"La Sapienza" University of Rome
Faculty of information engineering, information technology and statistics
Department of informatics, automation and control engineering
"ANTONIO RUBERTI"
Degree program: Artificial Intelligence and Robotics

SAPIENZA
UNIVERSITÀ DI ROMA

COURSE PROJECT

Training of the OpenAI Humanoid using the PPO algorithm

Student: OREL, Egor
Matricola: 1836231
Cloud storage containing the solution: https://yadi.sk/d/R0OKLHBiEuRYyQ
GitHub with the code: https://github.com/EgOrlukha/MuJoCo-PyTorch.git

Teacher: CAPOBIANCO, Roberto

Rome, 2019

Exercise: to train a Humanoid (a physical model with 27 dof which is made by OpenAI team) using the algorithm of PPO (Proximal Policy Optimization)

Introduction

PPO is a member of a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. Whereas standard policy gradient methods perform 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.

State of Art

The PPO algorithm is mostly equivalent with TRPO algorithm. The difference is that it improves the current state of affairs by introducing an algorithm that attains the data efficiency and reliable performance of TRPO, while using only first-order optimization. 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 $\hat{A}_t$ is an estimator of the advantage function at timestep t. Here, the expectation $\hat{E}_t[\ldots]$ 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:

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

Here, θ_{old} is the vector of policy parameters before the update. The theory justifying TRPO actually 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. It name's **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:

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

where epsilon is a hyperparameter, say, $\epsilon = 0.2$. The motivation for this objective is as follows. The first term inside the min is L^{CPI} . The second term, clip($r_t(\theta)$, $1-\epsilon$, $1+\epsilon$)$\hat{A}_t$, modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. 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. 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 $1 - \epsilon$ or $1 + \epsilon$ depending on whether the advantage is positive or negative.

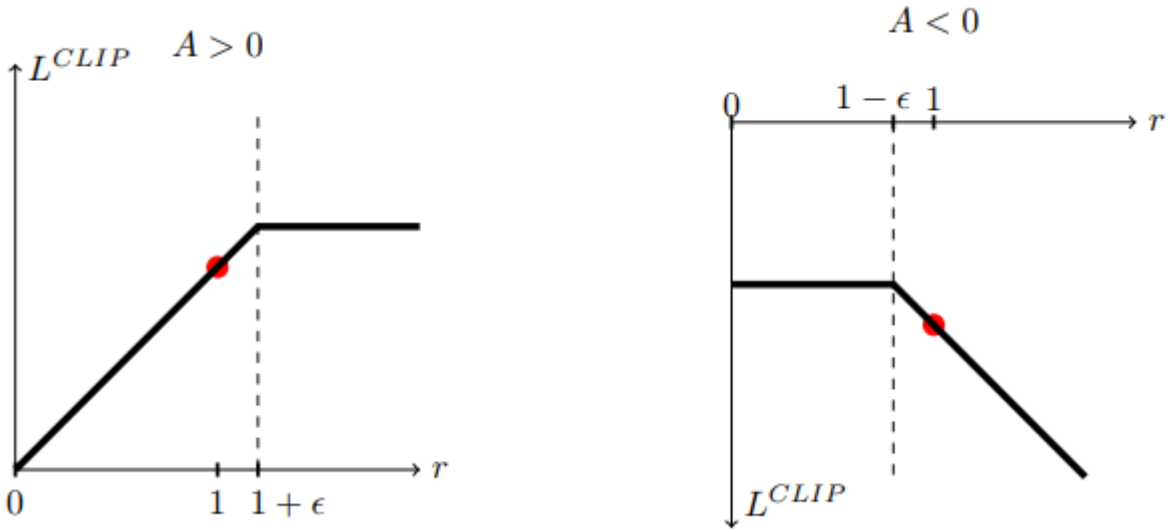

Figure 1 – Plots showing one term of "surrogate" loss $L^{CLIP}(r)$

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.

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

Results

First of all, I installed gym, MuJoCo and several packages which were needed. The main problem which required the longest solution was the installation of MuJoCo environment: I tried to install it to Windows 64x, had some troubles,

because Windows support is deprecated and has no current support. Then I tried to work with Ubuntu 16.04 using VirtualBox, but model rendering was too slow (because it uses reduced RAM) and I got some troubles with Anaconda. Then I came back to Windows platform and found a workable solution of the MuJoCo environment for Windows.

To find an approach of implementation of the algorithm I decided to start from the OpenAI Github (https://github.com/openai/baselines). The team propose a baselines algorithms package, which includes all the most important high-quality implementations of reinforcement learning algorithms. I tried to work with PPO2 part of the set: I trained model for 20 millions of timesteps and have got the following dynamics:

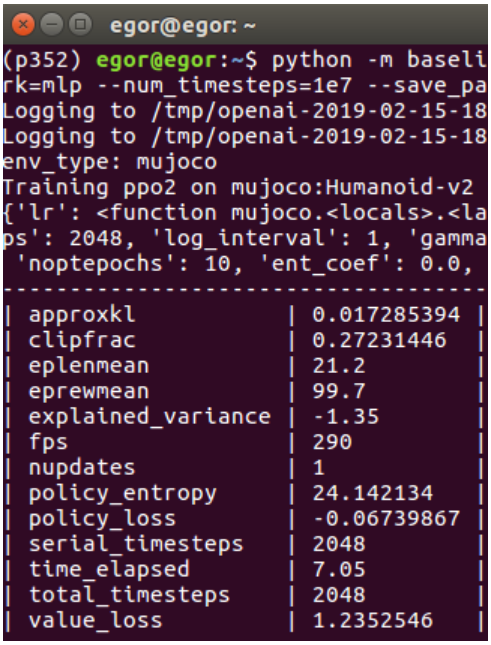

Figure 3 – Before training

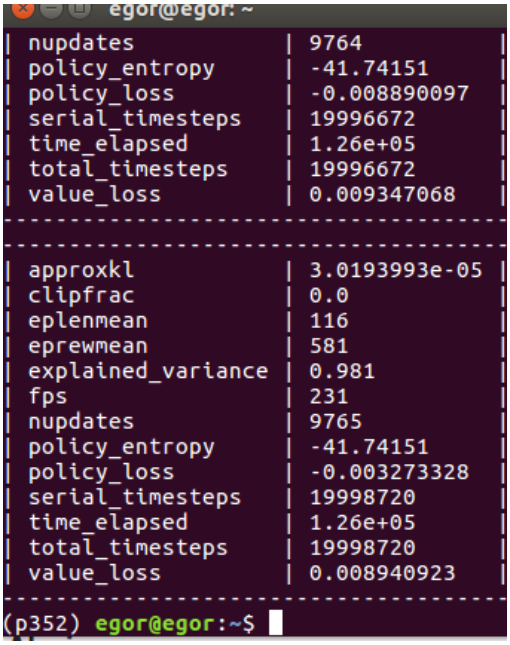

Figure 4 – After training

As we can see from the main settings: policy entropy and policy loss are decreasing during training, so it means that the model starts to make more and more efficient and balanced decisions about the next step, hence training is successful. But this algorithm has a stuck with loading of a trained model, so I had to find a new algorithm to work with.

Another variant, which I tried, is a kind of modified versions of OpenAI Baselines, but made using PyTorch library (https://github.com/dnddnjs/mujoco-pg). It provides training of continuous MuJoCo environments according to 4 algorithms: PG, NPG, TRPO, PPO. I modified the PPO algorithm in the following way:

— made the PPO algorithm and the Humanoid-v2 environment run by default;
— added Average score per Episodes dependency graph as an output of the algorithm;
— modified number of iterations counted by algorithm to avoid long training and to be able to see just tendencies.

I tried to train the system 3 times and stored results: twice per 20 iterations and once per 50 iterations (one iteration is about 90 episodes). Finally, I had following results:

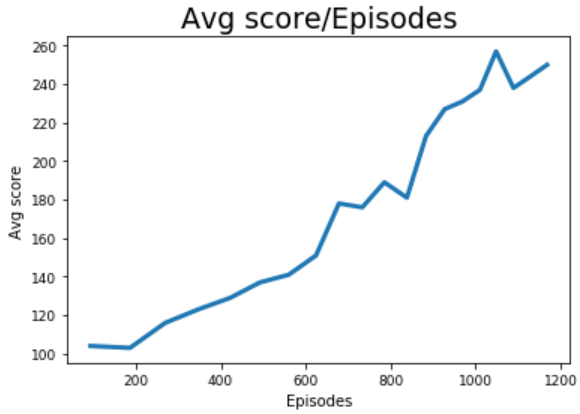

Figure 5 – Result after 20 iterations (1)

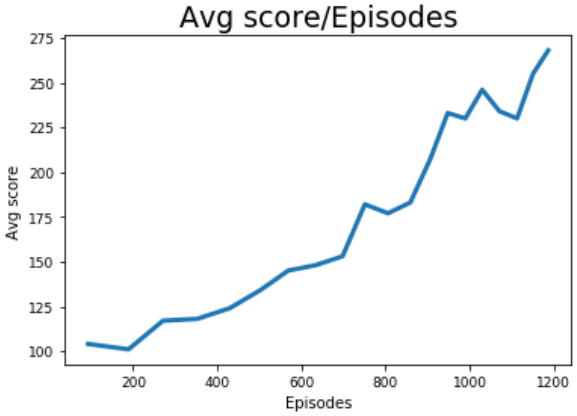

Figure 6 – Result after 20 iterations (2)

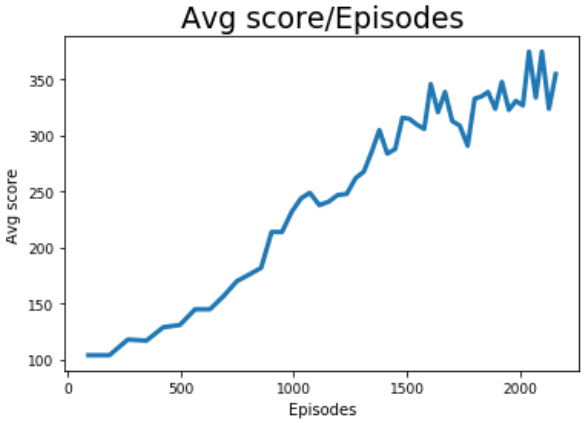

Figure 7 – Result after 50 iterations

Conclusions

Results of training of the system gives a positive tendency. It is possible to see from dependency graphs (average score grows with time) and from the video (https://yadi.sk/i/Z1mdYFsZGBnigA) of the trained Humanoid (it moves better gradually). Results of the PPO algorithm show good result of training and one of the best in comparison with other algorithms of reinforcement learning (as it is written in the paper https://arxiv.org/pdf/1707.06347.pdf).

The code that I used can be modified by adding of an Adaptive KL Penalty Coefficient algorithm to the Clipped Surrogate Objective one.