Proximal Policy Optimization: an Implementation Study

Manuel Mariani
University of Bologna
manuel.mariani2@studio.unibo.it

Abstract

The following is an implementation study on Proximal Policy Optimization, applied to the CoinRush OpenAI Gym environment. This practical study focuses on implementing a readable implementation of PPO, while also empirically exploring some of the algorithmic improvements that are overlooked in the original paper. After implementing the proposed improvements, the agent is able to quickly learn how to traverse the environment, achieving promising results.

1 Introduction

1.1 Goals & Requirements

The objective of the project is to apply Reinforcement Learning techniques to teach an agent to autonomously solve a simple video-game. The requirements and desired outcomes are:

- The algorithm should be efficient, such that it could be run and trained on consumer-level hardware.
- The algorithm should be reasonable fast, such that training, testing and hyper-parameter tuning could be done in reasonable time, ideally in less than 20 minutes to see performance improvements in the policy.
- The implementation should be readable and results reproducible.
- The final trained agent should have adequately good performance in the testing environment.

1.2 The environment

The environment used to assess the performance of the algorithm is CoinRun[3], in particular the version implemented in Procgen [2]. The environment is a simple platforming game, where an agent must reach the end of the level, represented as a coin, while avoiding a number of obstacles and performing well timed jumps.

At each timestep the environment provides as observation the raw frame data, centered on the agent, as a $64 \times 64 \times 3$ RGB image. The agent can perform 4 possible actions: moving left or right, jumping or waiting. The environment also provides a reward signal only if the episode is complete, that is when the agent reaches the coin.

This makes the environment not trivial to train an agent on, due to the fact that the observation space is quite large and rewards are also sparse.

1.3 The algorithm

To train the agent, Proximal Policy Optimization [12] is used. PPO is an on-policy gradient algorithm, where a surrogate objective is optimized. The main concept behind it is to limit the ratio between the

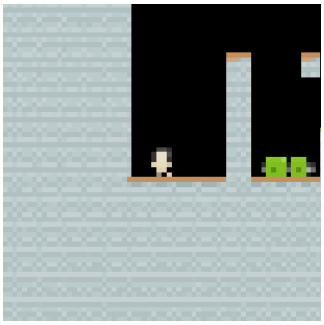

Figure 1: Example of raw frame data observation.

old policy (the one used to generate the training data) and the new learnt policy by clipping the ratio between the two.

PPO is also based on the Actor Critic[9], where an estimator of the expected returns of the current state is used to improve the performance.

Since PPO is an on-policy algorithm, it also benefits from an entropy regularization term to encourage exploration in the training phase, while also discouraging the agent from learning sub-optimal actions too quickly.

1.4 The model

The agent uses a variation of the IMPALA[7] network architecture. The network is composed by:

- A **convolutional feature extractor**, where the $64 \times 64 \times 3$ image is reduced to a vector of 256 features.
- A linear **policy network** where from the extracted features it outputs the probability distribution of the 4 actions.
- A linear **value network** where from the extracted features a single value representing the estimated expected returns of the current state is produced.

The main differences with the original network are:

- No memory layer (LSTM) used.
- Tanh activation on the last feature extractor layer instead of ReLU.
- Batch normalization at the start of each IMPALA block.

2 Implementation

Unfortunately, by only applying the PPO algorithm found in [12] the agent does not reach adequate levels of performance in reasonable time. Many studies have been performed analyzing in detail how much of the PPO paper performance comes from the algorithm itself versus other unspecified algorithmic improvements [8, 6]. In addition, the study conducted by Marcin Andrychowicz et al.[1] explores further algorithmic improvements and hyper parameter choices.

2.1 Algorithmic Improvements

In this section, a collection of tested improvements is listed. Improvements with an empirical positive impact are listed with ✓, while those with negative impact are listed as ✗.

2.1.1 Vectorized Environment ✓

The environment is parallelized, reducing the time needed to collect multiple trajectories. Instead of computing one long sequential list of episodes, it parallelizes shorter sequences of episodes.

2.1.2 Reward standardization ✓

Rewards are standardized by subtracting their mean and standard deviation. The reward statistics are not episode-based, but instead they are tracked using the Welford Algorithm[5], to obtain a better estimation of the true mean and standard deviation of the reward signal over the course of the whole training.

2.1.3 Generalized Advantage Estimation ✓

Instead of using the N-step return, Generalized Advantage Estimation[11] is used.

2.1.4 Reward Shaping ✗

It is possible to encourage the agent to perform in a certain manner by manipulating the reward signal it receives. In particular we want the agent to:

1. Reach the end of the level
2. Avoid losing by hitting obstacles
3. Reach the end of the level as quickly as possible

These requirements can be formalized by appropriately setting positive or negative value to the reward signal when the conditions are met.

In practice this approach doesn't work easily, because the choice of the shaped rewards values is very sensitive and impacts training dramatically. In particular, negative rewards for hitting obstacles can cause the agent to prefer "stalling" instead of exploring the environment, resulting in a performance degradation due to the less collected samples. Additionally, objective 3 can be achieved by reward discounting, thus this approach has been discarded in favor of the default rewards provided by the environment.

2.1.5 PPO-style value loss ✗

In the PPO implementation, the value loss is clipped in a way similar to the CLIP loss. Instead, better performance can be achieved by using the traditional MSE loss between the predicted value and the actual returns[1].

2.1.6 Fixed training seed ✓

Instead of randomly regenerating new initial states, the training is performed on always the same initial states. This allows the agent to learn faster, due to the fact that the episodes is explored multiple times, increasing the stability of the training. Note that since that training should be reasonably fast, as specified in section 1.1, few training steps are performed. If longer steps are used, it is beneficial to periodically reset the starting state seeds to improve generalization.

2.1.7 Frame stacking ✗

Frame stacking changes the input of the agent where, instead of the current frame, the agent observes the last ϕ frames of the episode. In the papers [12, 8] when training on Atari frame stacking is performed by first reducing the image from RGB to grayscale, and then stacking the last $\phi = 4$ greyscale frames. This results in an input of size 84x84x4.

Experimentally this approach didn't work as expected, because many of the obstacles in the CoinRun environment are color-coded and can become difficult to distinguish in grayscale. A better approach could have been to not reduce the images to grayscale, obtaining a 64x64x16 observation input, or only convert to grayscale the past observations to obtain a smaller 64x64x7 input. Those two approaches were not tested, due to increase in computation cost and training times.

2.1.8 Training ϵ-greedy ✓

By default the PPO policy chooses an action by sampling the distribution outputted by the policy-network. To further regularize the training and enforce exploration, a simple variation of ϵ-greedy

policy is used, where a random action is used with probability ϵ or a sample from the distribution with probability $1 - \epsilon$.

2.1.9 Prioritized experience replay ✓

To speed up training, instead of uniformly sampling all the collected experiences, it is possible to sample the experiences based on how unexpected and meaningful they are. To do so, a slight variant of Prioritized Experience Replay[10] is used, where each transition has a probability of being picked equal to

$$P(i) = \frac{p(i)^\alpha}{\sum_k p(k)^\alpha}$$
$$p(i) = |\text{Advantage}(i) + \epsilon|$$

where ϵ and α are parameters that determine the uniformity of the distribution.

2.1.10 Minibatch repeated training ✓

Instead of performing backpropagation using the whole collected trajectories, the trajectories are first shuffled, then partitioned into M minibatches. Also the backward pass is performed N times over the entire batch, effectively performing $M \times N$ backward passes per training step.

2.1.11 Advantages standardization ✗

When computing the loss, the original PPO implementation standardizes the advantages over the minibatch. This has been proven to be not beneficial to the training [1].

2.1.12 Orthogonal initialization ✓

Neural network weight initialization has been proven to be an important factor in many tasks, including reinforcement learning. As such it has been found that by initializing the policy network weights with an orthogonal initialization and standard deviation 0.01 and with biases set to 0, the agent performance increases [1, 8]

2.1.13 Learning rate scheduler and gradient clipping ✓

Training stability can be slightly improved by using a decaying learning rate scheduler. In addition, to avoid exploding gradients, gradient clipping is used.

3 Results

3.1 Experiment setup

To evaluate the performance of the algorithm, each training step is alternated with a validation step. The two steps are run on different environments with fixed seeds, to better grasp the training progress. Training took from start to finish only 40 minutes, using the hyperparameters and hardware described in Appendix A. All plots shown are smoothed using exponential smoothing with weight $\alpha = 0.1$, and can be interactively explored in the following link, alongside other plots from the experiment.

3.2 Agent win rate

From figure 2 we can see that after an initial exploratory phase (step < 20), the agent quickly learns how to navigate the environment. In the initial phase, the agent moves slowly and randomly with some sporadic wins but mostly unfinished episodes. After that, the agent quickly learns how to navigate the environment and learns to complete and win most of the simple episodes, while still struggling on harder episodes with more obstacles.

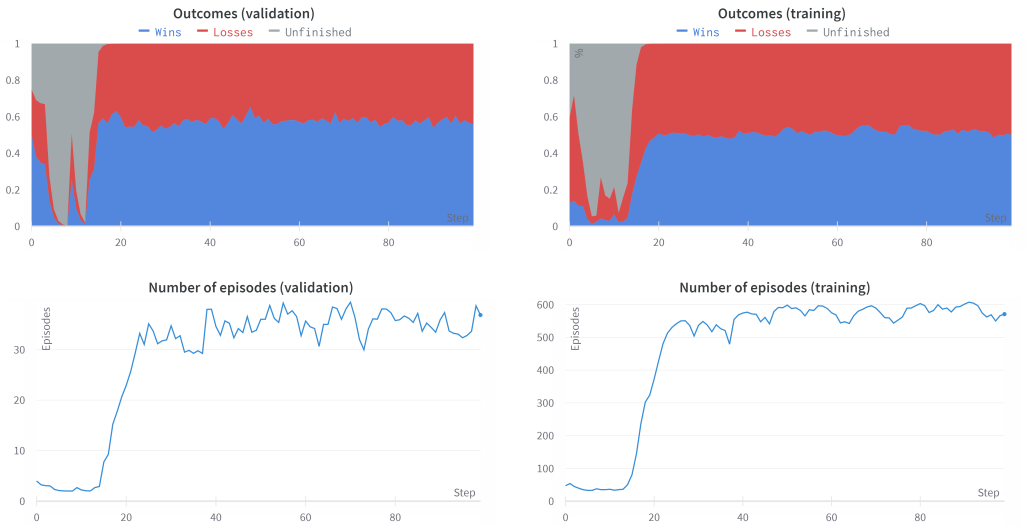

Figure 2: Top row: distribution of episodes won, lost and unfinished, per step. Bottom row: total number of episodes per step.

In summary:

- On the training environment: agent starts with 12% win rate over 47 explored episodes, and reaches a peak of 55% win rate over 600 explored episodes.
- On the validation environment: agent starts with 50% win rate over 4 explored episodes, and reaches a peak of 65% over 40 explored episodes.

3.3 Action distribution

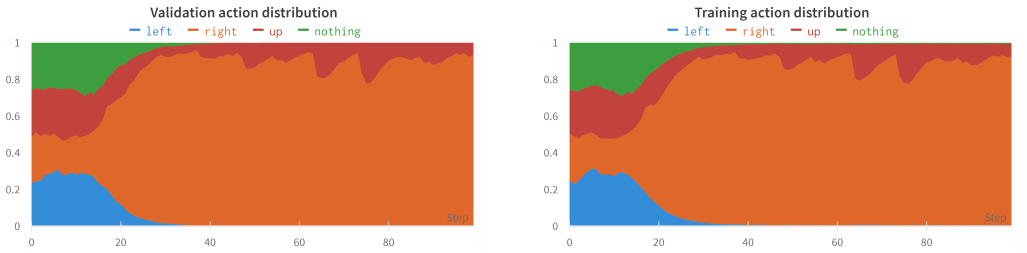

Figure 3: Action distribution in validation and training environment over training step.

From figure 3, we can see that initially the agent's action distribution is quite uniform, akin to a random agent. After episode 20, the agent quickly learns to do the actions `right` and `up`, which are the most valuable actions to reach the goal. Most interestingly, the agent almost completely avoids performing the actions `nothing` and `left`, suggesting strong overfitting despite the proposed regularization techniques.

4 Further improvements

While running the experiments, it has been found that this algorithm is extremely sensitive to the choice of hyperparameters. As such, automatic hyper-parameter tuning, in combination with early stopping can be used to further improve the scores.

Also the training has been performed only for 100 epochs, to emphasize the algorithms capabilities in quickly learning a policy. By training for longer and with better regularization, for example with Dropout Layers, Noise and environment seed reset, a more generalized and performant policy can be learnt.

Finally, it has been proven in [1, 8] that using different policy and value networks greatly stabilizes training and improves results. This can be achieved with the current algorithm by implementing two separate networks, but it greatly increases the computation cost and training times. A promising alternative that doesn't increase the computation cost would be to use Phasic Policy Gradient[4] — a PPO variant that splits the policy and value function training in two different non-interfering phases — alongside with the algorithmic improvements discussed in this study.

In addition, while end-to-end reinforcement learning has proven to be a viable solution to some problems, a better approach could have been to train an object detection network or a self-supervised feature extractor autoencoder, and feed its outputs to a reinforcement learning agent. This approach could be even integrated in the PPG algorithm, alternating policy-value-feature training regimes to achieve better stability.

5 Conclusion

Overall, considering the low number of training steps, the implementation achieves very good results and a good margin for improvement. The algorithms implementations were not trivial, but by far the hardest step, which was not documented in this study for brevity, was hyperparameter tuning. Proximal Policy Optimization turned out to be quite sensitive to the hyper parameters and neural network initialization, which suggest that many trials with longer trainings are needed to achieve state of the art performance.

References

[1] Marcin Andrychowicz et al. *What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study*. 2020. DOI: 10.48550/ARXIV.2006.05990. URL: https://arxiv.org/abs/2006.05990.

[2] Karl Cobbe et al. *Leveraging Procedural Generation to Benchmark Reinforcement Learning*. DOI: 10.48550/ARXIV.1912.01588. URL: https://arxiv.org/abs/1912.01588.

[3] Karl Cobbe et al. *Quantifying Generalization in Reinforcement Learning*. 2018. DOI: 10.48550/ARXIV.1812.02341. URL: https://arxiv.org/abs/1812.02341.

[4] Karl W Cobbe et al. "Phasic policy gradient". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 2020–2027.

[5] Andrey A Efanov, Sergey A Ivliev, and Alexey G Shagraev. "Welford's algorithm for weighted statistics". In: *2021 3rd International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE)*. IEEE. 2021, pp. 1–5.

[6] Logan Engstrom et al. *Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO*. 2020. DOI: 10.48550/ARXIV.2005.12729. URL: https://arxiv.org/abs/2005.12729.

[7] Lasse Espeholt et al. *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*. 2018. DOI: 10.48550/ARXIV.1802.01561. URL: https://arxiv.org/abs/1802.01561.

[8] Shengyi Huang et al. "The 37 Implementation Details of Proximal Policy Optimization". In: *ICLR Blog Track*. https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/. 2022. URL: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

[9] Vijay Konda and John Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems* 12 (1999).

[10] Tom Schaul et al. *Prioritized Experience Replay*. 2015. DOI: 10.48550/ARXIV.1511.05952. URL: https://arxiv.org/abs/1511.05952.

[11] John Schulman et al. "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438* (2015).

[12] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

A Appendix

A.1 Hardware

Laptop with Intel(R) Core(TM) i7-10750H CPU, 16.0 GB RAM, NVIDIA 3070 Mobile GPU.

A.2 Hyperparameters

Parameter	Value
Environment	
Difficulty	Easy
Parallel agents (train)	32
Parallel agents (validation)	2
Max steps per agent	1000
Training parameters	
Max Steps	100
Optimizer	Adam
Learning rate	2e-5
Learning rate decay	0.99
Batch size	512
Training epochs per step	2
Random action choice ϵ	0.02
Prioritized sampling α	0.5
Prioritized sampling ϵ	0.01
Returns γ	0.99
GAE λ	0.9
PPO	
PPO ϵ	0.2
Entropy coefficient	0.01
Value function coefficient	0.5