Introduction to Reinforcement Learning

February 2023

Proximal Policy Optimization Algorithms

Final Report

Master 2 - Data Science

Lilian BISCARRAT

Alexandre CHAUSSARD

Théo MORVAN

Contents

1 Introduction

Given a reinforcement learning framework, the main challenge is to determine the right policy to predict the best action our agent can take given the state of the environment it evolves in.

Finding a good policy can be done namely through optimization algorithms of the value function (or the action value function), which characterises the policy quality, or directly by optimizing a policy living in a given class of functions, which is called policy optimization.

In this report, we are interested in the latter situation: we define the policy as a given parameterized function living in the neural network functions class. We aim at finding the best set of policy parameters relatively to the reward the agent can get at each state. To achieve this, we will rely on the Proximal Policy Optimization (PPO) algorithm [1] introduced by OpenAI.

The PPO algorithm is one of the current state of the art policy optimization algorithms when it comes to neural net-like policies, as it is scalable, data efficient, quite robust, and without much hyperparameters tuning. It is also fairly easy to implement and understandable since it only uses first-order optimization techniques, which makes it a great candidates compared to other leading policy optimization algorithms like Trust Region Policy Optimization (TRPO) [2].

Therefore, the objective of this report is to explain the fundamental bricks of the PPO algorithm, set the frame of work, and to explain the mathematical foundations of the algorithm. Then, we will illustrate the characteristics of the PPO algorithm through an implementation of ours applied to various reinforcement learning environments. We will namely show the impact of the parameters on the performance of the algorithm, and compare the PPO to other optimization methods like Prolicy Gradient Optimization (PGO), Actor-Critic, or TRPO.

Our implementation of the various algorithms can be found on GitHub:

https://github.com/LiliBISC/RL

It includes the classical REINFORCE optimizer, Actor-Critic (A2C), TRPO and PPO in Actor-Critic style. Our experiments can be replicated through the *sandbox* folder and the various *playground* scripts into it.

2 Fundamentals of Policy Gradient Optimization

2.1 Application domain and notations

Before we get any further on policy optimization, we need to define the framework on which our policy optimization algorithm is applicable:

- $\mathcal{A}$ is a continuous or discrete space of actions the agent can take.
- $\mathcal{S}$ is a continuous or discrete space of states the environment can be into.
- π_θ is a given policy (possibly stochastic) parameterized by θ, differentiable relatively to θ. In our framework, we decide that π_θ belongs to the class of functions defined by any neural network of parameters $\theta = (W_i, b_i)_{i \in \{1,\ldots,L\}}$, so it matches the differentiability statement. One may then denote by $\pi_\theta(a|s)$ the probability of taking the action a at state s.

We will denote by $a_t \in \mathcal{A}$ the action of the agent at time t, $s_t \in \mathcal{S}$ the state of the environment at time t, and $r_t \in \mathbb{R}$ the obtained reward at time t.

For the described algorithms, we place ourselves in an episodic discounted setting parameterized by $\gamma \in (0,1)$, with the horizon $T \in \mathbb{N}$. We will also assume that the distribution of the states denoted by μ_{π_θ} is markovian, such that μ_{π_θ} the stationary distribution of the Markov chain is well defined.

2.2 Objective function & Monte Carlo approximation

Therefore, in the described framework, finding the optimal policy is now equivalent to find the optimal θ so that the policy π_θ maximizes the average discounted rewards on a given problem. The performance metric we then introduce in our continuing task is given by the reward function (for notation simplicity in the notations we omit the discounted setting parameter for now):

$$\begin{aligned} J(\theta) &= \sum_{s \in \mathcal{S}} \mu_{\pi_\theta}(s) v_{\pi_\theta}(s) \\ &= \sum_{s \in \mathcal{S}} \mu_{\pi_\theta}(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) q_{\pi_\theta}(s,a) \end{aligned}$$

Where is v_{π_θ} the value function characterizing π_θ, and q_{π_θ} the action-value function.

The optimal θ^* is then given by:

$$\theta^* = arg \max_\theta J(\theta)$$

Since we would like to use policy gradient optimization algorithms to solve that problem, we will assume that $\nabla_\theta J(\theta)$ is always well defined. However, we still have to compute that gradient, which is problematic since μ_{π_θ} is unknown. Thankfully, one should recall the policy gradient theorem [3] that enables us to compute the gradient of the reward in an episodic setting, without having to know the initial distribution of the states:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s \sim \mu_{\pi_\theta}, a \sim \pi_\theta} \left[q_{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(a|s) \right]$$

To approximate the unknown action-value function, one can use a Monte-Carlo policy gradient approach with the following approximation: $q_{\pi_\theta}(s,a) \approx G_k$ where G_k is the episodic return of episode k.

Hence, the estimator of the gradient of the reward in a discounted episodic setting parameterized by γ becomes the following:

$$\widehat{\nabla}_\theta J(\theta) = \sum_{t=1}^{T} \gamma^t \nabla \log \pi_\theta(a_t|s_t) G_t$$

Even if we have convergence guarantees in that setting [6], it faces two main downsides. First, we need to wait until the end of each episode to collect the expected return. Secondly, the estimator has a high variance. To reduce the variance issue, one can introduce the baseline, denoted by $b(s)$, generally picked as $b(s) = v_{\pi_\theta}(s)$, that results in the following estimator:

$$\widehat{\nabla}_\theta J(\theta) = \sum_{t=1}^{T} \gamma^t \nabla \log \pi_\theta(a_t|s_t) A_{\pi_\theta}(s_t, a_t)$$

where $A_{\pi_\theta}(s_t, a_t) = q_{\pi_\theta}(s_t, a_t) - v_{\pi_\theta}(s_t)$ is what we call the advantage function. The advantage function can be seen as how much better is a certain arbitrary action following π_θ compared to taking an action following π_θ given the current state s_t.

Introducing the baseline is a common technique in variance reduction that works through the theory of control variates [4]. In a few words, the baseline should be picked so that it is negatively correlated with the random variable we are trying to approximate, so when looking at the variance of the difference of the two, we obtain the original variance of the random variable plus a negative covariance term, so that is indeed decreasing the initial variance.

In practice, since v_{π_θ} is unknown, we only get to estimate the advantage function so that $A_{\pi_\theta}(s_t, a_t) \approx \widehat{A}_{\pi_\theta}(s_t, a_t)$. We will dive in the computation of the approximation $\widehat{A}_{\pi_\theta}(s_t, a_t)$ in the next section, when introducing the TD-error trick.

2.3 Variance reduction through Actor-Critic

So far, we have obtained an estimator of the gradient of the reward that is unbiased. However, despite the control variate trick, its high variance makes the computation time slower over the number of epochs. One would also want to address the issue of needing to compute a full episode in order to get the reward after a given episode.

To tackle these major issues, we introduce the Actor-Critic [5] framework: we rely on an additional estimator of q_{π_θ}, called the critic. The actor corresponds to the policy π_θ, while the critic is a Q-value function that is modeled by another predictor (a neural network for instance), and enables us to estimate $\widehat{q}_{\pi_\theta}$ in an online fashion way, so that we can stop at any time without having to wait the end of the episode. Basically, the critic evaluates the action of the actor in the current state to produce a Q-value that stands for the quality of that action. This critic feedback is then given to the actor in addition to the current state as input, so it can adjust its action accordingly. Note that in our case, the critic and the actor are both modelized by neural networks. However, this architecture requires to train values weight estimates w_v and w_q that increase the complexity:

$$\begin{aligned} v_{\pi_\theta}(x) &\approx \widehat{v}_{\pi_\theta}(s, w_v) \\ q_{\pi_\theta}(s, a) &\approx \widehat{q}_{\pi_\theta}(s, a, w_q) \\ A_{\pi_\theta}(s, a) &\approx \widehat{q}_{\pi_\theta}(s, a, w_q) - \widehat{v}_{\pi_\theta}(s, w_v) \end{aligned}$$

Thankfully, a little trick using the TD error enables to use only one train value weight estimate rather than two. The TD error is denoted by $\delta = r + \gamma v_{\pi_\theta}(s') - v_{\pi_\theta}(s)$. Notice that:

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}[\delta_{\pi_\theta}|s,a] &= \mathbb{E}_{\pi_\theta}[r + \gamma v_{\pi_\theta}(s') - v_{\pi_\theta}(s)] \\
&= q_{\pi_\theta}(s,a) - v_{\pi_\theta}(s) \\
&= A_{\pi_\theta}(s,a)
\end{aligned}$$

Hence the TD error can be used directly in the gradient of the reward instead of the advantage, leaving only w_v to be updated by the critic since we don't need have q_{π_θ} in that formulation anymore.

In our online framework, the critic therefore comes first looking at the state of the environment (policy evaluation step), outputting the TD error and the current state to the actor so it can adapt using both the state information and the critic information through the TD error (policy improvement step).

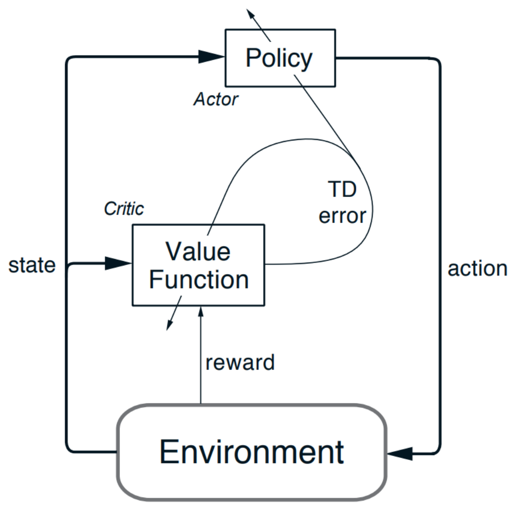

Figure 1: Actor-critic architecture (Sutton and Barto's Book [5])

The algorithm of the Actor-Critic we obtain will be detailed later in the report.

3 Key concepts of Proximal Policy Optimization

Before we can dive into the specificity of the Proximal Policy Optimization algorithm (PPO), we need to understand some corner stone algorithms that are at the origin of the PPO.

3.1 Policy Gradient Optimization algorithm: REINFORCE

In the previous section, we described the performance metric and its gradient so we could solve the following optimization problem to nail down the optimal policy:

$$\theta^* = arg \max_\theta J(\theta)$$

The most classical way to numerically implement this maximization problem is through a gradient ascent algorithm, for which the steps are as following:

Algorithm 1 Gradient ascent algorithm

```
Require: $\alpha > 0$ a given learning rate, $\theta_0$ an initialization of the parameter
  for $k \in \{0, ..., n_{iter}\}$ do
    $\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta)$
  end for
Output: $\theta_T$
```

Replacing $\nabla_\theta J(\theta)$ by the previously computed formulae using the expected return, we obtain the classical Policy Gradient Optimization that is generally called REINFORCE algorithm [6]:

Algorithm 2 REINFORCE algorithm

```
Require: $\alpha > 0$, $\theta$ an initialization of the parameter
  for $k \in \{0, ..., n_{iter}\}$ do
    Generate a trajectory following $\pi_\theta$ : $(s_t, a_t, r_t)_{t \in \{1,...,T\}}$
    for $t \in \{1, ..., T\}$ do
      Estimate the expected return $G_t$
      $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t, s_t)$
    end for
  end for
Output: $\theta$
```

We have implemented the REINFORCE algorithm, so we can benchmark it later with the PPO algorithm. Being the most basic algorithm one could think of when it comes to policy optimization, it will stand as our ground comparison with other policy optimization approaches.

Note that the vanilla REINFORCE algorithm does not use the control variate trick to reduce the variance, but one could easily implement it by replacing the estimation of G_t by the advantage at time t [8]. The next figure highlights the variance issue of the standard REINFORCE algorithm as we have implemented it for the *CartPole-v1* environment.

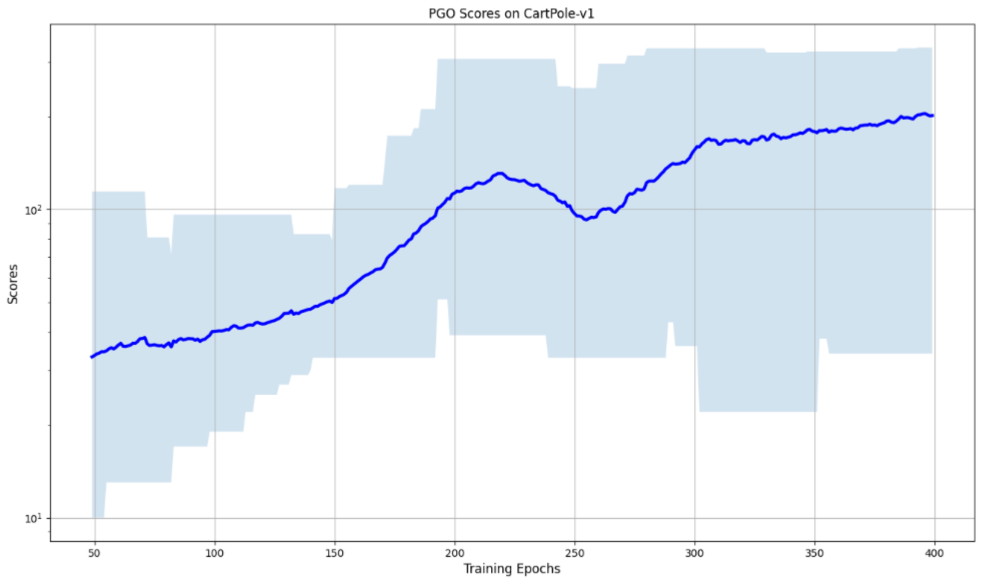

Figure 2: Typical REINFORCE performance on *CartPole-v1* environment. The blue area represents the range. The curve is a moving average of the performance over 50 iterations.

Also, the step size α is determinant in the algorithm convergence on average. Indeed, stochastic approximation claims that α must be sufficiently small so that the algorithm converges in expectation towards the right parameter θ^*. If the learning rate is too big, then REINFORCE may lead to a plateau value that is not the wanted optimum of the problem θ^*.

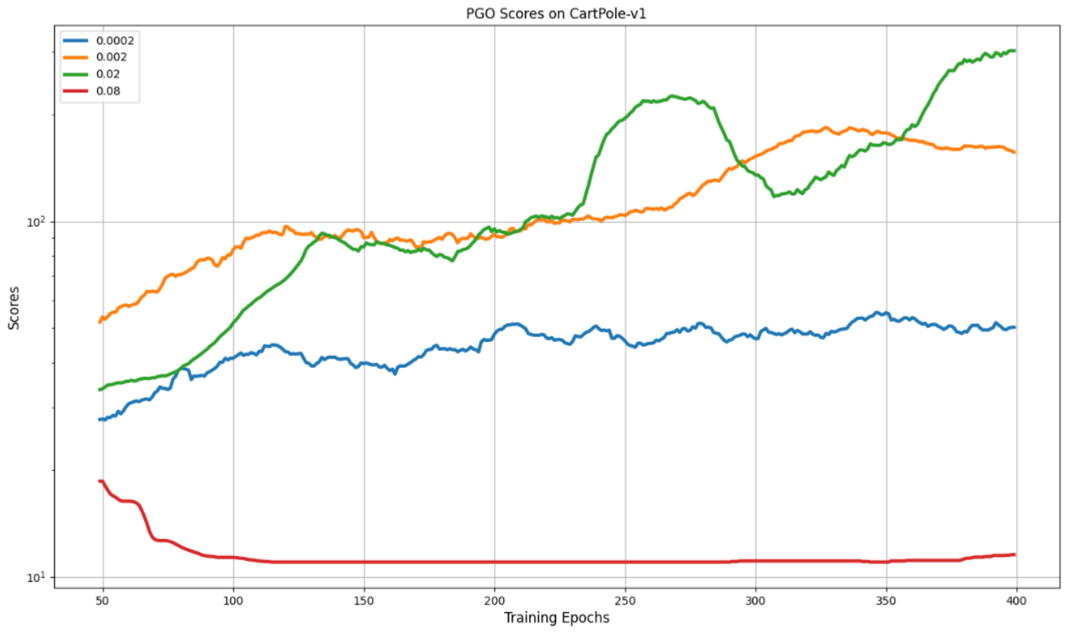

Figure 3: Typical REINFORCE performance on *CartPole-v1* environment with the learning rates $\alpha_1 = 2.10^{-4}$, $\alpha_2 = 2.10^{-3}$, $\alpha_3 = 2.10^{-2}$, $\alpha_4 = 8.10^{-2}$.

As we can see on the previous figure, if α is too large the policy update consistently fails at the targeted task. On the other hand, if the learning rate is too small, the training gets really slowed down even though it seems to consistently improve. Finding the right learning rate is therefore a challenge of policy gradient methods as an incorrect step size can severly slow down the training, and may even lead to convergence towards a poorly performing policy or not converge at all.

Finally, one may observe empirically some regular drops in performance when using REINFORCE. Indeed, REINFORCE and its variants may often leads to problematic large policy updates that may have a significant drop in performance that is hard to catch up once fallen into the pit. This issue is going to be addressed later namely by gradient clipping methods and KL constraint optimization.

3.2 Actor-Critic algorithm

As we've previously described it, the Actor-Critic algorithm stands as another key reference when it comes to solving policy optimization problems. It namely serves as a variance reduction technique to the policy gradient methods, additionally data efficient, and will be a corner stone in the PPO algorithm.

One of the key aspects of the Actor-Critic algorithm, is that it's located in between policy optimization and value-based optimization. Indeed, knowing the value function does assist the policy update in a very positive way by reducing its variance as we have discussed earlier, thanks to the critic feedback.

Implementing this algorithm is also fairly easy [5]:

- The critic feedback can either be computed through the estimated action-value $\widehat{q}_{\pi_\theta,w}$ or the estimated value function $\widehat{v}_{\pi_\theta,w}$, parameterized by w.
- The actor should update the policy π_θ considering both the observed state and the suggestion of the critic.

Algorithm 3 Actor-Critic algorithm (A2C)

```
Require: $\alpha_w, \alpha_\theta > 0$ learning rates, $\theta$, $w$
  Denote by $s$ the current state of the environment
  Sample the action $a \sim \pi_\theta(a|s)$
  for $k \in \{0, ..., n_{iter}\}$ do
    Sample reward $r_t \sim R(s, a)$
    Sample next state $s' \sim P(s'|s, a)$
    Sample next action $a' \sim \pi_\theta(a|s')$
    Update parameter $\theta \leftarrow \theta + \alpha_\theta \widehat{q}_{\pi_\theta,w}(s, a) \nabla_\theta \log \pi_\theta(a|s)$
    Compute TD error $\delta_t = r_t + \gamma \widehat{q}_{\pi_\theta,w}(s', a') - \widehat{q}_{\pi_\theta,w}(s, a)$
    Update the weight parameter $w \leftarrow w + \alpha_w \delta_t \nabla_w \widehat{q}_{\pi_\theta,w}(s, a)$
    $a \leftarrow a'$
    $s \leftarrow s'$
  end for
Output: $\theta$
```

Note that we need to define two learning rates, one being for the critic parameter update w, and the other for the policy update θ. Next figure shows the performances of the A2C on the *CartPole-v1* game as we have implemented it, compared to the REINFORCE method.

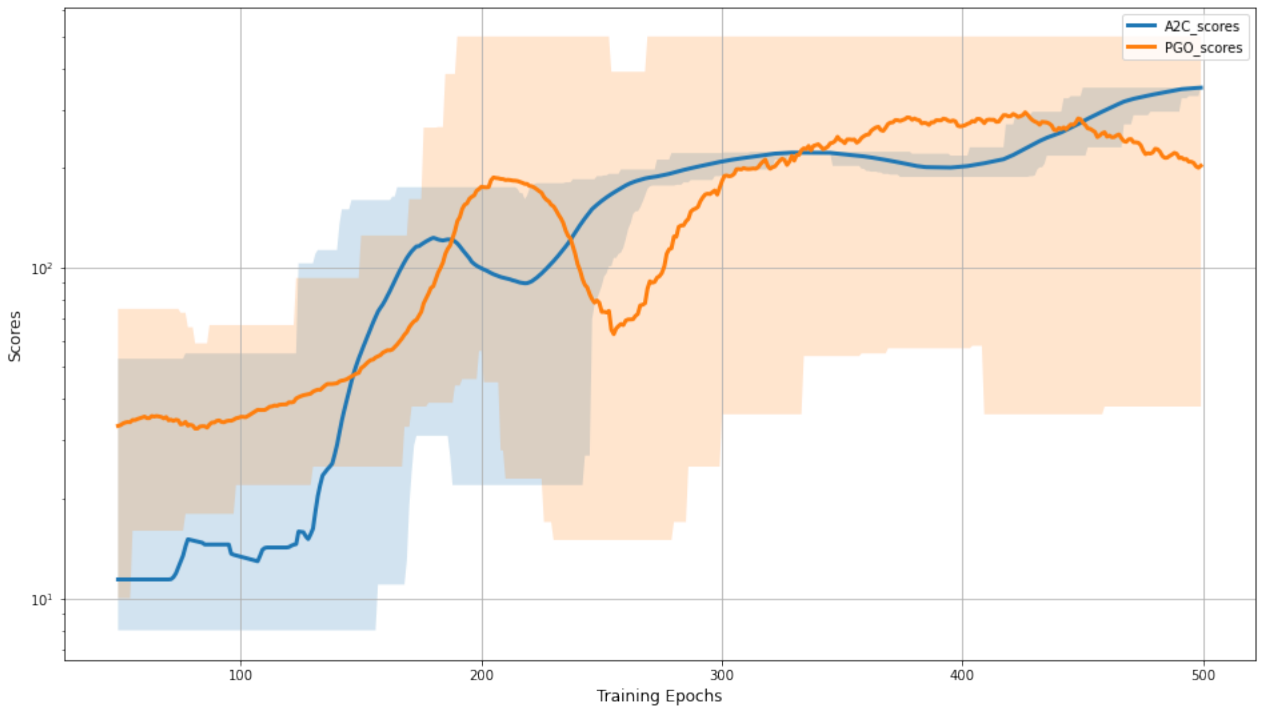

Figure 4: Typical A2C and REINFORCE performance on *CartPole-v1* environment. The colored areas represent the range. The curves are moving averages of the performance over 50 iterations. Blue curve/area represents A2C. Orange curve/area represents REINFORCE.

As we can see on the previous figure, the A2C variance reduction gain over REINFORCE is massive. Hence, even if the average performance of each algorithm is comparable at this stage, the variance reduction gain makes the A2C architecture a must-have for policy gradient based methods.

One variant of the Actor-Critic that we are interested in is the Asynchronous Advantage Actor-Critic (A3C) [7]. The main advantage of this approach is the parallel training with possible syncing of parameters during the procedure. Since the concept is quite close to the Actor-Critic, we decided not to dive in the technical details of this variant that allows parallelism. It is still worth noting that variant since it's one of the idea behind Proximal Policy Optimization.

3.3 Trust Region Policy Optimization algorithm

So far, we have discussed the variance issue of Proximal Gradient methods, and how to partially solve it using control variables or Actor-Critic architecture.

Yet, another major issue we barely introduced is the one of too large policy changes between iterations. Indeed, too large policy updates can lead to leaving the optimal zone and dropping in performance, which may be hard to catch up to good performances again.

One possible approach to prevent that issue is given by the Trust Region Policy Optimization (TRPO). The idea is to add a KL divergence constraint on the size of the policy update. Indeed, the Kullback-Leibler (KL) divergence is a measure of distance between probability distributions defined as follows for continuous densities (take counting measure of Dirac for discrete densities):

$$D_{KL}(f||g) = \int_{\mathcal{X}} f(x) \log \frac{f(x)}{g(x)} dx$$

Remark then that if $f = g$, $D_{KL}(f||g) = 0$. Also note that the KL divergence is positive, but asymmetric which may lead to buggy situations when f is close to 0 in the sense of the KL divergence. However, this issue is fixed if $g \gg 0$ in the sense of the KL divergence. Therefore, the problematic situation holds when we want to compare distributions of the same amplitude.

To implement that constraint, we use the trick of importance sampling, introducing a secondary policy instead of π_θ to generate the trajectories as in the REINFORCE algorithm and get the comparison with a secondary policy to appear. The said secondary policy in TRPO corresponds to the previous policy iterate denoted by $\pi_{\theta_{old}}$:

$$\begin{aligned}
J(\theta) &= \sum_{s \in \mathcal{S}} \mu(s) v_{\pi_\theta}(s) \\
&= \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} q_{\pi_\theta}(s, a) \pi_\theta(a|s) \\
&= \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} q_{\pi_\theta}(s, a) \pi_{\theta_{old}}(a|s) \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \\
&= \mathbb{E}_{s \sim \mu, a \sim \pi_{\theta_{old}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} q_{\pi_\theta}(s, a) \right]
\end{aligned}$$

The main idea of TRPO is then to apply the following KL divergence constraint on the optimization problem, for a given $\delta > 0$:

$$\mathbb{E}_{s \sim \mu}[D_{KL}(\pi_{\theta_{old}}(.|s)||\pi_\theta(., s))] \leq \delta$$

This constraint would ensure that the old and the new policy are not too far KL-wise, which guarantees a monotonic improvement during training.

Another major change in TRPO is also the predictor used when generating trajectories. Referring to the PGO algorithm **1**, we were generating trajectory using the current policy π_θ, while TRPO proposes to simulate using $\pi_{\theta_{old}}$ due to the importance sampling trick. This is meant to prevent policies that wouldn't respect the KL-divergence constraint to influence the current iteration while running the algorithm asynchronously.

Note that despite its robust qualities, the TRPO algorithm is quite hard to implement in practice, and can sometimes be difficult to interpret. One can find below a pseudo-code that uses conjugate gradient methods to handle the constraint [9]:

Algorithm 4 Trust Region Policy Optimization algorithm

```
Require: $\theta$, $\delta$, $\alpha \in (0,1)$ backtracking coefficient, $K$ number of backtracking steps
  Let $\theta_{old} \leftarrow \theta$
  for k $\in\{1, ..., n_{iter}\}$ do
    Generate a trajectory following $\pi_{\theta_{old}}$: $(s_t, a_t, r_t)_{t \in 1,...,T}$
    Estimate the advantage $\widehat{A}_{\pi_{\theta_{old}}}$
    Compute the reward gradient $\widehat{\nabla}_\theta J(\theta)$ using importance sampling with $\pi_{\theta_{old}}$ and $\widehat{A}_{\pi_{\theta_{old}}}$
    Estimate the Hessian $\widehat{H}$ of $\bar{D}_{KL} = \mathbb{E}_{s \sim \pi_{\theta_{old}}}[D_{KL}(\pi_{\theta_{old}}(.|s)||\pi_\theta(.|s)]$
    Running a step of conjugate gradient [10], compute $\widehat{x} \approx \widehat{H}\widehat{\nabla}_\theta J(\theta)$
    Find $j \in \{0, ..., K\}$ smallest value improving the sample loss, satisfying the KL constraint
    and producing a positive advantage.
    Update $\theta_{old} \leftarrow \theta$
    Update $\theta \leftarrow \theta + \alpha^j \sqrt{\frac{2\delta}{\widehat{x}^T \widehat{H} \widehat{x}}}\widehat{x}$
  end for
Output $\theta$
```

As one can tell straight ahead, this algorithm isn't easy to implement due to the constraint optimization. A lot of approximations are required to compute quickly this algorithm [9], such as order 1 Taylor expansions approximations of the reward $J(\theta)$ and order 2 Taylor expansion of $\bar{D}_{KL}$, so that it gives the following update rule:

$$\theta_{k+1} = arg \max_\theta \nabla_\theta J(\theta)(\theta - \theta_k)$$

$$s.t. \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k)(\theta - \theta_k) \leq \delta$$

This approximation is the reason why we have to introduce the backtracking linesearch with $\alpha \in (0,1)$ and $K \in \mathbb{N}$. Indeed, without the coefficient add-on, the iterates $(\theta_k)_k$ may not respect the KL constraint, nor improve the advantage [9].

We have implemented the TRPO (Actor-Critic style) algorithm to be able to compare it to the PPO, as it clearly stands as its main competitor at the moment. Typical performance on the *CartPole-v1* game (limited to 1000 iterations to win) is illustrated below:

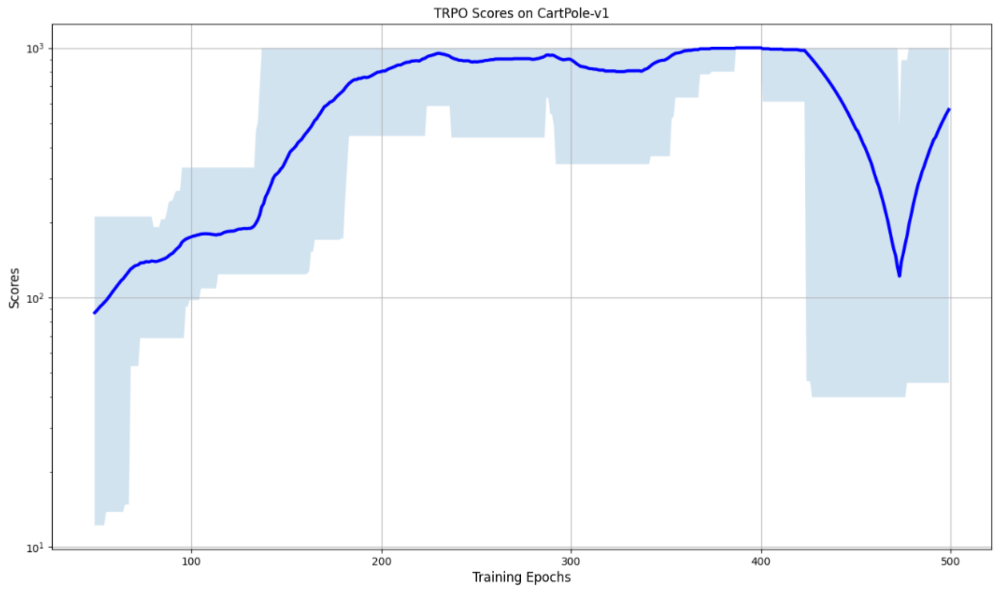

Figure 5: Typical TRPO performance on *CartPole-v1* environment. The colored area represents the range. The curves is a moving average of the performance over 50 iterations.

As we can see, the performance is outstanding. The TRPO iterates are reaching the maximum value of 1000 around 200 iterations, and stays in the neighbourhood for over 230 iterations after that. Note the drop around the end of the training that is due to a KL parameter δ that may be too big, leading to a policy update that is too large still and worsen the performances. One may address that issue by properly tuning the KL parameter. The following graph represents multiple performances of the TRPO algorithm on the *CartPole-v1* game with different KL parameters.

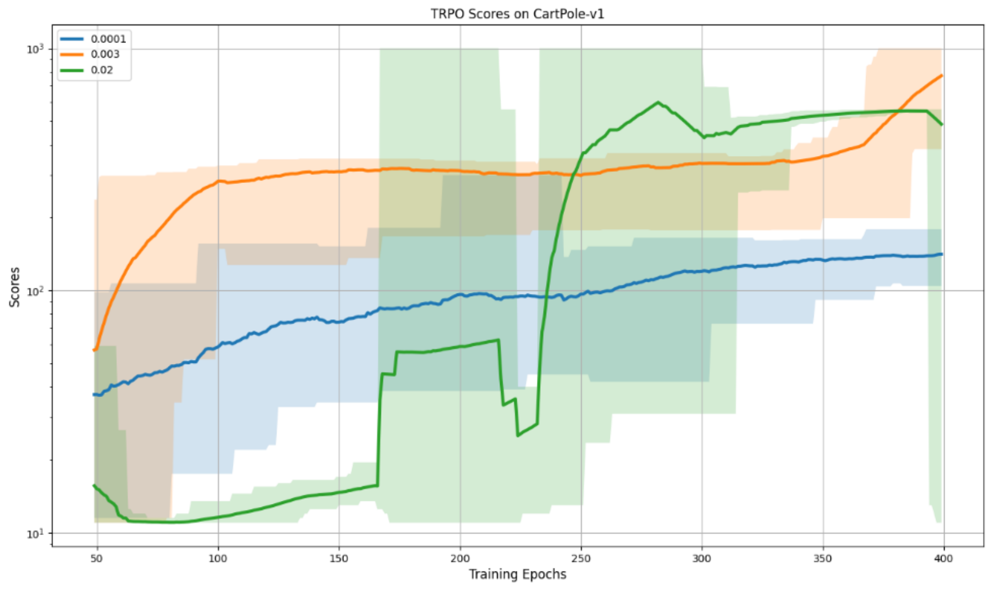

Figure 6: Typical TRPO performance on *CartPole-v1* environment using different KL parameters $\delta_1 = 10^{-4}$, $\delta_2 = 3.10^{-3}$, $\delta_3 = 2.10^{-2}$. The colored areas represent the range. The curves are moving averages of the performance over 50 iterations.

Looking at this figure, one may see that a very small KL parameter δ_1 limits the policy updates and therefore makes the improvements consistent but slow. On the other hand, a very large KL parameter δ_3 steers towards classical PGO updates with higher variance and risk of major drops in performance (see the end of the green area variance and drops). Therefore, an optimal δ has to be determined to achieve balance between the training time and a consistent improvement of the policy iterates of the TRPO.

Since the constraint can be tough to implement, note that there exists an alternative formulation of the KL constraint, that can be written using a penalty expression rather than a constraint on the optimization problem, so that for a given coefficient $\beta > 0$, using the advantage expression of the reward:

$$arg \max_{\theta} \mathbb{E}_{s\sim\mu, a\sim\pi_{\theta_{old}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\pi_\theta}(s,a) - \beta D_{KL}(\pi_{\theta_{old}}(.|s)||\pi_\theta(.,s)) \right]$$

This formulation actually comes from the fact that it is a pessimistic lower bound of the performance metric on the policy π_θ. Yet it is not used in TRPO since it's not strictly equivalent. Indeed, TRPO uses a hard constraint instead to enable single choices of β performing well for various problems. Here, the adaptive constraint steers towards the constrained space without necessarily having each policy iterate to be in the constraint space of limited KL divergence. As a result, it is not quite as certain as the classical TRPO hard constraint and could still lead to problematic updates if β isn't tuned properly.

Therefore, there is place for improvement here: finding an algorithm that can exploit a similar form of penalty rather than a hard constraint, with the same improvement as TRPO or better. That algorithm would be much more understandable and easier to implement. However, it can not be done as such since experiments have been showing that a fixed penalty is not enough to achieve the performances of TRPO. So additional findings are required, as introducing an adaptive penalty and a clipping method as presented in the following sections.

3.4 Objective clipping

In order to attend the problematic of too large policy updates without introducing constraints, we introduce the idea of objective clipping. The goal of clipping is to control the update by forcing the ratio of probabilities to be in a given range ϵ. The probability ratio is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Recall the reward objective function using importance sampling, rewritting it using the probability ratio:

$$\begin{aligned} J(\theta) &= \mathbb{E}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \widehat{A}_{\pi_\theta}(s_t,a_t) \right] \\ &= \mathbb{E}\left[r_t(\theta) \widehat{A}_{\pi_\theta}(s_t,a_t) \right] \end{aligned}$$

So to avoid the incentive of excessively large policy updates, we clip the probability ratio $r_t(\theta)$ denoting the clipped ratio at threshold ϵ:

$$\begin{aligned} r_t^{CLIP}(\theta) &= \max\left(\min(r_t(\theta), 1+\epsilon), 1-\epsilon\right) \\ &\in [1-\epsilon, 1+\epsilon] \end{aligned}$$

Finally, to build the clipped objective function we will replace the probability ratio by $\min(r_t(\theta), r_t^{CLIP}(\theta))$. The reason behind taking the minimum of the clipped ratio and the unclipped one is to obtain a pessimistic bound on the unclipped objective to reduce the interest in major increasing policy updates to prioritize a small reward improvements. Hence, we obtain the following clipped objective function to optimize:

$$J^{CLIP}(\theta) = \mathbb{E}\left[\min(r_t(\theta), r_t^{CLIP}(\theta))\widehat{A}_{\pi_\theta}(s_t, a_t)\right]$$

This clipped objective function is then well suited to obtain updates that are close to the previous policy, making small but safe steps as we limit the importance of promising actions, while for bad actions we decrease the probability accordingly. Overall, this procedure enables small but relevant updates to prevent the incentive of too big steps that may decrease performances drastically.

Note that this clipping approach has been criticized lately, either for instability reasons on the ratio, sudden flattening or optimization efficiency. Some recent works have been addressing these issues, like using early stopping [11] rather than clipping, or functional clipping approaches [12], that have shown improved results.

3.5 Adaptive KL Penalty

Another approach to implement the KL constraint during updates is to introduce an adaptive KL penalty. This penalty could be used as an alternative to the clipping objective presented above. Recall the KL-penalty approach presented as an almost equivalent to the TRPO constraint:

$$J^{KL}(\theta) = \mathbb{E}_{s\sim\mu, a\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}A_{\pi_\theta}(s,a) - \beta D_{KL}(\pi_{\theta_{old}}(.|s)||\pi_\theta(.,s))\right]$$

The idea of the adaptive KL-penalty is to adapt β so that we achieve a given target value of the KL distance d_{targ} between the previous policy and the updated one.

In practice, we compute the adaptive KL penalty as follows [1]:

Algorithm 5 Adaptive KL penalty

```
Require: $\theta_{old}$, $\beta$, $s_t$
  Compute $\theta$ using stochastic gradient ascent on $L^{KL}(\theta_{old})$:
  Compute $d = \mathbb{E}\left[D_{KL}\left(\pi_{\theta_{old}}(.|s_t)||\pi_\theta(.,s_t)\right)\right]$
  if $d < d_{targ}/1.5$ then
    $\beta \leftarrow \beta/2$
  end if
  if $d > d_{targ} \times 1.5$ then
    $\beta \leftarrow 2\beta$
  end if
Output $\beta$
```

The output penalty coefficient β has now become adaptive, and is going to be used at the next update of the policy. Experimentally, it still rarely happens that the KL distance we obtain is rather large in front of d_{targ}, but thankfully β adjustments are quite quick. As a result, we can also choose any initial value for β since it adjusts quickly.

However, experimental results from the original paper have shown that the adaptive KL penalty approach is not performing quite as great as the clipping objective method in most situations [1]. Hence in the PPO algorithm we are going to introduce now, we would stick to the clipped objective rather than the adaptive KL penalty for better results. It's still worth noting that it has this other alternative writing though, as it can be used to solve some problematic situations of PPO clipped version as described in the "PPO limits" section.

4 Proximal Policy Optimization

4.1 Standard PPO

Now that we have built alternative methods to achieve TRPO's logic without having to compute a constraint that make the algorithm relatively complicated, we are ready to introduce the Proximal Policy Optimization (PPO) algorithm.

PPO is one of the state of the art algorithm when it comes to policy optimization, and it just uses the various tricks we describe to get the best of each world, using clipped objective or the adaptive KL penalty. Therefore, standard PPO just consists in a policy gradient optimization for which you replace the objective function either by $J^{CLIP}(\theta)$ or $J^{KL}(\theta)$ depending on how you want to control the updates:

Algorithm 6 Standard PPO

```
Require: $\alpha > 0$, $\theta_{old}$
  for $k \in \{0, ..., n_{iter}\}$ do
    Generate a trajectory following $\pi_{\theta_{old}}$ : $(s_t, a_t, r_t)_{t \in \{1,...,T\}}$
    for $t \in \{1, ..., T\}$ do
      Compute $\theta$ by optimizing the chosen reward $J^{CLIP}(\theta_{old})$ or $J^{KL}(\theta_{old})$
      Update $\theta_{old} \leftarrow \theta$
    end for
  end for
Output: $\theta_{old}$
```

4.2 PPO enhancements: Actor-Critic, squared error & entropy

What we are more interested in though are some variants of the PPO algorithm that increase the quality of the algorithm. First, using the Actor-Critic architecture can lower the variance of the estimator, which is a game changer for policy gradient algorithms. An entropy term, denoted by $S(s_t, \pi_\theta(.))$, can be introduced: it enables exploration during optimization [13], like the Metropolis algorithms in Markov chains, so that we can leave possible local optima for better ones. Therefore, using A3C architecture [7] and with exploration noise, we obtain the following objective function of interest for the PPO, given $c_1, c_2 > 0$, respectively the squared error coefficient and the entropy coefficient:

$$J^{PPO}(\theta) = \mathbb{E}\left[J^{CLIP}(\theta) - c_1 \underbrace{(v_{\pi_\theta}(s) - v_{target})^2}_{actor-critic} + c_2 \underbrace{S(s_t, \pi_\theta(.))}_{entropy} \right]$$

This new objective functions will enable us to compute one of the most favorite algorithms when it comes to policy optimization for neural network class of policies. The idea is that we let our agent play in the environment with the current policy for a given time T to collect a serie of transitions (a_t, s_t, r_t). We then compute the advantage at each time $t \in [0, T]$ following the current policy using the given estimator:

$$\widehat{A}_{\pi_\theta,t} = -v_{\pi_\theta}(s_t) + \sum_{i=0}^{T-t+1} \gamma^i r_{t-i} + \gamma^{T-1} v_{\pi_\theta}(s_T)$$

Or using the δ_t TD-error trick from the Actor-Critic architecture:

$$\widehat{A}_{\pi_\theta,t} = \sum_{i=0}^{T-t+1} \gamma^i \delta_{t-i}$$

We can then write the Actor-Critic algorithm of the PPO that we implemented for our experiments too:

Algorithm 7 PPO Actor-Critic

```
Require: $\theta_{old}$, $c_1$, $c_2$, $\alpha$, $\gamma$
  for $k \in \{0, ..., n_{iter}\}$ do
    for each actor do
      Generate trajectory following $\pi_{\theta_{old}} : (s_t, a_t, r_t)_{t\in 1,...,T}$
      Compute the advantages $\widehat{A}_{\pi_\theta,1}, ..., \widehat{A}_{\pi_\theta,T}$
    end for
    Compute $\theta$ by optimizing $J^{PPO}(\theta_{old})$
    Update $\theta_{old} \leftarrow \theta$
  end for
Output $\theta_{old}$
```

Overall, even if some clipping or penalization is hidden behind the optimization of θ, this implementation is pretty easy to understand and compute. Note that this algorithm is also scalable, and can be performed in multiple pass on the data with minibatches.

However, one challenge remains in this implementation: we have tune the coefficients c_1, c_2 that respectively goes for the squared error of the critic coefficient and the entropy coefficient, as well as the clipping threshold of the policy iterates. The following figure illustrates the behavior for the of PPO performances for different values of the squared error coefficient c_1.

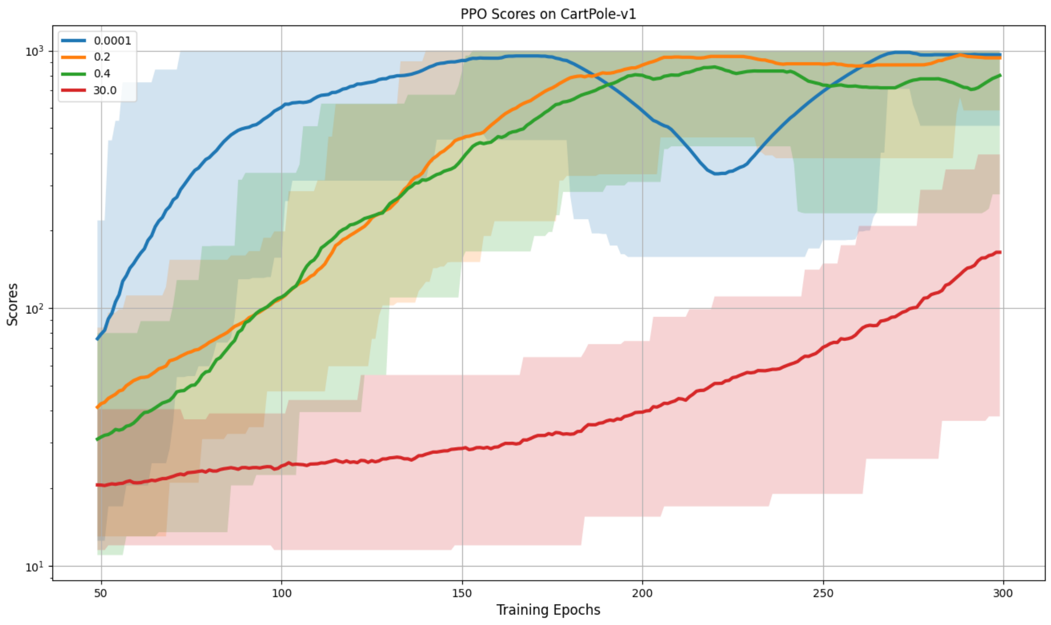

Figure 7: PPO performances on *CartPole-v1* environment using different values of the squared error coefficient of the critic c_1: parameters $c_1^{(1)} = 10^{-2}$, $c_1^{(2)} = 0.2$, $c_1^{(3)} = 0.4$, $c_1^{(4)} = 30$. The colored areas represent the range. The curves are moving averages of the performance over 50 iterations.

As we can observe, using a critic coefficient that is too low tends to increase the variance of the policy updates as it reduces the impact of the actor-critic architecture by diminishing the critic feedback. Higher values increase the impact of the critic, which reduces the overall variance. However, if the value of c_1 gets too high we might end up undermining the other terms (entropy and clipping), so it has to be tuned appropriately.

Looking at the entropy term, The following figure illustrates the performances of the PPO using multiple values of the entropy coefficient c_2.

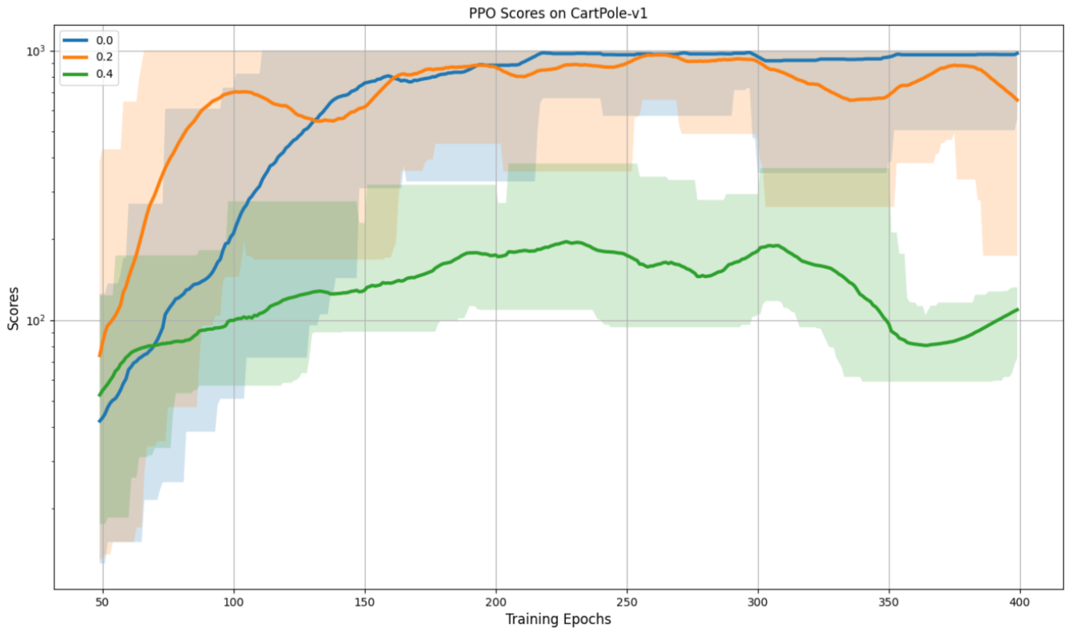

Figure 8: PPO performances on *CartPole-v1* environment using different values of the entropy coefficient c_2: parameters $c_2^{(1)} = 0$, $c_2^{(2)} = 0.2$, $c_2^{(3)} = 0.4$. The colored areas represent the range. The curves are moving averages of the performance over 50 iterations.

According to Ahmed et al. [13], as well as helping the exploration process, the entropy plays a role of regularizer of the optimization landscape. Therefore, as we can see on the previous figure, higher values of entropy tend to increase the variability of policy performance as it's exploring more in order to avoid local optima, up until a critical value of high entropy that prevents the convergence to happen: we are constantly leaving optimal areas due to the exploration being too large, so we can't converge. We would expect smaller entropy values to tend to get stuck in local optima of the problem, but we couldn't confirm that experimentally due to the nature of the *Cartpole-v1* game and the time limitation we fixed. Hence, one should try to find a coefficient for the entropy that is not too small to avoid local optima and keep exploring, while avoiding too large values that may lead to an unstable algorithm, not capable to converge to an optimal policy.

Finally, we would like to explore the impact of the clipping threshold of policy iterates on the performance of the PPO.

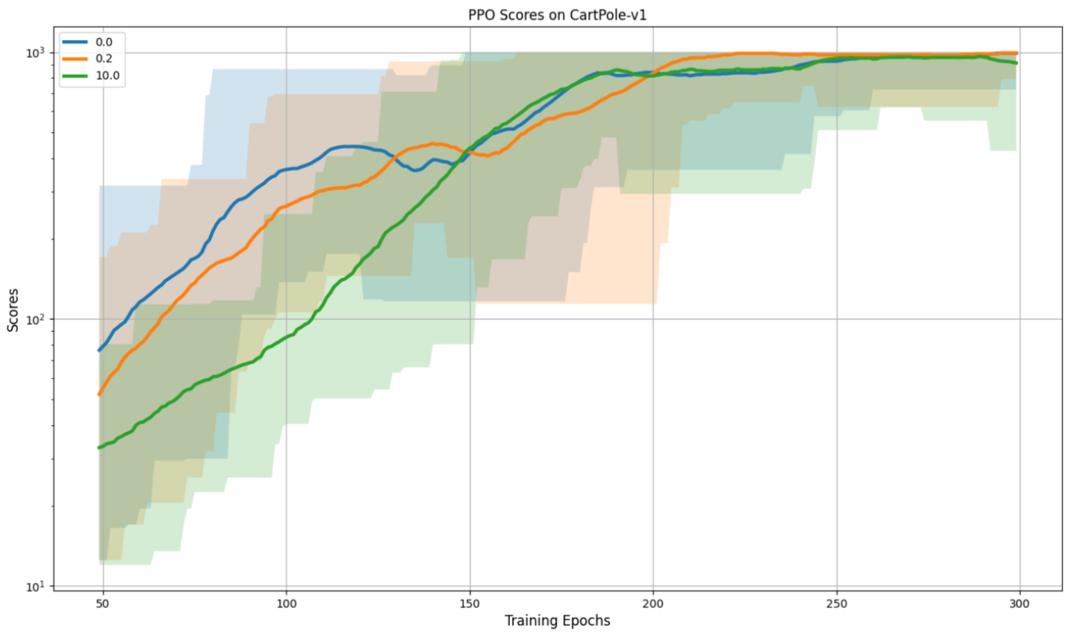

Figure 9: PPO performances (no entropy) on *CartPole-v1* environment using different values of the clipping threshold coefficient of the policy updates ϵ: parameters $\epsilon_1 = 10^{-2}$, $\epsilon_2 = 0.2$, $\epsilon_3 = 0.4$. The colored areas represent the range. The curves are moving averages of the performance over 50 iterations.

Recall that the clipping consists in projecting the probability ratio $r_t(\theta)$ in $[1-\epsilon, 1+\epsilon]$, where the clipping coefficient is denoted by ϵ. Hence, small values of ϵ are steering the policy updates towards minimal changes regarding the previous one ($r_t(\theta) \approx 1$). Conversely, using large values of ϵ enables much changes regarding the previous policy iterate. As we can observe on the graph, there seems to exist a right balance to find for ϵ that impacts the stability of the PPO iterates performance-wise. Indeed, if we use a value of ϵ that is too large, the surrogate objective first order approximation isn't valid anymore and we might end up with a policy iterate that is leaving the region of optimality (see the range of the green curve indicating a higher variance), leading to potential drops in performance and higher instability. On the other hand, using a very small value of ϵ results in sticking to the previous policy if the next iterate can not provide an improved reward, which may drastically slow down the progress of the iterates in most scenarios.

4.3 Some PPO limits

Despites the great promises offered by the PPO algorithm, a later paper by Hsu et al., 2020 [14] showed three modes in which PPO is failing:

- If $\mathcal{A}$ is continuous, PPO gets unstable when the reward vanishes outside bounded support.
- If $\mathcal{A}$ is discrete with sparse high rewards, PPO may get stuck in suboptimal solutions for θ.
- If J^{PPO} admits local optima close to the initialization θ_0, the policy gets highly sensitive.

Thankfully, the authors proposed two adjustments that may help fixing these issues. For instance, when using a gaussian policy model, discretizing the action space or using β-distributions can help fixing the first and third issues. Also, using the adaptive KL penalty rather than clipping objective may help solving the second issue.

5 Benchmarking Proximal Policy Optimization

5.1 Experimental setup

Now that we have presented these policy gradient methods orbiting around the PPO algorithm, we would like to compare them on different settings, trying to verify the results of the original paper [1]. To that end, we will be using the *gymnasium* package of OpenAi [15] to train a neural network policy using the different policy optimization algorithms we have implemented. The source code can be found on our GitHub at the following address: *https://github.com/LiliBISC/RL.*

So that we can compare the performances of algorithms, we need to set a common policy architecture as a comparison baseline. Therefore, we decided to go with a neural network policy with the following parameters:

- 2 layers with 256 neurons each
- ReLU activation function in between
- Softmax output function (choosing between multiple actions)

The network is not so complex so we can train it fast and focus on the optimization algorithms complexity and characteristics.

Regarding the environments, we decided to go with games for which the reward criterion is the duration of the episode. For instance in *CartPole-v1*, the longer we keep the stick up the better: so the reward can be defined as how long we manage to keep the stick up.

Note that we slightly modified the environments we trained into so that the episodes wouldn't last too long and be limited in time. Therefore, each environment is limited to up to 1000 iterations to avoid over-performing algorithms that could lead to hours of playing up until it fails.

5.2 Optimization algorithms comparison

The following figure represents the performance of each fine-tuned algorithm on various seeds of the *Cartpole-v1* game.

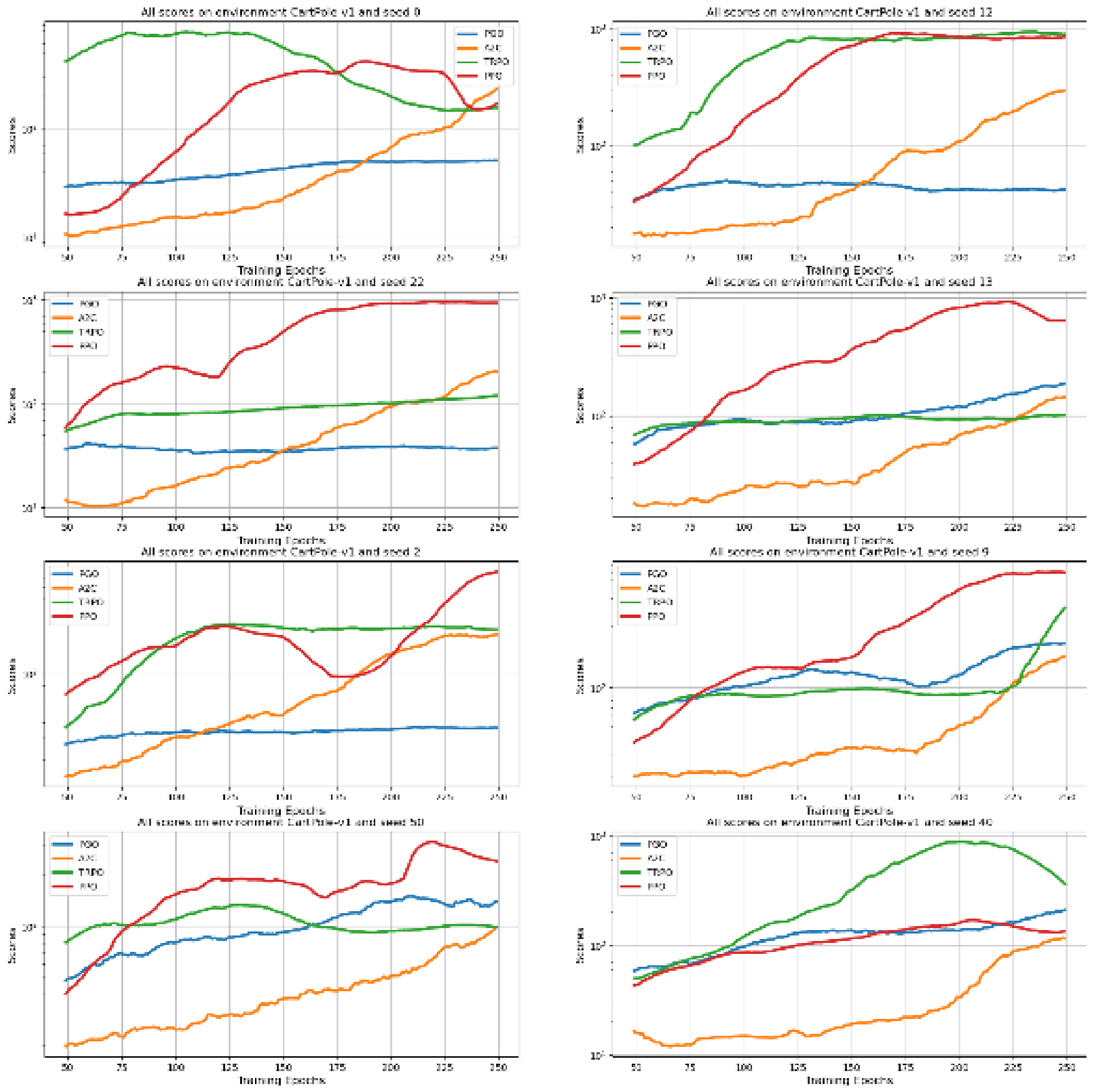

Figure 10: Benchmark of the different optimization algorithms on different initialization of the *Cartpole-v1* environment. The curves are moving averages of the performance over 50 iterations.

As we can see, the PPO Actor-Critic variant outperforms every algorithm in convergence speed and performances on most situations. Therefore, PPO clearly appears as a much more understandable technique than TRPO with at least as good or better performances.

In order to compare the algorithms computation time relatively to a given environment, we ran each method on the *Cartpole-v1* environment with multiple seeds and computed the average time to run a step of each optimizer. The following table summarize the average times of a given step on the *Cartpole-v1*:

Algorithm	Average time (s)
PGO	0.037s
A2C	0.0122s
TRPO	0.16s
PPO	0.135s

Table 1: Average execution time of each algorithm on the *Cartpole-v1* environment with maximum reward of 200s and an horizon $T = 100$

As we can see, the PPO average computation time on the *Cartpole-v1* environment is lower than the TRPO. Hence, for increased performances, we also have a lower complexity with PPO than TRPO that makes the computation faster on average with PPO.

6 Conclusion

Proximal Policy Optimization has revolutionized the policy gradient methods with its outstanding performances and computational improvements. Besides, it has built a much more interpretable and robust alternative to the TRPO by introducing a clipping surrogate objective to replace the constraint. Combined with the Actor-Critic architecture, we were able to significantly reduce the variance of the standard policy gradient methods. Finally, using entropy we were able to enhance the exploration process to avoid local optima and introduce explicit regularization to the optimization algorithm.

However, despite all the good promises of the PPO, the optimization method is still facing some limitations and issues, some of the challenges being addressed by Hsu et al. [14] that we described earlier. Other improvements are being explored regarding the clipping surrogate objective as addressed by Sun et al. [11] using early stopping instead, or Zhu et al. [12] introducing a functional clipping to prevent performance instability issues. Improvements are also being proposed regarding the entropy implementation looking at Zhang et al. [16].

Therefore, even though PPO has been a breakthrough in the field of policy optimization in deep reinforcement learning, there is still room for improvements.

7 Bibliography

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. Proximal Policy Optimization Algorithms, *arXiv:1707.06347*, 2017

[2] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel: Trust Region Policy Optimization. *arXiv:1502.05477*, 2015

[3] Sutton Richard S., Barto Andrew G.. Reinforcement Learning: An Introduction, Chap 13.2, *MIT Press*, 2018

[4] Control Variates, *Wikipedia*

[5] Sutton Richard S., Barto Andrew G.. Reinforcement Learning: An Introduction, Chap 11.1, *MIT Press*, 2018

[6] Sutton Richard S., Barto Andrew G.. Reinforcement Learning: An Introduction, Chap 13.3, *MIT Press*, 2018

[7] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning, *arXiv:1602.01783*, 2016

[8] Sutton Richard S., Barto Andrew G.. Reinforcement Learning: An Introduction, Chap 13.4, *MIT Press*, 2018

[9] Josh Achiam, Pieter Abbeel. Trust Region Policy Optimization. *OpenAI, Spinning Up*, 2020

[10] Stephen Boyd, Liven Vandenberghe. Convex Optimization, $CambridgeUniversityPress$, Chap 2-5, 2004

[11] Mingfei Sun, Vitaly Kurin, Guoqing Liu, Sam Devlin, Tao Qin, Katja Hofmann. Shimon Whiteson. You May Not Need Ratio Clipping in PPO, *arXiv:2202.00079*, 2022

[12] Wangshu Zhu, Andre Rosendo. A Functional Clipping Approach for Policy Optimization Algorithms, *IEEE, DOI: 10.1109/ACCESS.2021.3094566*, 2021

[13] Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, Dale Schuurmans. Understanding the impact of entropy on policy optimization, *arXiv:1811.11214*, 2018

[14] Chloe Ching-Yun Hsu, Celestine Mendler-Dünner, Moritz Hardt. Revisiting Design Choices in Proximal Policy Optimization, *arXiv:2009.10897*, 2020

[15] OpenAi, Farama Foundation. Gymnasium, an open source library for reinforcement learning environments in Python, *https://gymnasium.farama.org/*, 2016

[16] Junwei Zhang, Zhenghao Zhang, Shuai Han, Shuai Lü. Proximal policy optimization via enhanced exploration efficiency, *Information Sciences Volume 609*, 2022