REINFORCE vs Proximal Policy Optimization

Daniele Sirocchi
University of Bologna
`daniele.sirocchi@studio.unibo.it`

Abstract

Policy gradient methods have been already used for solving a huge variety of reinforcement learning (RL) problems, and there exist different algorithms based on them.The goal of this project is to implement and to compare two of these algorithms: the REINFORCE and the Proximal Policy Optimization (PPO).
The implementations and the experiments are based on three different environments, found in the OpenAI's Procgen Benchmark.

1 Policy gradient methods

The main idea behind *policy gradient methods* is that it is possible to find the optimal policy by maximizing a performance measure (i.e., cumulative reward). Mathematically, this can be obtained through the gradient ascent, which considers the gradient of the performance measure in the model updates. These methods rely on the Policy Gradient Theorem [1].

1.1 REINFORCE

REINFORCE [2] is a Monte Carlo policy gradient control algorithm, and it can be seen as a baseline among all the policy gradient methods available. The vanilla version of REINFORCE optimizes the following objective function:

$$L^{PG} = \hat{\mathbb{E}}_t \left[\nabla_\theta log \pi_\theta(a_t|s_t) G_t \right] \tag{1}$$

The implemented REINFORCE version used in this project considers also the baseline term in the updates, because it allows to have a much more stable training.

Thus, the final expression of the objective function is:

$$L^{PG} = \hat{\mathbb{E}}_t \left[\nabla_\theta log \pi_\theta(a_t|s_t)(G_t - \hat{v}(s_t)) \right] = \hat{\mathbb{E}}_t \left[\nabla_\theta log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{2}$$

where $\hat{A}_t$ is called *Advantage Function*.

1.2 Proximal Policy Optimization

Like REINFORCE, PPO [3] is a Monte Carlo policy gradient control algorithm, and it includes a series of stratagems that make it comparable with state-of-the-art approaches, while being much easier to implement and tune.

1.2.1 Clipped surrogate objective

PPO is an algorithm that leverages a key concept introduced in the Trust Region Policy Optimization (TRPO) [4], which consists in limiting the policy updates (i.e., the policy after an update cannot be too different from the old one).
From the TRPO paper, the surrogate objective maximized during the training is the following:

$$L^{UNCLIP}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right] \tag{3}$$

Where $r_t(\theta)$ is a probability ratio obtained from the old policy (i.e., the one used for creating the trajectory) and the new one, which provides a way for controlling the updates.
TRPO uses the KL divergence between the old and the new policy, so as to control the updates through a constraint. But this adds complexity to the problem, causing also poor scalability.

The PPO idea is that it is possible to limit the policy updates by clipping the surrogate objective (3), by penalizing the updates that move the ratio $r_t(\theta)$ too far away from 1:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t) \right] \tag{4}$$

where ϵ is a constant.

The min term creates a lower (i.e., pessimistic) bound of the objective function. In this way, the probability ratio change is considered only if the update would make the objective worse, otherwise the the same TRPO objective function L^{UNCLIP} is considered.

This clipping also increases the *sample efficiency*, because it allows to perform multiple policy updates using the same trajectory.

The objective function used in PPO also considers two additional terms: the value function loss L^{VF} (i.e., squared-error loss), and an entropy bonus S. The latter is used for promoting exploration, and reducing the possibility of being stuck in a local optima.
The final expression is the following:

$$L^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S(\pi_\theta)(s_t) \right] \tag{5}$$

where c_1 and c_2 are constants used for scaling the value function loss and the entropy bonus.

1.2.2 Generalized advantage estimate

As mentioned before, the *Advantage Function* used with REINFORCE is very useful for reducing the variance, and helping the training. Nevertheless, this could introduce a bias.
PPO counters this effect by using a different technique for computing the $\hat{A}_t$ term, which is called Generalized Advantage Estimate (GAE) [5].

Essentially, GAE makes a policy update by considering multiple steps in the environment, and combining them together through an exponential average.

2 Environments

The environments used in this project have been taken from the OpenAI's Procgen Benchmark [6], which is composed by 16 different games. In particular, the model has been trained and tested on 3 games, which have an increasing complexity: *Coinrun*, *Ninja* and *Leaper*. The latter is by far the most difficult one, where the agent must be able to dodge and surpass several obstacles in order to win a match.

The *ProcgenGym3Env* object has been used in order to parallelize multiple environments at once, and for speeding up the training. Essentially, it allows to simultaneously run an agent on multiple environments for creating the trajectory, which is afterward used for doing the policy updates.

The environments provide observations in the form of 64x64 RGB images.

3 The Model

The created model is composed by 2 elements: the actor (i.e., the policy learner) and the critic (i.e., the value function learner). The OpenAI's Procgen paper [6] mentions two different architectures that can be used for creating these two elements: Nature-CNN and Impala-CNN.

As the names suggest, both these architectures are based on CNNs. Nature-CNN is a pretty simple network, while Impala-CNN is much deeper and it allows to extract more complex features, at the cost of higher computational requirements. Both these architectures have been implemented and tested for comparison purposes. Figure 1 and 2 depict their general structures.

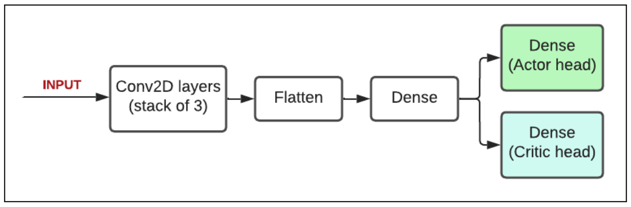

Figure 1: *Nature-CNN* overview.

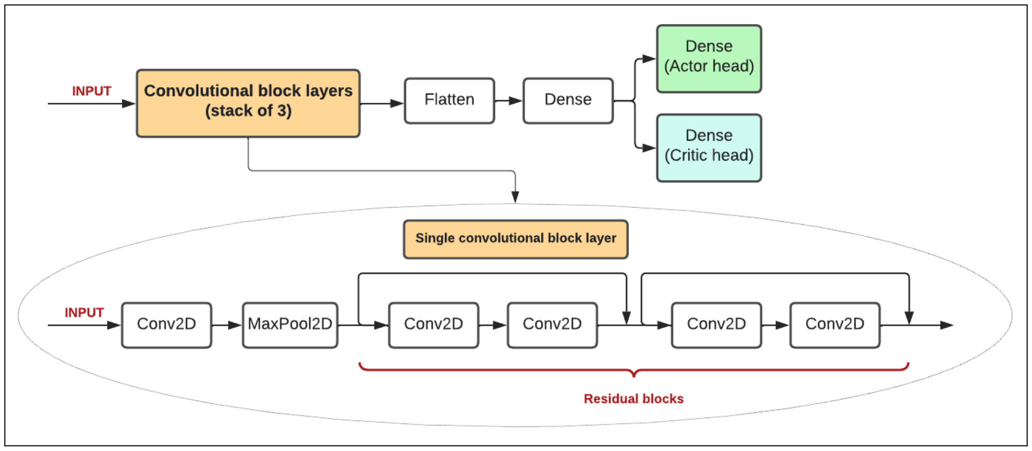

Figure 2: *Impala-CNN* overview.

4 Training

The general training procedure has been done by following the PPO paper, and it is composed by the following steps:

1. Iterate for a certain number of iterations: 768 for Coinrun and Ninja, 1536 for Leaper.
2. In each iteration create the trajectory. This is done by running 32 parallel environments, where agents can perform a maximum number of 1024 steps, while using the current policy.
3. Compute advantages (GAE) and returns for each collected episode.
4. Divide the trajectory in minibatches of 256 elements, and train the policy.
5. Only PPO: use the same trajectory for updating the policy multiple times.

Thus, the same training strategy has been used for REINFORCE and PPO, with the only exception of the last point just described. This choice has been made also for comparison purposes between the two algorithms.

4.1 Evaluation Metric

The main evaluation metric used is the *Win Ratio* (equation 6).

The implementation of this metric required alterations of the rewards, because the OpenAI's Procgen environments return a positive reward in case of a win, and a 0-reward in all other cases. Therefore, the trajectory has been modified so as to have a negative reward whenever the agent lose the game, or the game is incomplete (i.e., the agent is stuck and reaches the maximum number of time steps).

$$win\_ratio = \frac{n\_win}{n\_win + n\_loss + n\_incomplete} \tag{6}$$

4.2 Credit assignment problem

Initially, the final trajectory obtained after computing the evaluation metric (previous section) has been also used for computing returns and advantages. This strategy worked pretty well for *Coinrun* and *Ninja*, but it was destructive for *Leaper*.

Leaper is way more difficult than *Coinrun* and *Ninja*, and it was incredibly rare that the agent was able to win the game just by chance. Basically, while learning the agent was continuously losing games, and receiving only negative rewards. Therefore, the agent was able to learn only to stay still in order to maximize the rewards, and the majority of the games were just incomplete.

The winning strategy here was to remove all the negative rewards, just providing a positive reward in case of a win, and 0-rewards in all other cases. With this modification, the agents were able to improve their performances also on *Leaper*.

4.3 Other training tricks

In the literature [7, 8] there are many possible tricks could be helpful for the training of these models. For trying to improve the agents performance, some of these tricks has been implemented and tested, and what follows is a short list of those that actually brought improvements.

Batch-level advantages normalization coupled with *value function loss clipping* were beneficial, especially in the *Leaper* training stability. Without those, the *Win Ratio* was much less stable during the training, particularly after a large number of iterations.

Mini-batch samples shuffling. This technique is pretty common when training neural networks, and there are theoretical and practical proofs that state this is very often beneficial.

Layer weights *orthogonal initialization*. Layer weights initialization is generally very important, and can be foundamental in order to speed-up the training. In the experimentation, the initial phase of the training was faster when applying this kind of weight initialization.

5 Results

The training and the analysis of the results has been split in different stages, that are briefly explained in the following subsections.

5.1 Finding the best architecture and algorithm

The initial stage of training and testing was about finding the best possible architecture, mainly considering two different aspects (Figure 3).
The first is the **backbone**, and comparisons have been made between *Impala-CNN* and *Nature-CNN*. The second, and probably the most important, is related to the **algorithm**, and evaluations have been made between *REINFORCE* and *PPO*. These results depict a clear winner: *PPO* coupled with *IMPALA* is able to learn much faster, it is more stable, and it reaches higher scores.

5.2 Action learning

The OpenAI's Procgen environments give the possibility of using 15 different actions. However, in all the games that have been tested, just a few of these actions are actually required (i.e., just those for moving around in *Leaper* and *Coinrun*, and also those for firing in *Ninja*).

On the first attempts, the number of possible actions have been limited, with the idea of helping the training (i.e., limited the number of Actor's output). Nevertheless, both *PPO* and *REINFORCE* were actually able to autonomously learn which are the best actions, filtering out those that are not useful for winning a game.

Figure 4 shows this interesting aspect: during the first iterations the actions are completely random, but during the training the number of actions is heavily reduced to only those required for winning a game.

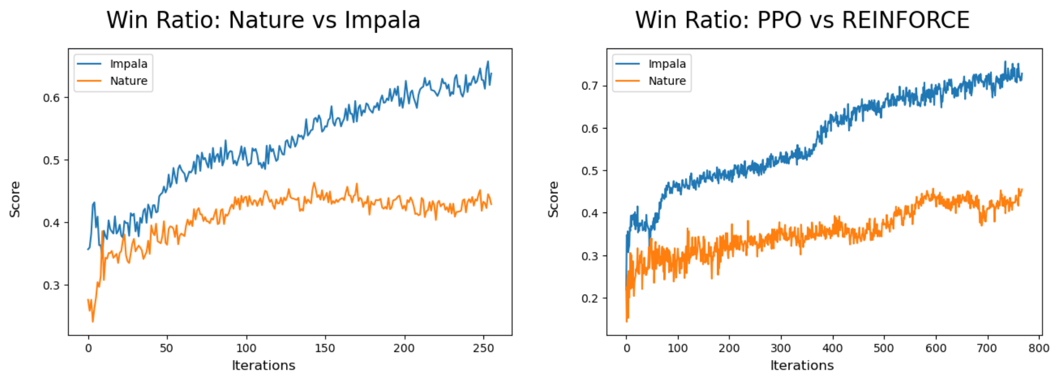

Figure 3: Comparison between different backbones and algorithms for controlling the policy updates

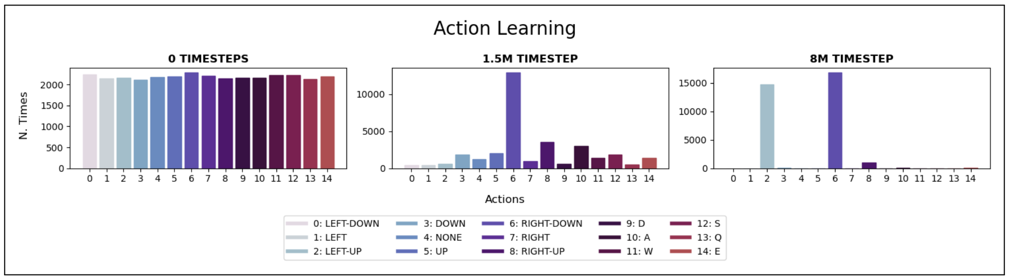

Figure 4: PPO action learning on the Ninja game

5.3 Win ratio

Finally, the best model found so far has been trained on all the three games, so as to evaluate whether the model was able to learn. The metric used for doing this evaluation is the **Win Ratio** explained before. Note that the three trainings were completely independent from each other. From these plots, it is possible to remark the following aspects:

1. The model is able to reach decent scores in every game.
2. The *Leaper* game has been trained for a larger number of iterations, so as to reach more satisfactory scores. Intuitively, this is something that could be expected, because this game is way more difficult compared to the others.

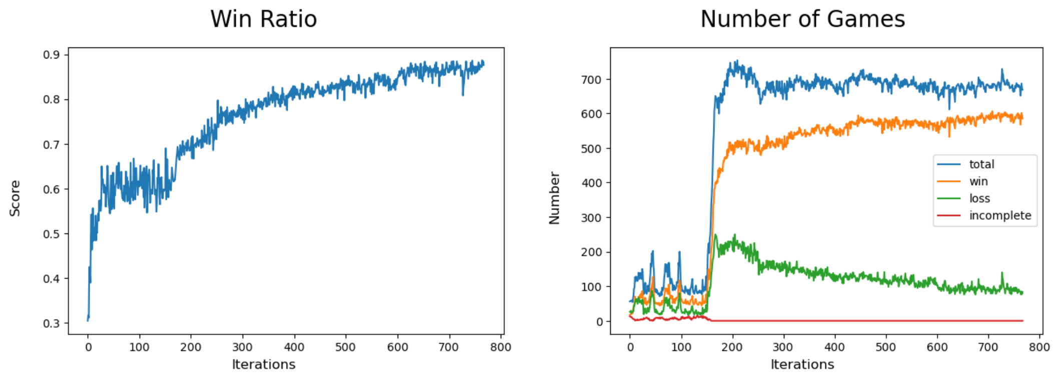

Figure 5: *Win Ratio* and *Number of games* recorded while playing **Coinrun**

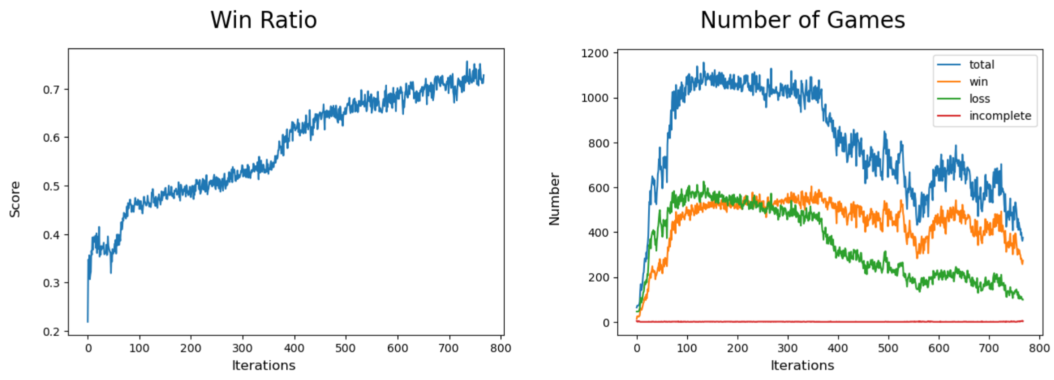

Figure 6: *Win Ratio* and *Number of games* recorded while playing **Ninja**

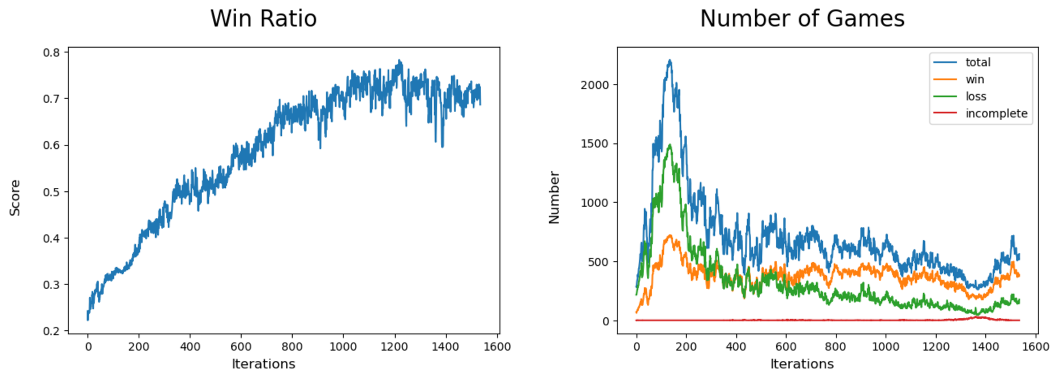

Figure 7: *Win Ratio* and *Number of games* recorded while playing **Leaper**

3. There are just a few incomplete games, typically focused on the very initial stage of the trainings. This means that the agent learns how to move in the first iterations, and then always tries to reach its goal of winning the game, and it does not stay still.
4. For each environment the number of games is initially very small. After some iterations, the agent is capable of moving around, and therefore the number of games increases.

6 Conclusions

Although the obtained results appear to be satisfactory, there is undoubtedly space for improvements.

As an example, in the *Ninja* game the *Firing* actions are rarely used, and most likely it would be necessary to promote more exploration (e.g., ϵ-greedy, or other more sophisticated techniques). Moreover, all the games have been played in the *easy* difficulty, and some tests should be done also using harder environments to assess how the model behaves. Additionally, new and alternative techniques have been proposed, such as [9], and could be interesting to implement them so as to do comparisons. Anti-overfitting techniques (e.g., dropout) could be beneficial for generalization, and possibly improve the model behaviour. Furthermore, a stack of observations [10] could be beneficial to surpass obstacles that require information about speed and direction of objects.

Nevertheless, this project gave the opportunity of implementing and testing two very widely used algorithms, and of deeply studying how they work. Moreover, it provided the chance of understanding the importance of the *Credit Assignment Problem*, which was the main cause of no-learning in the *Leaper* game. Finally, it has been really amazing seeing an agent going from a completely random behaviour, to a behaviour that seems actually intelligent.

References

[1] Richard S. Sutton, et al. (1998) Introduction to Reinforcement Learning. MIT press.

[2] Ronald J Williams. (1992) Simple statistical gradient-following algorithms for connectionist reinforcement learning (https://link.springer.com/content/pdf/10.1007/BF00992696.pdf)

[3] John Schulman, et al. (2017) Proximal Policy Optimization Algorithms (https://arxiv.org/pdf/1707.06347.pdf)

[4] John Schulman, et al. (2017) Trust Region Policy Optimization (https://arxiv.org/pdf/1502.05477.pdf)

[5] John Schulman, et al. (2018) HIGH-DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION (https://arxiv.org/pdf/1506.02438.pdf)

[6] Karl Cobbe, et al. (2020) Leveraging Procedural Generation to Benchmark Reinforcement Learning (https://arxiv.org/pdf/1912.01588.pdf)

[7] Huang Shengyi, et al. (2022) The 37 Implementation Details of Proximal Policy Optimization (https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/)

[8] Marcin Andrychowicz, et al. (2020) What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study (https://arxiv.org/pdf/2006.05990.pdf)

[9] Karl Cobbe, et al. (2020) Phasic Policy Gradient (https://arxiv.org/pdf/2009.04416.pdf)

[10] Volodymyr Mnih, et al. (2013) Playing Atari with Deep Reinforcement Learning (https://arxiv.org/pdf/1312.5602.pdf)