A. PPO

Proximal Policy Optimization (PPO) (Schulman et al., 2017) is an actor-critic 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:

$$J_{\text{PPO}} = J_\pi - \alpha_1 J_V + \alpha_2 S_{\pi_\theta}, \tag{5}$$

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

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

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):

$$J_{PG}(\theta) = \sum_{a \in \mathcal{A}} \pi_\theta(a|s) \hat{A}_{\theta_{\text{old}}}(s,a) = \mathbb{E}_{a \sim \pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \hat{A}_{\theta_{\text{old}}}(s,a) \right], \tag{6}$$

where $\hat{A}(\cdot)$ is an estimate of the advantage function, θ_{old} are the policy parameters before the update, $\pi_{\theta_{old}}$ 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

$$J_{PG}(\theta) = \mathbb{E}_{a \sim \pi_{\theta_{\text{old}}}} \left[r(\theta) \hat{A}_{\theta_{\text{old}}}(s,a) \right], \tag{7}$$

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 (5) becomes

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

where $\hat{A} = \hat{A}_{\theta_{\text{old}}}(s,a)$ for brevity. The function 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.

B. DAAC and IDAAC

As described in the paper, DAAC and IDAAC alternate between optimizing the policy network and optimizing the value network. The value estimates are used to compute the advantage targets which are needed by both the policy gradient objective and the auxiliary loss based on predicting the advantage function. Since we use separate networks for learning the policy and value function, we can now use different numbers of epochs for updating the two networks. We use E_π epochs for every policy update and E_V epochs for every value update. Similar to Cobbe et al. (2020), we find that the value network allows for larger amounts of sample reuse than the policy network. This decoupling of the policy and value optimization allows us to also control the frequency of value updates relative to policy updates. Updating the value function less often can help with training stability since can result in lower variance gradients for the policy and advantage losses. We update the value network every N_π updates of the policy network. See algorithms 1 and 2 for the pseudocodes of DAAC and IDAAC, respectively.

Algorithm 1 DAAC: Decoupled Advantage Actor-Critic

Hyperparameters: Total number of updates N, replay buffer size T, number of epochs per policy update E_π, number of epochs per value update E_V, frequency of value updates N_π, weight for the advantage loss α_a, initial policy parameters θ, initial value parameters ϕ.
for $n = 1, \ldots, N$ **do**
  Collect $\mathcal{D} = \{(s_t, a_t, r_t, s_{t+1})\}_{t=1}^{T}$ using $\pi(\theta)$.
  Compute the value and advantage targets $\hat{V}_t$ and $\hat{A}_t$ for all states s_t
  for $i = 1, \ldots, E_\pi$ **do**
    $L_{\mathrm{A}}(\theta) = \hat{\mathbb{E}}_t \left[\left(A_\theta(s_t, a_t) - \hat{A}_t \right)^2 \right]$ ▷ Compute the Advantage Loss
    $J_{\mathrm{DAAC}}(\theta) = J_\pi(\theta) + \alpha_{\mathrm{s}} \mathrm{S}_\pi(\theta) - \alpha_{\mathrm{a}} L_{\mathrm{A}}(\theta)$ ▷ Compute the Policy Loss
    $\theta \leftarrow \arg\max_\theta J_{\mathrm{DAAC}}$ ▷ Update the Policy Network
  if $n \,\%\, N_\pi \;=\; 0$ **then**
    for $j = 1, \ldots, E_V$ **do**
      $L_{\mathrm{V}}(\phi) = \hat{\mathbb{E}}_t \left[\left(V_\phi(s_t) - \hat{V}_t \right)^2 \right]$ ▷ Compute the Value Loss
      $\phi \leftarrow \arg\min_\phi L_{\mathrm{V}}$ ▷ Update the Value Network

Algorithm 2 IDAAC: Invariant Decoupled Advantage Actor-Critic

Hyperparameters: Total number of updates N, replay buffer size T, number of epochs per policy update E_π, number of epochs per value update E_V, frequency of value updates N_π, weight for the invariance loss α_i, initial policy parameters θ, initial value parameters ϕ.
for $n = 1, \ldots, N$ **do**
  Collect $\mathcal{D} = \{(s_t, a_t, r_t, s_{t+1})\}_{t=1}^{T}$ using $\pi(\theta)$.
  Compute the value and advantage targets $\hat{V}_t$ and $\hat{A}_t$ for all states s_t
  for $i = 1, \ldots, E_\pi$ **do**
    $L_{\mathrm{A}}(\theta) = \hat{\mathbb{E}}_t \left[\left(A_\theta(s_t, a_t) - \hat{A}_t \right)^2 \right]$ ▷ Compute the Advantage Loss
    $L_{\mathrm{E}}(\theta) = -\frac{1}{2}\log\left[\mathrm{D}_\psi\left(\mathrm{E}_\theta(s_i), \mathrm{E}_\theta(s_j)\right)\right] - \frac{1}{2}\log\left[1 - \mathrm{D}_\psi\left(\mathrm{E}_\theta(s_i), \mathrm{E}_\theta(s_j)\right)\right]$ ▷ Compute the Encoder Loss
    $J_{\mathrm{IDAAC}}(\theta, \phi, \psi) = J_\pi(\theta) + \alpha_{\mathrm{s}} \mathrm{S}_\pi(\theta) - \alpha_{\mathrm{a}} L_{\mathrm{A}}(\theta) - \alpha_i L_{\mathrm{E}}(\theta)$ ▷ Compute the Policy Loss
    $L_{\mathrm{D}}(\psi) = -\log\left[\mathrm{D}_\psi\left(\mathrm{E}_\theta(s_i), \mathrm{E}_\theta(s_j)\right)\right] - \log\left[1 - \mathrm{D}_\psi\left(\mathrm{E}_\theta(s_i), \mathrm{E}_\theta(s_j)\right)\right]$ ▷ Compute the Discriminator Loss
    $\theta \leftarrow \arg\max_\theta J_{\mathrm{IDAAC}}$ ▷ Update the Policy Network
    $\psi \leftarrow \arg\min_\psi L_{\mathrm{D}}$ ▷ Update the Discriminator
  if $n \,\%\, N_\pi \;=\; 0$ **then**
    for $j = 1, \ldots, E_V$ **do**
      $L_{\mathrm{V}}(\phi) = \hat{\mathbb{E}}_t \left[\left(V_\phi(s_t) - \hat{V}_t \right)^2 \right]$ ▷ Compute the Value Loss
      $\phi \leftarrow \arg\min_\phi L_{\mathrm{V}}$ ▷ Update the Value Network

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

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) as found in Table 3:

We ran a hyperparameter search over the number of epochs used during each update of the policy network $E_\pi \in [1, 3, 6]$ the number epochs used during each update of the value network $E_V \in [1, 5, 9]$, the number of value updates after which we perform a policy update $N_\pi \in [1, 8, 32]$, the weight for the advantage loss $\alpha_a \in [0.05, 0.1, 0.15, 0.2, 0.25, 0.3]$, and the weight for the instance-invariant (adversarial) loss $\alpha_i \in [0.001, 0.005, 0.01, 0.05, 0.1, 0.2]$. We found $E_\pi = 1, E_V = 9, N_\pi = 1, \alpha_a = 0.25, \alpha_i = 0.001$ to be the best hyperparameters overall, so we used these values to obtain the results reported in the paper. For all our experiments we use the Adam (Kingma & Ba, 2015) optimizer.

For all baselines, 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 entire distribution of levels for testing.

For Mixreg, PLR, UCB-DrAC, and PPG, we used the best hyperparameters reported by the authors, since all these methods use Procgen for evaluation and they performed extensive hyperparameter sweeps.

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).

D. Procgen Results

Figures 7 and 8 show the test and train performance for IDAAC, DAAC, PPG, UCB-DrAC, and PPO on all Procgen games. Our methods, IDAAC and DAAC demonstrate superior test performance, outperforming all other baselines on the majority of the games, while being comparable on most of the remaining ones.

Figures 9 and 10 show the test and train performance for PPO, DAAC, and two ablations, DVAC and AAC, on all Procgen games. On both train and test environments, our method is substantially better than all the ablations for most of the games and comparable on the remaining ones (*i.e.* CaveFlyer and Maze).

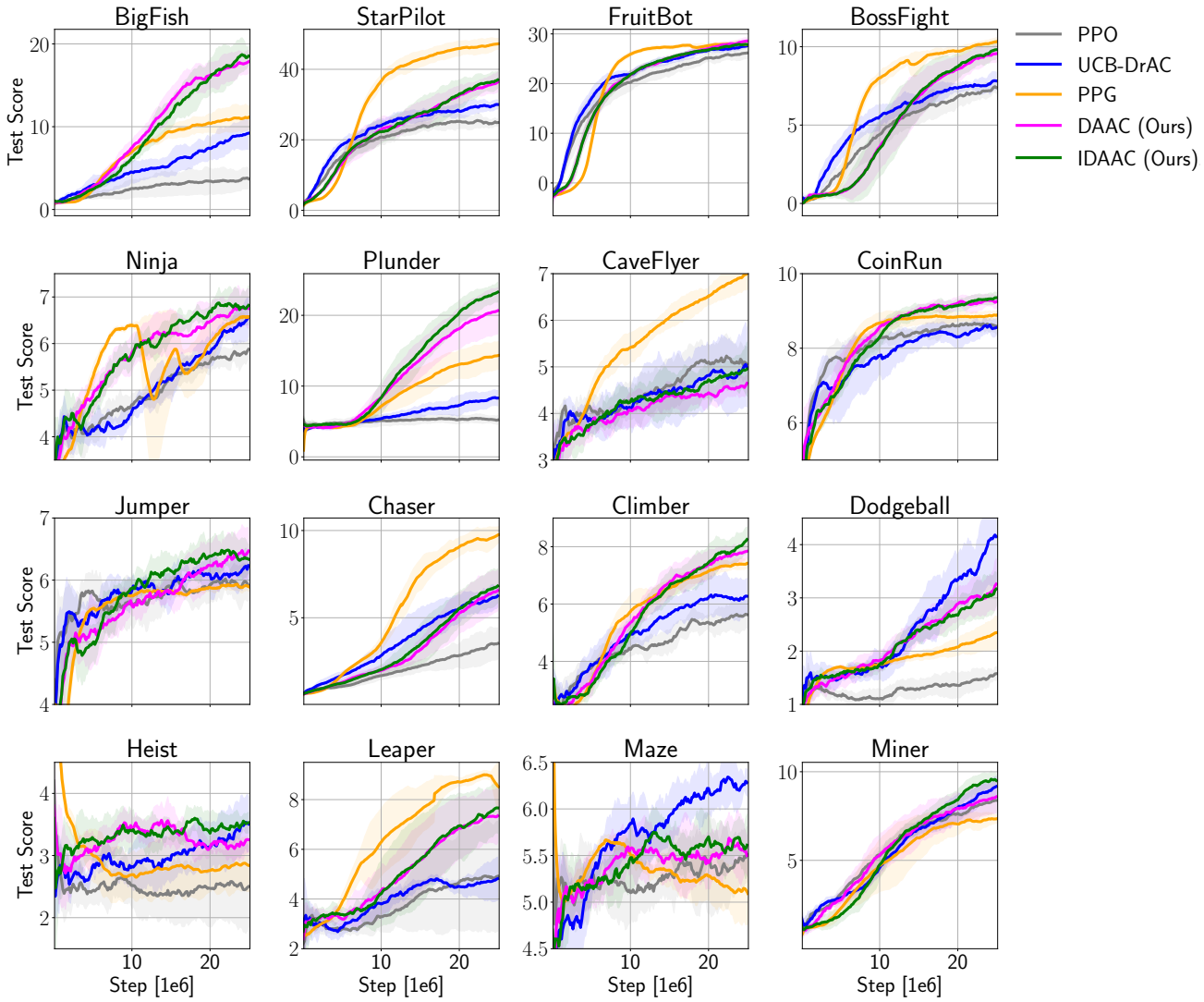


Figure 7. **Test Performance** of IDAAC, DAAC, PPG, UCB-DrAC, and PPO on all Procgen games. IDAAC outperforms the other methods on most games and is significantly better than PPO. The mean and standard deviation are computed over 10 runs with different seeds.

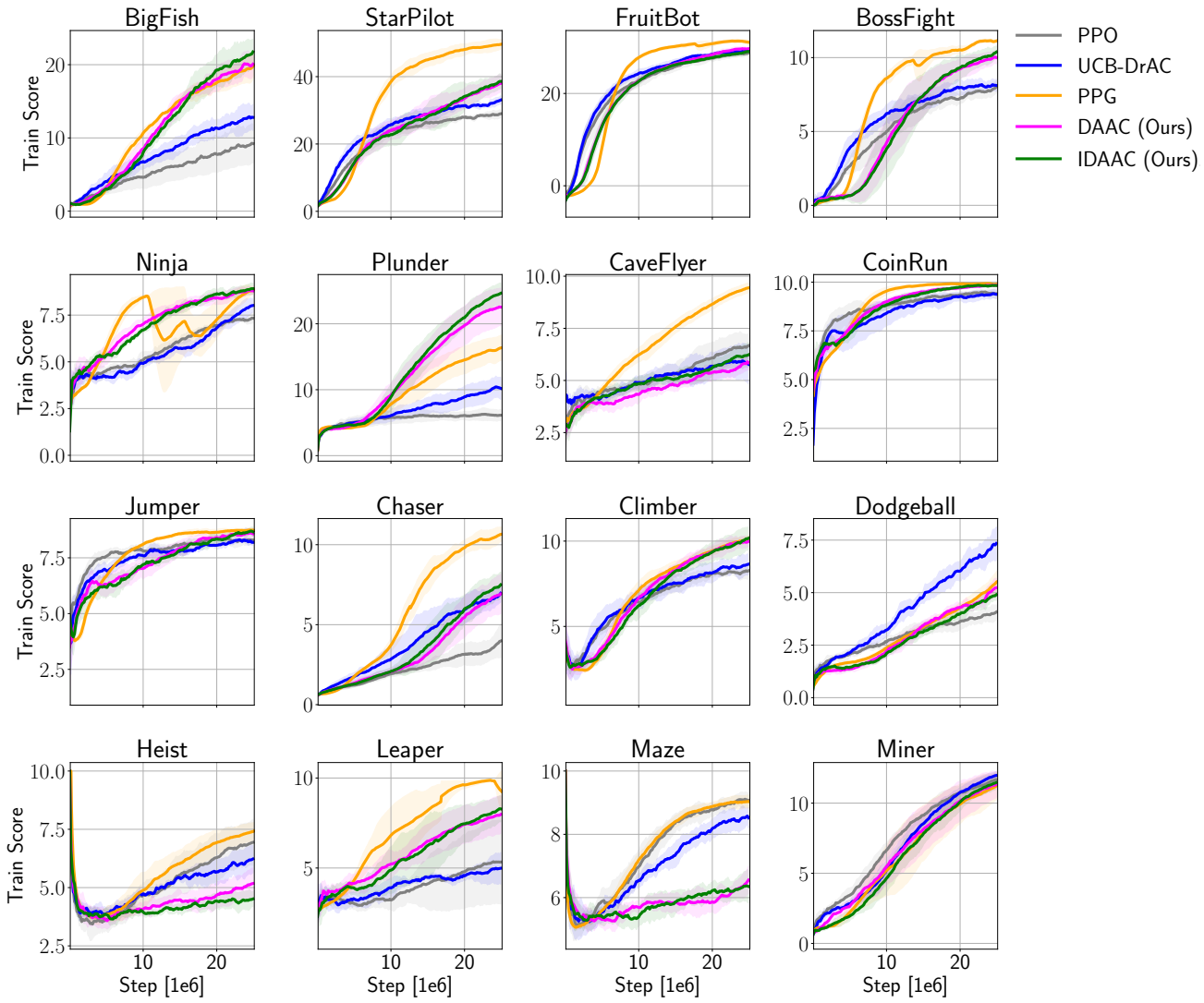


Figure 8. **Train Performance** of IDAAC, DAAC, PPG, UCB-DrAC, and PPO on all Procgen games. IDAAC outperforms the other methods on most games and is significantly better than PPO. The mean and standard deviation are computed over 10 runs with different seeds.

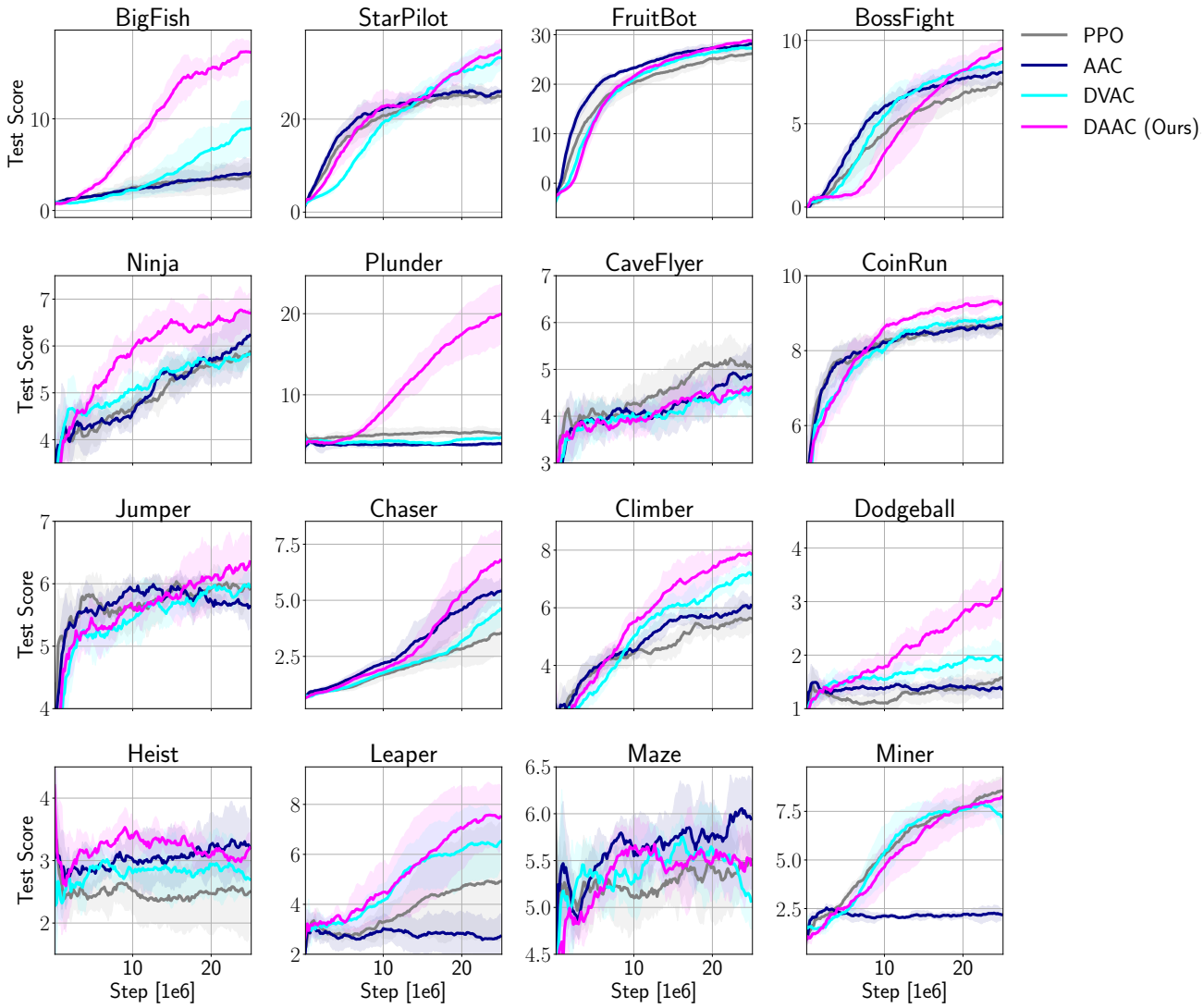


Figure 9. **Test Performance** of DAAC, DVAC, AAC, and PPO on all Procgen games. DVAC is an ablation of DAAC that replaces the advantage head of the policy network with a value head. AAC is similar to PPO but has an additional advantage head as part of the policy network. DAAC outperforms all these ablations, emphasizing the importance of all of its components. The mean and standard deviation are computed over 5 runs with different seeds.

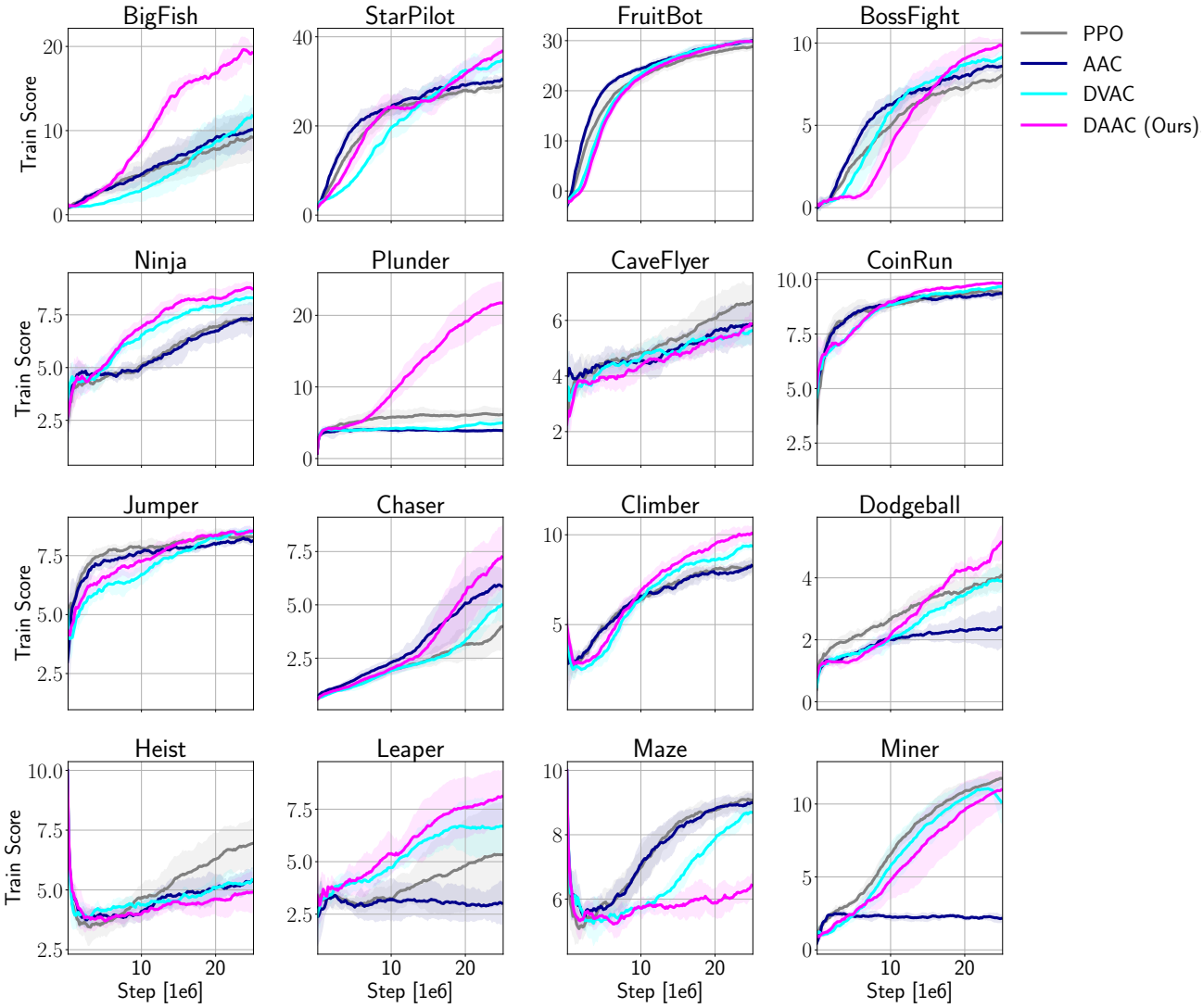


Figure 10. **Train Performance** of DAAC, DVAC, AAC, and PPO on all Procgen games. DVAC is an ablation of DAAC that replaces the advantage head of the policy network with a value head. AAC is similar to PPO but has an additional advantage head as part of the policy network. DAAC outperforms all these ablations, emphasizing the importance of all of its components. The mean and standard deviation are computed over 5 runs with different seeds.

Tables 4 and 5 show the final mean and standard deviation of the test and train scores obtained by our methods and the baselines on each of the 16 Procgen games, after 25M training steps. For brevity, we only include the strongest baselines in these tables. See Raileanu et al. (2020) for the scores obtained by Rand-FM and IBAC-SNI.

Table 4. Procgen scores on test levels after training on 25M environment steps. The mean and standard deviation are computed using 10 runs with different seeds.

Game	PPO	Mixreg	PLR	UCB-DrAC	PPG	DAAC (Ours)	IDAAC (Ours)
BigFish	3.7 ± 1.3	7.1 ± 1.6	10.9 ± 2.8	9.2 ± 2.0	11.2 ± 1.4	17.8 ± 1.4	$\mathbf{18.5 \pm 1.2}$
StarPilot	24.9 ± 1.0	32.4 ± 1.5	27.9 ± 4.4	30.0 ± 1.3	$\mathbf{47.2 \pm 1.6}$	36.4 ± 2.8	37.0 ± 2.3
FruitBot	26.2 ± 1.2	27.3 ± 0.8	28.0 ± 1.4	27.6 ± 0.4	27.8 ± 0.6	$\mathbf{28.6 \pm 0.6}$	27.9 ± 0.5
BossFight	7.4 ± 0.4	8.2 ± 0.7	8.9 ± 0.4	7.8 ± 0.6	$\mathbf{10.3 \pm 0.2}$	9.6 ± 0.5	9.8 ± 0.6
Ninja	5.9 ± 0.2	6.8 ± 0.5	$\mathbf{7.2 \pm 0.4}$	6.6 ± 0.4	6.6 ± 0.1	6.8 ± 0.4	6.8 ± 0.4
Plunder	5.2 ± 0.6	5.9 ± 0.5	8.7 ± 2.2	8.3 ± 1.1	14.3 ± 2.0	20.7 ± 3.3	$\mathbf{23.3 \pm 1.4}$
CaveFlyer	5.1 ± 0.4	6.1 ± 0.6	6.3 ± 0.5	5.0 ± 0.8	$\mathbf{7.0 \pm 0.4}$	4.6 ± 0.2	5.0 ± 0.6
CoinRun	8.6 ± 0.2	8.6 ± 0.3	8.8 ± 0.5	8.6 ± 0.2	8.9 ± 0.1	9.2 ± 0.2	$\mathbf{9.4 \pm 0.1}$
Jumper	5.9 ± 0.2	6.0 ± 0.3	5.8 ± 0.5	6.2 ± 0.3	5.9 ± 0.1	$\mathbf{6.5 \pm 0.4}$	6.3 ± 0.2
Chaser	3.5 ± 0.9	5.8 ± 1.1	6.9 ± 1.2	6.3 ± 0.6	$\mathbf{9.8 \pm 0.5}$	6.6 ± 1.2	6.8 ± 1.0
Climber	5.6 ± 0.5	6.9 ± 0.7	6.3 ± 0.8	6.3 ± 0.6	2.8 ± 0.4	7.8 ± 0.2	$\mathbf{8.3 \pm 0.4}$
Dodgeball	1.6 ± 0.1	1.7 ± 0.4	1.8 ± 0.5	$\mathbf{4.2 \pm 0.9}$	2.3 ± 0.3	3.3 ± 0.5	3.2 ± 0.3
Heist	2.5 ± 0.6	2.6 ± 0.4	2.9 ± 0.5	3.5 ± 0.4	2.8 ± 0.4	3.3 ± 0.2	$\mathbf{3.5 \pm 0.2}$
Leaper	4.9 ± 2.2	5.3 ± 1.1	6.8 ± 1.2	4.8 ± 0.9	$\mathbf{8.5 \pm 1.0}$	7.3 ± 1.1	7.7 ± 1.0
Maze	5.5 ± 0.3	5.2 ± 0.5	5.5 ± 0.8	$\mathbf{6.3 \pm 0.1}$	5.1 ± 0.3	5.5 ± 0.2	5.6 ± 0.3
Miner	8.4 ± 0.7	9.4 ± 0.4	9.6 ± 0.6	9.2 ± 0.6	7.4 ± 0.2	8.6 ± 0.9	$\mathbf{9.5 \pm 0.4}$

Table 5. Procgen scores on train levels after training on 25M environment steps. The mean and standard deviation are computed using 10 runs with different seeds.

Game	PPO	Mixreg	PLR	UCB-DrAC	PPG	DAAC (Ours)	IDAAC (Ours)
BigFish	9.2 ± 2.7	15.0 ± 1.3	7.8 ± 1.0	12.8 ± 1.8	19.9 ± 1.7	20.1 ± 1.6	$\mathbf{21.8 \pm 1.8}$
StarPilot	29.0 ± 1.1	28.7 ± 1.1	2.6 ± 0.3	33.1 ± 1.3	$\mathbf{49.6 \pm 2.1}$	38.0 ± 2.6	38.6 ± 2.2
FruitBot	28.8 ± 0.6	29.9 ± 0.5	15.9 ± 1.3	29.3 ± 0.5	$\mathbf{31.1 \pm 0.5}$	29.7 ± 0.4	29.1 ± 0.7
BossFight	8.0 ± 0.4	7.9 ± 0.8	8.7 ± 0.7	8.1 ± 0.4	$\mathbf{11.1 \pm 0.1}$	10.0 ± 0.4	10.4 ± 0.4
Ninja	7.3 ± 0.3	8.2 ± 0.4	5.4 ± 0.5	8.0 ± 0.4	8.9 ± 0.2	8.8 ± 0.2	$\mathbf{8.9 \pm 0.3}$
Plunder	6.1 ± 0.8	6.2 ± 0.3	4.1 ± 1.3	10.2 ± 1.76	16.4 ± 1.9	22.5 ± 2.8	$\mathbf{24.6 \pm 1.6}$
CaveFlyer	6.7 ± 0.6	6.2 ± 0.7	6.4 ± 0.1	5.8 ± 0.9	$\mathbf{9.5 \pm 0.2}$	5.8 ± 0.4	6.2 ± 0.6
CoinRun	9.4 ± 0.3	9.5 ± 0.2	5.4 ± 0.4	9.4 ± 0.2	$\mathbf{9.9 \pm 0.0}$	9.8 ± 0.0	9.8 ± 0.1
Jumper	8.3 ± 0.2	8.5 ± 0.4	3.6 ± 0.5	8.2 ± 0.1	8.7 ± 0.1	8.6 ± 0.3	$\mathbf{8.7 \pm 0.2}$
Chaser	4.1 ± 0.3	3.4 ± 0.9	6.3 ± 0.7	7.0 ± 0.6	$\mathbf{10.7 \pm 0.4}$	6.9 ± 1.2	7.5 ± 0.8
Climber	6.9 ± 1.0	7.5 ± 0.8	6.2 ± 0.8	8.6 ± 0.6	10.2 ± 0.2	10.0 ± 0.3	$\mathbf{10.2 \pm 0.7}$
Dodgeball	5.3 ± 2.3	9.1 ± 0.5	2.0 ± 1.1	$\mathbf{7.3 \pm 0.8}$	5.5 ± 0.5	5.2 ± 0.4	4.9 ± 0.3
Heist	7.1 ± 0.5	4.4 ± 0.3	1.2 ± 0.4	6.2 ± 0.6	$\mathbf{7.4 \pm 0.4}$	5.2 ± 0.7	4.5 ± 0.3
Leaper	5.5 ± 0.4	3.2 ± 1.2	6.4 ± 0.4	5.0 ± 0.9	$\mathbf{9.3 \pm 1.1}$	8.0 ± 1.1	8.3 ± 0.7
Maze	$\mathbf{9.1 \pm 0.2}$	8.7 ± 0.7	4.1 ± 0.5	8.5 ± 0.3	9.0 ± 0.2	6.6 ± 0.4	6.4 ± 0.5
Miner	11.7 ± 0.5	8.9 ± 0.9	9.7 ± 0.4	$\mathbf{12.0 \pm 0.3}$	11.3 ± 1.0	11.3 ± 0.9	11.5 ± 0.5

E. DeepMind Control Suite Experiments

For our DMC experiments, we followed the protocol proposed in Zhang et al. (2020b) to modify the tasks so that they contain natural and synthetic distractors in the background. For each DMC task, we split the generated environments (each with a different background video) into training (80%) and testing (20%). The results shown here correspond to the average return on the test environments, over the course of training. Note that this setting is slightly different from the one used in Raileanu et al. (2020) which shows results on all the generated environments, just like Zhang et al. (2020b). As Figure 11 shows, our methods outperform the baselines on these continuous control tasks.

In line with standard practice for this benchmark, we use 8 action repeats for Cartpole Swingup and 4 for Cartpole Balance and Ball In Cup. We also use 3 stacked frames as observations. To find the best hyperparameters, we ran a grid search over the learning rate in $[0.0001, 0.0003, 0.0007, 0.001]$, the number of minibatches in $[32, 8, 16, 64]$, the entropy coefficient in $[0.0, 0.01, 0.001, 0.0001]$, and the number of PPO epochs per update in $[3, 5, 10, 20]$. We found 10 ppo epochs, 0.0 entropy coefficient, 0.0003 learning rate, and 32 minibatches to work best across these environments. We use $\gamma = 0.99$, $\lambda = 0.95$ for the generalized advantage estimates, 2048 steps, 1 process, value loss coefficient 0.5, and linear rate decay over 1 million environment steps. Following this grid search, we used the best values found for all the methods. Any other hyperaparameters not mentioned here were set to the same values as the ones used for Procgen as described above. For UCB-DrAC, we used the best hyperparameters found by the corresponding authors. For PPG, we ran the same hyperparameter search as the one performed in the original paper for Procgen and found $N_\pi = 32$, $E_\pi = 1$, $E_V = 1$, $E_{aux} = 6$, and $\beta_{clone} = 1$ to be the best. Similarly, for DAAC and IDAAC, we ran the same hyperparameter search as for Procgen and found that $E_V = 9$, $N_\pi = 32$, $\alpha_a = 0.1$, and $\alpha_i = 0.1$ worked best across all environments.

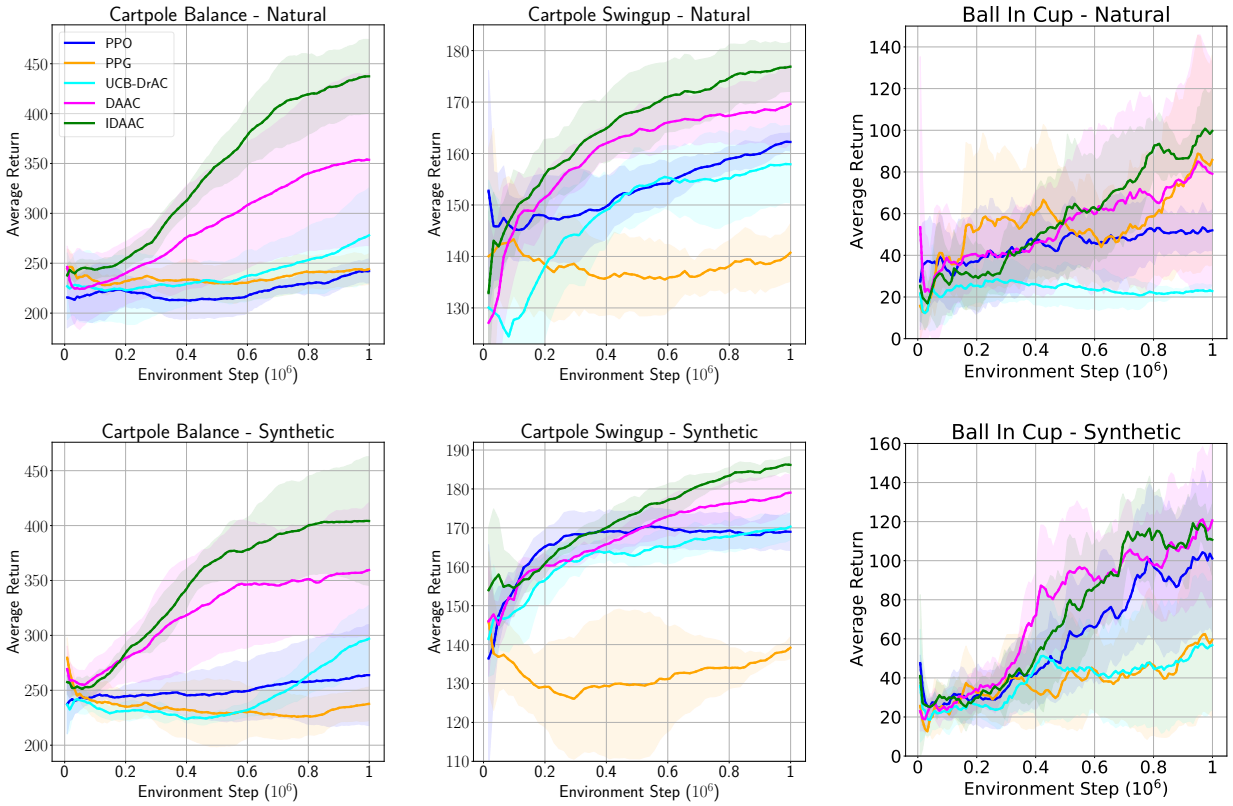


Figure 11. **Average return on three DMC tasks, Cartpole Balance (left), Cartpole Swingup (center), and Ball In Cup (right), with natural (top) and synthetic (bottom) video backgrounds.** The mean and standard deviation are computed over 10 runs with different seeds. DAAC and IDAAC outperform PPO, PPG, and UCB-DrAC.

F. Value Loss and Generalization

In this section, we look at the relationship between the value loss, test score, and number of training levels for all Procgen games (see Figures 12, 14, and 13). As discussed in the paper, the value loss is *positively* correlated with the test score and number of training levels. This result goes against our intuition from training RL algorithms on single environments where in general, the value loss is *inversely* correlated with the agent's performance and sample efficiency. This observation further supports our claim that having a more accurate value function can lead to representations that overfit to the training environments. When using a shared network for the policy and value, this results in policies that do not generalize well to new environments. By decoupling the representations of the policy and value function, our methods DAAC and IDAAC can achieve the best of both worlds by (i) learning accurate value functions, while also (ii) learning representations and policies that better generalize to unseen environments.

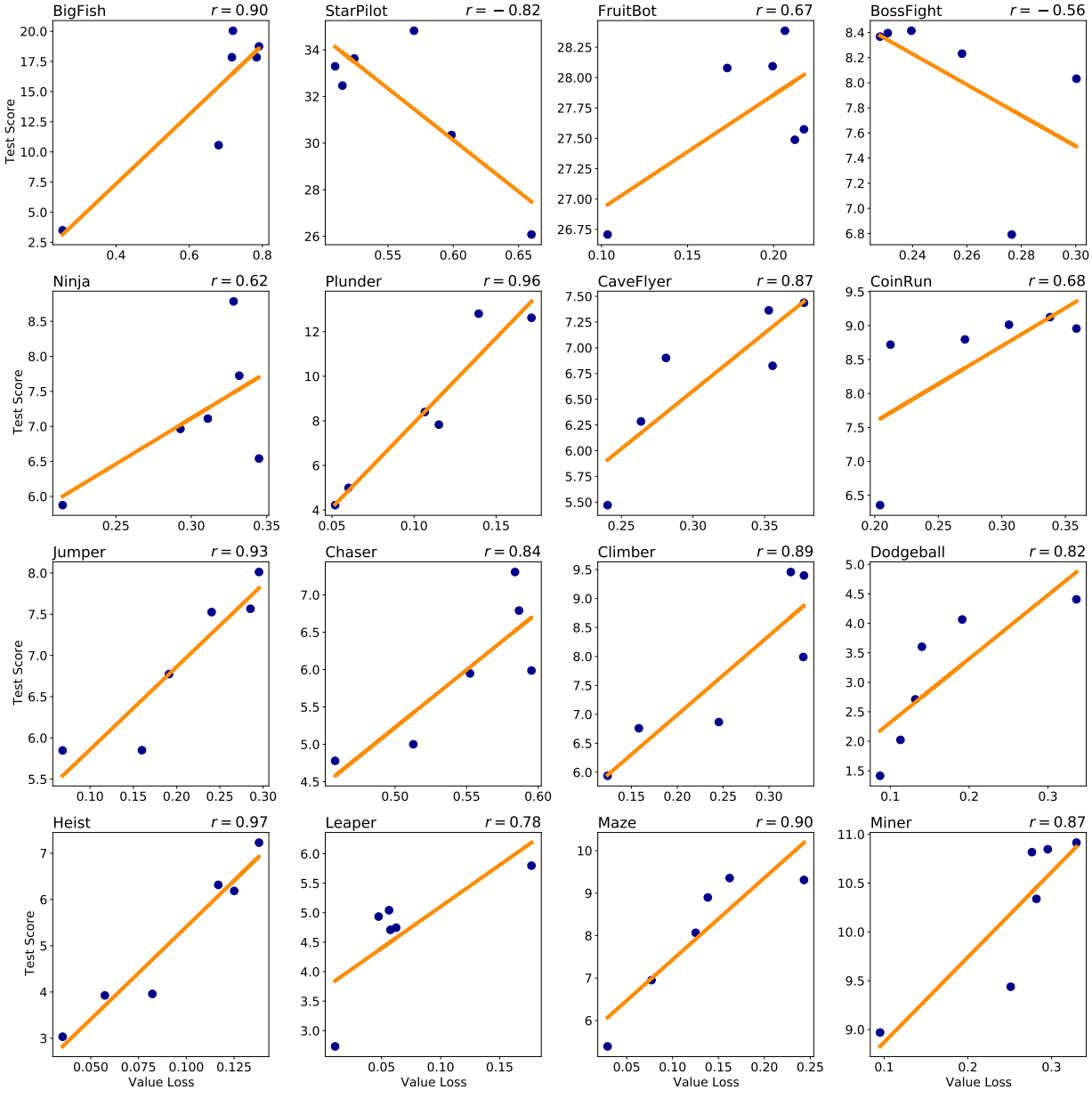


Figure 12. **Correlation of the value loss and test score for PPO agents trained on varying numbers of levels: 200, 500, 1000, 2000, 5000, and 10000.** Surprisingly, the value loss is positively correlated with the test score for most games, suggesting that *models with larger value loss generalize better*.

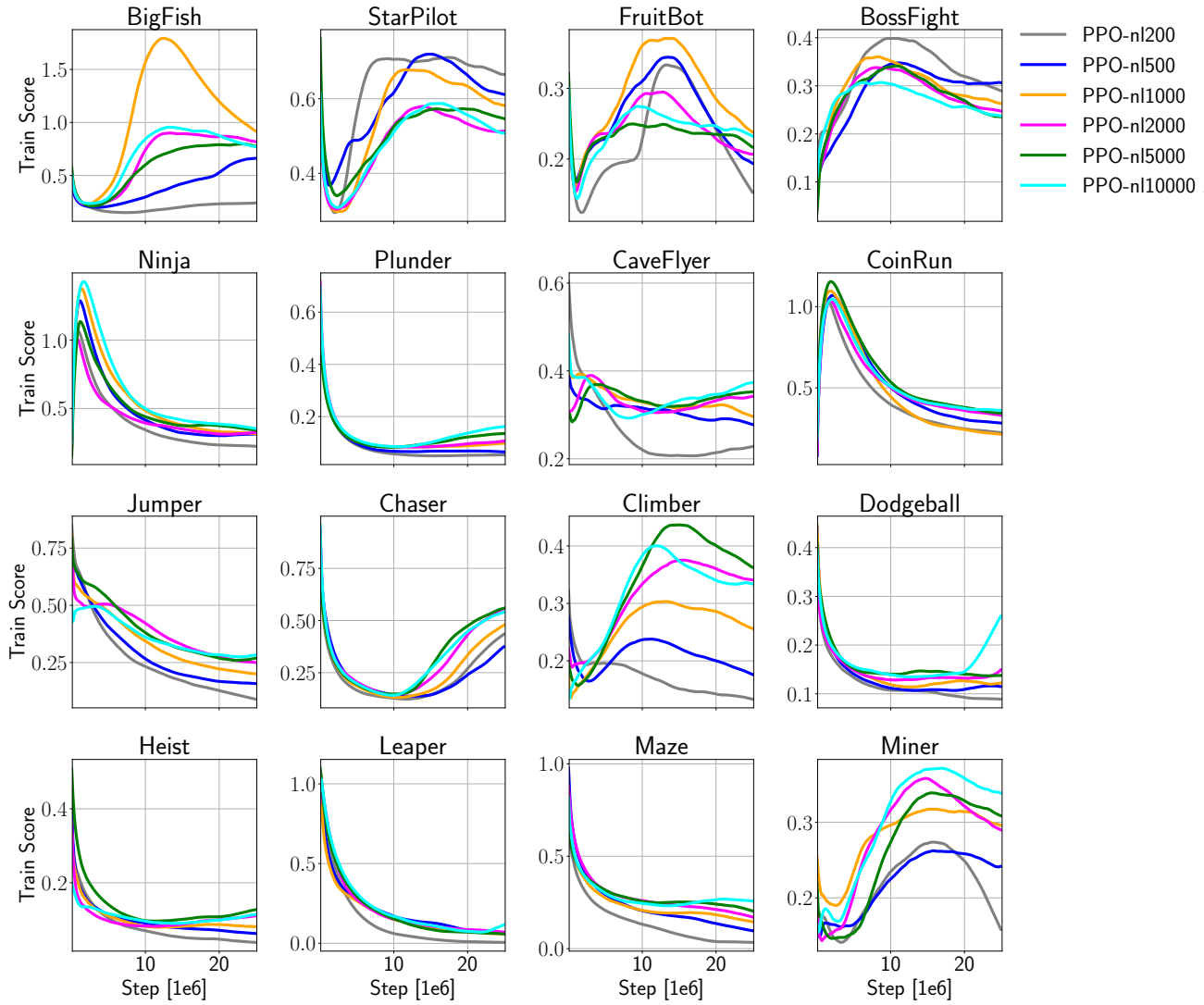


Figure 13. **Test score for PPO agents trained on varying numbers of levels: 200, 500, 1000, 2000, 5000, and 10000.** For most games, the test score increases with the number of training levels, suggesting that models trained on more levels generalize better to unseen levels, as expected.

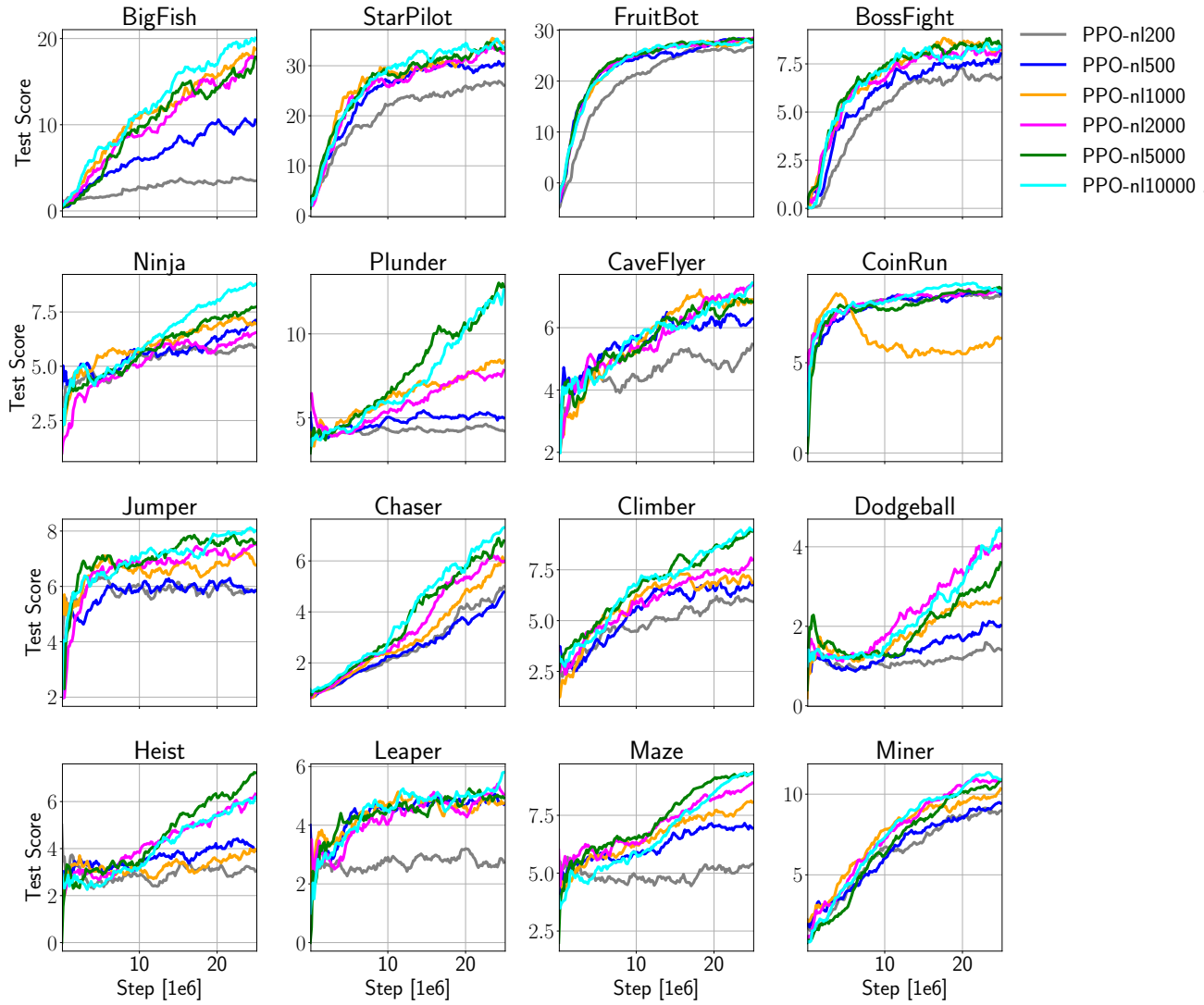
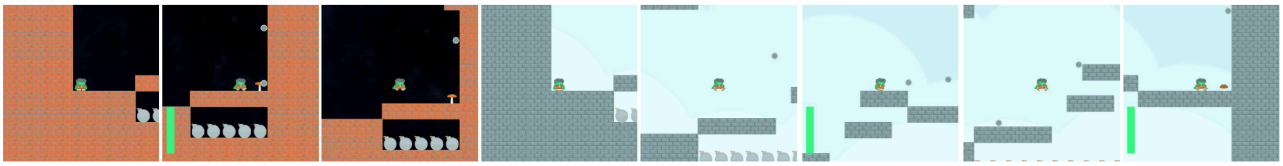


Figure 14. **Value loss for PPO agents trained on varying numbers of levels: 200, 500, 1000, 2000, 5000, and 10000.** For most games, the value loss increases with the number of training levels used, suggesting that models trained on more levels (and thus with better generalization) have higher value loss.

G. Advantage vs. Value During an Episode

Figure 15 shows two different Ninja levels and their corresponding true and predicted values and advantages (by a PPO agent trained on 200 levels). The true values and advantages are computed assuming an optimal policy. For illustration purposes, we show the advantage for the noop action, but similar conclusions apply to the other actions. As the figure shows, the true value function is different for the two levels, while the advantage is (approximately) the same. This holds true for the agent's estimates as well, suggesting that using the value loss to update the policy parameters can lead to more overfitting than using the advantage loss (since it exhibits less dependence on the idiosyncrasies of a level such as its length or difficulty).



Level 1 $V_1(s_0) = 7.9,\ \hat{V}_1(s_0) = 8.0$ $A_1(s_0, a_0) = -0.1,\ \hat{A}_1(s_0, a_0) = 0.05$

Level 2 $V_2(s_0) = 6.1,\ \hat{V}_2(s_0) = 6.3$ $A_2(s_0, a_0) = -0.1,\ \hat{A}_2(s_0, a_0) = 0.04$

Figure 15. **Policy-Value Asymmetry.** Two Ninja levels with initial observations that are *semantically identical but visually different*. Level 1 (first three frames from the left with black background) is much shorter than Level 2 (last six frames with blue background). Both the true and the estimated values (by a PPO agent trained on 200 levels) of the initial observation are higher for Level 1 than for Level 2 *i.e.* $V_1(s_0) > V_2(s_0)$ and $\hat{V}_1(s_0) > \hat{V}_2(s_0)$. Thus, to accurately predict the value function, the representations must capture level-specific features (such as the backgrounds), which are irrelevant for finding the optimal policy. Consequently, using a common representation for both the policy and value function can lead to overfitting to spurious correlations and poor generalization to unseen levels. In contrast to the value, the true advantage of the initial states and noop action has the same values for the two levels, and the advantages predicted by the agent also have very similar values.

We also analyze the timestep-dependence of the predictions (*i.e.* value or advantage) made by various models over the course of an episode. Figures 16, 17, 18, and 19 show examples from CoinRun, Ninja, Climber, and Jumper, respectively. While both PPO and PPG (trained on 200 levels) learn a value function which is increasing almost linearly with the episode step, DAAC (also trained on 200 levels) learns an advantage function which does not have a clear dependence on the episode step. Thus, DAAC is less prone to overfitting than PPO or PPG. Similarly, a PPO model trained on 10k levels with good generalization performance does not display a linear trend between value and episode step. This suggests there is a trade-off between value accuracy and generalization performance. However, by decoupling the policy and value representations, DAAC is able to achieve both accurate value predictions and good generalization abilities.

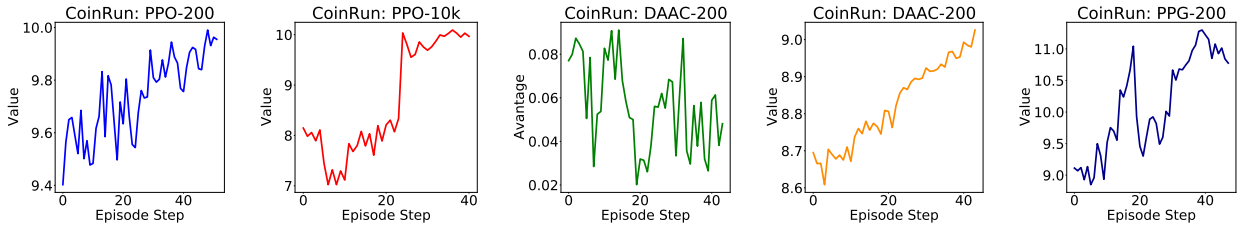


Figure 16. **Examples illustrating the timestep-dependence of the value function for a single CoinRun level.** The dark and light blue curves show the value as a function of episode step for PPG and PPO, respectively, each trained on 200 levels. Note the near linear relationship, indicating overfitting to the training levels. By contrast, a PPO model (red) trained on 10k levels (thus exhibiting far less overfitting) does not show this relationship. Our DAAC model trained on 200 levels (green) also lacks this adverse dependence in the advantage prediction which is used for training the policy network, thus is able to generalize better than the PPO model trained on the same amount of data (see Fig. 3). Nevertheless, DAAC's value estimate still have a linear trend (orange) but, in contrast to PPO and PPG, this does not negatively affect the policy since we use separate networks for learning the policy and value.

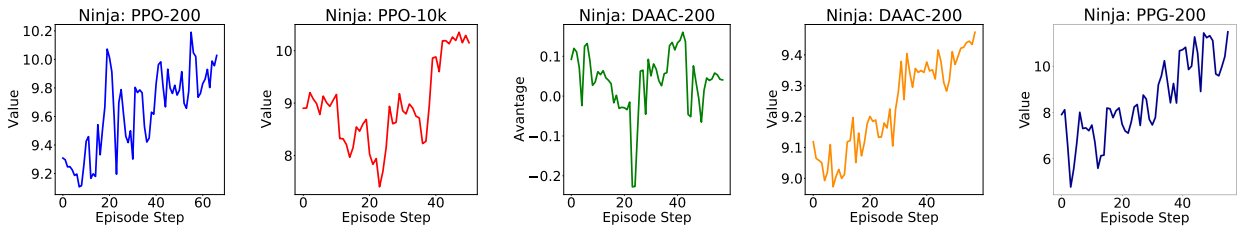


Figure 17. **Examples illustrating the timestep-dependence of the value function for a single Ninja level.** The dark and light blue curves show the value as a function of episode step for PPG and PPO, respectively, each trained on 200 levels. Note the near linear relationship, indicating overfitting to the training levels. By contrast, a PPO model (red) trained on 10k levels (thus exhibiting far less overfitting) does not show this relationship. Our DAAC model trained on 200 levels (green) also lacks this adverse dependence in the advantage prediction which is used for training the policy network, thus is able to generalize better than the PPO model trained on the same amount of data (see Fig. 3). Nevertheless, DAAC's value estimate still have a linear trend (orange) but, in contrast to PPO and PPG, this does not negatively affect the policy since we use separate networks for learning the policy and value.

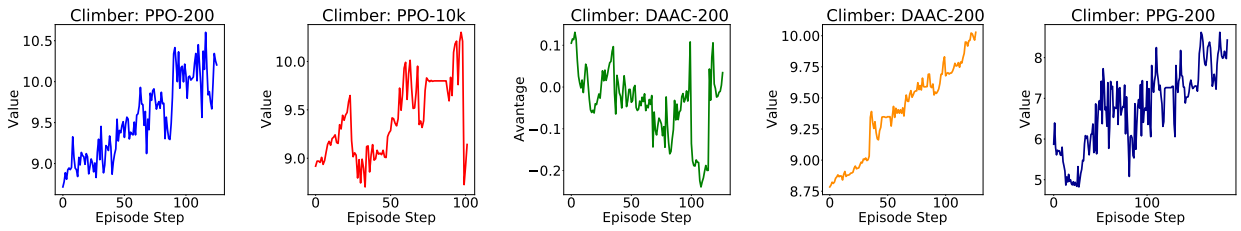


Figure 18. **Examples illustrating the timestep-dependence of the value function for a single Climber level.** The dark and light blue curves show the value as a function of episode step for PPG and PPO, respectively, each trained on 200 levels. Note the near linear relationship, indicating overfitting to the training levels. By contrast, a PPO model (red) trained on 10k levels (thus exhibiting far less overfitting) does not show this relationship. Our DAAC model trained on 200 levels (green) also lacks this adverse dependence in the advantage prediction which is used for training the policy network, thus is able to generalize better than the PPO model trained on the same amount of data (see Fig. 3). Nevertheless, DAAC's value estimate still have a linear trend (orange) but, in contrast to PPO and PPG, this does not negatively affect the policy since we use separate networks for learning the policy and value.

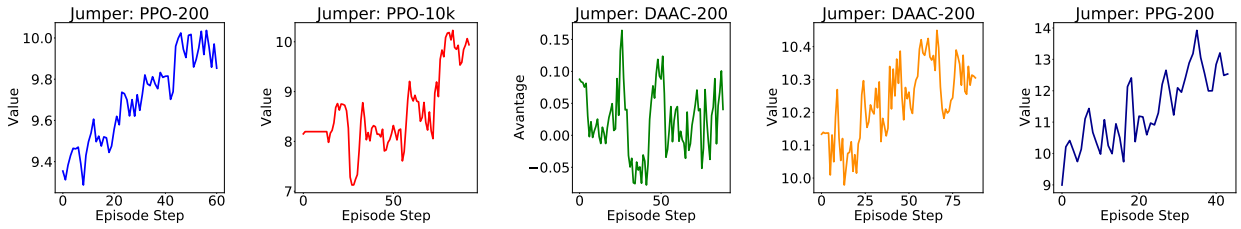


Figure 19. **Examples illustrating the timestep-dependence of the value function for a single Jumper level.** The dark and light blue curves show the value as a function of episode step for PPG and PPO, respectively, each trained on 200 levels. Note the near linear relationship, indicating overfitting to the training levels. By contrast, a PPO model (red) trained on 10k levels (thus exhibiting far less overfitting) does not show this relationship. Our DAAC model trained on 200 levels (green) also lacks this adverse dependence in the advantage prediction which is used for training the policy network, thus is able to generalize better than the PPO model trained on the same amount of data (see Fig. 3). Nevertheless, DAAC's value estimate still have a linear trend (orange) but, in contrast to PPO and PPG, this does not negatively affect the policy since we use separate networks for learning the policy and value.

H. Value Variance

In this section, we look at the variance in the predicted values for the initial observation, across all training levels. In partially-observed procedurally generated environments, there should be no way of telling how difficult or long a level is (and thus how much reward is expected) from the initial observation alone since the end of the level cannot be seen. Thus, we would expect a model with strong generalization to predict similar values for the initial observation irrespective of the environment instance. If this is not the case and the model uses a common representation for the policy and value function, the policy is likely to overfit to the training environments. As Figure 20 shows, the variance decreases with the number of levels used for training. This is consistent with our observation that models with better generalization predict more similar values for observations that are semantically different such as the initial observation. In contrast, models trained on a low number of levels, memorize the value of the initial observation for each level, leading to poor generalization in new environments. Note that we chose to illustrate this phenomenon on three of the Procgen games (*i.e.* Climber, Jumper, and Ninja) where it is more apparent due to their partial-observability and substantial level diversity (in terms of length).

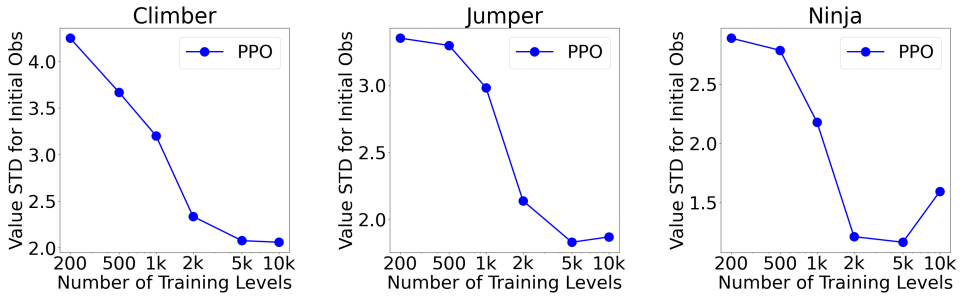


Figure 20. **Standard deviation of the predicted values for the initial observations from 200 levels, as a function of the number of training levels.** From left to right: Climber, Jumper, and Ninja. The 200 levels used to compute the standard deviation were part of the training set for all the agents. Note that, in general, the variance of the value decreases with the number of training levels. This is consistent with our claim that when sharing parameters for the policy and value, models which generalize better predict close values for observations that are semantically similar (*e.g.* the initial observation) since they learn representations which are less prone to overfitting to level-specific features.

I. Robustness to Spurious Correlations

In this section, we investigate how robust the learned features, policies, and predicted values or advantages, are to spurious correlations. To answer this question, we measure how much the features, policies, and predictions vary when the background of an observation changes. Note that the change in background does not change the underlying state of the environment but only its visual aspect. Hence, a change in background should not modify the agent's policy or learned representation. For this experiment, we collect a buffer of 1000 observations from 20 training levels, using a PPO agent trained on 200 levels. Then, we create 10 extra versions of each observation by changing the background. We then measure the L1-norm and L2-norm between the learned representation (*i.e.* final vector before the policy's softmax layer) of the original observation and each of its other versions. We also compute the difference in predicted outputs (*i.e.* values for PPO and PPG or advantages for DAAC and IDAAC) and the Jensen-Shannon Divergence (JSD) between the policies. For all these metrics, we first take the mean for all 10 backgrounds to obtain a single point for each observation, and then we report the mean and standard deviation across all 1000 observations, resulting in an average statistic of how much these metrics change as a result of varying the background.

Figures 21, 22, and 23 show the results for Ninja, Jumper, and Climber, respectively, comparing IDAAC, DAAC, PPG, as well as PPO trained on 200 and 10k levels. In particular, the results show that both our methods learn representations which are more robust to changes in the background than PPO and PPG (assuming all methods are trained on the same number of levels *i.e.* 200). Overall, the differences due to background changes in the auxiliary outputs of the policy networks for DAAC and IDAAC (*i.e.* the predicted advantages) are smaller than those of PPO and PPG (*i.e.* the predicted values). These results indicate that DAAC and IDAAC are more robust than PPO and PPG to visual features which are irrelevant for control.

We do not observe a significant difference across the JSDs of the different methods. However, in the case of Procgen, the

JSD isn't a perfect measure of the semantic difference between two policies because some of the actions have the same effect on the environment (*e.g.* in Ninja, there are two actions that move the agent to the right) and thus are interchangeable. Two policies could have a large JSD while being semantically similar, thus rendering the policy robustness analysis inconclusive.

In some cases, PPO-10k exhibits better robustness to different backgrounds than DAAC and IDAAC by exhibiting a lower feature norm and value or advantage difference. However, PPO-10k is a PPO model trained on 10000 levels of a game, while DAAC and IDAAC are trained only on 200 levels. For most Procgen games, training on 10k levels is enough to generalize to the test distribution so PPO-10k is expected to generalize better than methods trained on 200 levels. In this paper, we are interested in generalizing to unseen levels from a small number of training levels, so PPO-10k is used as an upper-bound rather than a baseline since a direct comparison wouldn't be fair. Hence, it is not surprising that some of these robustness metrics are better for PPO-10k than DAAC and IDAAC.

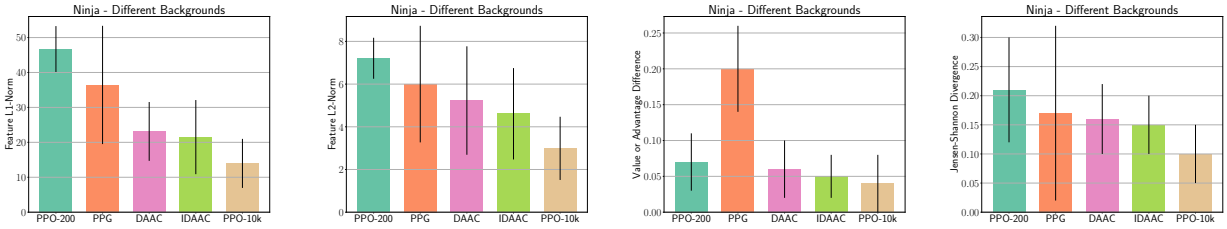


Figure 21. **Variations in the learned features, policies, and values or advantages when changing the background in Ninja.** From left to right we report the L1 and L2-norm for the features, the value or advantage difference, and the Jensen-Shannon Divergence for the policy. We compare PPO trained on 200 and 10k levels with PPG, DAAC, and IDAAC. Our models are more robust to changes in the background (which does not affect the state). The means and standard deviations were computed over 10 different backgrounds.

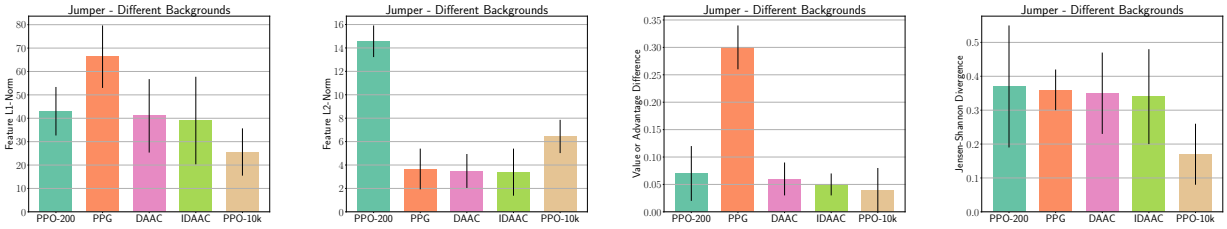


Figure 22. **Variations in the learned features, policies, and values or advantages when changing the background in Jumper.** From left to right we report the L1 and L2-norm for the features, the value or advantage difference, and the Jensen-Shannon Divergence for the policy. We compare PPO trained on 200 and 10k levels with PPG, DAAC, and IDAAC. Our models are more robust to changes in the background (which does not affect the state). The means and standard deviations were computed over 10 different backgrounds.

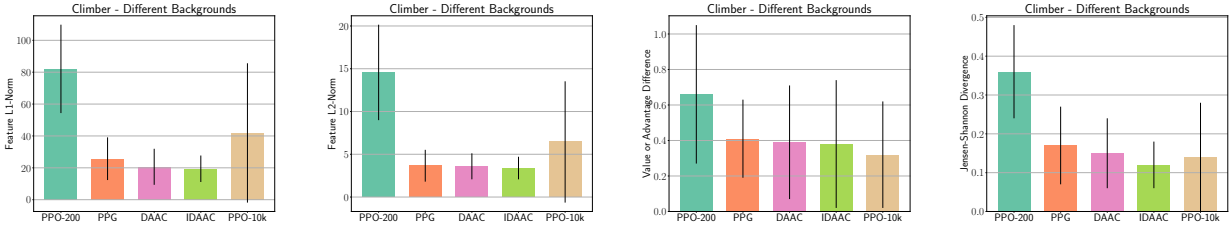


Figure 23. **Variations in the learned features, policies, and values or advantages when changing the background in Climber.** From left to right we report the L1 and L2-norm for the features, the value or advantage difference, and the Jensen-Shannon Divergence for the policy. We compare PPO trained on 200 and 10k levels with PPG, DAAC, and IDAAC. Our models are more robust to changes in the background (which does not affect the state). The means and standard deviations were computed over 10 different backgrounds.