PROXIMAL POLICY OPTIMIZATION ALGORITHM ON OPEN-AI GYM CAR RACING V0 ENVIRONMENT

AI – 2B

Ahmed El Sheikh
1873337

SAPIENZA
UNIVERSITÀ DI ROMA

Table of Contents

Introduction

PPO is a member of policy gradient methods family for deep reinforcement learning, which alternate between sampling data through interaction with the environment (step no.1) and optimizing a surrogate objective loss function using stochastic gradient approach (step no.2). Whereas a standard policy gradient method performs one gradient update per data sample, a novel objective function that enables multiple epochs of mini-batch updates is proposed. The proximal policy optimization methods have some of the benefits of trust region policy optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). PPO were experimentally tested on a collection of benchmark tasks, including simulated robotic locomotion and Atari game playing. Results of these experiments showed that PPO outperforms other online policy gradient methods, and overall strikes a favorable balance between sample complexity, simplicity, and wall-time.

As per mentioned above PPO outperforms other online policy gradient methods, but what are online policy gradient methods? They are methods where

1. Agents can pick actions on their own
2. Follows most obvious setup (Learn with exploration, Play with exploitation)
3. Agent follows his own policy (Learn from either experts (Imperfect), or from recorded sessions (Recorded Data))

How it outperforms TRPO? TRPO is a trust region policy optimizer, function to optimize and some local approximation of this function which is accurate locally but goes inaccurate if we go far away from the starting point, so we have a trust region where we trust our local approximator & as long as we are in this region, we are willing to optimize are local approximation a lot. TRPO uses conjugate gradient approach to solve the problem, conjugate gradient approximates the constraint quadratically, and the objective function linearly, PPO simply uses linear approximation, apart from that TRPO had a constraint that it needs to optimize our function according to it, PPO includes this constraint in the equation as a penalty.

Moreover, PPO uses multiple epochs mini-batches updates instead of performing only one gradient update as per sample like policy gradient methods.

State of the Art

Authors of the PPO propose a novel objective with clipped probability ratios, which forms a pessimistic estimate (i.e., lower bound) of the performance of the policy. To optimize policies, they alternate between sampling data from the policy and performing several epochs of optimization on the sampled data.

Development of the PPO is built on basic policy gradient methods. Policy gradient methods work by computing an estimator of the policy gradient and plugging it into a stochastic gradient ascent algorithm. The most commonly used gradient estimator has the form:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

where π_θ is a stochastic policy and $\widehat{A_t}$ is an estimator of the advantage function at timestep t. Here, the expectation $\hat{E}[\ldots]$ indicates the empirical average over a finite batch of samples, in an algorithm that alternates between sampling and optimization. Implementations that use automatic differentiation software work by constructing an objective function whose gradient is the policy gradient estimator; the estimator $\hat{g}$ is obtained by differentiating the objective:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

While it is appealing to perform multiple steps of optimization on this loss L^{PG} using the same trajectory, doing so is not well justified, and empirically it often leads to destructively large policy updates.

In TRPO, an objective function (the "surrogate" objective) is a maximized subject to a constraint on the size of the policy update:

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t [\mathrm{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta. \end{aligned}$$

Here, θ_{old} is the vector of policy parameters before the update. The theory justifying TRPO suggests using a penalty instead of a constraint, i.e., solving the unconstrained optimization problem for some coefficient β:

$$\underset{\theta}{\text{maximize}} \, \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t - \beta \, \mathrm{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)] \right]$$

This follows from the fact that a certain surrogate objective (which computes the max KL over states instead of the mean) forms a lower bound (i.e., a pessimistic bound) on the performance of the policy π. TRPO uses a hard constraint rather than a penalty because it is hard to choose a single value of β that performs well across different problems — or even within a single problem, where the characteristics change over the course of learning. Hence, to achieve our goal of a first-order algorithm that emulates the monotonic improvement of TRPO, experiments show that it is not sufficient to simply choose a fixed penalty coefficient β and optimize the penalized objective this Equation with SGD; *additional modifications are required*.

So, the best approach to improve the situation was proposed by the OpenAI team. Its name is **Clipped Surrogate Objective**.

TRPO maximizes a "surrogate" objective:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right]$$

The superscript CPI refers to conservative policy iteration, where this objective was proposed. Without a constraint, maximization of L^{CPI} would lead to an excessively large policy update; hence, we now consider how to modify the objective, to penalize changes to the policy that move $r_t(\theta)$ away from 1.

The main objective we propose is the following:

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

where epsilon is a hyperparameter, as proposed by the paper, ε = 0.2. The motivation for this objective is as follows. The first term inside the min is L^{CPI}. The second term, clip($r_t(\theta)$, 1− ε, 1+ ε) $\widehat{A_t}$, modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving r_t outside of the interval [1 − ε, 1 + ε]. Finally, we take the minimum of the clipped and unclipped objective, so the final objective

is a lower bound (i.e., a pessimistic bound) on the unclipped objective function. With this scheme, we only ignore the change in probability ratio when it would make the objective improve, and we include it when it makes the objective worse. Note that $L^{CLIP}(\theta) = L^{CPI}(\theta)$ to first order around θ_{old} (i.e., where r = 1), however, they become different as θ moves away from θ_{old}. Figure 1 plots a single term (i.e., a single t) in L^{CLIP}; note that the probability ratio r is clipped at 1 − or 1 + depending on whether the advantage is positive or negative.

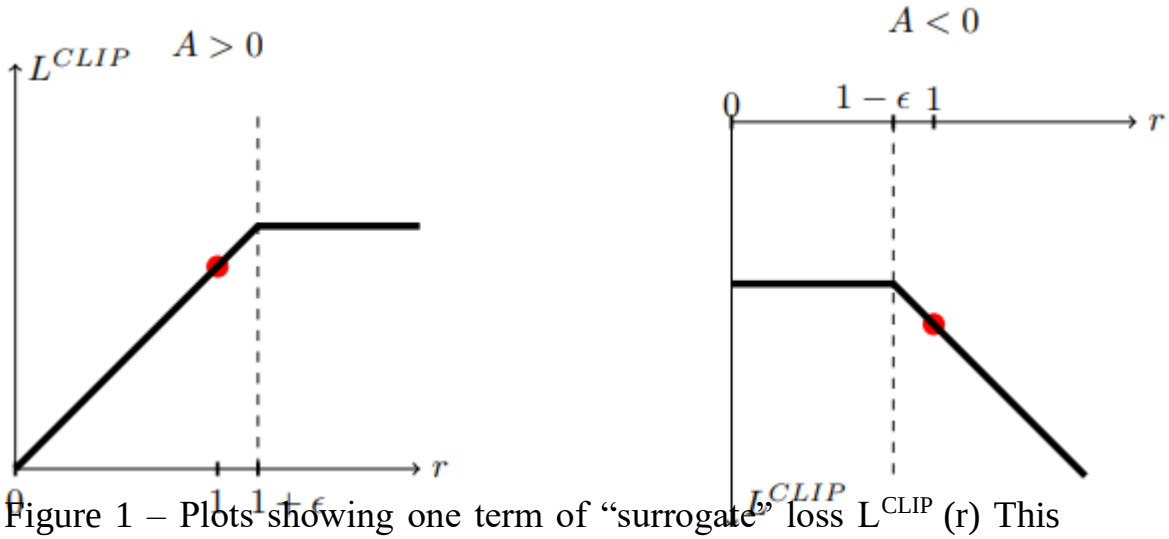


Figure 1 – Plots showing one term of "surrogate" loss $L^{CLIP}(r)$ This approach was used.

The surrogate losses from the previous sections can be computed and differentiated with a minor change to a typical policy gradient implementation. For implementations that use automatic differentiation, one simply constructs the loss L^{CLIP} instead of L^{PG}, and one performs multiple steps of stochastic gradient ascent on this objective.

Methodology

Briefly, the PPO algorithm can be described as follows at the figure 2:

Algorithm 1 PPO, Actor-Critic Style

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

Figure 2 – PPO algorithm description

We start with processing the frames for our network to take it as input, so we have to remove the 3 color channels RGB for reduced computational parameters, as well as, color adds useless information added to cropping irrelevant information, as this is our input frame

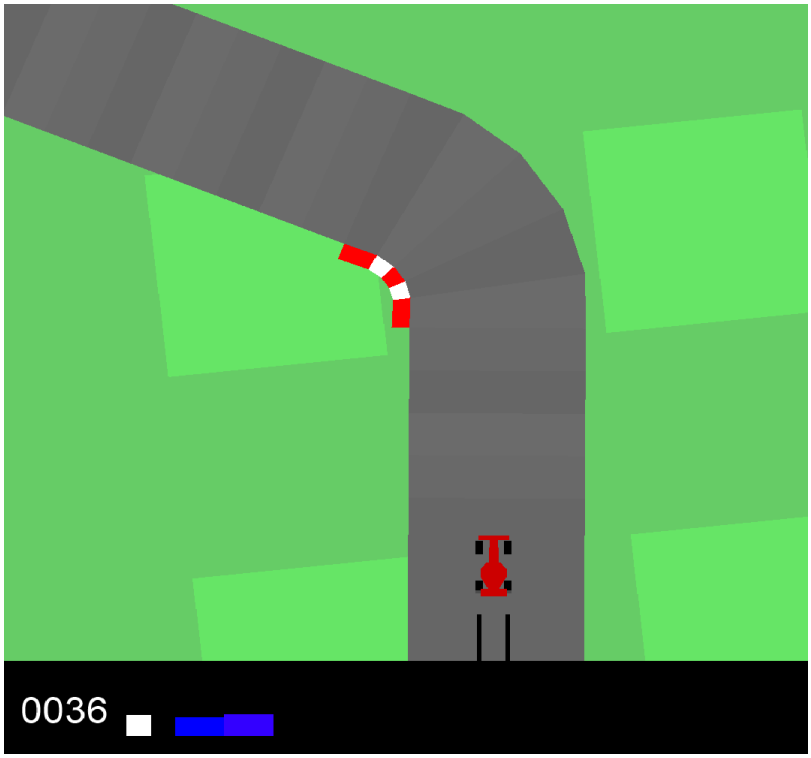


Figure 3 Input frame from the environment

So our processing steps are as follows (in the following figure)

```python
def crop(frame):
    # Crop to 84x84
    return frame[:-12, 6:-6]


def rgb2grayscale(frame):
    # change to grayscale
    return np.dot(frame[..., 0:3], [0.299, 0.587, 0.114])


def normalize(frame):
    return frame / 255.0


def preprocess_frame(frame):
    frame = crop(frame)
    frame = rgb2grayscale(frame)
    frame = normalize(frame)
    frame = frame * 2 - 1
    return frame

```

Figure 4 Preprocessing steps followed

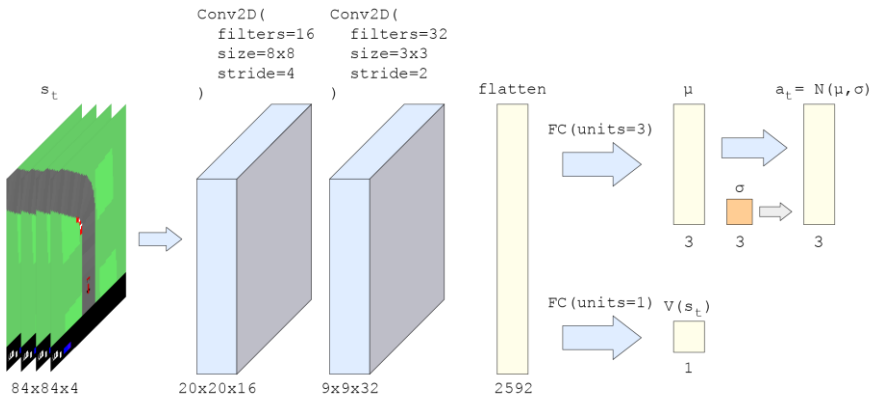
And our Training parameters (hyperparameters) are

Initial learning rate	0.0003
Discounting factor	0.99
GAE Lambda	0.95
Epsilon (Clipping coefficient for PPO)	0.2
Value Scaling coefficient	0.5
Entropy Scaling coefficient (to encourage exploration)	0.01
Horizon	128
Num of epochs	10
Batch size	128
Number of environments (agents)	16

Non training params

Save interval	1000
Evaluate interval	200
Record episodes	True

From the above we can see number of environments as my algorithm is based on A3C (Async Advantage Actor Critic network), which is basically based on an actor network as well as a critic network

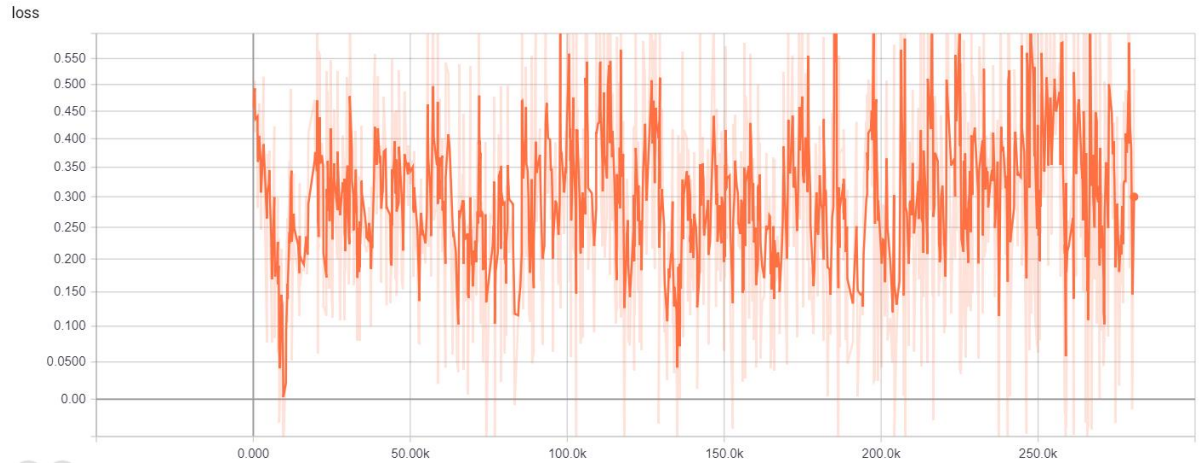


And our loss function is as follows,

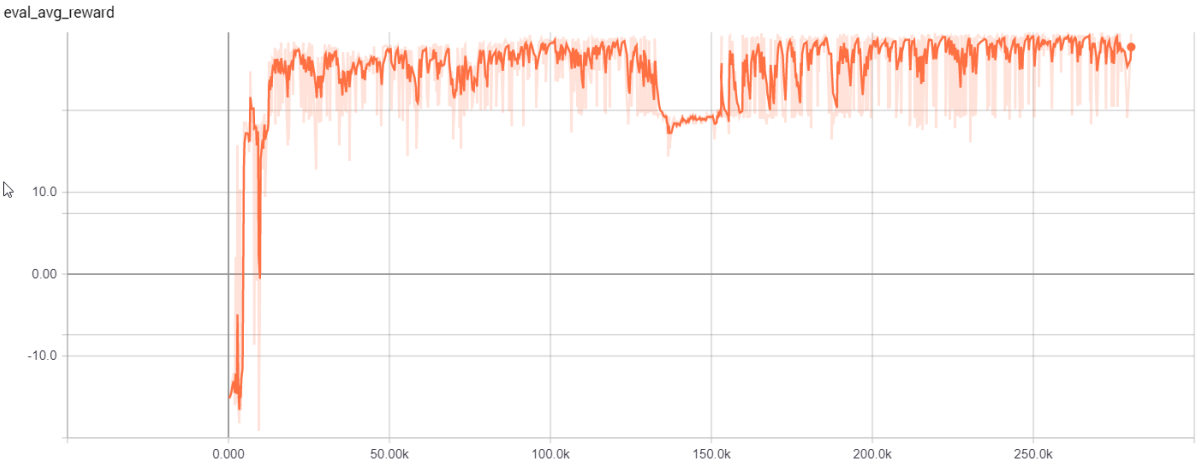
$$L^{CLIP+VF+S}(\theta) = -\hat{\mathbb{E}}_t\left[L^{CLIP}(\theta) - \alpha L^{VF}(\theta) - \beta\frac{1}{2}\left(\log\left(2\pi\sigma^2\right)+1\right)\right]$$

Results

Policy Loss



Evaluation Reward as per steps



Returns