Reinforcement Learning, Implementation project

Riccardo Caprari, ID number 1743168

Winter 2020

Contents

1 Introduction

Aim of this work is to implement, train and test the Soft Actor Critic algorithm [1]. This recent reinforcement learning algorithm is defined *off-policy* because it actually is independent of the policy used, meaning that the agent could behave randomly. In fact, in the first part of the learning process the agent will gather information from random moves and this randomness will slow down over time. In particular, with SAC we speak about entropy-regularized reinforcement learning which will be explained in the next chaper (2).

Coding frameworks used for this work are **Python** v3.7, **Tensorflow** v2.1.0, **CUDA** v10.1, **CUDNN** v7.6.5. Training and test processes have been executed using a GeForce GTX 1650 4GB GPU.

The evaluation of the algorithm has been done using two OpenAI Gym environments:

- *MountainCarContinuous*: In this first environment, which is a continuous action space, given a car and a final goal the objective is to let the car reach the goal. The difficulty lies in the fact that the goal is positioned on the top of a mountain and the car, due to its power limitations, must exploit the slope of the mountains (cumulated momentum) to reach the top. The reward is **sparse**, meaning that it will be positive only when the goal is reached and negative when a move, which is a limited real value, is performed.
- *Walker2d-v2*: Here the objective is to let a two-dimensional bipedal robot walk forward as fast as possible. In this case the reward is **infinite**: there's not a specified reward threshold at which the experiment is considered solved.

2 Soft Actor Critic

SAC, as introduced in Chapter 1, is an off-policy actor-critic algorithm based on the maximum entropy reinforcement learning framework. The entropy is defined as a measure of uncertainty of a probability distribution and it plays a fundamental role in the objective function. Usually, in standard reinforcement learning we try to maximize the expected sum of rewards i.e. $\sum_t \mathbb{E}_{(s_t,a_t)\ p_\pi}[r(s_t,a_t)]$. Instead, with the addition of entropy we have:

$$\sum_t \mathbb{E}_{(s_t,a_t)\ p_\pi}[r(s_t,a_t) + \alpha H(\pi(\cdot|s_t))] \tag{1}$$

where α is the temperature and controls the stochasticity of the algorithm, i.e. the contribute importance of entropy. In fact, for $\alpha = 0$ we obtain the standard objective function. Therefore, SAC is defined *entropy regularized* due to presence of H, the entropy term, in the formulation of the objective. $H(\pi(\cdot|s_t))$ borns like an arbitrary term and can depend on the problem the algorithm is applied to. However, in [2] the entropy is defined as:

$$H(\pi(\cdot|s_t)) = -log(\pi(\cdot|s_t)) \tag{2}$$

Using entropy, the agent will focus on the exploration of new combinations and this favors the use of techniques like gradient descent or ascent because of the smooth shape of the objective. Every action returned by the policy will be computed w.r.t. the maximization of the function mentioned in Eq. 1.

After this step, soft policy iteration, which is a value iteration algorithm, is introduced to let the policy converge to optimum using a soft q-function and a value function. In particular, from Bellman operator we know that:

$$V(s_t) = \mathbb{E}_\pi[Q_\theta(s_t,a_t) - log(\pi(a_t|s_t)] \tag{3}$$

In [1] the authors suggest the use of a separate function approximator for the state value in addition of the policy, the soft q-function and the value function. This consists, in an implementation view, of four neural networks associated to each function. This approximation function should speed up

and stabilize the training process. Assuming the use of a value function, it's error function which has to be minimized is the squared residual error, defined as:

$$J_V = \mathbb{E}_{s_t \sim D}[\tfrac{1}{2}(V(s_t) - \mathbb{E}_{a_t \sim \pi_\theta}[Q_\theta(s_t, a_t) - log(\pi_\theta(a_t|s_t)])^2] \tag{4}$$

However, in this implementation project, it's been decided to use **two soft q-functions** with their relative approximation functions instead of a single q-function with a value one. With this choice, the error functions of the two q-functions will be:

$$J_Q(\theta) = \mathbb{E}_{(s_t, a_t) \sim D}[(Q_\theta(s_t, a_t) - y(r_t, s_t, d_t))^2] \tag{5}$$

where y is defined as:

$$y(r_t, s_t, d_t) = r(s_t, a_t) + \gamma(1 - d) * (min_{i=1,2}\hat{Q}(s_t, a_t) - \alpha log(\pi(a_t|s_t))) \tag{6}$$

Using these approximation functions and the error in Eq. 5 it's actually possible to speed up the convergence of the policy. The policy will be optimized according to:

$$max_\theta \, \mathbb{E}_{s \sim D, \xi \sim N}[K_{adv} log(\pi(\tilde{a}_\theta(s, \xi)|s))] \tag{7}$$

$$K_{adv} = min_{i=1,2} Q_\theta(s, \tilde{a}_\theta(s, \xi)) - log(\pi(\tilde{a}_\theta(s, \xi)|s)) \tag{8}$$

which is slightly different from the one in the pseudocode showed in Fig. 1. In fact, this change leads to a faster and better convergence [4] to the optimum.

An important feature used to optimize the policy is the *reparameterization trick* which permits to obtain the sample $\tilde{a}_\theta(s, \xi)$. This method exploits a squashed Gaussian policy:

$$\tilde{a}_\theta(s, \xi) = tanh(\mu_\theta(s) + \sigma_\theta(s) + \xi) \tag{9}$$

In Fig. 1 it's possible to look at the pseudocode of the implemented SAC algorithm with the already mentioned change at step 14.

Algorithm 1 Soft Actor-Critic

1: Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
3: **repeat**
4: Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
5: Execute a in the environment
6: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
7: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
8: If s' is terminal, reset environment state.
9: **if** it's time to update **then**
10: **for** j in range(however many updates) **do**
11: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12: Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

13: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

14: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right) \Big),$$

where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
15: Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

16: **end for**
17: **end if**
18: **until** convergence

Figure 1: SAC pseudocode.

3 Training and evaluation of the model

In this section the training and the outcomes of the SAC implementation will be analyzed in the two different environments. Training is done using a loop over the number of episodes, which can be set by the user, and another loop of time steps which varies w.r.t. the environment (e.g. the maximum number of steps in MountainCarContinuous is 1000). In the first *replay_start_size* steps, a BufferReplay, mentioned in Fig. 1, is filled with random samples composed of: actual state, action, next state, reward and done flag. After this number of steps is reached, the agent starts to train, which means next actions won't be randomized but will, instead, be chosen by the policy or the so called *actor*.

3.1 MountainCarContinuous

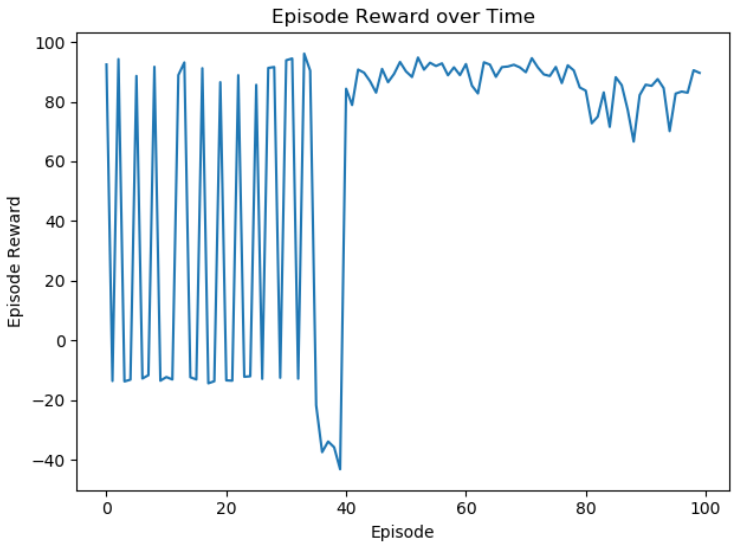

Figure 2: Reward trend in the MountainCarContinuous env.

As mentioned before and looking at Fig. 2 it's actually possible to deduce that the first 38/39 episodes are dedicated for filling the ReplayBuffer with random actions. Some of these episodes are favorable, which means the

goal has been reached. Without these samples the algorithm would converge much slower or even not converge at all. After part of the ReplayBuffer is filled the training phase starts and it's evident how the policy starts to choose in a "bad manner". Anyway, in subsequent episodes, the policy will be capable to learn the correct behaviour sampling from the ReplayBuffer.

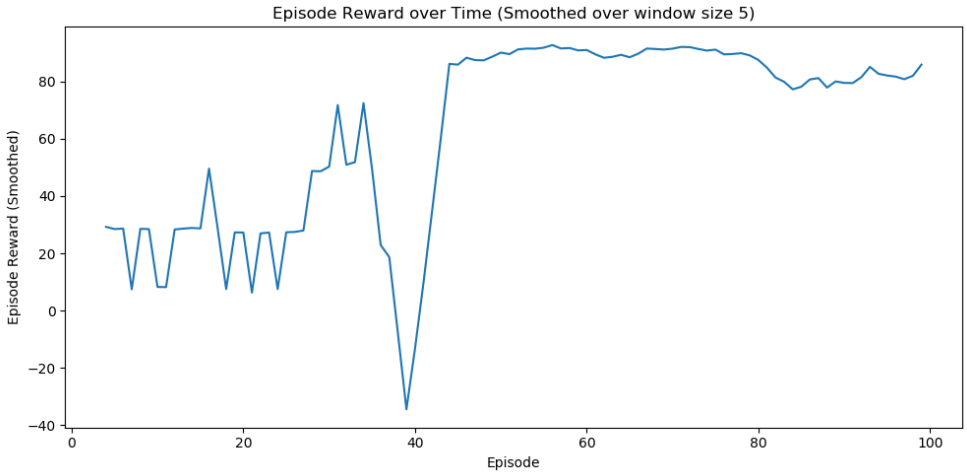

Figure 3: Smoothed reward trend in the MountainCarContinuous env.

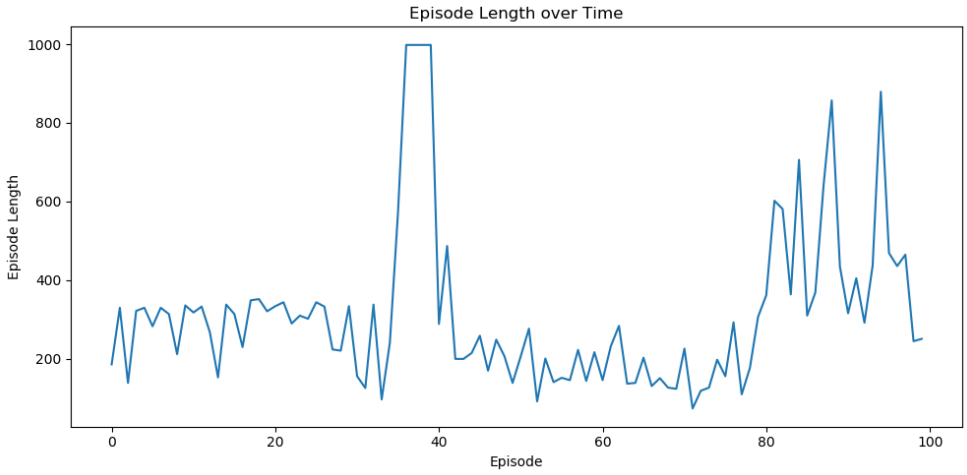

Figure 4: Episode length (in steps) in the MountainCarContinuous env.

3.2 Walker2d-v2

This second environment, as mentioned in Section 1, hasn't a reward threshold and this makes it really complex and hard in terms of learning. In general, there won't be a convergence since there's not a precise threshold and this could mean that the higher are the episodes the higher is the reward reached. Of course, this may not be always true but it actually happened in this implementation.

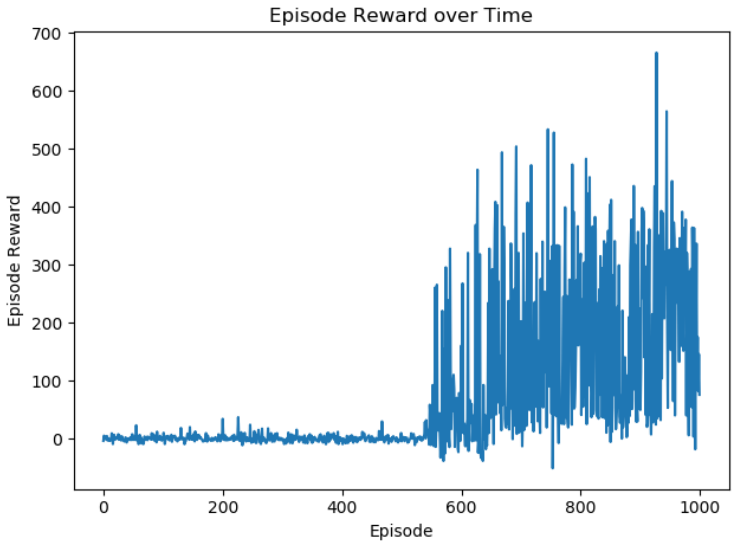

Figure 5: Reward trend in the Walker2d-v2 env.

In Fig. 5 it's possible to notice how "noisy" the reward looks over time. At episode 516 it's clear how the training process starts in the same manner discussed in 3.1. From that point the agent starts to reach different scores and tries to improve over time.

In Fig. 6 there's a better image of what is happening during the training, showing how the agent is learning to reach better scores as the episodes go by.

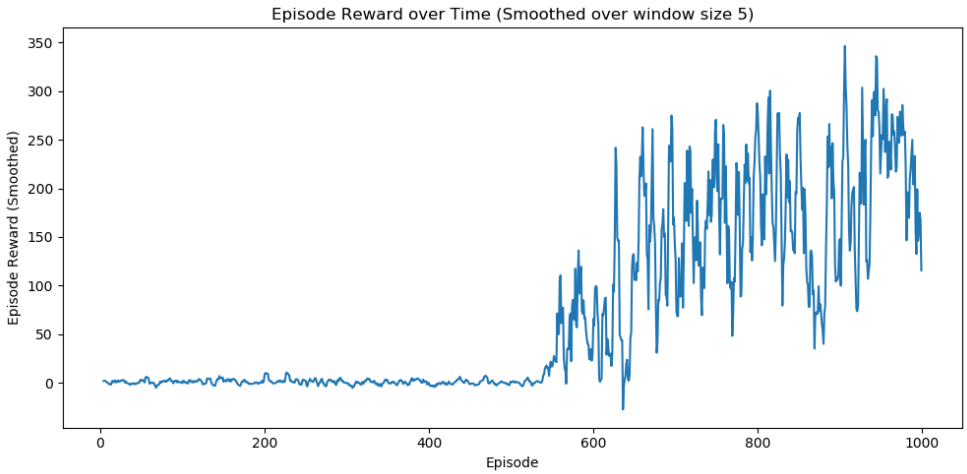

Figure 6: Smoothed reward trend in the Walker2d-v2 env.

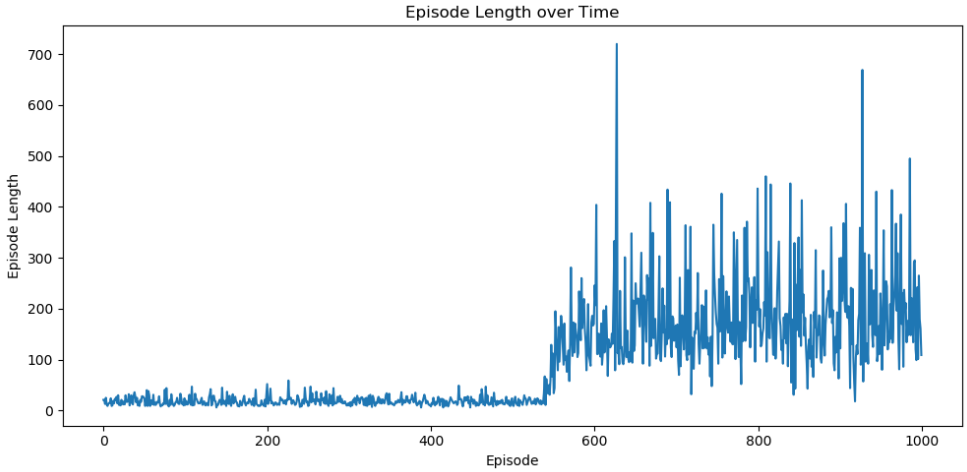

Figure 7: Episode length (in steps) in the Walker2d-v2 env.

It's clear to see that the peaks in reward function correspond to the ones in Fig. 7 meaning that reward is computed also taking into account the number of steps done in the environment, aka the alive period of the agent.

4 References

[1] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.*

[2] Tuomas Haarnoja, Aurick Zhou. *Soft Actor-Critic Algorithms and Applications.*

[3] mrahtz's Soft Actor-Critic implementation.
`github.com/mrahtz/tf-sac`

[4] Reinforcement Learning Library, SAC Algorithm.
`github.com/arnomoonens/yarll/tree/tf2`

[5] Soft Actor Critic Demystified.
`towardsdatascience.com/soft-actor-critic-demystified-b8427df61665`

[6] Spinning up Open AI - SAC.
`spinningup.openai.com/en/latest/algorithms/sac.html`