RL Final Project: TRPO implementation in a Mujoco environment

Sergio Calo
sergio.calo01@estudiant.upf.edu

Aaron Verdaguer
aaron.verdaguer01@estudiant.upf.edu

Keywords: TRPO, Reinforcement Learning, MuJoCo

***Abstract*—In this project we have implemented the Trust Region Policy Optimization algorithm on different agents in the MuJoCo physical simulation environment with continuous action spaces. In this report, we will explain the motivations of the work, analyze and understand the algorithm and present the results obtained in different training processes. Thanks to these results, we will conclude with a discussion of the effects that the different parameters have on the learning behavior, which will help us to understand in depth the details of the algorithm.**

I. Introduction

A. Motivation

The main motivation behind this project was getting hands-on experience with more complex algorithms that the ones used in previous lab sessions. Reinforcement learning (RL) can be used for many tasks and many different algorithms have been presented in class but the one that drew the most attention to us was the Trust Region Policy Optimization (TRPO) [1]. More specifically the part where the authors talk about locomotion experiments. Replicating, understanding, and using TRPO to solve different environments is the main idea behind the project.

In order to define our project we need to define some key concepts and terminology for Markov Decision Processes (MDPs), a formal description of Reinforcement Learning (RL) problems. As it has been discussed in this course MDPs can be defined as MDP = (S, A, P, R, γ) where:

- S is a finite set of states
- A is a finite set of actions
- P is a state transition probability
- R is a reward function
- γ is a discount factor $\gamma \in$ [0, 1]

The idea behind RL is to study how well agents perform in a given environment. What determines what action an agent does in a given state is called policy (π). Depending on the action an agent takes in a given state he is either rewarded or punished based on the reward function. Thus RL formilizes rewarding or punishing an agent based on how well it performs, making it more likely for the agent to undergo actions that will maximize the final reward.

Since the intention of this project is to replicate the locomotion part of the TRPO original paper (section 8.1) we have also conducted the robotic locomotion experiments using the MuJoCo simulator [2]. Because we also wanted to get a better understanding of how parameters affect TRPO and see how it performed with different models we have created a control panel to choose (see Figure 2 at the Annex) the value of the variables and to choose different models to compare different complexities. The states of the problem are their generalized positions and velocities, and the actions are the joint toques just like in the paper trying to replicate. Also agents are rewarded for standing up and moving forward and an episode ends when it falls. The three simulated agents are depicted in Figure 1 at the Annex.

B. TRPO

TRPO is a RL algorithm that seeks to update policies taking the biggest step possible to improve performance but within its trusted region. This trusted region is constrained by Kullback–Leibler divergence (KL-Divergence) which can be seen as a measure to measure how similar two probability distributions are and is defined as follows for discrete probability distributions:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right).$$

This makes a huge difference from normal policy gradient algorithms by ensuring that policy performances are close by. Normal policy gradient keeps old and updated policies close in their parameter space but this cannot ensure that difference in performance is not big. This is the main reason for giving small learning rates to vanilla policy gradient algorithms. TRPO tends to improve performance monotonically and quicker than other algorithms and it can be used in discrete and continuous environments.

The algorithm itself consists in improving policy parameters as follows. Allow π_θ to be a policy with parameters θ then the TRPO update goes as follows:

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}(\theta_k, \theta) \quad \text{s.t. } \bar{D}_{KL}(\theta || \theta_k) \leq \delta$$

As mentioned previously the distance between parameters performance based on an average KL-divergence between policies across states visited by the old policy ($D_{KL}(\theta||\theta_k)$) which can't be higher than the threshold δ. The parameter selection has to be made in such a manner that the surrogate advantage ($\mathcal{L}(\theta_k, \theta)$) , which measures policy performance comparing the new policy to the old policy, is maximized, always staying within the trust region thanks to the constraints. The average KL-divergence and the surrogate advantage measures the difference between policies as follows:

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right],$$

$$\bar{D}_{KL}(\theta||\theta_k) = \underset{s\sim\pi_{\theta_k}}{\mathrm{E}} \left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right].$$

Due to the computational power it takes and the complexity behind computing the formulas as expressed previously, TRPO uses Taylor expand to make approximations and reducing problem complexity as follows:

$$\begin{aligned} \mathcal{L}(\theta_k, \theta) &\approx g^T(\theta - \theta_k) \\ \bar{D}_{KL}(\theta||\theta_k) &\approx \frac{1}{2}(\theta-\theta_k)^T H (\theta-\theta_k) \end{aligned}$$

$$\begin{aligned} \mathcal{L}(\theta_k, \theta) &\approx g^T(\theta - \theta_k) \\ \bar{D}_{KL}(\theta||\theta_k) &\approx \frac{1}{2}(\theta-\theta_k)^T H (\theta-\theta_k) \end{aligned}$$

Which leaves us with the following approximate optimization problem:

$$\begin{aligned} \theta_{k+1} = \arg\max_\theta \; & g^T(\theta-\theta_k) \\ \text{s.t. } & \frac{1}{2}(\theta-\theta_k)^T H(\theta-\theta_k) \le \delta. \end{aligned}$$

This optimization problem can be solved using the method of Lagrangian duality leading to this result:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g.$$

This would give us the Natural Policy Gradiant [3] which basically represents the steepest descent direction based on the underlying structure of the parameter space to move toward choosing a greedy optimal action rather than just a better action. But a problem that comes with the approximation made previously is that it cannot be ensured that this follows the KL-Divergence constraint nor that improves performance. Thus TRPO adds and extra step to this update rule, a backtracking line search that goes as follows:

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

Here $\alpha \in (0,1)$ is the backtracking coefficient and j is the smallest non-negative integer such that $\pi_{(k+1)}$ satisfies the KL constraint and produces a positive surrogate advantage. Keep in mind that you don't really need to go over all π previously evaluated since very distant policies in time will most likely have a higher performance difference than the constraint we chose.

Final but not least computing H^{-1} is very computationally expensive when dealing with very complex problems, thus TRPO proposes this solution shown down below which consists in computing the matrix-vector Hx to avoid computing and storing the whole matrix.

$$Hx = \nabla_\theta \left(\left(\nabla_\theta \bar{D}_{KL}(\theta||\theta_k) \right)^T x \right),$$

Figure 1 goes over how to approach each of the previously mentioned steps as a pseudo code. It is very important to keep in mind the role that exploration and exploitation play. TRPO trains as an on-policy a stochastic policy which means that it explores by sampling data points from the previous policy. Even though the initial policy may contain a random sampling of actions, as training goes on, this will change and becomes less random. This is due to the fact that the update rule tries to exploit already found rewards.

Algorithm 1 Trust Region Policy Optimization

```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: Hyperparameters: KL-divergence limit $\delta$, backtracking coefficient $\alpha$, maximum number
   of backtracking steps $K$
3: for $k = 0, 1, 2, ...$ do
4:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
5:     Compute rewards-to-go $\hat{R}_t$.
6:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
7:     Estimate policy gradient as
           $\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$
8:     Use the conjugate gradient algorithm to compute
           $\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$
       where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
9:     Update the policy by backtracking line search with
           $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$
       where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and
       satisfies the sample KL-divergence constraint.
10:    Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$
       typically via some gradient descent algorithm.
11: end for
```

Fig. 1: Pseudo algorithm for Trust Region Policy Optimization

II. Methodology

In this section we will talk about the process followed for the development of the work, from the first implementations, to the extraction of results and conclusions.

First, we start by installing and configuring the MuJoCo (which stands for Multi-Joint dynamics with Contact) physics simulation engine, which will allow us to run simulations with our agents in a realistic environment that follows the laws of physics. On top of this simulation engine we use OpenAI Gym, a library that facilitates the connection between our algorithms and MuJoCo.

Secondly, we continue with an implementation of the TRPO algorithm. The selected implementation was based on the Pytorch library, in the Python programming language. PyTorch is an open source machine learning library, developed by the Facebook AI lab. For the development of the code and the functions necessary for the algorithm to work, we relied on the following GitHub repository: https://github.com/ikostrikov/pytorch-trpo from the user ikostrikov. We adapted this implementation for our environment.

Once the algorithm is implemented and running in our simulation engine, the task is to select the agents we want to use. First, we decided to use two agenters widely used in the literature, such as Humanoid and Hopper. These two agents are created by default thanks to the use of the OpenAI Gym library. In addition to these two, we decided to create our own agent. Taking the Humanoid as a model, we simplified the body of the agent from its xml file, to obtain a similar agent, with the same objective but with less degrees of freedom, by removing a good part of the upper body. In this way, we expect to obtain a shorter computation time for this humanoid in relation to the original. Sample images of the three agents can be seen in the appendix of this report.

Once we had this, we developed a web application using the streamlit python library, with which we were able to generate an interface that allowed us to generate the results and visualize the agents in a simpler and faster way, as well as to vary the different parameters more easily. A sample image of the interface is also available in the annex.

Now with the interface, we decided to define some standard values for the parameters. From these standard values, which we will use as a pivot, what we will do is to vary one by one those key parameters for the algorithm, and visualize how the learning curves are. In this way, we will be able to understand the influence that these parameters have on the performance of the algorithm. The standard values are as follows:

- gamma = 0.995
- tau = 0.97
- l2 reg = 1e-3
- max kl = 1e-2
- damping = 1e-1
- batch size = 1000
- alpha = 0.5

III. Results

In this section we will detail the results obtained in the different training sessions carried out. Our objective will be to show the influence that the different parameters have on the training behavior, in order to compare it with the theoretical intuition discussed in previous sections.

First of all, as we have already mentioned, we have three types of agents. Depending on their complexity (their number of degrees of freedom), training will be faster for those with lower complexity (as in the case of the Hopper). Therefore, we will use different agents for the convenience of our computation time, and we will make explicit in each graph which agent it is.

The second analysis we performed is then the comparison between learning curves and computation times for the three proposed agents:

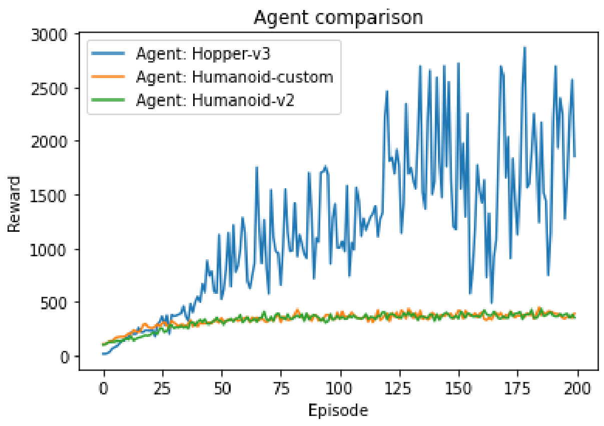


Fig. 2: Training results for the different agents, with standard parameter values for each of them.

- Hopper-v3 time: 0.5362 s/iter
- Humanoid-custom time: 1.628 s/iter
- Humanoid-v2 time: 1.895 s/iter

The second analysis we performed is that of the influence of the gamma parameter. For this purpose, we will set the values of other parameters to values that we consider reasonable, as they are typical values in this type of implementations. The agent used is the Humanoid custom. The result can be seen in Figure 3.

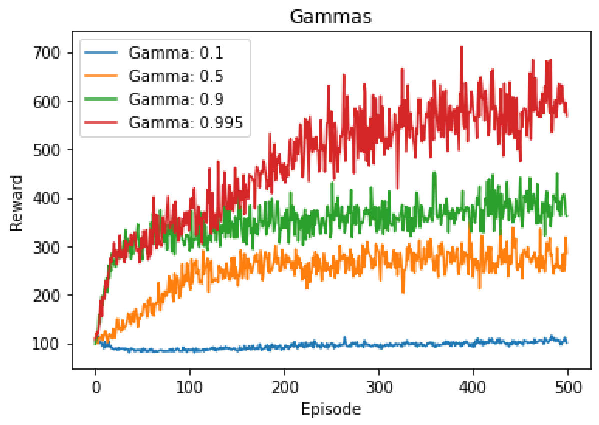
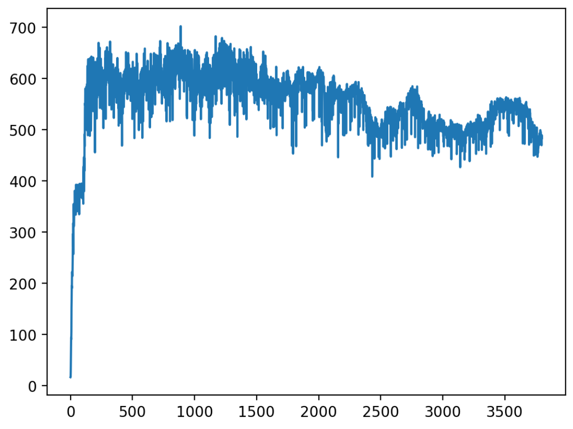


Fig. 3: Training results at different values of the gamma parameter. The other parameters set as follows: ($tau = 0.97$, $l2reg = 1e-3$, $maxkl = 1e-2$, $damping = 1e-1$, $batchsize = 1000$, $alpha = 0.5$, Humanoid-custom)

The following two graphs (Figure 2a and Figure 2b) show the training results for two different tau values: 0.97 and 0.5. As we wanted to train for a considerable number of


(a) Training result with tau = 0.97.

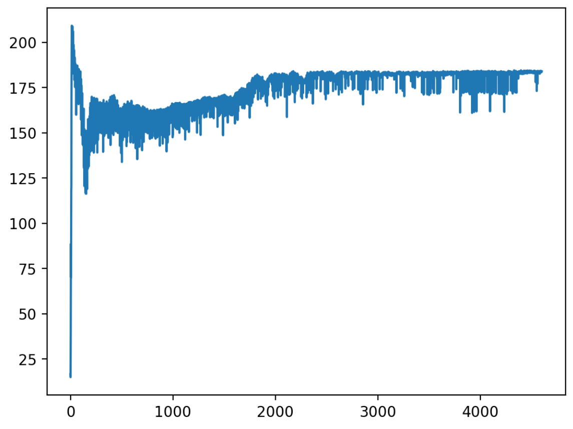

(b) Training result with tau = 0.5.

Fig. 4: Analysis of the effect of Tau on performance. The other parameters set as follows: ($gamma = 0.995$, $l2reg = 1e-3$, $maxkl = 1e-2$, $damping = 1e-1$, $batchsize = 1000$, $alpha = 0.5$, Hopper)

episodes, this time the agent chosen was the Hopper. The other parameters, again, have been set to standard values

It is also key to study how such an important value as δ (also called here max kl) parameter affects our training. This parameter determines how restrictive our constraint is when choosing the policy to update.

In order to get a better grasp in the understanding of batch size we have tried a batch size of 10 and 2000. Batch size is the number of samples used to estimate the gradient direction for each update of the policy parameters, which affects computational time and algorithm performance. Figure 7 show the difference obtained reward at each episode between the two chosen values for the batch size. We can see a big difference in time as these are the results obtained:

- batch size = 10 time: 27.40 s
- batch size = 2000 time: 182.15 s

IV. Discussion

Judging the times obtained, we see that the more complex agents take more time per iteration (the Hopper is much faster),

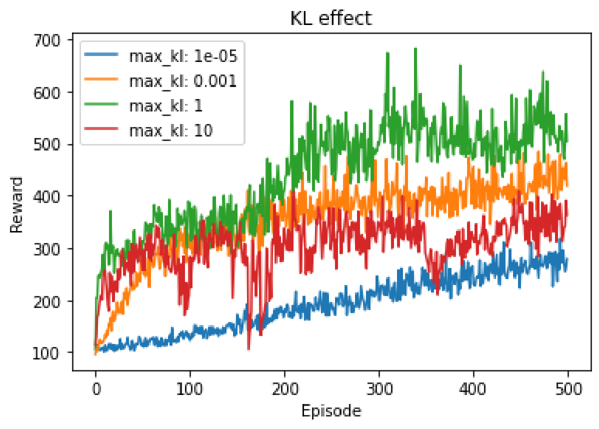


Fig. 5: Results at different values of the constrain parameter, defined here as kl max, which characterizes the maximum allowed Kullback–Leibler divergence between the new and the old policy distribution. (The other parameters set as follows: ($gamma = 0.995$, $tau = 0.97$, $l2reg = 1e-3$, $damping = 1e-1$, $batchsize = 1000$, $alpha = 0.5$, Humanoid custom)

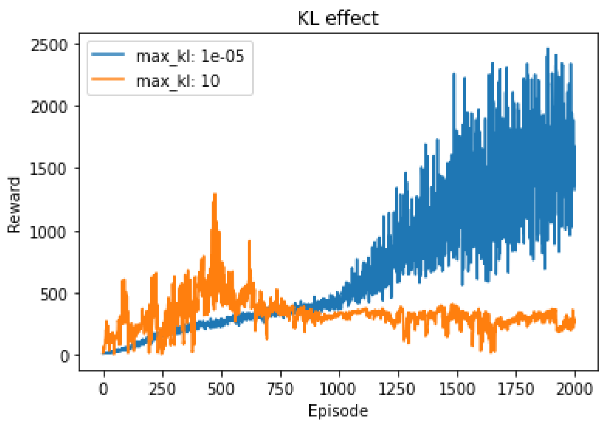


Fig. 6: Results at different values of the constrain parameter, defined here as kl max, which characterizes the maximum allowed Kullback-Leibler divergence between the new and the old policy distribution. (The other parameters set as follows: ($gamma = 0.995$, $tau = 0.97$, $l2reg = 1e-3$, $damping = 1e-1$, $batchsize = 1000$, $alpha = 0.5$, Hopper)

which agrees perfectly with the initial hypothesis. Looking at Figure 2, we can see graphically how humanoids have a less noisy training but their learning is also very small.

Looking at the next graph, Figure 3, we see that high gammas close to 1 produce good results. Very small gammas, however, prevent improvement.

In the theory we exposed how the tau (Figures 4a and 4b) corresponds to a kind of training attenuation. Therefore, we see in the graphs how when we lower its value too much, the agent stops training, it only reaches 175 reward. However, in graph 4a) there is something else that catches our attention. We see that the reward at a certain point of the training starts to decrease. Does this go against the statement of the

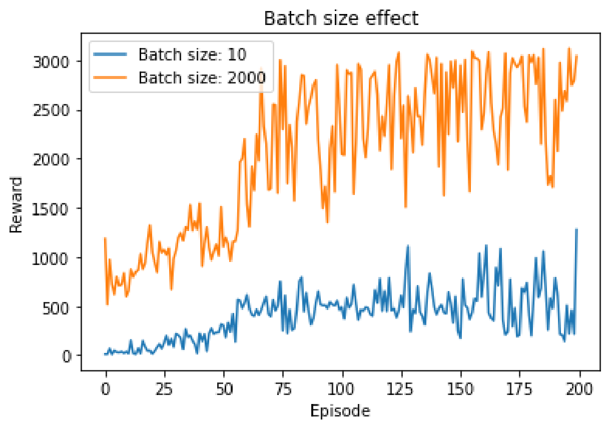


Fig. 7: Results with different values of the batch size, the number of samples used in each iteration. (The other parameters set as follows: ($gamma = 0.995$, $tau = 0.97$, $maxkl = 1e-2$, $l2reg = 1e-3$, $damping = 1e-1$, $alpha = 0.5$, Hopper)

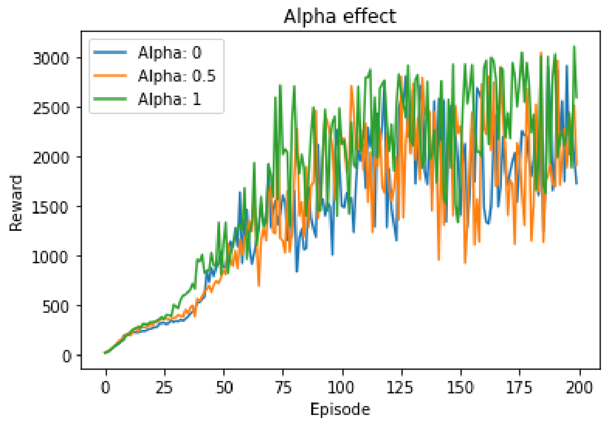


Fig. 8: Results with different values of alpha. (The other parameters set as follows: ($gamma = 0.995$, $tau = 0.97$, $maxkl = 1e-2$, $l2reg = 1e-3$, $damping = 1e-1$, Hopper)

monotonic improvement guarantee? Is perhaps the constrain too soft because of the max kl parameter set, and therefore this guarantee is not fulfilled?

To learn more about this issue, we will analyze what is the effect of this parameter kl just mentioned. Is it possible, therefore, that it is the cause of the breakdown of the monotonic improvement guarantee? If we look at the graph in Figure 5, when this value is small, for example 10^{-5}, this guarantee is very clearly fulfilled. However, when this value starts to rise, especially for value 10, we see that there are more or less sharp drops in the reward, especially in the short term. These sharp drops may be due to the fact that a very high value of this parameter causes too large steps to be taken in the wrong direction, making the training more unstable. One can clearly see these instabilities for max kl = 10 in the area near episode 150, for example.

In Figure 6, we look for the more long-term effects of this parameter. To this end, in order to be able to run many more episodes, we decided to set Hopper as our agent, because of its higher computational speed. In this case, choosing the extreme values (10^{-5} and 10), we train for 2000 episodes. Here we can see how for a small value the mean tends to rise steadily (although also to become noisier between episodes) while when the value is very high, we even see a worsening between episodes 500 to 100. After episode 1000, the training with very low max kl becomes stagnant, which could be interpreted as falling into a local minimum.

Analyzing Figure 7 and the times obtained for it, we see that the higher batch size makes the training go faster, since in each episode more data are executed to update the policy, but it also requires more computation time per iteration, which is in accordance with what was expected. Perhaps it was not as expected that training with large batch size would be noisier, as in this case. Could it be because of the scale and that the smaller batch is more variable as a percentage of the mean?

Finally, we look at Figure 8. This graph presents the results of training the Hopper with different values of alpha. We can see in it something different from all the other graphs, and that is that the value of alpha does not seem to affect the training behavior at all. Why does this phenomenon occur? We know that the value of alpha is raised to an integer that has to do with backtracking and previous states, as we explained in more detail in section 1 of the introduction. By analyzing it carefully, we have found that this value, which corresponds to the exponent, turns out to be 0 for all cases during training. Therefore, any value we give to alpha will be raised to zero and will therefore be worth 1 constantly.

V. Conclusion

This project has brought light to some aspects that for us were unclear at the beginning of the project. The effects that all the studied variables $gamma$, tau, $maxkl$, $l2reg$, $alpha$, $batchsize$ are now more clear. Although some concepts where already clear as we have seen them in class, such as gamma, others were harder to comprehend their role. The Kullback-Leibler divergence was a concept that made sense theoretically but we lacked the knowledge to understand its effect on training. After this project we can state that if you allow high difference in performance between the new and the old policy ($maxkl$), policy improvement might not be done in the right direction, leaving our trust region, due to the Taylor expansion used to reduce computational complexity of the algorithm.

On the hand a Kullback-Leibler divergence too low can also have a negative impact in computational time, since it does not allow to make reasonable performance improvements between policies. Batch size was also a big question since such topic has not been discussed in class deeply due to the lack its necessity for previously taught algorithms (or addressed problems). The bigger the batch size the more data points will be used for training in an episode. For this specific situation the higher the batch size, the better results were obtained. The last variable that was interesting to study was τ, which it has been shown that the closer it is to one the better results, since it represents the state action sequence generated by following π_k, the policy at time step k.

With this project not only theoretical aspects have been covered. We have learnt how hard it is to visualize trained agents and how time consuming it can be. Besides this inconveniences, it has been very useful to see how rewards are visually represented in MuJoCo. Truth is that at the beginning it was very challenging to even get MuJoCo running from python, but as we got more familiar with the library Mujoco-py, the easier it became to make some sense of the visualizations we were getting and how where visualized high rewarded episodes. For example, one might think that for the Hooper, the further the Hooper gets, the higher the reward, but the truth is that sometimes Hooper agents that stayed the longest standing up beat other Hopper agents that get further away but fall very quick. This has been very use full to fully understand how our reward worked in a problem which can be visualized.

Even though not much attention has been brought to the control panel, it has been the most use full tool of the project. It has helped us to play around with different values of the studied variables in order to get a much better understanding of the little details behind complex algorithms such as the one studied in this project, TRPO. As mentioned in previous sections, this control panel also allows you to select three different problems (agents) and different values for the algorithm parameters.

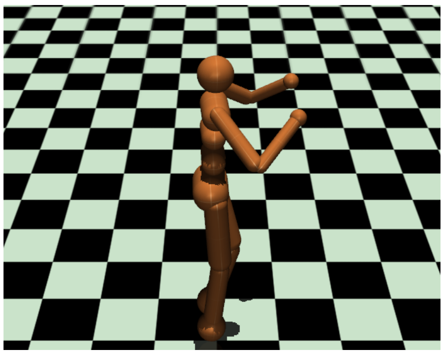
This gives us a lot of flexibility to propose future experiments. It would be very interesting to be able to pick all the algorithms seen in class, as well as more complex algorithms, keeping in mind that some limitations that simpler algorithms have. Also adding different agents than the ones presented at this project and showing computing time could be further additions. Real time visualization of the accumulated reward for each episode and the policy the agent followed in the same window would be very practical.

VI. Resources

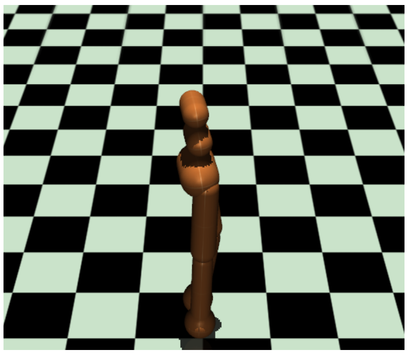
1) Schulman, John, et al. “Trust Region Policy Optimization.” ArXiv.org, 20 Apr. 2017, https://arxiv.org/abs/1502.05477.
2) Todorov, Emanuel, Erez, Tom, and Tassa, Yuval. MuJoCo: A physics engine for model-based control. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pp. 5026–5033. IEEE, 2012.
3) Kakade, Sham, et al. “A Natural Policy Gradient.” Guide Proceedings, 1 Jan. 2001, https://dl.acm.org/doi/10.5555/2980539.2980738.
4) TRPO Pytorch implementation https://github.com/ikostrikov/pytorch-trpo

VII. ANNEX

a.



b.



c.

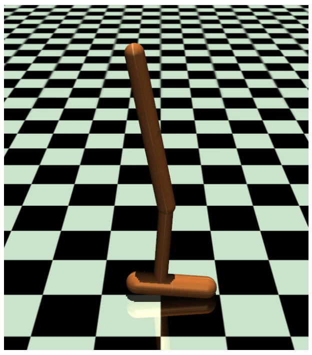
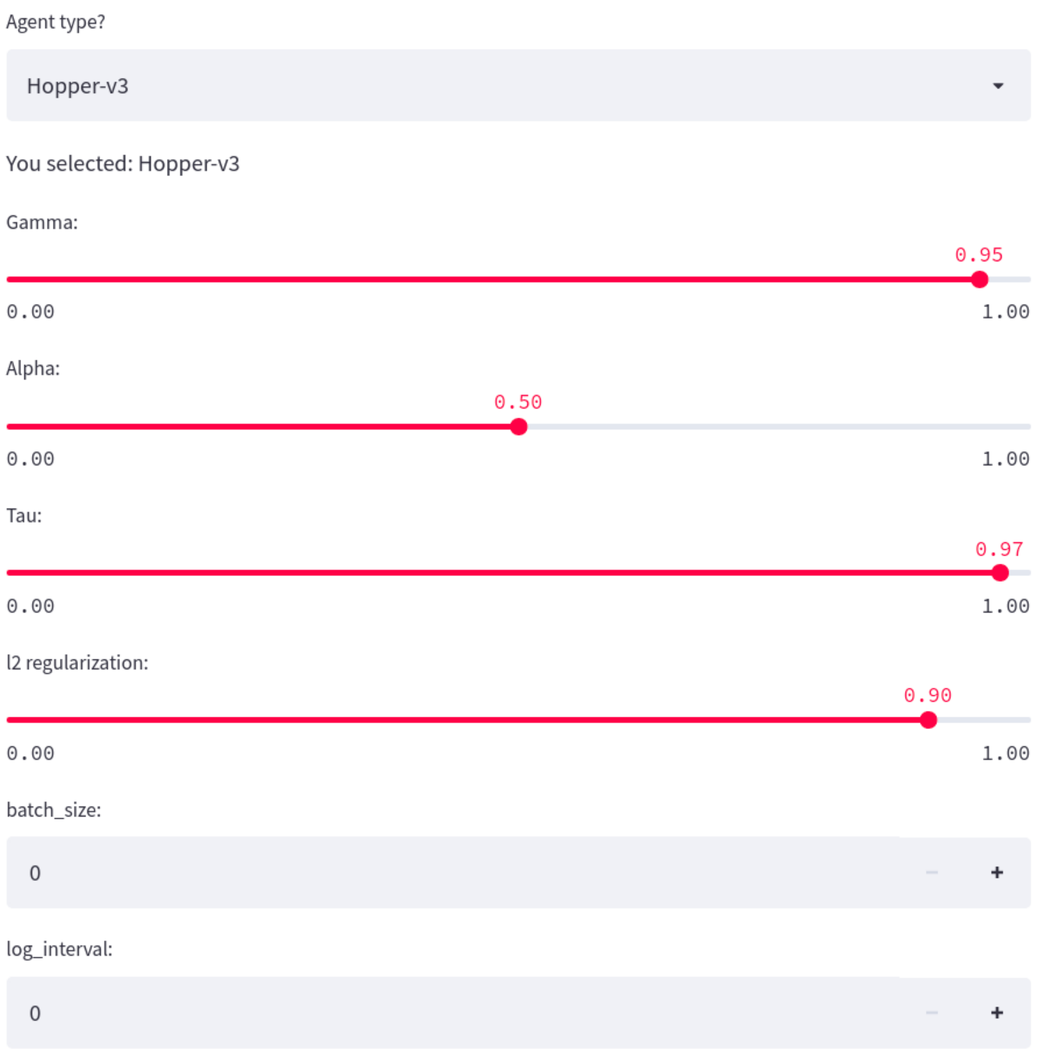


Fig 1. Possible models to select: a. Humanoid b. Humanoid custom c. Hopper


Agent type?

Hopper-v3

You selected: Hopper-v3

Gamma: 0.95 (0.00 – 1.00)

Alpha: 0.50 (0.00 – 1.00)

Tau: 0.97 (0.00 – 1.00)

l2 regularization: 0.90 (0.00 – 1.00)

batch_size: 0

log_interval: 0

Fig 2. Sample image of the web app interface. Control panel for parameter selection with default values.