DDPG vs SAC - comparison between off-policy algorithms in continuous action space

Simone Giordani
Alma Mater Studiorum - University of Bologna
`simone.giordani2@studio.unibo.it`

Abstract

In this project is presented a comparison between the algorithms Deep Deterministic Policy Gradient (DDPG) and Soft Actor Critic (SAC) for the control in an environment with continuous actions space. Here is used the environment "Walker2d-v4" provided in the *openai-gym* package. Code of this project can be found on GitHub [1]

1 Introduction

When we are dealing with environments with continuous action spaces, we cannot use the traditional DQN algorithm, because it relies on finding the action that maximize the action-value function, which can be handle in discrete and low-dimensional action spaces, but in continuous ones it require an iterative optimization process at every step. So instead policy gradient methods are used, in particular actor critic ones, with a network that learns the policy (the actor), and another one that learn the values (the critic). Most of them are *on-policy* methods, which requires new samples to be collected for each gradient step. Instead *off-policy* methods can reuse past experience, stored in a *Replay Buffer*, and this speed up learning. Here two off-policy actor-critic methods are compared: Deep Deterministic Policy Gradient (DDPG) [1] and Soft Actor Critic (SAC) [2, 3]

1.1 The Environment

The environment used for the comparison is the "Walker2d-v4"[2] provided in the *openai-gym* package. In this environment we have to make a robot walk. The robot has 2 legs, with 3 hinge joints for each one: the hip, the knee and the ankle, and in each joint there is a motor, that can be controlled applying a torque.

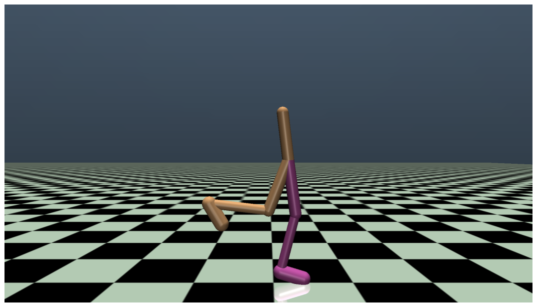


Figure 1: Frame of the environment

[1]https://github.com/Zumo09/AAS_Project
[2]https://www.gymlibrary.ml/environments/mujoco/walker2d/

So the agent take a 6-element vector for actions, with action space in $[-1, 1]$. The observation space is 17-dimensional array with information about the position and the velocity of the joints and of the (x, z) coordinate of the head of the robot (only the x position is omitted).

Each step the environment produces a reward of $+1$ for being alive, plus an amount proportional to the x-displacement between the two actions, positive if the robot moves forward, plus a negative reward if the agent takes actions that are too large. An episode terminates if it reaches 1000 time steps, or if the robot torso is too much tilted, or the tip of the head is too much low.

2 Deep Deterministic Policy Gradient

It is an extension of the DPG algorithm [4] that allows to use neural networks function approximators. It maintains a parameterized actor function $\mu_\phi(\mathbf{s})$ which specifies the current policy by deterministically mapping states to specific actions, a critic $Q_\theta(\mathbf{s}, \mathbf{a})$ that is learned using the Bellman equation as in DQN, and two target networks $\mu_{\bar{\phi}}(\mathbf{s})$ and $Q_{\bar{\theta}}(\mathbf{s}, \mathbf{a})$ used to calculate target values.

At each step the actions are selected using an exploration policy μ', obtained by adding noise sampled from a noise process $\mathcal{N}$ to the policy

$$\mu'_\phi(\mathbf{s}) = \mu_\phi(\mathbf{s}) + \mathcal{N} \tag{1}$$

Here $\mathcal{N}$ is chosen to be an Ornstein-Uhlenbeck process [5], that generate temporally correlated noise, useful to explore efficiently in physical control problems with inertia.

Then a batch of N transitions $(\mathbf{s}_t, \mathbf{a}_t, r_t, d_t, \mathbf{s}_{t+1})$ is sampled from the replay buffer $\mathcal{R}$. The critic network is optimized by minimizing the loss

$$L = \frac{1}{N} \sum_t \left[r_t + \gamma(1 - d_t) Q_{\bar{\theta}}(\mathbf{s}_{t+1}, \mu_{\bar{\phi}}(\mathbf{s}_{t+1})) - Q_\theta(\mathbf{s}_t, \mathbf{a}_t) \right]^2 \tag{2}$$

The actor instead is optimized taking a step in the direction of the *policy gradient*

$$\nabla_\phi J \propto \mathbb{E}_{\mathbf{s}_t \sim \mathcal{R}} \left[\nabla_\phi Q_\theta(\mathbf{s}_t, \mu_\phi(\mathbf{s}_t)) \right] = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{R}} [\nabla_{\mathbf{a}} Q_\theta(\mathbf{s}_t, \mathbf{a})|_{\mathbf{a}=\mu_\phi(\mathbf{s}_t)} \nabla_\phi \mu_\phi(\mathbf{s}_t)] \tag{3}$$

which, with the sampling of the states, can be approximated as

$$\nabla_\phi J \propto \frac{1}{N} \sum_t \nabla_{\mathbf{a}} Q_\theta(\mathbf{s}_t, \mathbf{a})|_{\mathbf{a}=\mu_\phi(\mathbf{s}_t)} \nabla_\phi \mu_\phi(\mathbf{s}_t) \tag{4}$$

At last the target networks are updated using the soft update rule, that constrains the target values to change slowly, greatly improving the stability of the learning

$$\bar{\theta} \leftarrow \tau\theta + (1 - \tau)\bar{\theta}, \quad \bar{\phi} \leftarrow \tau\phi + (1 - \tau)\bar{\phi}, \quad (\tau \ll 1) \tag{5}$$

At test time the policy is evaluated without the exploration noise.

3 Soft Actor Critic

It draw on the maximum entropy framework, which augments the standard maximum reward reinforcement learning objective with an entropy maximization term. The maximum entropy formulation provides a substantial improvement in exploration and robustness, because it optimizes policies to maximize both the expected return and the expected entropy of the policy. Here, instead of the deterministic actor used in DDPG, it is used a stochastic policy $\pi(\mathbf{a}_t|\mathbf{s}_t)$, and the optimal policy is defined as

$$\pi^* = \arg\max_\pi \sum_t \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[r(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathcal{H}(\pi(\cdot|\mathbf{s}_t)) \right] \tag{6}$$

where α is the temperature parameter that determined the relative importance of the entropy term versus the reward, and thus controls the stochasticity of the optimal policy.

In SAC are considered two parameterized soft Q-function $Q_{\theta_i}(\mathbf{s}_t, \mathbf{a}_t)$, a stochastic policy $\pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ and two target value functions $Q_{\bar{\theta}_i}(\mathbf{s}_t, \mathbf{a}_t)$. The functions Q can be modelled as a neural networks, and the policy as a Gaussian with mean and covariance given by neural networks. The two critic functions are trained independently to optimize $J_Q(\theta_i)$, and for the gradient steps is taken the minimum of them.

Also here for each gradient step a batch of N transitions $(\mathbf{s}_t, \mathbf{a}_t, r_t, d_t, \mathbf{s}_{t+1})$ is sampled from the replay buffer $\mathcal{R}$. The Q-functions parameters are optimized to minimize the soft Bellman residual

$$J_Q(\theta_i) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\mathcal{R}}\left[\frac{1}{2}\left(Q_{\theta_i}(\mathbf{s}_t, \mathbf{a}_t) - \left(r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1}\sim p}\left[V_{\bar{\theta}_i}(\mathbf{s}_{t+1})\right]\right)\right)^2\right] \tag{7}$$

with

$$V_{\bar{\theta}_i}(\mathbf{s}_{t+1}) = \mathbb{E}_{\mathbf{a}_{t+1}\sim\pi}\left[\min_{j\in\{1,2\}} Q_{\bar{\theta}_j}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \alpha \log \pi_\phi(\mathbf{a}_{t+1}|\mathbf{s}_{t+1})\right] \tag{8}$$

That becomes the gradient step

$$\nabla_\theta J_Q(\theta_i) = \nabla_\theta\left[\frac{1}{2}\left(Q_{\theta_i}(\mathbf{s}_t, \mathbf{a}_t) - y_t\right)^2\right] \tag{9}$$

with

$$y_t = r_t + \gamma(1 - d_t) \min_{j\in\{1,2\}} Q_{\bar{\theta}_j}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \alpha \log \pi_\phi(\mathbf{a}_{t+1}|\mathbf{s}_{t+1}) \tag{10}$$

The target networks are updated using the soft update rule (equation 5) as before.

The policy parameters instead can be learned by minimizing the following equation

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t\sim\mathcal{R}}\left[\mathbb{E}_{\mathbf{a}_t\sim\pi_\phi}\left[\alpha \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) - \min_{i\in\{1,2\}} Q_{\theta_i}(\mathbf{s}_t, \mathbf{a}_t)\right]\right] \tag{11}$$

To get the gradient step, a reparameterization trick has to be applied. The policy is reparametrized using a neural network transformation (more about it in appendix A)

$$\mathbf{a}_t = f_\phi(\epsilon_t; \mathbf{s}_t), \quad \epsilon_t \sim \mathcal{N}(0, I) \tag{12}$$

and then the step for the optimization can be approximated as

$$\nabla_\phi J_\pi(\phi) = \nabla_\phi\left[\alpha \log \pi_\phi(f_\phi(\epsilon_t; \mathbf{s}_t)|\mathbf{s}_t) - \min_{i\in\{1,2\}} Q_{\theta_i}(\mathbf{s}_t, f_\phi(\epsilon_t; \mathbf{s}_t))\right] \tag{13}$$

Also here at test time the stochasticity is removed, and the agents are evaluated using as actions the mean values predicted by the actor network.

4 Experiments

Besides the two algorithm presented above, another experiment done is to add the double Q-function trick also on the DDPG. The experiments has been conducted by setting the limit on the number of episodes used in the training to 15k, in this way less promising runs ends quickly, without wasting time and energy. In fact a good run, resulting in good performances, took about 16-20h to complete all the 15K episodes, running on a Nvidia GeForce GTX 1060 with 6 GB of memory. A small hyperparameters search has been done, and it has been noted that SAC and DDPG with the double critic are robust to the change of the parameters, instead vanilla DDPG is very brittle to the changes, and in lot of run it diverges quickly.

Below are showed the experiments that has been done using the hyperparameters listed in the respective papers, reported in table 1.

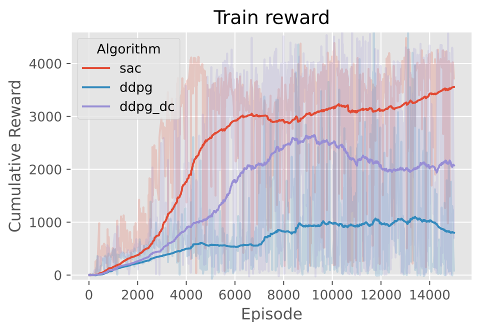
Table 1: Hyperparameters

Common		DDPG		SAC	
Param	Value	Param	Value	Param	Value
Optimizer	Adam	Hidden units	400, 300	Hidden units	256, 256
Discount (γ)	0.99	Soft update (τ)	10^{-3}	Soft update (τ)	5×10^{-3}
Buffer size	10^5	LR (A-C)	10^{-4}, 10^{-3}	LR (A-C)	3×10^{-4}
Sample size	256	Weight Decay	10^{-9}, 10^{-2}	Weight Decay	10^{-9}
Num. Episodes	15000	Noise (θ, σ)	0.15, 0.2	Temperature	0.5

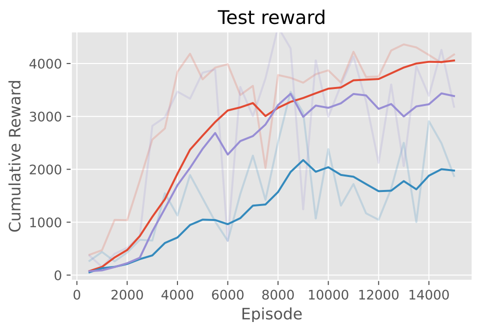
As it is shown in figure 2, all the agents improves their performances during training, and for all of them removing stochasticity at test time improves their results. But there are some significant difference.

Vanilla DDPG is the worst of them, both at train and test time. Introducing the double Q-function evaluation resolves in better performances, almost doubling the average cumulative reward. In fact, as explained also by [6], the double Q-learning trick mitigates the positive bias in policy improvement. This phenomenon can be seen in action looking to the Actor Loss in the DDPG training (figure 2c), that continues to decrease, even if the performances are not improving. It is due to the overestimation of the value of the states made by the critic.

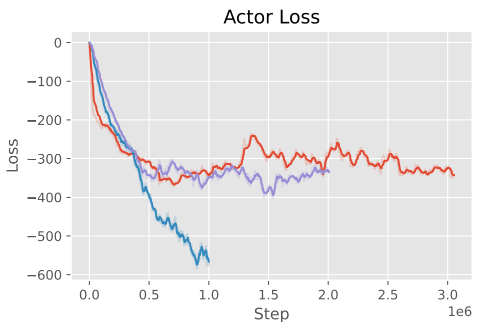
SAC is the one that performs better, possibly due to the better exploration. It can also be noted that the difference in cumulative reward between train and test time is lower in the case of SAC. That's because here also the stochasticity of the policy is optimized during training, instead in the DDPG algorithm, a noise is always added to the action selected by the agent to maintain exploration.



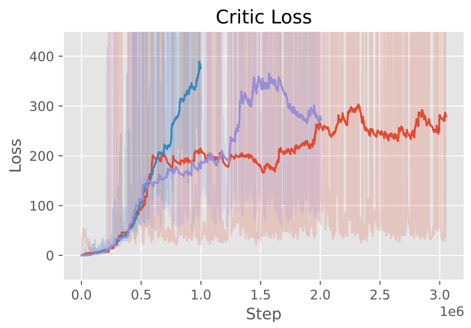
(a)



(b)



(c)



(d)

Figure 2: Training over 15k episodes

References

[1] Timothy P. Lillicrap et al. “Continuous control with deep reinforcement learning”. In: (2015). DOI: `10.48550/ARXIV.1509.02971`. URL: `https://arxiv.org/abs/1509.02971`.

[2] Tuomas Haarnoja et al. “Soft Actor-Critic Algorithms and Applications”. In: (2018). DOI: `10.48550/ARXIV.1812.05905`. URL: `https://arxiv.org/abs/1812.05905`.

[3] Tuomas Haarnoja et al. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: (2018). DOI: `10.48550/ARXIV.1801.01290`. URL: `https://arxiv.org/abs/1801.01290`.

[4] David Silver et al. “Deterministic Policy Gradient Algorithms”. In: Proceedings of Machine Learning Research 32.1 (June 2014). Ed. by Eric P. Xing and Tony Jebara, pp. 387–395. URL: `https://proceedings.mlr.press/v32/silver14.html`.

[5] G. E. Uhlenbeck and L. S. Ornstein. “On the theory of Brownian Motion”. In: *Phys. Rev.* 36 (1930), pp. 823–841. DOI: `10.1103/PhysRev.36.823`.

[6] Scott Fujimoto, Herke van Hoof, and David Meger. “Addressing Function Approximation Error in Actor-Critic Methods”. In: *CoRR* abs/1802.09477 (2018). arXiv: `1802.09477`. URL: `http://arxiv.org/abs/1802.09477`.

Appendix

A Enforcing Action Bounds

The actions are sampled from a Gaussian distribution

$$\mathbf{u} \sim \pi'(\mathbf{u}|\mathbf{s}) = \mathcal{N}(\mu_\phi(\mathbf{s}), \sigma_\phi(\mathbf{s})) \tag{14}$$

$$\mathbf{u} = \mu_\phi(\mathbf{s}) + \sigma_\phi(\mathbf{s}) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{15}$$

$$\mathbf{a} = f_\phi(\epsilon, \mathbf{s}) = \tanh(\mathbf{u}) \tag{16}$$

The network predicts $\mu(\mathbf{s})$ and $\log(\sigma(\mathbf{s}))$, with $\log(\sigma(\mathbf{s}))$ that is clamped to be in a range $[-10, 10]$ before exponentiation, due to numerical stability.

The invertible squashing function $\mathbf{a} = \tanh(\mathbf{u}) \in \mathbb{R}^D$ is applied in order to enforce a bound to the actions. So the log-likelihood of the policy π should can be calculated as (see appendix C of [3]):

$$\begin{aligned}
\log \pi(\mathbf{a}|\mathbf{s}) &= \log \pi'(\mathbf{u}|\mathbf{s}) - \sum_{i=1}^{D} \log(1 - \tanh^2(u_i)) \\
&= \log \pi'(\mathbf{u}|\mathbf{s}) - \sum_{i=1}^{D} \log\left(1 - \frac{(e^{2u_i} - 1)^2}{(e^{2u_i} + 1)^2}\right) \\
&= \log \pi'(\mathbf{u}|\mathbf{s}) - \sum_{i=1}^{D} 2(\log(2) + u_i + \log(1 + e^{2u_i}))
\end{aligned} \tag{17}$$

The last derivation has been done due to numerical stability.