# Proximal Policy Optimization

## Deep Reinforcement Learning

## MVA 2021-2022

**Théo VINCENT**
theo.vincent@eleves.enpc.fr

**Nicolás VIOLANTE**
nicolas.violante@ens-paris-saclay.fr

**Pauline SERT**
pauline.sert@ens-paris-saclay.fr

**Elías MASQUIL**
elias.masquil@ens-paris-saclay.fr

## Abstract

This project focuses in the theory and the implementation of Proximal Policy Optimization (PPO) [7], which aims to overcome the poor sampling efficiency of traditional Policy Gradient methods by introducing a clipping loss to enforce new policies to be close to the sampling policy. This allows to reuse the same sampled trajectory to perform several policy updates.

## 1 Introduction

Traditional Policy Gradient methods suffer from poor sampling efficiency. They iterate between two phases: sample a full trajectory under the current policy (rollout phase) and then use it to perform a single gradient update on the parameters of the policy network (learning phase). This is clearly inefficient. One could try to reuse the same trajectory and perform several gradient updates with the same data, but naively doing so just does not work because Policy Gradient methods are on-policy. In the learning phase, after the first gradient update we end up with a new policy. The following updates do not have any theoretical justification and indeed this performs poorly in practice.

To overcome this issue, Proximal Policy Optimization (PPO) [7] propose to perform several gradient updates with the same trajectory while constraining the new policies to be close to the one used during the rollout phase. This way the training becomes stable.

## 2 Proximal Policy Optimization

In this section we describe the main components of PPO and some implementation details needed to make it work properly. For the full algorithm please refer to algorithm 1. The "tricks" that we implemented are coming from the the work of S. Huang et. al. [5, 4], P. Dhariwal et. al. [2], A. Raffin et. al. [6] and J. Achiam et. al. [1].

**Algorithm 1** Proximal Policy Optimization

**Inputs:** n_training_iterations: number of training iterations, $\rho$: initial state distribution, n_rollout: number of samples to collect, $\mathcal{D}$: replay buffer, n_epoch: number of epochs, $B$: batch size
$\theta$ initial weights for value network, $\phi$ initial weights for policy network
$\alpha_\pi$: policy learning_rate, $\alpha_v$: value learning_rate

**for** t = 1, . . . , n_training_iterations **do**
  $\mathcal{D} = \emptyset$
  $S_1 \sim \rho$
  **for** t = 1, . . . , n_rollout **do** ▷ Rollout Phase (2.2)
    $V_t = v_\phi(S_t)$
    $(\mu_t, \sigma_t) = \pi_\theta(S_t)$
    $A_t \sim \mathcal{N}(\mu_t, \sigma_t^2)$
    $S_{t+1}, R_{t+1}, D_{t+1}$ = step$(V_t, A_t)$
    $\mathcal{D} = \mathcal{D} \cup \{\big(S_t, V_t, \log \mathcal{N}(A_t; \mu_t, \sigma_t^2), A_t, R_{t+1}, D_{t+1}\big)\}$
    **if** $D_{t+1}$ is True **then**
      $S_t \sim \rho$
    **else**
      $S_t = S_{t+1}$

  Add the GAE to the transitions of $\mathcal{D}$ ▷ Generalized Advantage Estimation (2.3)

  **for** i = 1, . . . , n_epoch **do** ▷ Learning Phase
    shuffle$(\mathcal{D})$
    **for** $\mathcal{B}$ in $\mathcal{D}$ **do**
      $S_k, A_k, V_k, \log \mathcal{N}(A_k; \mu_k, \sigma_k^2), \mathcal{A}_k = \mathcal{B}(k)$ for $k = 1, \ldots, B$
      ▷ Value loss (2.4)

$$\mathcal{L}_{value}(\phi) = \frac{1}{B} \sum_{k=1}^{B} (v_\phi(S_k) - (V_k + \mathcal{A}_k))^2$$

      $\phi \leftarrow \phi - \alpha_v \nabla_\phi \mathcal{L}_{value}$
      ▷ Policy loss (2.5)
      $(\bar{\mathcal{A}}_k)_k$ = normalize$((\mathcal{A}_k)_k)$
      $\mu_k(\theta), \sigma_k(\theta) = \pi_\theta(S_k)$
      $r(\theta) = \exp(\log \mathcal{N}(A_k; \mu_k(\theta), \sigma_k^2(\theta)) - \log \mathcal{N}(A_k; \mu_k, \sigma_k^2))$
      $r_\epsilon(\theta) = \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon)$

$$\mathcal{L}_{policy}(\theta) = -\frac{1}{B} \sum_{k=1}^{B} \min(r(\theta) \cdot \bar{\mathcal{A}}_k, r_\epsilon(\theta) \cdot \bar{\mathcal{A}}_k)$$

      $\theta \leftarrow \theta - \alpha_\pi \nabla_\theta \mathcal{L}_{policy}(\theta)$

### 2.1 Neural Networks

For the policy network, we use a 2-layer MLP with 64 units. We experiment both with relu and tanh activations. For some experiments we include a final tanh activation. In the following, the policy network will be noted $\pi_\theta$. For a state $s$, it outputs the mean $\mu$ and the variance $\sigma$ of a gaussian distribution i.e. $(\mu, \sigma) = \pi_\theta(s)$. Then, the sampling policy will be denoted by $\pi_{\widetilde{\theta}}$. It samples the out coming action $a$ with $a \sim \pi_{\widetilde{\theta}}(s)$ where $\pi_{\widetilde{\theta}}(s)$ is the gaussian law $\mathcal{N}(\mu, \sigma^2)$. Once the training is complete, we commit to a greedy policy by selecting the maximal action $a = \mu$.

For the value network $v_\phi$ we also use a 2-layer MLP with 64 units and tanh activations.

### 2.2 Rollout phase

The first step of PPO is to sample a sequence of transitions $\tau = S_1, A_1, R_2, S_2, \dots$ using the sampling policy for the current weights $\theta_{old}$, which we denote $\pi_{\widetilde{\theta_{old}}}$. This sequence can be composed of several episodes with the last one not even finished. For each state $S_t$ we estimate its value with the value network, $v_\phi(S_t)$. During this phase we also compute the log-probabilities $\log \pi_{\widetilde{\theta_{old}}(A_t|S_t)}$, which we will need for the clipping loss during the learning phase. Since we are using gaussian distributions, the log-probability is just the density evaluated at the action $A_t$, $\log \mathcal{N}(A_t; \mu, \sigma^2)$ where $(\mu, \sigma^2) = \pi_{\theta_{old}}(S_t)$

### 2.3 Generalized Advantage Estimation

First we consider the n-step bootstraped approximations $\mathcal{A}_t^{(n)}$ of the advantage using the estimations $V_t$ we got with our value network and the rewards we collected.

$$\begin{aligned}
\mathcal{A}_t^{(1)} &= R_{t+1} + \gamma V_{t+1} - V_t \\
\mathcal{A}_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+2} - V_t \\
&\vdots \\
\mathcal{A}_t^{(\infty)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + \dots - V_t
\end{aligned}$$

We can combine all these estimators into a single one, the so-called Generalized Advantage Estimator

$$\mathcal{A}_t = (1-\lambda)\left(\sum_{l=1}^{\infty} \lambda^{l-1} \mathcal{A}_t^{(l)}\right)$$

This estimator has the advantage of combining low variance and high bias estimators (small $n$) with high variance and low bias estimators (large $n$)

By introducing the temporal difference $\delta_t = R_{t+1} + \gamma V_{t+1} - V_t$ and recognizing a geometric sum on $\lambda$, we can rewrite the Generalized Advantage Estimator as

$$\mathcal{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

In practice we use a truncated version of this expression given by the length of the sampled trajectories in the sequence of transitions $\tau$ collected during the rollout phase. Moreover, during the learning phase, these estimators are normalized with the mean and variance of the estimators of the batch for the policy loss. For the value loss we use them unnormalized.

### 2.4 Value loss

The value loss is the mean squared error between the current value $v_\phi(S_k)$ and the sum of the value predicted during the rollout phase $V_k$ and the advantage $\mathcal{A}_k$. To have the intuition on why this sum has to be done, we can take the case of a trajectory of length one, the advantage is then: $\mathcal{A}_k = R_{k+1} + \gamma V_{k+1} - V_k$, then the target is equal to $R_{k+1} + \gamma V_{k+1}$ which is equal to the bootstrapped version of the return which is what the network has to learn.

### 2.5 Policy loss

To enforce new policies $\pi_{\widetilde{\theta}}$ to be similar to the sampling policy $\pi_{\widetilde{\theta_{old}}}$ we consider a clipped loss based on the ratio $r(\theta) = \frac{\pi_{\widetilde{\theta}}(a|s)}{\pi_{\widetilde{\theta_{old}}}(a|s)}$. For similar policies, this ratio should be $r(\theta) \simeq 1$ for all actions and

states. In order to penalize large deviations between policies, we first clip the ratio into the interval $(1-\epsilon, 1+\epsilon)$, obtaining $r_\epsilon(\theta) = \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)$ and then consider the followig loss function

$$\mathcal{L}_{policy} = \mathbb{E}_{s,a\sim\pi_{\widetilde{\theta_{old}}}}\left[\min\left(r(\theta)\cdot\mathcal{A}^{\pi_{\widetilde{\theta_{old}}}}(s,a), r_\epsilon(\theta)\cdot\mathcal{A}^{\pi_{\widetilde{\theta_{old}}}}(s,a)\right)\right] \tag{1}$$

where $\mathcal{A}^{\pi_{\widetilde{\theta_{old}}}}(s,a)$ is the advantage function of the policy $\pi_{\widetilde{\theta_{old}}}$. In practice we use an estimator of the advantage function based on our value network.

# 3 Experiments

## 3.1 Methodology

To prevent technical and functional bugs, we did the following:

- Established good coding practices using PR's, Issues and Discussions.
- Monitored metrics and values during training on a TensorBoard:
  - **Per timestep:** reward, mean, episodic return, and standard deviation of the distribution over the actions, the action taken.
  - **Per training iteration:** average episodic return (average of all the episodes' returns collected during the current training step)
  - **Per epoch:** value loss, policy loss, Kullback-Leibler divergence, the learning rates of the policy and value networks
  - We included TensorBoards and videos of the agents in our repository
- Implemented tests for critical components of our project such as the `FixedReplayBuffer` and the `general_advantage_estimation`.
- Reduced the time complexity of the algorithm in order to iterate faster on hyperparameter tuning. We optimized as much as we could the rollout phase and the sampling of batches during training, switching between NumPy and Jax as needed and 'jitting' heavy calculations, such as the forward and backward pass of the neural networks.

## 3.2 Implementation details

Here we present a list of implementation details. Depending on the experiment, some of them weren't included. In the results section we specify the setting for each experiment. Across all the experiments we used: clipped gradients, the same networks initialization, learning rate annealing, and minibatches. The exact configuration for each of the best agents can be found in the repository of the project.

- **Environment wrappers**: normalization and clipping of the observation and the rewards. Also action normalization so the agent can predict actions between $[-1, 1]$, and the wrapper scale them back to the environment action range.
- **Learning rate annealing**: we implement linear learning rate annealing. Our implementation supports the usage of two different learning rates: one for the actor, and other for the critic.
- **Networks initialization**: Orthogonal initialization of the weights and constant initialization for the biases.
- **Final tanh activation**: In the cases where the actions are normalized, we can add a final tanh activation to the policy network, in order to constrain the mean and the variance of the policy so that they do not explode.
- **Clipped gradients**: To further stabilize training, we clip the gradient updates for both losses. This is particularly useful to prevent exploding the value loss when working with unnormalized rewards.

- **Fixed policy variance**: In some experiments we fix the variance of the policy, so the actor only needs to predict the mean. In the cases we predicted the variance, we follow a similar same strategy as in [3]. Then we apply a softplus activation and scale it by a given factor, we also add a small constant to keep a minimum variance during all training.
- **Minibatches**: Our implementation uses mini-batches to compute the gradient and update the policy instead of the whole batch.
- **Advantage normalization**: The advantages are normalized in all minibatches for computing the policy loss.

# 4 Results

In this section we summarize our results and comment on the different variants we tried to improve the agent's training . The full training log of the best agents can be found in the project's repository.

## 4.1 InvertedPendulum-v2

We observed two important hyperparameters to stabilize training in this environment. Firstly the learning rate. We originally started with $10^{-3}$ but after observing a divergent policy loss we set it to $10^{-5}$ and achieved better results. In both cases we included a linear annealing scheduler for the learning rate. Secondly, fixing the policy network $\sigma$ to a small value. We found that setting it according to the range of the actions, in particular $\sigma = 0.05(a_{max} - a_{min})$ works well, but using larger values produced divergent training.

We obtain basically the same results with and without reward normalization. However, we verify that the value loss increases significantly when the rewards are not normalized, so we believe it's safer to include this normalization. Nevertheless, we also clip the value loss's gradients to prevent exploding gradients.

For this environment we found thtat using action normalization and an extra tanh layer doesn't work (at least with the other set of hyperparameters); the agent wasn't able to learn anything useful.

Figure 1 shows the training progress without action normalization, and with both reward and observations normalization.

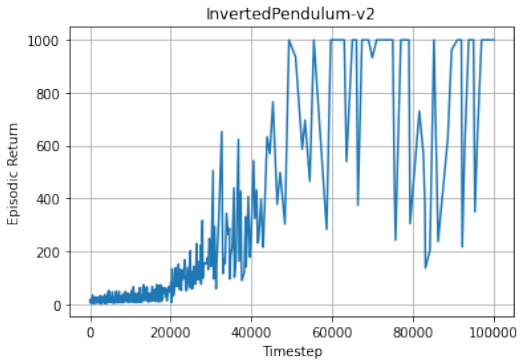


Figure 1: Episodic return for InvertedPendulum-v2. Reaching a 1000 returns means that the agent managed to complete the game.

## 4.2 Reacher-v2

For Reacher-V2 we compare different implementations: fixed std, learned std, normalized, and unnormalized actions. Here we show the comparison between these approaches by evaluating the average episodic return at each training iteration. The average is taken among all the episodes collected in the same training iteration (without a gradient update).

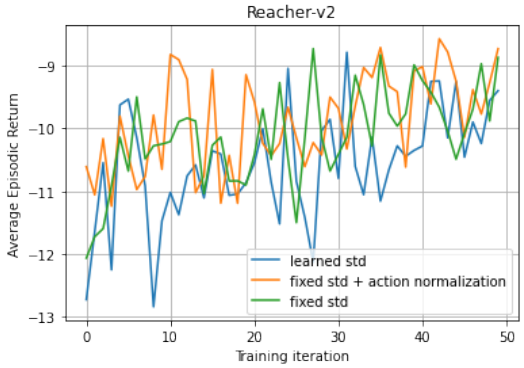


Figure 2: Average episodic return for Reacher-v2 per training iteration. Comparison between learned std, normalized, and unnormalized actions.

The best results for Reacher are obtained using normalized actions and fixed std.

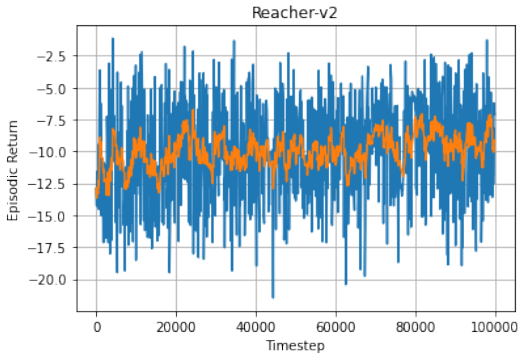


Figure 3: Episodic return for Reacher-v2. .

During the final evaluation after training was completed, the agent achieved an average score of $-8.6$ over 10 different episodes.

### 4.3 Pendulum-v0

This environment was by far the hardest we faced. With the proposed time budget of 100k interactions we couldn't get great results. Out of all our trials, only after one particular execution we managed to get one test episode in which the agent could balance the pendulum.

In this case we found that normalizing actions and adding a final tanh layer was helpful to the model. We also used reward normalization, observations normalization. Maybe we could have got better results by training for more steps or adding more epochs to each training iteration, as we found on some reference implementations.

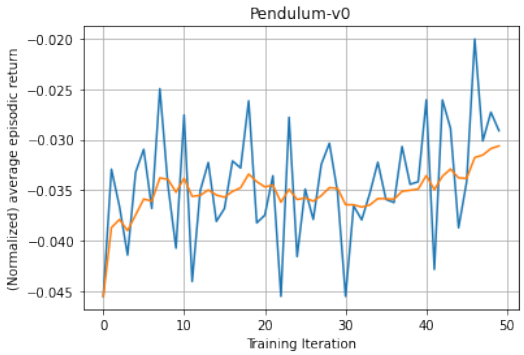


Figure 4: Average episodic return for Pendulum-v0.

## 5 Conclusion

PPO is one of the standard Deep Reinforcement Learning algorithms in the literature. It introduces several improvements respect to other PG methods, and achieves good results in a wide variety of environments. However, implementing it is far from being straightforward. Most of the available implementations, including the original one, rely on a set of tricks that are important to make it work. Most of these tricks are not mentioned in the original paper but are key to make PPO work.

Moreover, tricks and hyperparameters that work well in one environment doesn't necessarily generalize to other environments. It's necessary to carefully monitor the training process and tune parameters for each environment in order to get a good agent.

With this project we got a deep understanding of PPO and what does it take to make it work. It's not enough to grasp the theoretical details but it's also important to apply good coding practices, and be mindful of the implementation details for stable training.

## References

[1] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

[2] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[3] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.

[4] S. Huang. The 32 Implementation Details of Proximal Policy Optimization (PPO) Algorithm. `https://costa.sh/blog-the-32-implementation-details-of-ppo.html`.

[5] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, and Jeff Braga. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. 2021.

[6] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.