Trust Region Policy Optimization for HalfCheetah-v2

Luca Polenta 1794787, Michela Proietti 1739846

February 5, 2021

Abstract

In this project, the TRPO (Trust Region Policy Optimization) algorithm has been implemented using python3 and tensorflow 2. Mujoco's HalfCheetah-v2 environment has been used to test it and different results obtained by changing the values of several hyperparameters are shown. In addition, an alternative implementation that has some features of PPO algorithm has been developed and tested on the same environment.

1 Policy gradient methods

In policy gradient methods, we try to optimize a parametrized policy directly using a function approximator that has as weights the parameters of the policy. The policy can therefore select the actions to execute without using a value function, that is still useful to learn the parameters of the policy. The most common gradient estimator that is used to perform a stochastic gradient ascent algorithm in order to optimize the parameters of the policy is the following:

$$\hat{g} = \hat{\mathbf{E}}_t[\nabla_\theta log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{1}$$

where $\hat{\mathbf{E}}_t$ is an empirical average over a finite batch of samples, π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at timestep t. This estimator is obtained by differentiating the objective function:

$$J(\theta) = \hat{\mathbf{E}}_t[log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2}$$

The advantage function is given by the following formula:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \tag{3}$$

where $Q^\pi(s_t, a_t)$ is the expected discounted cumulative reward given state s_t and action a_t, while $V^\pi(s_t)$ is the value function for state s_t, that measures potential future rewards we may get from being in this state. A good estimation of the advantage function that we have used in our implementation of the TRPO algorithm is the temporal difference error, that is defined as follows:

$$\delta(s, a, s') = R(s, a, s') + \gamma V(s') - V(s) \tag{4}$$

where $R(s, a, s')$ is the reward we get in state s' after executing action a from state s, γ is the discount factor, $V(s')$ is the value function for state s' and $V(s)$ is the value function for state s.

1.1 Trust Region Policy Optimization (TRPO)

TRPO is an on-policy method, which means that we try to optimize the same policy that we use to make decisions and therefore data that comes from the past or from versions of the policy that are different from the one we are learning about cannot be reused, because otherwise we may have unstable performance and updates. So, we can use data coming from the same trajectory as long as the policy does not move too far from the one that originated the data. Therefore, in TRPO we use an objective function, called surrogate objective, that is maximized subject to a trust region constraint, which allows us to take the largest step to improve performance, while still having that the new and old policies are not too far away. We measure the distance between the two policies by using the KL divergence, that is defined as follows:

$$KL(\pi_\theta, \pi_k) = \sum \pi_k log \frac{\pi_k}{\pi_\theta} \tag{5}$$

where π_k is the output of the neural network representing the policy at iteration k with parameters θ_k, that represents a probability distribution over the action space, while π_θ is the new version of the policy.
So, the problem is to find the set of parameters that allows us to maximize the surrogate objective, with a sufficiently small distance between the two policies and it can be formalized in the following way:

$$\underset{\theta}{argmax}\ \hat{\mathbf{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \hat{A}_t \right] \text{ subject to } \hat{\mathbf{E}}_t \left[KL \left[\pi_{\theta k}(\cdot|s_t), \pi_\theta(\cdot|s_t) \right] \right] \leq \delta \tag{6}$$

where $\pi_\theta(a_t|s_t)$ is the new policy, $\pi_{\theta_k}(a_t|s_t)$ is the old version of the policy, $\hat{A}_t$ is the estimation of the advantage function, and the trust region constraint is the average KL divergence:

$$\overline{KL}(\theta, \theta_k) \leq \delta \tag{7}$$

that imposes that the difference between the distribution coming out from the old policy and the one coming out from the new policy must be smaller than a certain value δ. In order to compute the search direction and therefore the update, we can use a linear approximation of the objective function and a quadratic approximation of the constraint:

$$L(\theta) \approx g^T(\theta - \theta_k) \text{ with g = conjugate gradient} \tag{8}$$

$$\overline{KL}(\theta, \theta_k) \approx \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \text{ with } H = \nabla\nabla\overline{KL}(\theta, \theta_k) \tag{9}$$

In this way, we get to the approximated optimization problem:

$$\theta_{k+1} = \underset{\theta}{argmax} g^T(\theta - \theta_k) \text{ subject to } \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta \tag{10}$$

It can be solved analytically and then we add to this solution a backtracking line search by using a backtracking coefficient α:

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g \tag{11}$$

where $\alpha \in (0, 1)$ and j is the smallest non-negative integer such that $\pi_{\theta k+1}$ satisfies the KL constraint and produces a positive surrogate advantage. However, H^{-1} is very expensive to compute, therefore we use the conjugate gradient algorithm to solve $Hx = g$ instead of $x = H^{-1}g$.

1.1.1 Conjugate Gradient and Backtracking Line Search

If we use steepest descent to optimize a nonlinear function, it may happen that some steps are along directions that undo the progress of the others, so the idea of the conjugate gradient method is to move in non-interfering directions. Suppose we move along the direction u, then we should have that the gradient ∇f at the current point is perpendicular to u because otherwise we would have been able to move further along u. If the next direction along which we move is v, we want ∇f to be perpendicular to u before and after we move along v. This means we want v to be Hf-orthogonal to u:

$$\mathbf{u}^T A \ \mathbf{v} = 0 \text{ with } A = Hf \tag{12}$$

If A is symmetric and positive definite and the vectors $p_1, ..., p_n$ are conjugate w.r.t. A, then we express the solution of our optimization problem $Ax = b$ as:

$$x_* = \sum_{i=1}^{n} \alpha_i p_i \tag{13}$$

We implemented the conjugate gradient method iteratively, by starting from an initial guess of $x_* = x_0 = 0$, which means we have $p_0 = b - Ax_0 = b$. We can notice that p_0 is also the residual of the initial step of the algorithm, and in general we have that the residual at the kth step is:

$$r_k = b - Ax_k \tag{14}$$

and it corresponds to the negative gradient of f at $x = x_k$, so in the gradient descent method we would move along the direction r_k. However, in this case we want the directions p_k to be conjugate and this can be obtained by requiring that the next search direction is built out of the current residual and all previous search directions:

$$p_k = r_k - \sum_{i<k} \frac{p_i^T A r_k}{p_i^T A p_i} p_i \tag{15}$$

After having chosen the search direction, we use a backtracking line search to determine how far we want to move along the computed direction, and by doing this the next optimal location we obtain is:

$$x_{k+1} = x_k + \alpha_k p_k \text{ where } \alpha_k = \frac{r_k^T r_k}{p_k^T A p_k} \tag{16}$$

while the residuals are updated as follows:

$$r_{k+1} = r_k - \alpha_k A p_k \tag{17}$$

Then, if the residuals we get is sufficiently small, we exit the loop and return x_{k+1} we have computed, otherwise we can compute the next direction as:

$$p_{k+1} = r_{k+1} + \beta_k p_k \text{ where } \beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \tag{18}$$

1.2 KL Penalty and Proximal Policy Optimization (PPO)

In TRPO we could also consider the unconstrained problem:

$$\underset{\theta}{argmax}\ \hat{\mathbf{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \hat{A}_t - \beta KL\left[\pi_{\theta k}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right] \right] \tag{19}$$

where we consider a penalty instead of a constraint. However, in this case if we use a fixed penalty coefficient β it is unlikely that we will get good results, because it is very hard to find a value for β that works well during all the training. Therefore, in PPO method one option is to use an adaptive KL penalty coefficient in order to get to a target value of the KL divergence. So, the algorithm performs several epochs of SDG to optimize the objective function, and then it computes the KL divergence, and depending on its value w.r.t. the target KL, it updates the β coefficient, that is used for the next policy update.

2 Implementation

The implementation of the code that applies TRPO algorithm has been split into 4 principal files: policy.py, value.py, utils.py and main.ipynb .
The "main.ipynb" file allows to run the entire algorithm. Initially, some hyperparameters and global variables are defined, including the environment, the policy, the value function, the logger and the directories for saving the logs. Then, there is a loop over the number of episodes in which various functions are called. First of all, the run_policy function is called and it calls the run_episode function for batch_size episodes in order to collect some trajectories. The run_episode function samples the actions and executes the steps in the environment until the boolean variable "done" returns True. Meanwhile, it also computes observations, actions, rewards and the entropy. When the set of trajectories is obtained, the estimate of the value function is computed and added to them. After this, the algorithm computes the sum of discounted rewards and the advantage. When it obtains these values, they will be used for determining the policy update and the fit of the value function. At the end of each cycle, the algorithm prints on the terminal some useful information in order to understand how the training progresses.
The "policy.py" file contains the definition of the TRPO policy. It is characterized by

3 functions: the initialization of the hyperparameters, the sampling of an action w.r.t. a normal distribution and the update of the policy w.r.t. observations, actions and advantages. Furthermore, within this last function, there are defined other functions that allow to perform the update. The most important ones are the surrogate loss, the line search and the conjugate gradient. The surrogate loss allows us to compute the surrogate objective defined in equation 6. The conjugate gradient implements the conjugate gradient algorithm to speed up the search for the optimal direction to follow. Finally, the line search function allows us to identify the next optimal set of parameters according to the direction computed by using the conjugate gradient method and the step size defined through backtracking line search.
The "value.py" file contains the definition of the neural network for the value function. There are four main functions: the initialization of the hyperparameters, the initialization of the network model, the fit function and the predict function. Both the policy and the value function have a neural network made up of only Dense layers. The policy neural network is composed of 2 Dense layers, while the neural network of the value function is composed of 3 Dense layers. The number of nodes of each layer is computed based on the number of nodes of the neighbouring layers and, in the case of the input layer, based on the size of the observation space, and some constant multipliers chosen experimentally have been used to increment the number of nodes in the first and last hidden layers. Finally, for both networks, the activation functions are all Leaky-ReLU because they have experimentally reported better results.
In the end, the "utils.py" file contains some utilities for computing the mean and variance of the observations and scaling them according to these values.

2.1 Alternative Implementation

In order to improve the results of the algorithm, there is also an alternative implementation of the code, in which all the files are the same as before and the main difference is in the policy, that now is called "policy_v2.py". In this variant of the policy, the algorithm has been modified by integrating some aspects of the PPO algorithm in order to compute the policy update. Specifically, there is no explicit definition of line search and the conjugate gradient. Instead, the surrogate loss is optimized by performing the fit of the policy neural network. Moreover, in this case the loss is computed by summing two terms: the surrogate loss as defined in the previous implementation and the KL penalty, which consists in the product between the KL distance and the β value. Furthermore, the value of the learning rate of the neural network and the beta value used in the computation of the loss are modified according to KL divergence in order to reach a desired kl_target. These corrections are made because we do not want the new policy to be too far away when we compute an update. Finally, in this variant the policy network is larger than the one defined previously. In fact, it is now composed by 3 Dense layers with a greater number of nodes, computed in the same way as in the previous implementation, but using larger constant multipliers that allowed to get better results. In addition, the learning rate and logvar_speed are set based on the size of the network, while the activation functions are all Leaky-ReLU like before.

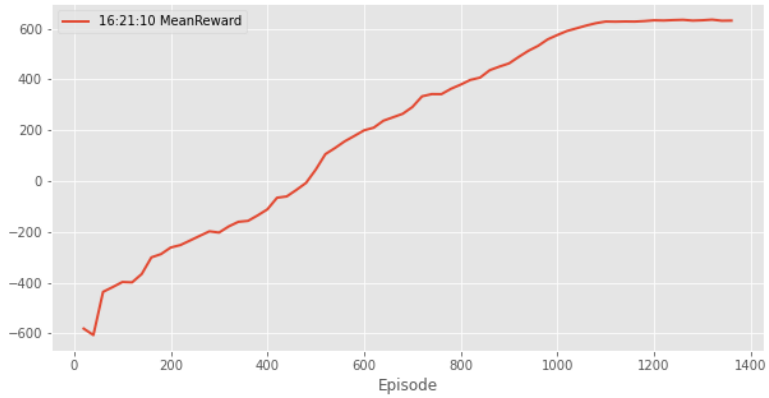
3 Experiments and Results

Once the code has been structured, some tests have been carried out in order to obtain the best possible results in the considered environment. Starting from the recommended values for the most important hyperparameters, the following values have been identified as optimal:

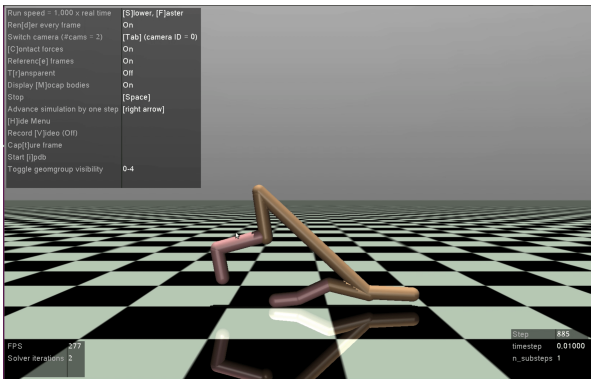
- **num_episodes:** it is the maximum number of episodes. It was set to 1400;
- **batch_size:** it is the number of times (episodes) run_episode is executed when the collections of trajectories are computed at each cycle in the main. It was set to 20;
- **gamma:** it is a value defined between 0 to 1, called discount factor, which is used to scale the rewards of the trajectories. It was set to 0.995;
- **hid1_size:** it is an integer number multiplied by the size of the observation space in order to obtain the number of nodes of the first Dense layer of the policy neural network. It was set to 8;
- **epsilon:** it is a value defined between 0 and 1 and it is used to understand whether to perform a random action or to perform exploitation. If the number obtained randomly is greater than epsilon then exploitation is performed, otherwise exploration is performed. It was set to 0.4;
- **epsilon_decay:** it decreases the epsilon at the end of each policy update, so after each batch_size episode, by a certain amount. The update of the epsilon value is done by taking the maximum between 0 and the subtraction of a certain amount from the current epsilon value (0.008 in this case) so that epsilon never becomes negative.

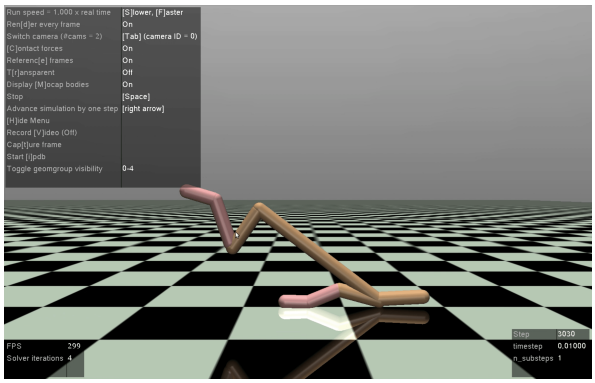
Many of these values have been chosen as they guarantee maximum performance. For example, epsilon is updated at the end of each policy update because keeping it at a stable value would not allow for enough exploration at the beginning of the process or enough exploitation at the end of the process. Consequently, the value is decreased a little bit at each update of the policy, thus having greater randomness (exploration) and only when the agent begins to have greater knowledge there is more exploitation. Furthermore, the value is chosen as the maximum between the decrease of the actual value of epsilon and a minimum fixed value in order to prevent epsilon to become negative.
Another issue was choosing the exact value of batch_size. In fact, values that were too small or too large did not allow to follow an optimal path and to correct errors in its computation in time. Furthermore, it has also been tried to vary the value of batch_size during execution: it was set to a smaller value (10 to be exact) after a fixed number of episodes, because from then on the values of the reward and of the loss started to be unstable. Sometimes they decreased and increased alternatively, and then improved their trends again between 20 and 60 episodes later. So, supposing that the algorithm had already learned enough how to move in the environment, in order to avoid such losses, we tried to carry out a smaller number of iterations in an attempt to correct the losses in advance and thus improve the trend. However, the obtained result was different. In fact, what was reported in the output was a

trend that could no longer find the direction towards the optimum and therefore continued to oscillate around the same reward value. So, at the end, it was decided to keep the batch_size value as constant at 20.
Another interesting aspect to note is how the node numbers of the Dense layers are computed. Networks were originally created with a fixed number of nodes. Initially set to 64 and then raised to 96 in some layers and finally to 128, the networks did not report any noteworthy improvement. Subsequently, it was decided to approach the definition of the nodes of the layers trying to compute the number of nodes based on the number of nodes of the neighbouring layers and, in the case of the input layer, based on the size of the observation space. Some values used in these operations were obtained empirically based on the problem under consideration and they reported notable improvements.
In general, the obtained results showed a sufficiently increasing trend of the reward, starting from a value of -580 to arrive at a final reward of +630 in about 1050 epochs, as shown in the next image.

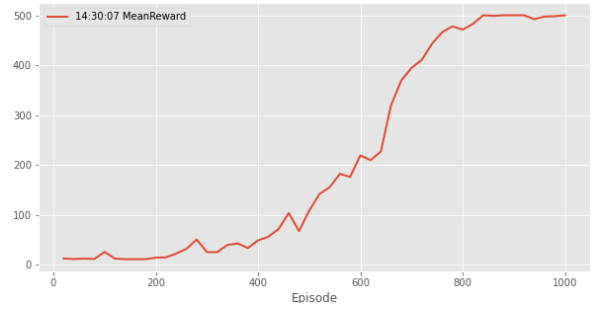


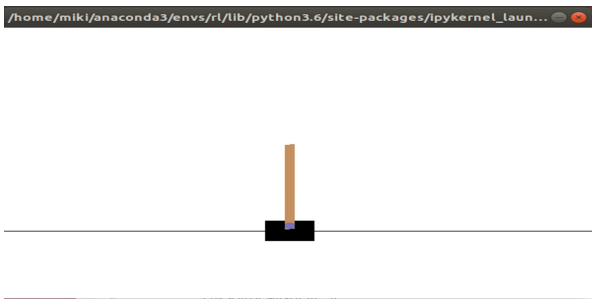
An observation that can be made is that when the training reaches 1100 epochs it is no longer possible to notice improvements in the reward, even if the training has continued up to 1400 epochs. This trend is not positive, as it means that the agent is no longer able to improve his behavior. In fact, even though there has been a good improvement from the beginning, they did not make the agent move in an optimal way: the hind limb moves sufficiently well, while the front limb seems to move much less, causing the head to rub on the floor. An example of its position as it moves is in the following images.





Failing to train HalfCheetah-v2 environment, it was necessary to understand if it was the algorithm that had bugs or if it worked in other environments, therefore it was decided to try this algorithm on a simpler environment. The chosen environment is "CartPole-v1", one of the classic gym environments. The result was excellent and the pole remained balanced on the cart, as shown in the next image.

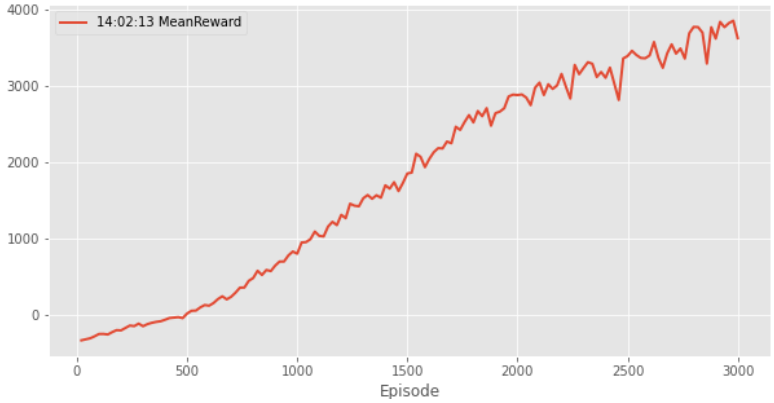




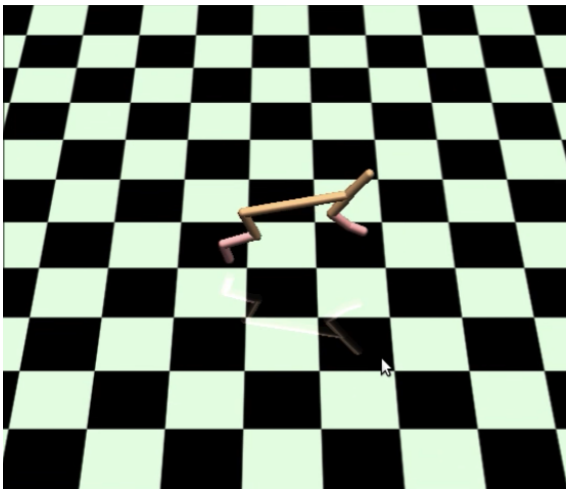
Therefore, it can be concluded that our implementation of the TRPO algorithm works well on simpler environments. However, in order to present an implementation that works also with Mujoco's HalfCheetah environment, a slightly different algorithm has been developed and tested.

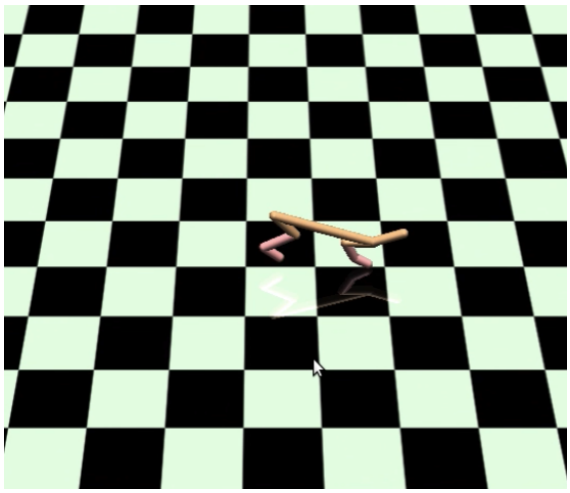
3.1 Experiments and Results of the Alternative Implementation

Even in this second implementation, some hyperparameters have been set and tested, just as it was done in the previous case. Some of them are the same and have been set to the same values as before, including for example the maximum number of epochs, the batch_size and the value of gamma, while the number of episodes has been set to 3000. Some additional hyperparameters are instead the kl_target, which has been set to 0.01, and the adaptive β coefficient.
In this second case, the agent starts with a reward of about -300 and ends at 3000 episodes with 3600. During the training, a peak of 3847 peaks was also reached. The trend is shown in the following image.



It is possible to notice that for the first 2000 epochs there was an increase in the reward and just in a few episodes there were small degradations. Instead, from 2000 epochs the trend began to be more unstable, but with general improvements. In any case, the algorithm was sufficiently adequate to recover the losses and to continue increasing the reward. All this then turned into an almost perfect performance of the agent, as shown by the following images.





4 Conclusion

In its purest and most faithful form, the TRPO algorithm is unable to correctly train Mujoco's HalfCheetah-v2 environment, although it works perfectly on simpler environments such as CartPole-v1. Instead, the TRPO algorithm enhanced with some aspects of PPO was able to train even a more complex environment.

References

[1] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel, *Trust Region Policy Optimization*, 2017, avaiable at https://arxiv.org/abs/1502.05477.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov, *Proximal Policy Optimization Algorithms*, 2017, avaiable at https://spinningup.openai.com/en/latest/algorithms/ppo.html

[3] *Conjugate Gradient*, available at https://web.cs.iastate.edu/~cs577/handouts/conjugate-gradient.pdf and https://en.wikipedia.org/wiki/Conjugate_gradient_method .

[4] *Mujoco - HalfCheetah-v2*, available at https://gym.openai.com/envs/HalfCheetah-v2

[5] *Gym - CartPole-v1*, available at https://gym.openai.com/envs/CartPole-v1