exploration, J_V is the value function loss defined as

$$J_V = (V_\theta(s) - V_t^{\text{target}})^2.$$

The policy objective term J_π is based on the policy gradient objective which can be estimated using importance sampling in off-policy settings (*i.e.* when the policy used for collecting data is different from the policy we want to optimize):

$$J_{PG}(\theta) = \sum_{a \in \mathcal{A}} \pi_\theta(a|s) \hat{A}_{\theta_{\text{old}}}(s, a) = \mathbb{E}_{a \sim \pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \hat{A}_{\theta_{\text{old}}}(s, a) \right], \quad (8)$$

where $\hat{A}(\cdot)$ is an estimate of the advantage function, θ_{old} are the policy parameters before the update, $\pi_{\theta_{\text{old}}}$ is the behavior policy used to collect trajectories (*i.e.* that generates the training distribution of states and actions), and π_θ is the policy we want to optimize (*i.e.* that generates the true distribution of states and actions).

This objective can also be written as

$$J_{PG}(\theta) = \mathbb{E}_{a \sim \pi_{\theta_{\text{old}}}} \left[r(\theta) \hat{A}_{\theta_{\text{old}}}(s, a) \right], \quad (9)$$

where

$$r_\theta = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$$

is the importance weight for estimating the advantage function.

PPO is inspired by TRPO (Schulman et al., 2015), which constrains the update so that the policy does not change too much in one step. This significantly improves training stability and leads to better results than vanilla policy gradient algorithms. TRPO achieves this by minimizing the KL divergence between the old (*i.e.* before an update) and the new (*i.e.* after an update) policy. PPO implements the constraint in a simpler way by using a clipped surrogate objective instead of the more complicated TRPO objective. More specifically, PPO imposes the constraint by forcing $r(\theta)$ to stay within a small interval around 1, precisely $[1 - \epsilon, 1 + \epsilon]$, where ϵ is a hyperparameter. The policy objective term from equation (7) becomes

$$J_\pi = \mathbb{E}_\pi \left[\min \left(r_\theta \hat{A}, \text{clip}(r_\theta, 1 - \epsilon, 1 + \epsilon) \hat{A} \right) \right],$$

where $\hat{A} = \hat{A}_{\theta_{\text{old}}}(s, a)$ for brevity. The function $\text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon)$ clips the ratio to be no more than $1 + \epsilon$ and no less than $1 - \epsilon$. The objective function of PPO takes the minimum one between the original value and the clipped version so that agents are discouraged from increasing the policy update to extremes for better rewards.

Note that the use of the Adam optimizer (Kingma & Ba, 2015) allows loss components of different magnitudes so we can use G_π and G_V from equations (3) and (4) to be used as part of the DrAC objective in equation (5) with the same loss coefficient α_r . This alleviates the burden of hyperparameter search and means that DrAC only introduces a single extra parameter α_r .

Table 3: List of hyperparameters used to obtain the results in this paper.

Hyperparameter	Value
γ	0.999
λ	0.95
# timesteps per rollout	256
# epochs per rollout	3
# minibatches per epoch	8
entropy bonus	0.01
clip range	0.2
reward normalization	yes
learning rate	5e-4
# workers	1
# environments per worker	64
# total timesteps	25M
optimizer	Adam
LSTM	no
frame stack	no
α_r	0.1
c	0.1
K	10

B CYCLE-CONSISTENCY

Here is a description of the cycle-consistency metric proposed by Aytar et al. (2018) and also used in Lee et al. (2020) for analyzing the learned representations of RL agents. Given two trajectories V and U , $v_i \in V$ first locates its nearest neighbor in the other trajectory $u_j = \operatorname{argmin}_{u \in U} \|h(v_i) - h(u)\|^2$, where $h(\cdot)$ denotes the output of the penultimate layer of trained agents. Then, the nearest neighbor of $u_j \in V$ is located, *i.e.*, $v_k = \operatorname{argmin}_{v \in V} \|h(u_j) - h(v)\|_2$, and v_i is defined as cycle-consistent if $|i - k| \leq 1$, *i.e.*, it can return to the original point. Note that this cycle-consistency implies that two trajectories are accurately aligned in the hidden space. Similar to Aytar et al. (2018), we also evaluate the three-way cycle-consistency by measuring whether v_i remains cycle-consistent along both paths, $V \rightarrow U \rightarrow J \rightarrow V$ and $V \rightarrow J \rightarrow U \rightarrow V$, where J is the third trajectory.

C HYPERPARAMETERS

We use Kostrikov (2018)’s implementation of PPO (Schulman et al., 2017), on top of which all our methods are build. The agent is parameterized by the ResNet architecture from (Espeholt et al., 2018) which was used to obtain the best results in Cobbe et al. (2019). Unless otherwise noted, we use the best hyperparameters found in Cobbe et al. (2019) for the easy mode of Procgen (*i.e.* same experimental setup as the one used here), namely:

For DrAC, we did a grid search for the regularization coefficient $\alpha_r \in [0.0001, 0.01, 0.05, 0.1, 0.5, 1.0]$ used in equation (5) and found that the best value is $\alpha_r = 0.1$, which was used to produce all the results in this paper.

For UCB-DrAC, we did grid searches for the exploration coefficient $c \in [0.0, 0.1, 0.5, 1.0, 5.0]$ and the size of the sliding window used to compute the Q-values $K \in [10, 50, 100]$. We found that the best values are $c = 0.1$ and $K = 10$, which were used to obtain the results shown here.

For RL2-DrAC, we performed a hyperparameter search for the dimension of recurrent hidden state $h \in [16, 32, 64]$, for the learning rate $l \in [3e - 4, 5e - 4, 7e - 4]$, and for the entropy coefficient $e \in [1e - 4, 1e - 3, 1e - 2]$ and found $h = 32$, $l = 5e - 4$, and $e = 1e - 3$ to work best. We used Adam with $\epsilon = 1e - 5$ as the optimizer.

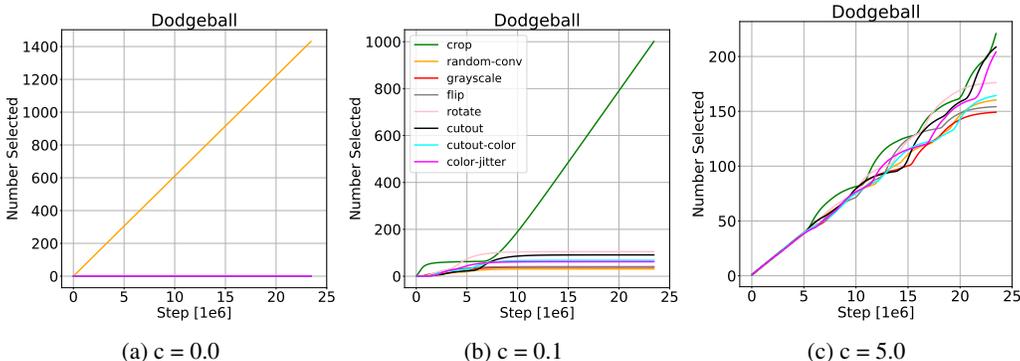


Figure 4: Behavior of UCB for different values of its exploration coefficient c on Dodgeball. When c is too small, UCB might converge to a suboptimal augmentation. On the other hand, when c is too large, UCB might take too long to converge.

For Meta-DrAC, the convolutional network whose weights we meta-learn consists of a single convolutional layer with 3 input and 3 output channels, kernel size 3, stride 1 and 0 padding. At each epoch, we perform one meta-update where we unroll the inner optimizer using the training set and compute the meta-test return on the validation set. We did the same hyperparameter grid searches as for RL2-DrAC and found that the best values were $l = 7e - 4$ and $e = 1e - 2$ in this case. The buffer of experience (collected before each PPO update) was split into 90% for meta-training and 10% for meta-testing.

For Rand-FM Lee et al. (2020) we use the recommended hyperparameters in the authors’ released implementation, which were the best values for CoinRun (Cobbe et al., 2018), one of the Procgen games used for evaluation in (Lee et al., 2020).

For IBAC-SNI Igl et al. (2019) we also use the authors’ open sourced implementation. We use the parameters corresponding to IBAC-SNI $\lambda = .5$. We use weight regularization with $l_2 = .0001$, data augmentation turned on, and a value of $\beta = .0001$ which turns on the variational information bottleneck, and selective noise injection turned on. This corresponds to the best version of this approach, as found by the authors after evaluating it on CoinRun (Cobbe et al., 2018). While IBAC-SNI outperforms the other methods on maze-like games such as heist, maze, and miner, it is still significantly worse than our approach on the entire Procgen benchmark.

For both baselines, Rand-FM and IBAC-SNI, we use the same experimental setup for training and testing as the one used for our methods. Hence, we train them for 25M frames on the easy mode of each Procgen game, using (the same) 200 levels for training and the rest of the levels for testing.

We use the Adam (Kingma & Ba, 2015) optimizer for all our experiments. Note that by using Adam, we do not need separate coefficients for the policy and value regularization terms (since Adam rescales gradients for each loss component accordingly).

D ANALYSIS OF UCB’S BEHAVIOR

In Figure 3, we show the behavior of UCB during training, along with train and test performance on the respective environments. In the case of Ninja, UCB converges to always selecting the best augmentation only after 15M training steps. This is because the augmentations have similar effects on the agent early in training, so it takes longer to find the best augmentation from the given set. In contrast, on Dodgeball, UCB finds the most effective augmentation much earlier in training because there is a significant difference between the effect of various augmentations. Early discovery of an effective augmentation leads to significant improvements over PPO, for both train and test environments.

Another important factor is the exploration coefficient used by UCB (see equation (6)) to balance the exploration and exploitation of different augmentations. Figure 4 compares UCB’s behavior for different values of the exploration coefficient. Note that if the coefficient is 0, UCB always selects the

augmentation with the largest Q-value. This can sometimes lead to UCB converging on a suboptimal augmentation due to the lack of exploration. However, if the exploration term of equation (6) is too large relative to the differences in the Q-values among various augmentations, UCB might take too long to converge. In our experiments, we found that an exploration coefficient of 0.1 results in a good exploration-exploitation balance and works well across all Progen games.

E PROCGEN BENCHMARK

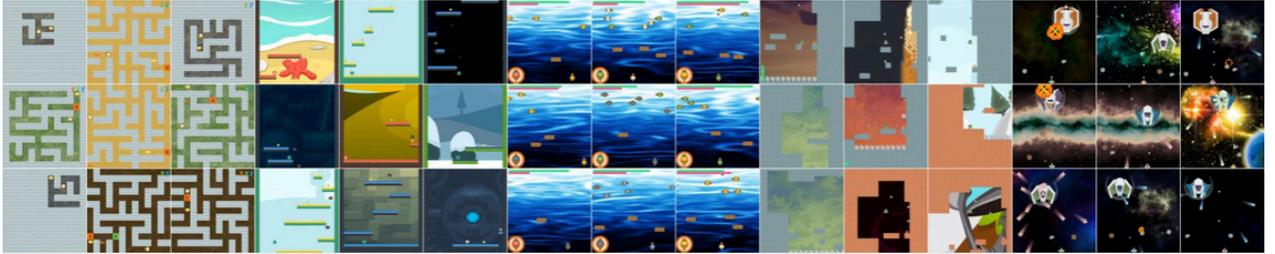


Figure 5: Screenshots of multiple procedurally-generated levels from five Progen environments: Maze, Climber, Plunder, Ninja, and BossFight (from left to right).

F BEST AUGMENTATIONS

Table 4: Best augmentation type for each game, as evaluated on the test environments.

Game	BigFish	StarPilot	FruitBot	BossFight	Ninja	Plunder	CaveFlyer	CoinRun
Best Augmentation	crop	crop	crop	flip	color-jitter	crop	rotate	random-conv

Table 5: Best augmentation type for each game, as evaluated on the test environments.

Game	Jumper	Chaser	Climber	Dodgeball	Heist	Leaper	Maze	Miner
Best Augmentation	random-conv	crop	color-jitter	crop	crop	crop	crop	color-jitter

G BREAKDOWN OF PROCGEN SCORES

Table 6: Procgen scores on train levels after training on 25M environment steps. The mean and standard deviation are computed using 10 runs. The best augmentation for each game is used when computing the results for DrAC and RAD.

Game	PPO	Rand + FM	IBAC-SNI	DrAC	RAD	UCB-DrAC	RL2-DrAC	Meta-DrAC
BigFish	8.9 ± 1.5	6.0 ± 0.9	19.1 ± 0.8	13.1 ± 2.2	13.2 ± 2.8	13.2 ± 2.2	10.1 ± 1.9	9.28 ± 1.9
StarPilot	29.8 ± 2.3	26.3 ± 0.8	26.7 ± 0.7	38.0 ± 3.1	36.5 ± 3.9	35.3 ± 2.2	30.6 ± 2.6	30.5 ± 3.9
FruitBot	29.1 ± 1.1	29.2 ± 0.7	29.4 ± 0.8	29.4 ± 1.0	26.1 ± 3.0	29.5 ± 1.2	29.2 ± 1.0	29.4 ± 1.1
BossFight	8.5 ± 0.7	5.6 ± 0.7	7.9 ± 0.7	8.2 ± 1.0	8.1 ± 1.1	8.2 ± 0.8	8.4 ± 0.8	7.9 ± 0.5
Ninja	7.4 ± 0.7	7.2 ± 0.6	8.3 ± 0.8	8.8 ± 0.5	8.9 ± 0.9	8.5 ± 0.3	8.1 ± 0.6	7.8 ± 0.4
Plunder	6.0 ± 0.5	5.5 ± 0.7	6.0 ± 0.6	9.9 ± 1.3	8.4 ± 1.5	11.1 ± 1.6	5.3 ± 0.5	6.5 ± 0.5
CaveFlyer	6.8 ± 0.6	6.5 ± 0.5	6.2 ± 0.5	8.2 ± 0.7	6.0 ± 0.8	5.7 ± 0.6	5.3 ± 0.8	6.5 ± 0.7
CoinRun	9.3 ± 0.3	9.6 ± 0.6	9.6 ± 0.4	9.7 ± 0.2	9.6 ± 0.4	9.5 ± 0.3	9.1 ± 0.3	9.4 ± 0.2
Jumper	8.3 ± 0.4	8.9 ± 0.4	8.5 ± 0.6	9.1 ± 0.4	8.6 ± 0.4	8.1 ± 0.7	8.6 ± 0.4	8.4 ± 0.5
Chaser	4.9 ± 0.5	2.8 ± 0.7	3.1 ± 0.8	7.1 ± 0.5	6.4 ± 1.0	7.6 ± 1.0	4.5 ± 0.7	5.5 ± 0.8
Climber	8.4 ± 0.8	7.5 ± 0.8	7.1 ± 0.7	9.9 ± 0.8	9.3 ± 1.1	9.0 ± 0.4	7.9 ± 0.9	8.5 ± 0.5
Dodgeball	4.2 ± 0.5	4.3 ± 0.3	9.4 ± 0.6	7.5 ± 1.0	5.0 ± 0.7	8.3 ± 0.9	6.3 ± 1.1	4.8 ± 0.6
Heist	7.1 ± 0.5	6.0 ± 0.5	4.8 ± 0.7	6.8 ± 0.7	6.2 ± 0.9	6.9 ± 0.4	5.6 ± 0.8	6.6 ± 0.6
Leaper	5.5 ± 0.4	3.2 ± 0.7	2.7 ± 0.4	5.0 ± 0.7	4.9 ± 0.9	5.3 ± 0.5	2.7 ± 0.6	3.7 ± 0.6
Maze	9.1 ± 0.3	8.9 ± 0.6	8.2 ± 0.8	8.3 ± 0.7	8.4 ± 0.7	8.7 ± 0.6	7.0 ± 0.7	9.2 ± 0.2
Miner	12.2 ± 0.3	11.7 ± 0.8	8.5 ± 0.7	12.5 ± 0.3	12.6 ± 1.0	12.5 ± 0.2	10.9 ± 0.5	12.4 ± 0.3

Table 7: Procgen scores on test levels after training on 25M environment steps. The mean and standard deviation are computed using 10 runs. The best augmentation for each game is used when computing the results for DrAC and RAD.

Game	PPO	Rand + FM	IBAC-SNI	DrAC	RAD	UCB-DrAC	RL2-DrAC	Meta-DrAC
BigFish	4.0 ± 1.2	0.6 ± 0.8	0.8 ± 0.9	8.7 ± 1.4	9.9 ± 1.7	9.7 ± 1.0	6.0 ± 0.5	3.3 ± 0.6
StarPilot	24.7 ± 3.4	8.8 ± 0.7	4.9 ± 0.8	29.5 ± 5.4	33.4 ± 5.1	30.2 ± 2.8	29.4 ± 2.0	26.6 ± 2.8
FruitBot	26.7 ± 0.8	24.5 ± 0.7	24.7 ± 0.8	28.2 ± 0.8	27.3 ± 1.8	28.3 ± 0.9	27.5 ± 1.6	27.4 ± 0.8
BossFight	7.7 ± 1.0	1.7 ± 0.9	1.0 ± 0.7	7.5 ± 0.8	7.9 ± 0.6	8.3 ± 0.8	7.6 ± 0.9	7.7 ± 0.7
Ninja	5.9 ± 0.7	6.1 ± 0.8	9.2 ± 0.6	7.0 ± 0.4	6.9 ± 0.8	6.9 ± 0.6	6.2 ± 0.5	5.9 ± 0.7
Plunder	5.0 ± 0.5	3.0 ± 0.6	2.1 ± 0.8	9.5 ± 1.0	8.5 ± 1.2	8.9 ± 1.0	4.6 ± 0.3	5.6 ± 0.4
CaveFlyer	5.1 ± 0.9	5.4 ± 0.8	8.0 ± 0.8	6.3 ± 0.8	5.1 ± 0.6	5.3 ± 0.9	4.1 ± 0.9	5.5 ± 0.4
CoinRun	8.5 ± 0.5	9.3 ± 0.4	8.7 ± 0.6	8.8 ± 0.	9.0 ± 0.8	8.5 ± 0.6	8.3 ± 0.5	8.6 ± 0.5
Jumper	5.8 ± 0.5	5.3 ± 0.6	3.6 ± 0.6	6.6 ± 0.4	6.5 ± 0.6	6.4 ± 0.6	6.5 ± 0.5	5.8 ± 0.7
Chaser	5.0 ± 0.8	1.4 ± 0.7	1.3 ± 0.5	5.7 ± 0.6	5.9 ± 1.0	6.7 ± 0.6	3.8 ± 0.5	5.1 ± 0.6
Climber	5.7 ± 0.8	5.3 ± 0.7	3.3 ± 0.6	7.1 ± 0.7	6.9 ± 0.8	6.5 ± 0.8	6.3 ± 0.5	6.6 ± 0.6
Dodgeball	11.7 ± 0.3	0.5 ± 0.4	1.4 ± 0.4	4.3 ± 0.8	2.8 ± 0.7	4.7 ± 0.7	3.0 ± 0.8	1.9 ± 0.5
Heist	2.4 ± 0.5	2.4 ± 0.6	9.8 ± 0.6	4.0 ± 0.8	4.1 ± 1.0	4.0 ± 0.7	2.4 ± 0.4	2.0 ± 0.6
Leaper	4.9 ± 0.7	6.2 ± 0.5	6.8 ± 0.6	5.3 ± 1.1	4.3 ± 1.0	5.0 ± 0.3	2.8 ± 0.7	3.3 ± 0.4
Maze	5.7 ± 0.6	8.0 ± 0.7	10.0 ± 0.7	6.6 ± 0.8	6.1 ± 1.0	6.3 ± 0.6	5.6 ± 0.3	5.2 ± 0.6
Miner	8.5 ± 0.5	7.7 ± 0.6	8.0 ± 0.6	9.8 ± 0.6	9.4 ± 1.2	9.7 ± 0.7	8.0 ± 0.4	9.2 ± 0.7

H AUTOMATIC DATA AUGMENTATION

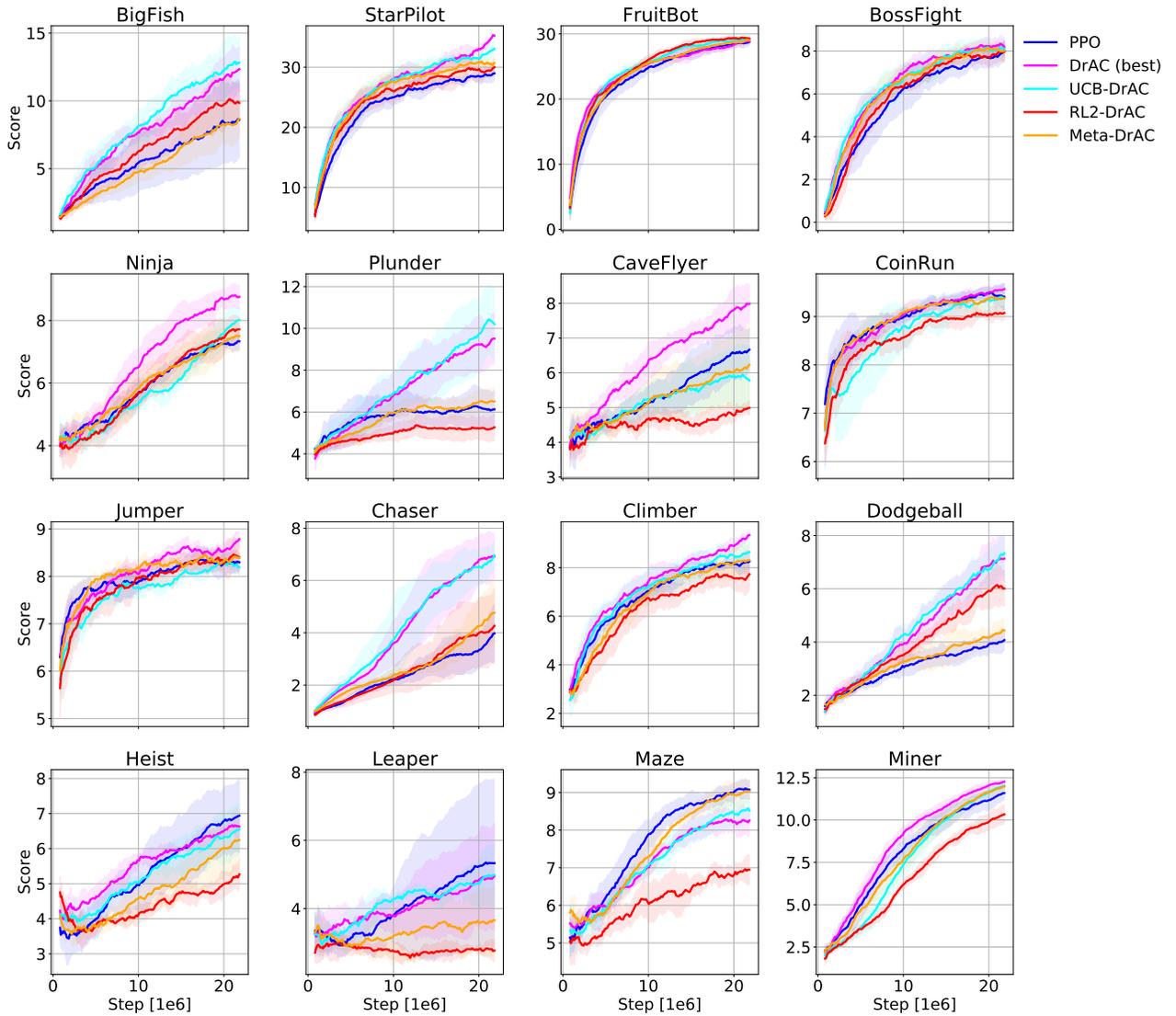


Figure 6: Train performance of various approaches that automatically select an augmentation, namely UCB-DrAC, RL2-DrAC, and Meta-DrAC. The mean and standard deviation are computed using 10 runs.

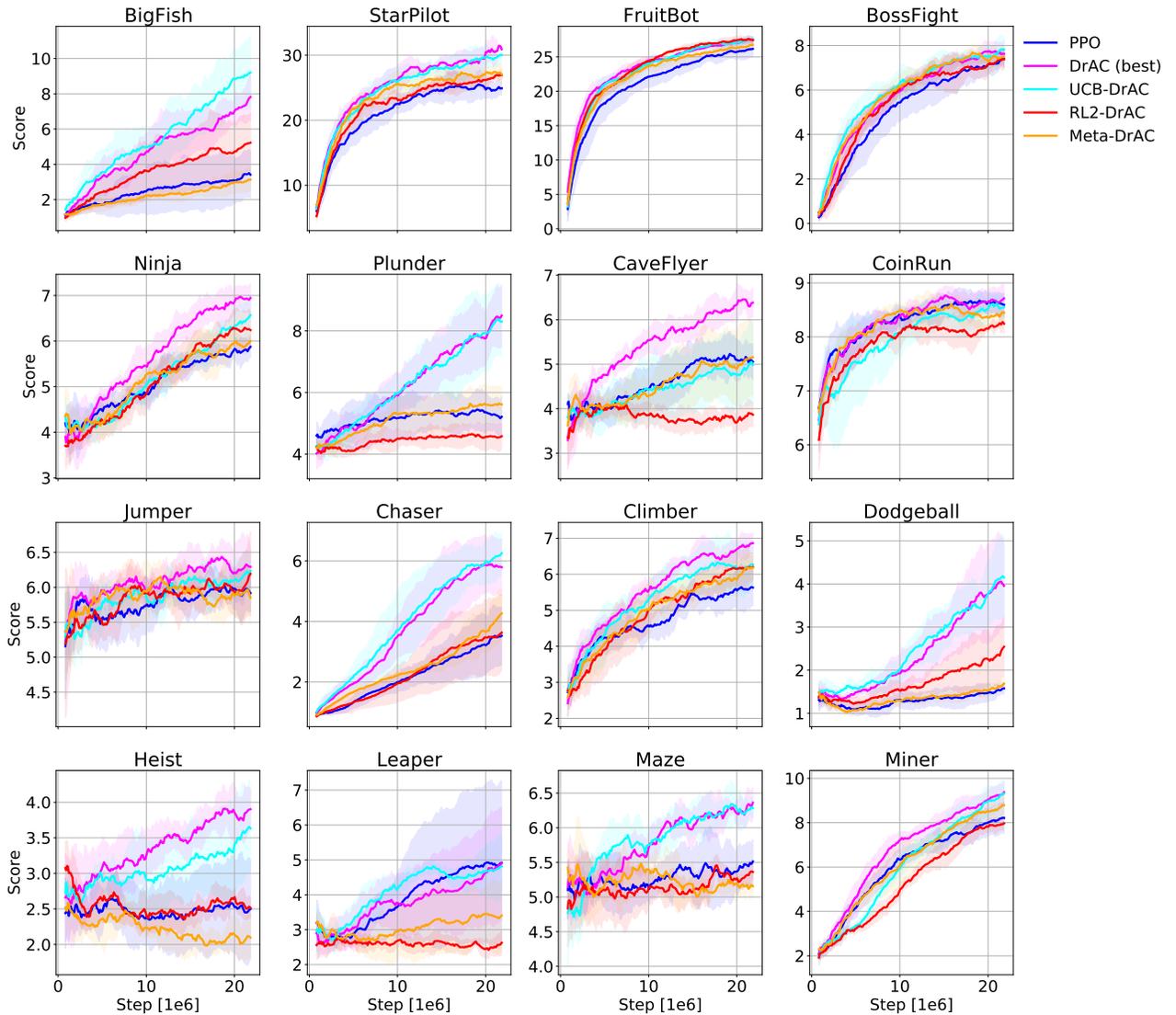


Figure 7: Test performance of various approaches that automatically select an augmentation, namely UCB-DrAC, RL2-DrAC, and Meta-DrAC. The mean and standard deviation are computed using 10 runs.