PPO performance on Procgen Benchmark

Muhammad Saleem Ghulam
Autonomous and Adaptive Systems
University of Bologna, MSc in Artificial Intelligence
`muhammad.ghulam@studio.unibo.it`

Abstract

This paper describes the implementation of a Reinforcement Learning agent using the Proximal Policy Optimization algorithm (PPO)[4]. The experiment purpose is to evaluate the PPO agent generalization capabilities on a subset of environments, during the learning process. The environments are procedurally generated and provided by the Procgen benchmark[1].

1 Introduction

The purpose of reinforcement learning (RL) is to build agents that are able to make optimal and autonomous decisions in complex and dynamic environments. Generalization represents a fundamental challenge in RL, as in real-world scenarios it's essential that the agents are prepared to deal with unseen situations during the learning phase. The Procgen Benchmark[1] was designed to prevent agents from overfitting to specific environments and to counteract their inability to play in unseen variations of those environments.

1.1 Procgen Environment

The Procgen Benchmark consists of 16 distinct procedurally generated environments, having as aim the improvement of sample efficiency and generalization performance of the RL agents. The benchmark allows training and evaluation of the agents on diverse sets of levels, randomly initialized, for each environment, preventing overfitting and helping to build more robust and stable agents. In this work, we are going to evaluate the generalization capability of a PPO agent on a subset of the available environments in the benchmark (Bigfish, Dodgeball, Fruitbot, Maze, Ninja, Starpilot).

Bigfish The agent is a small fish, and should eat other fishes to become bigger. The agent should only eat fishes smaller than itself, otherwise, this leads to failure. The level is complete when the agent is bigger than all the other fishes.

Dodgeball The agent should navigate in a room with walls, that if touched, leads to failure. Moreover, the agent should eliminate all the enemies moving in the room by throwing the balls at them, and meanwhile also avoid the balls thrown from the enemies. Once all the enemies are eliminated, the agent can move on a platform to complete the level.

Fruitbot A scrolling game where the agent should reach the end (large positive reward) by navigating through holes in the walls, and on the way collect fruit objects (positive reward) and avoid non-fruit objects (negative reward), and also avoid the walls that cause failure.

Maze The agent should navigate the maze and look for a piece of cheese to obtain the reward of 10.

Preprint. Under review.

Ninja A platforming game in which the agent must jump on narrow platforms while avoiding bombs. When the agent collects the mushroom, then a reward of 10 is returned and the episode terminates.

Starpilot A side scrolling shooter game, in which the agent must dodge the enemy fire and destroy enemies to increase the final score. Moreover, it should avoid obstacles.

```
Algorithm 1 PPO, Actor-Critic Style
  for iteration=$1, 2, \ldots$ do
      for actor=$1, 2, \ldots, N$ do
          Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
          Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
      end for
      Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
      $\theta_{\text{old}} \leftarrow \theta$
  end for
```

Figure 1: PPO training algorithm

2 Methodology

This section will discuss the implementation of the PPO algorithm, the architecture of neural networks, and the experimental setup.

2.1 Algorithm

The Proximal Policy Optimization [4] (PPO) algorithm is a policy gradient method, which means it directly optimizes the policy such that the expected rewards are maximized. The algorithm alternates between data collection from multiple parallel environments and optimization of a clipped loss function. PPO improves the Trust Region Policy Optimization (TRPO) algorithm, making the algorithm more stable, efficient, and easier to implement. A clipping mechanism is used to prevent drastic updates between the new policy and the old one, by ensuring a smoother and more stable learning. PPO also improves sample efficiency compared to other policy gradient methods, by using mini-batches and multiple epochs on a single collection of data. As shown in Figure 1, the PPO algorithm for each iteration runs N parallel actors, each actor collects T timesteps of data, and the clipped loss is then optimized for K epochs over the NT batch of data, updating the policy network parameters.

The loss function combines two terms, a ratio of probabilities of the actions between the new policy and the old policy, and the advantage function, which estimates how much an action in a state is better than an average action in that state. The calculation of advantage relies on the Generalized Advantage Estimation algorithm (GAE)[3]. This technique provides a stable and efficient estimate of the advantage, considering a weighted estimate of the advantages across multiple time steps. As mentioned in the PPO paper, an entropy bonus is added to the objective function to improve exploration.

2.2 Architecture

The network architectures of the actor and critic are shown in the Figure 2. In this project, the actor and critic neural networks are implemented with an independent set of parameters, and each one is optimized with its own loss function.

The backbone architecture of both networks is inspired by the Impala architecture [2] used in the Procgen paper [1], with some additions. The classification heads of the two networks are different, the actor model outputs action probabilities, so we have a 1x1 convolution and Global Average Pooling (GAP) layer followed by a Softmax layer. Instead, in the critic, the conv-net is followed by fully connected layers that output the state value. In the actor classification head, the GAP is used instead of the fully connected layer, which reduces each feature map to a single value. The result is a vector

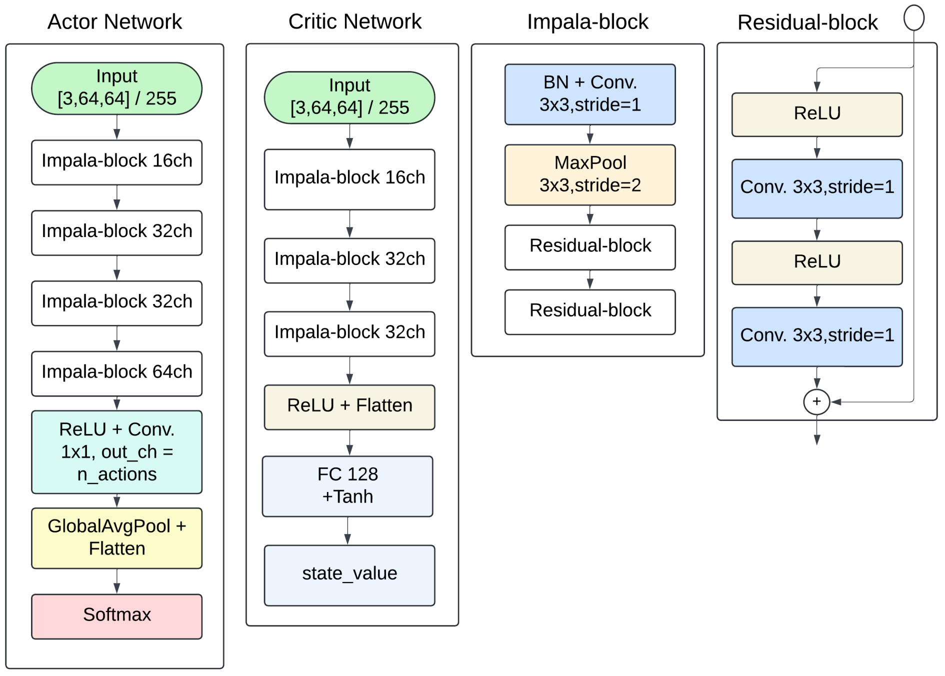

Figure 2: Actor and Critic architectures with Impala backbone

that contains one value at each channel for each output. The output is then flattened and passed through a Softmax layer to get action probabilities. This choice is inspired by the ResNet architecture. In the Impala block, a Batch Normalization (BN) layer is placed in front of the Convolutional layer. This layer normalizes the inputs of the next layer by adjusting and scaling the weights on the mean and variance of the mini-batch. In the following sections we will discuss about how it affects the learning stability and the generalization performance.

2.3 Implementation

The implementation is done by using the Open AI Gym API, which provides a collection of environments and defines a simple and consistent interface, simplifying the interaction between agents and environments. Gym API allows running multiple independent copies of the same environment in parallel using multiprocessing through the vectorized environments, for faster convergence. In this project, we decided to run 48 environments in parallel, in order to find a balance between resources and speed. Each time an environment terminates, a new one is automatically instantiated.

We also implemented a linear learning rate scheduler, that gradually decreases the learning rate as the number of iterations increases. This approach is implemented to prevent large updates in the later iterations, that may lead to a drastic drop in the agent's performance. In addition, we implemented the normalization of the advantages of each batch, such that the updates are on the same scale. This approach affects positively the learning process, by an increase in stability and speed.

2.4 Experimental setup

For each game, the notebook has been executed with three different seeds (42, 1377, 47981) for more significant results. The execution is done using Kaggle platform.

We decided to run 48 environments in parallel for faster convergence, for a total of 1000 iterations, and in each iteration, 256 steps are played. For each game, the agent performs a total of 12 million steps.

In the training phase, for each Procgen game a specific vectorized gym environment of 48 parallel environments is set with easy difficulty, the number of levels is fixed to 200, and the starting level and random seed are fixed for code reproducibility.
In the evaluation phase, we instantiated two different non-vectorized environments, in order to evaluate the agent's performance on seen and unseen levels. The environment to evaluate training performance has the same settings as the training one. Instead, the test environment has an unlimited number of levels and the same fixed random seed for reproducibility reasons. In order to produce plots, the scores are computed every two iterations, as a mean of rewards across ten episodes.
During training, every 250 iterations, actor and critic checkpoints are created. All the necessary information are saved in a pickle file. The PPO hyperparameters can be found in the *notebook*.

3 Results

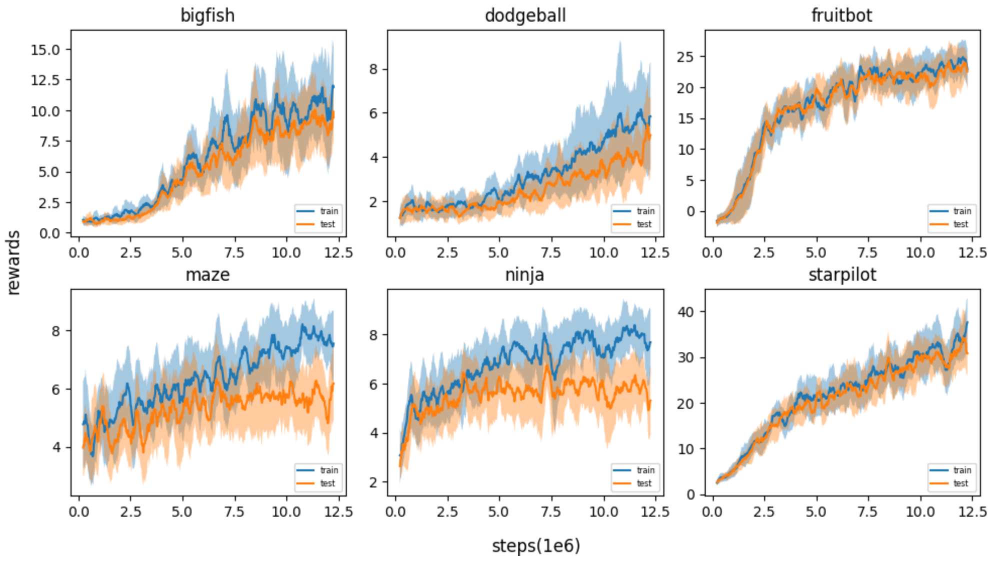

Figure 3: Agent's performance, trained on 200 levels with easy difficulty, and evaluated on the full distribution of levels.

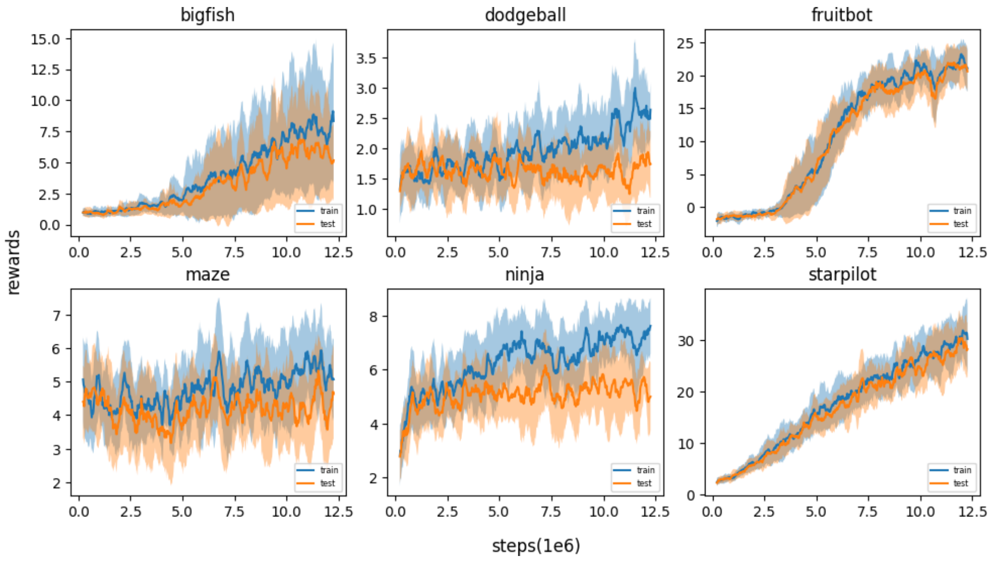

Figure 4: Agent's performance removing the batch normalization layer from the architecture, trained on 200 levels with easy difficulty, and evaluated on the full distribution of levels.

In this section, we present plots illustrating the agent's performance in both the training and test environments at different steps of the training process across the different games, as in Figure 3. We also compare the agent's behavior in case the BN layer is removed from the network architecture, as in Figure 4. This approach allows for a clear comparison of how BN influences the agent's learning and generalization capabilities across different environments.

As previously mentioned, during training, the agent is evaluated on both the training and test sets every two iterations. This process results in collecting a total of 500 values for each set. In each set, the agent plays ten episodes, and the final score is calculated as the mean of these rewards. Each plot shows the mean and the standard deviation values of the scores across the 3 seeds. For a better and smoother visualization of the results, a rolling operation is applied to the final values with a window size of 10.

4 Discussion

For resource constraints, we consider a training process of about 12 million steps for all the games. So the following discussion will be about this particular experiment.

As shown in Figure 3, the PPO agent is able to learn how to solve most of the games successfully, managing also to generalize quite well on the unseen levels. The generalization capabilities of the agent depend on the environment in which it's interacting. Indeed, in games like Bigfish, Dodgeball, Fruitbot and Starpilot, the performance on seen and unseen levels is almost equal. In contrast, in games like Maze and Ninja, the agent struggles to generalize within 12 million steps. This is evident from the slower increase in test scores compared to training ones. In environments like Bigfish, Fruitbot and Starpilot, the agent not only performs greatly on unseen levels, but also manages to obtain a good convergence speed.

The poor generalization performance observed in Maze and Ninja environments may be due to several factors, including how the rewards are collected during the gameplay, the number of episodes played, or the lack of training data to understand the underlying pattern. It's important to note that until 6 million steps, in both the games, the test score is increasing along with the training scores, indicating that the agent is effectively learning. However, after that point the agent starts to overfit over the training data, resulting in a stalemate situation for the test score.
In scrollable environments like Fruitbot and Starpilot, frequent feedback is provided and failure conditions are implemented before exceeding maximum allowed steps(1000), allowing the execution of more episodes. These factors noticeably speed up the learning and generalization performance. Instead, in environments like Maze, the episode terminates only if the agent finds the object and gets the reward, or it fails after 1000 steps. So basically, this type of game requires more diverse data and more steps to effectively understand and navigate the correct path to achieve the reward.

The use of Global Average Pooling in the actor classification head reduces computational overhead, as it has less number of parameters, and this may help to prevent overfitting and enhance generalization performance.
The Batch Normalization (BN) layer plays a crucial role in learning and enhancing generalization score. The removal from the networks architecture leads to poor performance, as shown in Figure 4. Without BN layer, the agent struggles on unseen levels and even fails to learn in certain environments, like Dodgeball and Maze. In other environments like Bigfish, Fruitbot and Starpilot, the agent is able to learn, but the learning speed is slower and the maximum score is lower compared to the variant with BN layer. Another important aspect that is highlighted from the use of BN layer is the reduction of variance across different seeds, making the training process more robust and stable.

5 Conclusions

In this project, we have implemented the PPO agent to test its generalization capabilities, using the Procgen benchmark. This benchmark provides different types of environments, allowing to access to an unlimited number of levels produced using procedural generation. We took a subset of environments, trained the PPO agent over 200 levels for each environment, and then evaluated the agent's performance during the learning process. Furthermore, we also evaluated the impact of BN layer on the agent's behavior.

This experiment highlights the ability of PPO agent to learn quite well across different environments using only a limited quantity of training data, and being also an easy algorithm to be implemented. Furthermore the PPO algorithm is structured in such a way that it makes possible to run multiple parallel environments, using the vectorized environments provided by Gym API. This tool speeds up the data collection and training process. From the results obtained, we can conclude that the Batch Normalization layer reveals to be quite important in learning stability and inference performance, in almost all the environments. In fact it's exclusion results in poor performance and increased noise across the different seeds.

In conclusion, the agent performs quite well in environments like Bigfish, Dodgeball, Fruitbot and Starpilot, but it overfits in environments like Maze and Ninja. The results suggest that the generalization performance depends on many factors, as the reward structure, the number of episodes played, the termination conditions and the reward frequency. In fact, environments like Fruitbot and Starpilot, that provide frequent feedback and allow for more exploration, leading to faster learning and improved generalization performance.

Future improvements can be an exhaustive search of optimal hyperparameters that may lead to better performance. Other ideas can be to train the agent for more number of steps, or to increase the number of levels for environments that tend to overfit, like Maze and Ninja. Another possible study can be about how the agent behaves in other environments provided by the benchmark, in order to understand the limitations and possible improvements.

6 Useful links

Project code: github.com

The following articles were read:

The 37 implementation details of PPO was an interesting read to understand details about PPO implementation, annealing learning rate and advantages normalization: iclr-blog-track.github.io
Openai PPO implementation This read clarifies some implementation aspects of PPO algorithm: spinningup.openai.com/en/latest/algorithms/ppo.html

The phrasing is corrected using ChatGPT (openai.com)

References

[1] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning, 2020.

[2] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018.

[3] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.