Exercise 4

Ludovica Mazzucco
Department of Control and Computer Engineering
Politecnico di Torino
Italy
s315093@studenti.polito.it

Abstract: The purpose of this report is to inspect the abilities of Policy Gradient and Actor-Critic reinforcement learning algorithms both applied to the same continuous version of the CartPole Gym environment in order to compare the performances.

1 Introduction

1.1 Policy Gradient (PG) Algorithms and Reinforce

Differently to what happens with value-based reinforcement learning algorithms, in which the main goal of the agent is to learn the optimal behaviour through the evaluation of the underlying policy with updates to the estimation of the value (or action-value) function and a subsequent greedy improvement as General Policy Iteration dictates, Policy Gradient methods make use of function approximation to let the agent directly learn the policy due to proper parameter tuning. The way those parameters are updated at each time step relies on the maximization of a *performance function* $J(\theta)$, that for episodic problems is considered as the value of the starting state under the running policy depending on the forementioned parameters. In particular, the update rule is computed as follows:

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t) \tag{1}$$

where α is the learning rate of the gradient ascent step and $\nabla \hat{J}(\theta_t)$ is a term whose expectation is the performance measure gradient. Through the Policy Gradient Theorem we are able to find a close form for the latter, that is a fundamental term for any PG algorithm, that is:

$$\nabla \hat{J}(\theta_t) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{2}$$

where μ is the state distribution, useful for giving more importance to some recurrent states rather than others and it is frequently used in the field of function approximation, accounting for the probabilities of a certain state to appear. As a consequence, the expression above can be reformulated as:

$$\nabla \hat{J}(\theta_t) = \mathbb{E}_\pi \left[\textstyle\sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \right]$$

The quantity in brackets can be thus exploited for parameter updates. Eventually, with some mathematical tricks equation [1] is rewritten in the following expression:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{3}$$

in which G represents the (possibly discounted) unbiased sample return computed as Monte Carlo methods do. Since it is practically useful to work with log-probabilities especially when the underlying probabilities assume very low values, the right-most part of the above equation can be rearranged

7th Conference on Robot Learning (CoRL 2023), Atlanta, USA.

due to the derivative chain rule assuming this final form:

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla log\pi(A_t|S_t, \theta_t)$$

The latter update rule is what `Reinforce` algorithm is based on. It has to be underlined that it is not incremental as the full episode return has to be computed first, thus if it lasts long enough it would not be so efficient. Anyway, equation [3] is helpful in giving a clear interpretation of what the Reinforce agent is driven to do, in the sense that it is enforced to "move" the parameters in a way that increases more the probabilitites of taking those actions that led to higher returns while normalizing them by the probabilities theirselves to overcome preferences for actions happening more frequently and then to manage the possibility to be stuck in local optima.
A further adjustment that can be made to the Reinforcement algorithm is to include in the update expression a so called $baseline$ ($b(s)$), a term that modifies the return value G_t to decrease its variance and thus to speed up convergence. The gradient ascent step will be then computed as follows:

$$\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t)) \nabla log\pi(A_t|S_t, \theta_t)$$

The baseline term can be any expression, with the only constraint that is does not depend on the action a, so that its addition does not influence the overall expression shown in [2].

1.2 Actor-Critic Algorithms

As it was stated in the previous section, Reinforce algorithm follows Monte Carlo approach of computing all the updates on parameters only at the end of the episode, after having experienced all the actions following the parameterized policy initialized to standard values. The downside is that the agent has to wait until the end of the episode to effectively know if the actions it took were good or not, when the policy is updated with the modified parameters and another sequence of states and actions is generated. In order to immediately be aware of the quality of the current actions so to change promptly behaviour and speed up learning, Reinforce with baseline structure is modified to become an incremental, online algorithm that estimates in the same time and at each time step either the policy and the value function of the current states, the latter emboding a sort of baseline. In Actor-Cristic methods, then, the update rule is formulated as follows:

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})) \nabla log\pi(A_t|S_t, \theta_t) \tag{4}$$

The quantity inside brackets looks like the TD-error, it is otherwise referred to as the *advantage* obtained from taking an action out policy and it exploits the approximation of the value function depending on the vector of weights $\mathbf{w}$. This is indeed a way of judging actions along the way the agent experiences the environment, this is where Actor-Critic algorithms derive their name from, since the approximated policy takes the role of actor (driving the behaviour of the agent) while the approximated policy embodies the critic (assessing the actor moves). In this case, two updates are performed at each iteration, the one to policy parameters θ following rule in [4] and the other to value function weights $\mathbf{w}$ as semi-gradient descent for function approximation.

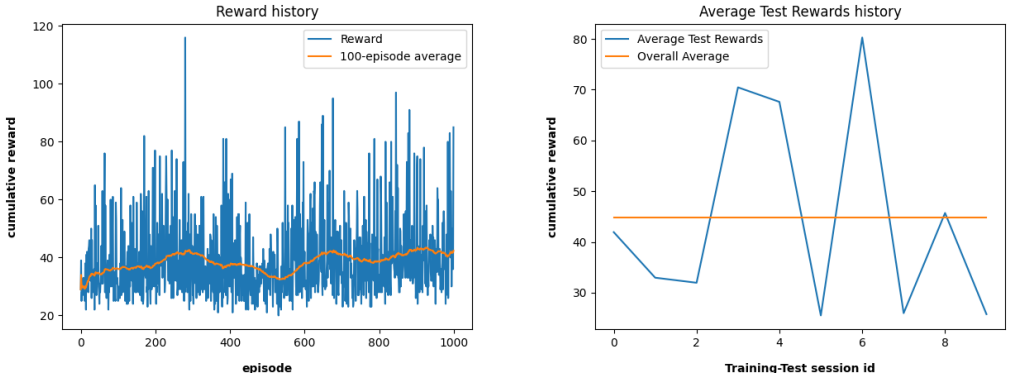
2 Experimental Results

2.1 Reinforce and the Effect of the Baseline

A it was disclosed in Section [1.1], the inclusion of a baseline function inside the update rule of Reinforce algorithm has the advantage of reducing the variance during training time and thus speeding up convergence. This is due to the fact that the value of the return is somehow normalized to the one of the baseline and compared to it, hence if the baseline is cautiously chosen the output of their difference will cause limited updates in modulus to the parameters. Furthermore, the baseline can represent a sort of treshold for the return: a greater return will end up in a positive increment to action probabilities while a lower return will cause a negative step and so the probabilities of those action taken are decreased, in other words, the agent will choose the actions that led to positive steps in future visites of the current state.

In order to demostrate the theoretical results, different simulations with the CartPole environment have been performed, each with a different setting of Reinforce.

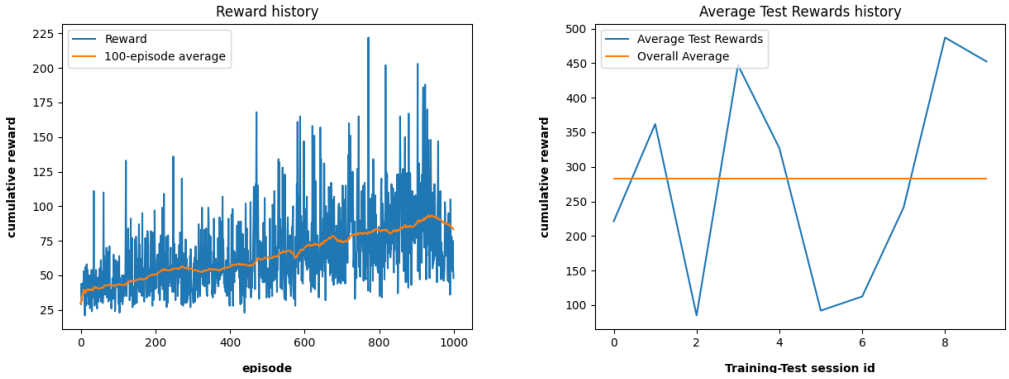
Case A. Reinforce without baseline.
This is the basic case to visualize the performance of Reinforce without the use of any baseline, hence we would expect relatively low returns and a slow learning. The plots below show in one hand the output of a single simulation (the left one) and in the other a series of 10 subsequent train-test sessions are performed in order to deal with the intrinsic stochasticity of the problem and better understand the results.



(a) single run (b) 10 runs with mean average test reward

Figure 1: Reinforce without baseline

Case B. Reinforce with baseline 20.
This time we perform the training of the CartPole introducing a constant baseline and, stating to all it was said before, we are anticipating a superior performance with respect to the previous one. The figures below have been collected following the same approach as the former ones.



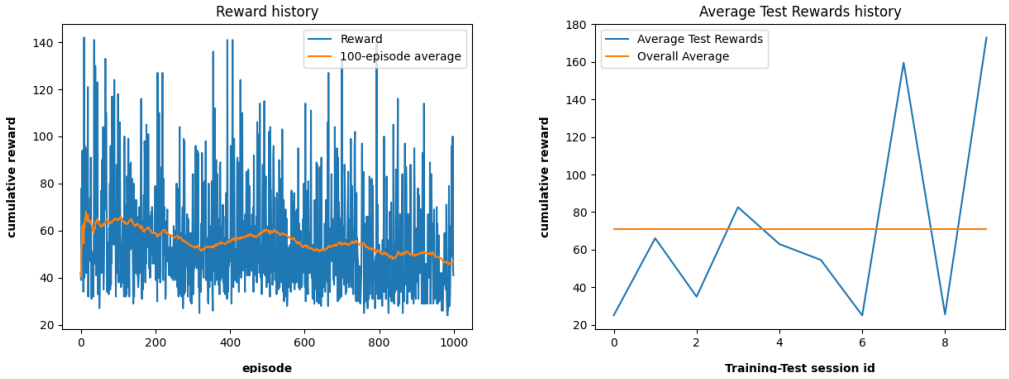
(a) single run (b) 10 runs with mean average test reward

Figure 2: Reinforce with constant baseline set to 20

It is trivial to catch the paramount improvement in performance between *Case A* and *Case B* just by looking at the high rewards reached.

Case C. Reinforce without baseline but with normalized rewards.
This is a special case in which instead of explicitly inserting a baseline we normalize the discounted returns to null mean and unitary variance. Actually, this is a trick to obtain similar benefits to the

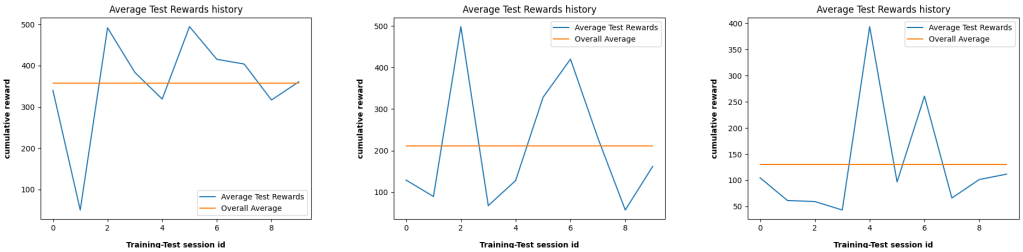
case with a baseline but without having a baseline and in fact, as the plots highlight, although the single run simulation seems to perform poorly with respect to the first case, the overall view reflects an enhancement.



(a) single run (b) 10 runs with mean average test reward

Figure 3: Reinforce with constant baseline set to 20

This is due to the fact that in this way we are forcing the rewards to fall in a certain interval across zero, so that it will be easier for the agent to recognize good actions from the bad ones.
Taking into account all the different situations, it is clear that the best performance has been achieved in *Case B*, where a constant baseline of 20 was exploited. Its effect is similar to the one provoked by setting non null initial values to the Q-function in value-based algorithms, that is encouraging the agent to find better actions by a higher degree of exploration. Setting the baseline to a predefined constant value requires a careful tuning for it, since it strongly depends on the current problem setting, for example 20 seems to be a reasonable value for the case under study, but a simulation with 30 outputs far better results, as Figure is underlining. This does not mean that the higher the baseline the better the performance, because if it becomes too high compared to any short-term rewards the agent can ever acquire then it will have mainly negative feedbacks, or well, no action would guide him to a direction of certain improvement and it will continue to randomize actions without reaching any good result. To test this, three different plots are proposed here representing average test rewards with different values of baseline, respectively 30, 40 and 100.



(a) baseline set to 30 (b) baseline set to 40 (c) baseline set to 100

Figure 4: Change in performance at different values of the baseline. While it is improved with 30, it progressively gets worse with 40 and 100

Setting the baseline as the average of rewards could seem a rational option, but it is not the optimal one since if the agent performs all suboptimal actions then it would not be completely aware about that because the average reflects what it has experienced. At this point, the best choice possible for the baseline would be something depending on the current state so that for high-valued states this

would set a higher treshold while for those states with a lower value it will be automatically lowered; this something would obviously be the value function.

2.2 Real-World Control Problems

The system exploited here to perform the Policy Gradient algorithms on is a modified version of the CartPole with continuous action space. Hence, the Neural Network used for the learning of the optimal policy (a Multi Layer Perceptron with two fully connected hidden layers) outputs the mean of the Gaussian distribution representing the probabilities of each action in a given state considering a fixed standard deviation of 5, rather then giving as result the output of a softmax layer confering finite probabilities to each single action as was done in the discrete case. In a real world setting we would not have episodes but a continuous sequence of states and actions plus we are hypothesizing to have an unbounded continuous action space. If we add that for any of the infinite actions possible the agent will get a +1 reward then the return could grow indefinetely causing big gradient steps to be performed and consequently a big change in parameters that could take far away from the optimal solutions and generate divergence. To manage that, a varying learning rate α could be chosen and a proper value for the discounting term γ.

2.3 Discrete Action Spaces

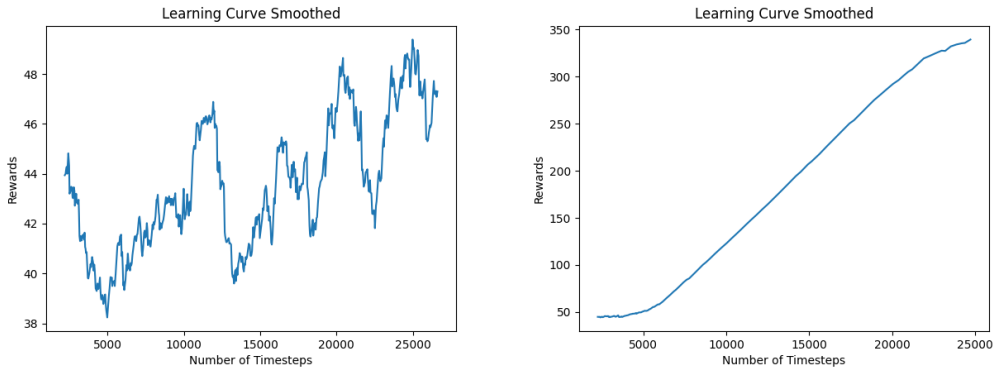
Policy gradient algorithms are powerful since they can be exploited for both discrete and continuous action spaces. A problem that could rise in the former case would be that the parameter update would not affect only the probability of the action taken into consideration but even the others since them all are dependent on θ, as a consequence it would not be always guaranteed that all the probabilities in a certain state sum up to one and this fact, other than being theoretically unfeasible, would cause the falling of the assumption that the insertion of the baseline term inside equation [2] would not change it. Inspite of that, resort to a softmax function applied to preferences functions depending on actions and parameters can overcome this issue.

2.4 Actor-Critic Algorithms: PPO and SAC

In this task we move our focus to the study of two particular Actor-Critic algorithms, Proximal Policy Optimization and Soft Actor Critic.
The main feature of the first is that it keeps the update step of the policy limited since a big step could move it away from the optimal generating a lot of variance and instability but a tiny one would slow too much the training. This goal is fulfilled substituting to the log-probability the ratio between the updated policy and the old one, representing how the probability of taking a certain action in a certain state has changed with the update:

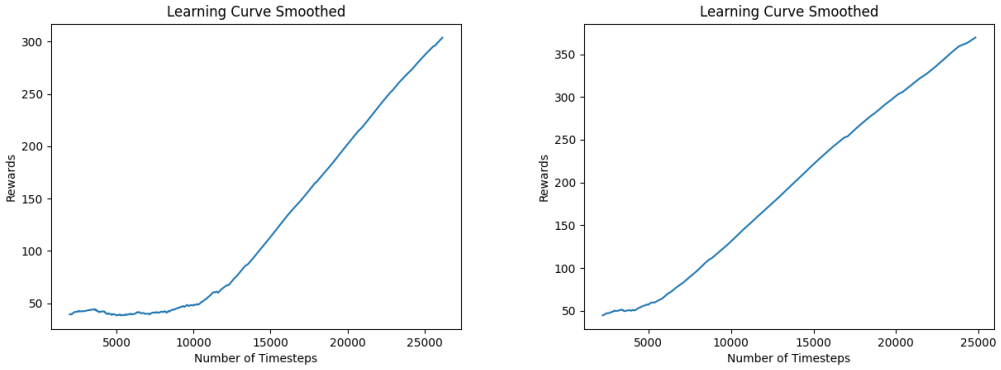
$$r_t(\theta) = \frac{\pi_{\theta new}(a_t, s_t)}{\pi_{\theta old}(a_t, s_t)}$$

This term is then multiplied by the advantage as any Actor-Critic method. Additionally, a "clipped" version of the loss function is used, to be sure that the ratio above falls in between the interval $[1-\epsilon, 1+\epsilon]$. In a nutshell, PPO tries to maintain the updated policy fairly close to the previous one to guarantee stability.
Conversely, SAC introduces in the loss function a term depending on the entropy of the current policy paired with its entropy regularization parameter, in a way that increasing it will cause a preference to the exploration rather than exploitation. The main differences with respect PPO are that this is an off-policy algorithm, with a Q-value critic and a target Q-value critic, and it accepts only continuous action spaces.
To compare PPO and SAC performances two simulations per algorithm have been run, one with learning rate $\alpha = 0.0003$ and one with $\alpha = 0.003$



(a) PPO (b) SAC

Figure 5: Comparison between PPO and SAC with $\alpha = 0.0003$



(a) PPO (b) SAC

Figure 6: Comparison between PPO and SAC with $\alpha = 0.003$

It is immediate to notice the big difference in the performance of PPO algorithm just increasing by an order of magnitude the learning rate, while SAC behaves quite the same in both the cases, being very efficient since the first trial. This can be explained saying that PPO is more likely to be stuck in local optima and it less likely to explore with respect SAC, given that it forces limited changes to policy parameters, in fact it is only needed to increase the learning rate to obtain results comparable to SAC and far more stable compared to the first case. Managing in a successful way the trade-off between exploration and exploitation SAC seems to work better in any case, but it is undoubtly more expensive in terms of time. Those considerations are verified by the test results:

```
PPO:
(lr = default)
Test reward (avg +/- std): (46.21 +/- 19.55826935083981) - Num episodes: 100
(lr = 0.003)
Test reward (avg +/- std): (500.0 +/- 0.0) - Num episodes: 100

SAC:
(lr = default)
Test reward (avg +/- std): (332.39 +/- 62.82943498074768) - Num episodes: 100
(lr = 0.003)
Test reward (avg +/- std): (490.22 +/- 39.983641655056886) - Num episodes: 100
```

With respect to Reinforce (with and without baseline), SAC and PPO obtain overall higher results and are less affected by the variance, thus are more stable.