Appendix A Application of mixreg to PPO and Rainbow

A.1 PPO

Background Proximal Policy Optimization (PPO) [12] introduces a novel objective function, resulting in a policy gradient method which is simpler, more general and performs better than previous work Trust Region Policy Optimization (TRPO) [10]. In each update, the algorithm collects a batch of transition samples using a rollout policy \( \pi_{\text{old}}(a_t|s_t) \) and maximizes a clipped surrogate objective:

\[
L_{\text{CLIP}}(\theta) = \mathbb{E}_t \left[ \min \left( r_t(\theta) A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_t \right) \right],
\]

where \( r_t(\theta) \) denotes the probability ratio \( r_t(\theta) = \frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)} \), \( A_t \) is the advantage at timestep \( t \) estimated by generalized advantage estimation (GAE) [11] and \( \epsilon \) is a hyperparameter controlling the width of the clipping interval. GAE makes use of a learned state-value function \( V(s) \) for computing the advantage estimation. To learn the value function, the following loss is adopted:

\[
L^V(\theta) = \mathbb{E}_t \left[ \frac{1}{2} \max \left( (V_\theta(s_t) - V_t^{\text{targ}})^2, \text{clip}(V_\theta(s_t), V_\theta^{\text{old}} - \epsilon, V_\theta^{\text{old}} + \epsilon) - V_t^{\text{targ}} \right)^2 \right],
\]

where \( V_t^{\text{targ}} \) is the bootstrapped value function target. Besides, an entropy bonus is often added to ensure sufficient exploration:

\[
L^H(\theta) = \mathbb{E}_t \left[ H[\pi_{\theta}](s_t) \right].
\]

Putting the above losses together, the overall minimization objective of PPO is:

\[
L^\text{PPO}(\theta) = -L_{\text{CLIP}}(\theta) + \lambda_V L^V(\theta) - \lambda_H L^H(\theta),
\]

where \( \lambda_V, \lambda_H \) are the coefficients to adjust the relative importance of each component. The algorithm alternates between sampling trajectory data using the policy and performing optimization on the collected data based on the above loss.

Applying mixreg At each update, we randomly draw two transitions \( i, j \) from the collected batch of transitions and convexly combine the observations and the associated supervision signals as follows:

\[
\begin{align*}
\bar{s} &= \lambda s_i + (1 - \lambda) s_j, \\
\bar{\pi}_{\text{old}} &= \lambda \pi_{\text{old}}(a_t|s_t) + (1 - \lambda) \pi_{\text{old}}(a_j|s_j), \\
\bar{V}_{\text{old}} &= \lambda V_{\text{old}}(s_i) + (1 - \lambda) V_{\text{old}}(s_j), \\
\bar{V}^{\text{targ}} &= \lambda V^{\text{targ}} + (1 - \lambda) V^{\text{targ}}, \\
\bar{A} &= \lambda A_i + (1 - \lambda) A_j.
\end{align*}
\]

Since we deal with discrete actions, the interpolated action \( \bar{a} \) is simply \( a_i \) if \( \lambda \geq 0.5 \) or \( a_j \) otherwise. The optimization is performed on the generated batch of mixed transitions and each part of the overall objective in Eqn. (12) now becomes

\[
\begin{align*}
\bar{L}_{\text{CLIP}}(\theta) &= \mathbb{E}_{i,j} \left[ \min \left( \hat{r}(\theta) \bar{A}, \text{clip}(\hat{r}(\theta), 1 - \epsilon, 1 + \epsilon) \bar{A} \right) \right] \quad \text{where} \quad \hat{r}(\theta) = \frac{\pi_{\theta}(\bar{a}|\bar{s})}{\pi_{\text{old}}(a_t|s_t)}, \\
\bar{L}^V(\theta) &= \mathbb{E}_{i,j} \left[ \frac{1}{2} \max \left( (V_\theta(\bar{s}) - V^{\text{targ}})^2, \text{clip}(V_\theta(\bar{s}), V_{\text{old}} - \epsilon, V_{\text{old}} + \epsilon) - V^{\text{targ}} \right)^2 \right],
\end{align*}
\]

A.2 Rainbow

Background Rainbow [6] combines the following six extensions for the DQN algorithm: double DQN [14], prioritized replay [9], dueling networks [15], multi-step learning [13], distributional RL [1] and Noisy Nets [5]. As in distributional RL [1], Rainbow learns to approximate the distribution of returns instead of the expected return. At each update, the algorithm samples a batch of transitions from the replay buffer and minimizes the following Kullback-Leibler divergence between the predicted distribution and the target distribution of returns:

\[
\mathbb{E}_t \left[ D_{\text{KL}}(\Phi_2 d_t^{(n)} || d_t) \right].
\]
where \( \widetilde{\gamma} \) denotes the estimated empirical Lipschitz constant.

Appendix B: On the smoothness of the learned policy and value function

Figure 10: The distribution of the calculated ratios from \( 10^6 \) randomly sampled pairs of observations. The largest ratio corresponds to the estimated empirical Lipschitz constant.

In this part, we provide additional results to demonstrate that mixreg helps learn a smooth policy and value function. Figure 11 plots the learned value function in Figure 2 in 3D space for better illustration. The color map is slightly different from the one used in Figure 2 but this does not affect the result. Moreover, we compute the empirical Lipschitz constant [17] of the trained network, i.e. calculating the following ratio for each pair of observations \( s_i, s_j \) and taking the maximum:

\[
\frac{\| f(s_i) - f(s_j) \|}{\| s_i - s_j \|}
\]

where \( f(s) \) denotes the latent representation of observation \( s \). Specifically, we collect a batch of observations and sample \( 10^6 \) pairs for estimating the empirical Lipschitz constant. The results are aggregated in box plots, shown in Figure 10. We can see that the network trained with mixreg has smaller empirical Lipschitz constant compared to the PPO baseline, implying that mixreg helps learn smoother policy.
Figure 11: 3D visualization of the learned state-value functions $V(s)$ in PPO for 6 game environments.
Appendix C Implementation details and hyperparameters

Due to high computation cost, we choose 6 out of 16 environments from the Procgen benchmark. Among the 16 environments, we first exclude 3 environments (Chaser, Leaper, Bossfight) which do not exhibit large generalization gap under 500 level generalization protocol. We then exclude two difficult games (Maze and Heist) where we find the trained policy by PPO performs comparably to a random policy. In the remaining 11 games, we randomly choose 6 environments (Caveflyer, Climber, Dodgeball, Fruitbot, Jumper, Starpilot) for evaluating different methods.

Following [2], we use the convolutional architecture proposed in IMPALA [4]. When applying batch normalization, we add a batch normalization layer after each convolution layer following the implementation\(^1\) in [3]. When applying \(\ell_2\) regularization, we use a weight \(10^{-4}\) as suggested by [3]. For cutout-color and random convolution, we follow the official implementation\(^2\) in [7]. As the official implementation of random crop is problematic\(^3\) (cropping 64 \(\times\) 64 window out of 64 \(\times\) 64 observation), we implement our own version of random crop by first resizing observations to 75 \(\times\) 75 and then randomly cropping with a 64 \(\times\) 64 window. For network randomization method [8], we follow their implementation\(^4\) but do not adopt the Monte Carlo approximation with multiple samples during inference.

For both PPO and Rainbow experiments, we use the same hyperparameters as in [2] except for the ones with \(\dagger\) in the following tables. For PPO experiments, we halve the number of workers but double the number of environments per worker to fit our hardware. This should result in little difference on performance and we are able to reproduce the results in [2]. For Rainbow experiments, as the implementation in [2] is not available, we follow the implementation in anyrl-py\(^5\) and some hyperparameters (denoted with \(\ddagger\)) in retro-baselines\(^6\). \(R_{\text{max}}\) is the normalization constant used in [2] and the distributional min value is changed to -5 in FruitBot.

<table>
<thead>
<tr>
<th>PPO</th>
<th></th>
<th>Rainbow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Env. distribution mode</td>
<td>Hard</td>
<td>Env. distribution mode</td>
</tr>
<tr>
<td>(\gamma)</td>
<td>.999</td>
<td>(\gamma)</td>
</tr>
<tr>
<td>(\lambda)</td>
<td>.95</td>
<td>Learning rate</td>
</tr>
<tr>
<td># timesteps per rollout</td>
<td>256</td>
<td># workers</td>
</tr>
<tr>
<td>Epochs per rollout</td>
<td>3</td>
<td># environments per worker</td>
</tr>
<tr>
<td># minibatches per epoch</td>
<td>8</td>
<td># env. steps per update per worker</td>
</tr>
<tr>
<td>Entropy bonus coefficient ((\lambda_H))</td>
<td>.01</td>
<td>Batch size per worker</td>
</tr>
<tr>
<td>Value loss coefficient ((\lambda_V))</td>
<td>.5</td>
<td>Reward clipping?</td>
</tr>
<tr>
<td>Gradient clipping ((\ell_2) norm)</td>
<td>.5</td>
<td>Distributional min/max values</td>
</tr>
<tr>
<td>PPO clip range</td>
<td>.2</td>
<td>(\dagger)Memory size</td>
</tr>
<tr>
<td>Reward normalization?</td>
<td>Yes</td>
<td>(\dagger)Min history to start learning</td>
</tr>
<tr>
<td>Learning rate</td>
<td>(5 \times 10^{-4})</td>
<td>Exploration (\epsilon)</td>
</tr>
<tr>
<td>(\dagger)# workers</td>
<td>2</td>
<td>Noisy Nets (\sigma_0)</td>
</tr>
<tr>
<td>(\dagger)# environments per worker</td>
<td>128</td>
<td>(\dagger)Target network period</td>
</tr>
<tr>
<td>Total timesteps</td>
<td>200M</td>
<td>Adam (\epsilon)</td>
</tr>
<tr>
<td>LSTM?</td>
<td>No</td>
<td>Prioritization exponent (\omega)</td>
</tr>
<tr>
<td>Frame stack?</td>
<td>No</td>
<td>(\dagger)Prioritization importance sampling (\beta)</td>
</tr>
<tr>
<td>beta distribution parameter (\alpha)</td>
<td>0.2</td>
<td>Multi-step returns (n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Distributional atoms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total timesteps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSTM?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frame stack?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>beta distribution parameter (\alpha)</td>
</tr>
</tbody>
</table>

\(^1\)https://github.com/openai/coinrun
\(^2\)https://github.com/pokaxpoka/rad_procgen
\(^3\)https://github.com/pokaxpoka/rad_procgen/issues/1
\(^4\)https://github.com/pokaxpoka/netrand
\(^5\)https://github.com/unixpickle/anyrl-py
\(^6\)https://github.com/openai/retro-baselines
Appendix D  Ablation results of varying the beta distribution parameter $\alpha$

For the beta distribution $\text{Beta}(\alpha, \alpha)$ used to draw mixing coefficient $\lambda$, we choose $\alpha = 0.2$ from the interval $[0.1, 0.4]$ suggested by [16]. We also test different $\alpha$ and plot the results in Figure 12. Using too large $\alpha$ (e.g. 1.0) leads to performance degradation in certain environments.

![Figure 12: Training and testing performance of mixreg with different parameter $\alpha$ for $\text{Beta}(\alpha, \alpha)$](image-url)

Figure 12: Training and testing performance of mixreg with different parameter $\alpha$ for $\text{Beta}(\alpha, \alpha)$. 
Appendix E  Additional results and training curves on Procgen

E.1  Comparing different methods on 500 level generalization in 6 environments

Figure 13: Training and testing performance of different methods on 500 level generalization.

Figure 14: Mean normalized score of different methods on 500 level generalization.
E.2 Comparing mixreg with PPO baseline on 500 level generalization in all 16 environments

Figure 15: Training and testing performance on 500 level generalization in 16 environments.

Figure 16: Mean normalized score on 500 level generalization averaged over 16 environments.
E.3 Combining mixreg with other methods on 500 level generalization

Figure 17: Training and testing performance of combining mixreg and other methods on 500 level generalization.

E.4 Scaling model size

Figure 18: Mean normalized score w.r.t. different model sizes on 500 level generalization.
E.5 Varying the number of training levels

Figure 19: Training and testing performance w.r.t. different number of training levels.

Figure 20: Mean normalized score w.r.t. different number of training levels.
References


