Proximal Policy Optimization implementation for Procgen Benchmark

Edoardo Merli
edoardo.merli@studio.unibo.it

Abstract

The Procgen Benchmark comprises 16 procedurally generated games designed to measure the generalization capabilities of an agent. In this report, we present the design choices and results obtained by tackling a subset of games in this benchmark using our implementation of Proximal Policy Optimization (PPO), with the IMPALA-CNN as our neural network choice. We experimented with changes to the PPO hyperparameters and the IMPALA architecture. Results on a subset of the 16 games yielded mixed quantitative results with agents being stuck in suboptimal policies. As these are trained for a limited number of steps, more computing time would be needed to assess the goodness of the implementation.

1 Introduction

The Procgen benchmark (Cobbe et al., 2020) has been widely used to assess sample efficiency and generalization in Reinforcement Learning. It has been designed to generate games of high diversity, tunable difficulty and requiring visual recognition to be solved. Due to limited GPU computing power, we chose a subset of the 16 games to tackle: Coinrun, Bossfight, Miner and Climber.

We implemented Proximal Policy Optimization (PPO) (Schulman et al., 2017b) as our agent, a policy gradient algorithm extensively used in the literature for addressing RL problems. As network architecture, we selected the IMPALA-CNN architecture (Espeholt et al., 2018), as it is feasible and reasonably fast to train using free online GPU computing resources (like Kaggle in our case) and its capabilities in the Procgen benchmark have been widely validated (Mohanty et al., 2021).

We experimented with changing some internal components of PPO, its hyperparameters, IMPALA's network architecture and training recipe.

The rest of this report is organized as follows: in section 2 we describe briefly the implemented PPO algorithm and the idea behind it, in section 3 we illustrate the experiments carried forward to reach the final configuration, of which we present the results in section 4. We discuss the results in section 5 and give our conclusions in section 6.

2 Background

2.1 Generalized Advantage Estimation

Vanilla policy gradient algorithms (e.g. REINFORCE), optimize the following loss function:

$$L^{PG}(\theta) = -\hat{\mathbb{E}}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t]$$

where $\hat{A}_t = G_t$, the discounted cumulative reward, or an advantage function, e.g. $\hat{A}_t = \hat{A}_t^{GAE(\gamma,\lambda)}$ using Generalized Advantage Estimation, or GAE (Schulman et al., 2018). The second choice is the most common; more precisely:

$$\begin{aligned}\hat{A}_t^{GAE(\gamma,\lambda)} &= (1-\lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + \dots) \\ &= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V\end{aligned}$$

with δ_t^V being the difference between the baseline $V(s_t)$ and the bootstrapped value of the discounted cumulative reward, namely:

$$\delta_t^V = -V(s_t) + r_t + \gamma V(s_{t+1})$$

$\hat{A}_t^{GAE(\gamma,\lambda)}$ can be rewritten, to ease intuition, as:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = (1-\lambda)(-V(s_t) + r_t + \gamma V(s_{t+1}) + \lambda(-V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2})) + \dots)$$

So the GAE advantage can be interpreted as an exponentially-weighted average, over increasing values of n, of the difference between the baseline value function at state s_t ($V(s_t)$) and the TD(n) update ($\sum_{i=0}^{n}\gamma^i r_{t+i} + \gamma^{n+1}V(s_{t+n+1})$).

For low values of λ, it weights more terms with TD(n) having low values of n, which have high bias and low variance. On the other hand, for high values of λ, the estimator weights more terms with TD(n) having high values of n, which have low bias and high variance. In this way, the λ parameter governs the balance of the interpolation between bias and variance in the estimate.

2.2 PPO

TRPO (Schulman et al., 2017a) introduced the loss function

$$L^{TRPO}(\theta) = -\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right]$$

The ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ makes it so that actions that are more probable to be taken under the updated policy, compared to before the optimization, weight more in the loss, and also the opposite (relatively less probable actions are down-weighted).

The main innovation introduced by the PPO algorithm is the *Clipped Surrogate Objective Function*, i.e.

$$L^{CLIP}(\theta) = -\hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

To avoid updates with too drastic changes of actions distributions from the frozen policy, the ratio is clipped (usually $\epsilon = 0.2$, as in our case). This helps to have more stable updates, and for this reason, makes it possible to train the network for multiple epochs on the same set of collected transitions.

The resulting algorithm is the following:

Algorithm 1 PPO

Input: Initialized π_θ, $\pi_{\theta_{\text{old}}}$, v_w
  for $iter = 1, \dots, N$ **do**
    Collect T transitions by following π_θ
    Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$ using v_w
    for $epoch = 1, \dots, M$ **do**
      Optimize $L^{CLIP}(\theta)$ wrt θ, using π_θ and frozen $\pi_{\theta_{\text{old}}}$
      Optimize $L^{MSE}(w)$ wrt w with $\hat{A}_1, \dots, \hat{A}_T$ as targets
      $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
    end for
  end for

We chose this algorithm as it is widely applied as a standard go-to choice for RL problems and a common baseline for other algorithms (which makes its study and implementation useful both as a didactic exercise and as a starting point for future personal learning in the RL field) and exhibits state-of-the-art performance while being relatively simple to implement.

3 Experiments and design choices

The experiments below were run on the game *Coinrun*.

3.1 PPO

Policy and Value networks. The policy network and the value network can either share weights and have different heads or be two totally separate networks. We chose the second option, which, as indicated by Andrychowicz et al. (2020), resulted in a substantial improvement in performance. For more details on the network architecture, refer to section 3.2 below.

Value targets. As value targets V_{target}, we experimented with both G_t and $\hat{A}_t^{GAE(\gamma,\lambda)} + V(s_t) \approx V^{\pi_\theta}(s_t)$ (as from Andrychowicz et al. (2020)). The second choice resulted in superior results. Furthermore, as the paper also highlights the importance of value normalization, we normalized the value targets by keeping running averages v_μ and v_ρ, let v_w predict $(V_{target} - v_\mu)/\max(v_\rho, 10^{-6})$ and de-normalized the value network predictions accordingly at inference time.

Entropy loss. In the Procgen paper Cobbe et al. (2020), the PPO loss also includes an entropy component, to incentivize exploration, with a coefficient of 0.01. We noticed this value to weight exploration too much and found a lower value of 1×10^{-5} to help exploration while not risking to make training diverge.

Early stopping. As, at each iteration, PPO runs a few epochs of optimization, a common early stopping strategy, to interrupt training and continue with collecting the next set of trajectories, is to measure the Kullback–Leibler divergence between the old, frozen policy $\pi_{\theta_{\text{old}}}$ and the policy being optimized π_θ, and stop the training if this value exceeds a threshold.

Tests were run with thresholds in $\{0.015, 0.02, 0.03, 0.05, \infty\}$ and we found that while performance wasn't really sensible to its value, its presence (so any value different from ∞) helped to avoid catastrophic forgetting. We settled for the value of 0.02.

An important note is that while the KL divergence on the whole policy distribution had extremely high variance, restricting it to the set of actions in the minibatch made its value much more stable. We found the resource Schulman (2020) useful to approximate the KL divergence from a set of samples, namely using the formula $KL = \mathbb{E}_t\left[\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} - 1\right) - \log\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\right)\right]$. This change notably improved training stability.

3.2 IMPALA network

The standard IMPALA implementation has 3 stages of respectively 16, 32 and 32 channels each, composed only of Convolutional layers, ReLU activations and Max pooling operations.

For the policy network π_θ, the head of the network is a MLP with 256 hidden neurons and as many output neurons as the number of actions possible in the environment. The logits of the last activations are then converted to a categorical distribution to sample from.

For the value network v_w, the head of the network is a MLP with 256 hidden neurons and a single output neuron. The loss is a standard MSE loss.

Input size. We experimented with stacking together either 2 or 4 frames in order to give the network the temporal information needed for the trajectories and velocities of objects in the scene. We settled for the setting with 4 frames.

Network width. We tried to increase the number of channels of each stage, up to doubling them ([32, 64, 64]). We didn't see any particular performance boost, so the cost of the increase in FLOPS wasn't motivated and we maintained the standard network width of [16, 32, 32].

Last layer initialization. As Andrychowicz et al. (2020) has shown, the initial policy has a high impact on training performance. They suggest initializing the last layer of the policy network with small weights, such that the initial action is almost independent of the observation. We do so by setting them to be orthogonal and with a value of 0.01, on top of setting bias to 0. We did the same for the value network but with the initial value set to 1.0.

Batch normalization. As batch normalization has shown to improve generalization in RL (Cobbe et al., 2019) we decided to test it in our problem scenario. We didn't observe any meaningful performance improvement, so we decided to ignore this addition.

Global average pooling. More modern image classification networks (starting from Szegedy et al. (2014)) add a Global average pooling operation before the classification head. We tried to add this component but we didn't see particular benefits from it, so we excluded it from our final model.

Squeeze and excitation block. We even tested the addition of Squeeze and Excitation block (Hu et al., 2019) to each stage, but as it didn't yield a concrete improvement, we continued with the standard stage block of IMPALA.

As most of the experiments aiming to improve performance by changing the network didn't have the desired effect, we hypothesize that the network capacity isn't the bottleneck in this problem setting. This didn't surprise us though, as simple networks perform remarkably well in RL problems in the literature.

3.3 Methods

The final configuration that we used to obtain the results of section 4 is shown in table 1 below:

Hyperparameter	Value
Backgrounds	False
Total steps	2M
# steps per iteration	1024
Batch size	64
Epochs per iteration	3
PPO clip ϵ	0.2
Separate networks	Yes
Learning rate	5×10^{-4}
LR schedule	Cosine
Value targets normalization	Yes
Entropy loss coefficient	1×10^{-5}
KL divergence limit	0.02
Frames stack size	4
IMPALA stages # channels	[16, 32, 32]
Batchnorm	No
Global average pooling	No
SE Block	No

Table 1: Configuration and hyperparameters

We run the training on one GPU NVIDIA Tesla T4 using Kaggle free computing resources, training took approximately 6 hours per game with the above total number of steps.

4 Results

4.1 Quantitative results

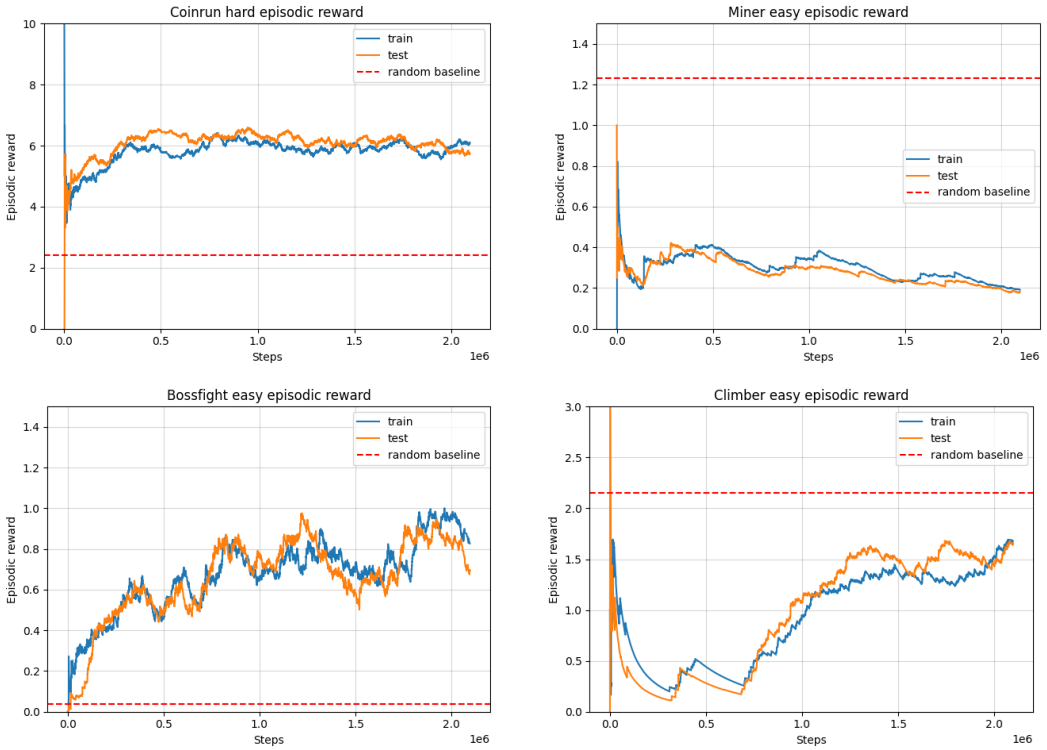

Figure 1: Episodic reward in the train and test environments compared against a random baseline, for the four games selected.

In Figure 1 above we show the performance of our agent both in the training environment (restricted to 200 levels for the easy difficulty and 500 levels for the hard difficulty) and the test environment (unlimited number of levels). We also plotted the mean episodic reward over 1000 episodes of a random baseline for comparison.

As we can see, the agent outperforms the random baseline in the games of Coinrun and Bossfight, while doing worse than it at Miner and Climber.

All curves have train and test values really close, and the rewards (except for Coinrun) tend to have low absolute value compared to the rewards that can be obtained in the respective game and are reached (with substantially more training steps) in Cobbe et al. (2019). This could indicate that the agent is not learning properly or that it requires further learning steps.

4.2 Qualitative results

Qualitative evaluations of the gameplay generated by the trained agents showed that the agent learned a repetitive strategy that doesn't depend on the state of the game:

- Coinrun: continually jump while moving right
- Miner: repeatedly move back and forth horizontally
- Bossfight: stand still and continually fire
- Climber: continually jump while moving slightly left or right, but independently of the near platforms

5 Discussion

While the quantitative results might have seemed promising, looking at the actual policy learned showed that a simplistic strategy (as for Coinrun and Bossfight) was enough to reach episodic reward considerably higher than the random baseline.

The findings of subsection 4.2 suggest that the agent is not exploring enough, and would motivate an increase in the exploration coefficient of the entropy. Nonetheless, we saw that this tentative fix didn't work as expected as the episodic reward decreased throughout the majority of the learning phase and resulted in even worse (almost random) policies in the end. We hypothesize that this decrease might have been only temporary, representing an exploration phase that would have eventually led to more high-quality behaviour.

This is in accordance with the fact that PPO agents in the literature are trained for hundreds of millions of steps. Our reduced computing power though didn't enable us to test this hypothesis.

6 Conclusions

In this report, we presented the design choices for our implementation of the PPO algorithm using the IMPALA architecture as the neural network choice. We listed the experiments carried forward to try to improve the algorithm's sample efficiency and performance on the Procgen benchmark.

The results obtained on a subset of four games, while promising at first, turned out to be generated by "collapsed" policies, generating repetitive behaviour independent from the input state. Theoretically, increasing exploration (in the form of a higher entropy coefficient) should have been beneficial, but its potentially positive effect couldn't be measured in the number of total steps of our configuration.

In general, in order to make a more informed statement regarding the quality of the implementation, it would be interesting to run the algorithm for orders of magnitude more steps (especially with a higher entropy coefficient), such that comparisons with the literature's results could be made.

References

Andrychowicz Marcin, Raichuk Anton, Stańczyk Piotr, Orsini Manu, Girgin Sertan, Marinier Raphael, Hussenot Léonard, Geist Matthieu, Pietquin Olivier, Michalski Marcin, Gelly Sylvain, Bachem Olivier. What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study. 2020.

Cobbe Karl, Hesse Christopher, Hilton Jacob, Schulman John. Leveraging Procedural Generation to Benchmark Reinforcement Learning. 2020.

Cobbe Karl, Klimov Oleg, Hesse Chris, Kim Taehoon, Schulman John. Quantifying Generalization in Reinforcement Learning. 2019.

Espeholt Lasse, Soyer Hubert, Munos Remi, Simonyan Karen, Mnih Volodymir, Ward Tom, Doron Yotam, Firoiu Vlad, Harley Tim, Dunning Iain, Legg Shane, Kavukcuoglu Koray. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. 2018.

Hu Jie, Shen Li, Albanie Samuel, Sun Gang, Wu Enhua. Squeeze-and-Excitation Networks. 2019.

Mohanty Sharada, Poonganam Jyotish, Gaidon Adrien, Kolobov Andrey, Wulfe Blake, Chakraborty Dipam, Šemetulskis Gražvydas, Schapke João, Kubilius Jonas, Pašukonis Jurgis, Klimas Linas, Hausknecht Matthew, MacAlpine Patrick, Tran Quang Nhat, Tumiel Thomas, Tang Xiaocheng, Chen Xinwei, Hesse Christopher, Hilton Jacob, Guss William Hebgen, Genc Sahika, Schulman John, Cobbe Karl. Measuring Sample Efficiency and Generalization in Reinforcement Learning Benchmarks: NeurIPS 2020 Procgen Benchmark. 2021.

Schulman John. Approximating KL Divergence — joschu.net. 2020.

Schulman John, Levine Sergey, Moritz Philipp, Jordan Michael I., Abbeel Pieter. Trust Region Policy Optimization. 2017a.

Schulman John, Moritz Philipp, Levine Sergey, Jordan Michael, Abbeel Pieter. High-Dimensional Continuous Control Using Generalized Advantage Estimation. 2018.

Schulman John, Wolski Filip, Dhariwal Prafulla, Radford Alec, Klimov Oleg. Proximal Policy Optimization Algorithms. 2017b.

Szegedy Christian, Liu Wei, Jia Yangqing, Sermanet Pierre, Reed Scott, Anguelov Dragomir, Erhan Dumitru, Vanhoucke Vincent, Rabinovich Andrew. Going Deeper with Convolutions. 2014.