Lab 7
Off-policy Actor-Critic methods: DDPG and TD3

November, 2025

1 Goal of the Lab

We have implemented, on one hand, value based algorithms like `DQN` that learn with sample efficiency but only in environments that are controlled using a set of discrete actions. On the other hand, we have implemented methods like `VPG` and `A3C` that learn not so efficiently but can deal with environments with continuous actions. Now we will implement first Deep Deterministic Policy Gradient (DDPG) [1] and and improvement over it called Twin Delayed Deep Deterministic policy gradient (TD3) [2] that are more sample efficient than on-policy gradient methods but at the same time can control continuous actions. They are described in the slides, and you can find the references in the bibliography.

2 Deep Deterministic Policy Gradient (DDPG) [8pts]:

In this exercise, we will implement the DDPG method (see figure 1). To show that it works in a continuous environment setting, we will work in the **continuous** version of the LunarLander-v2 environment that is created as follows:

```
env = gym.make("LunarLander-v3",continuous=True, render_mode='rgb_array')
```

In this environment, that you know well, actions are redefined to be 2 continuous actions, each in the range $[-1..1]$: the first one, controls the main engine and the second one controls left and right engines.

Note about parameters used for training: Use the ones described in Supplementary Information (section 7) of Experiment details in the original paper [1]. Even when the authors did not apply DDPG to the `LunarLander` problem, they seem to work also well in this problem. **Notice that DDPG is very sensitive to hyper-parameters**, architecture of the neural network and initialization of weights. I strongly warn you against trying to find the parameters by yourself instead of following the recommendations of the paper![1].

2.1 Understand the auxiliary classes:

In order to program the DDPG algorithm, I supply you with some classes implemented in different files. To implement DDPG you have to understand them.

1. **Network Architecture:** In file `ActorCriticNetworks.py` you will find implemented the Actor and Critic that will be used for DDPG. Do not change anything. In figures 2 and 3 you can see the architecture of the Actor and the Critic. Observe the following:
   (a) Use of batch normalization layers as suggested in the paper [1]
   (b) Use of custom initialization of weights, also as recommended in [1], section 7.

[1] You can try but I will not pay for your therapist afterwards...

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 1: DDPG algorithm

(c) Architecture for the Critic in which the state and action follow different paths of different length, so preprocessing of the state is done independently of the action. Finally, both paths converge to one layer where we sum the last layers of both paths.

2. **Noise classes:** DDPG is Deterministic in the sense that the policy always return the "best action", like DDQN. So, an exploration method has to be added in the method, like epsilon greedy in DQN. In DDPG exploration is implemented by adding a random value (noise) to the action. Random value can be generated using different stochastic procedures. The simplest one is sampling from a Normal distribution $N(0, \sigma)$. Another way to explore that seems to help better to DDPG for exploration is to follow the *Ornstein-Uhlenbeck* stochastic noise, that is very similar, but takes into consideration the last random value generated (see here for details). All these methods are implemented in the file `Noise.py` that defines classes for each kind of noise with a `sample` procedure that is used to choose the noise to add for exploration.

2.2 Programming Tasks:

1. **Replay Buffer:** You should implement a `ReplayBuffer` for DDPG. Remember that it is an off-policy method, so we need it! You can reuse the one we used for DQN. Notice that most of the program we will implement is very similar to DQN. That is because DDPG can be seen as an extension of DQN to continuous action spaces with the help of an Actor that is trained to find the maximum of the Q-value network.

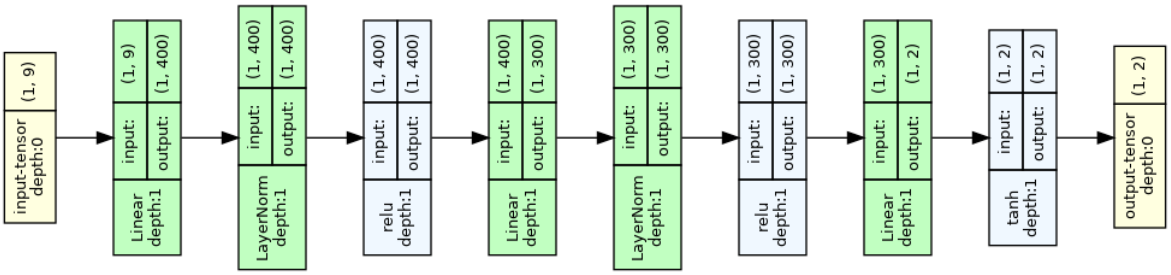

Figure 2: Actor architecture consists of 2 linear layers followed by Batch normalization and `Relu` function activation layers, and ending in a linear layer with a `tanh` activation layer that ensures that output is in range -1..1 which is the range of the actions in this continuous problem.

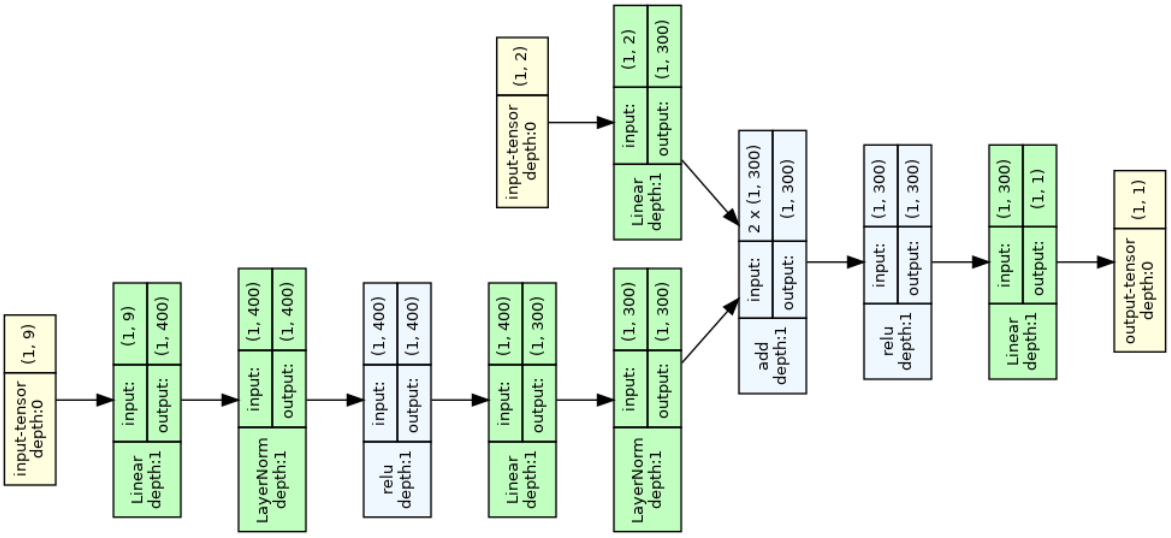

Figure 3: Critic architecture consists of 2 paths, lower one for the state and upper one for the action, that are combined with a sum of layers. The output is a single unbounded (no function activation) value - $Q(s, a)$

2. **Init of DDPG agent:** In init of the `DDPG` class in `DDPG.py` file, define the Critics and the Actors.
3. **Choose action:** In `DDPG.py` file, implement the method `choose_action` that given an state, returns the action to execute according to the Actor.
4. **Critic loss:** Implement the method `compute_critic_loss` that, given a batch of data, computes the Minimum Square Error for the Bellman equation similarly as it is done in DQN.
5. **Actor loss:** Implement the method `compute_actor_loss` that try to modify the actor to maximize the Q-values for a batch of states.
6. **Learn method: Apply gradients** In the learn method, when `ReplayBuffer` has enough data, get a batch of data and compute and update parameters of the Critic and Actor
7. **Learn method: Update targets** Do a soft update of the target neural networks with $\tau = 0.001$ as sugested in the paper.

After you have ended the programming tasks, run your program to see the performance. I reommend you to use GoogleColab with acces to a GPU to run the program because it takes some time to learn. For that I supply you with the file `RunDDPG_Colab.ipynb`. In figure 4 you can see the evolution of learning of a typical run. It takes about 1h and 1h30 to end the training in GoogleColab in a GPU node.

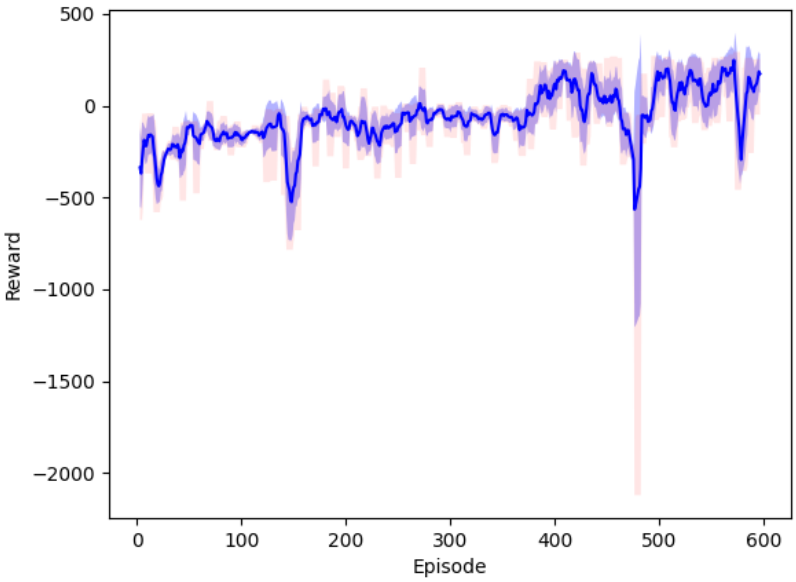

Figure 4: Typical run of learning. An anomalous bad trial reduces the scale for the reward, but the learning goes from -400 to +200. We end learning when in 5 consecutive episodes, reward is higher than 220.

3 Twin Delayed DDPG (TD3) [2pts]

DDPG is an off-policy method able to learn with continuous action spaces. However, it is very sensible to parameters and takes some time to learn. Some improvements can be made. This is the aim of [2] that introduces three simple changes that you should implement in a copy of your solution file `DDPG.py` that you will call `TD3.py`. The changes are the following (see full algorithm in figure 5):

3.1 Programming Tasks:

1. **Clipped Gaussian noise for exploration:** In TD3 implementation, they used for exploration regular Gaussian Noise (that is also implemented in the file `Noise.py`). In addition, they clipped the noise to be not very different of the recommended action of the policy. Implement this change.

2. **Twin neural networks:** In order to avoid the overestimation of Q-values, due to random initialization and the application of the *max* operator, they used two neural networks Q_{θ_1} and Q_{θ_2} (plus their target counterparts) to estimate the Q-values. When computing the target for the calculation of the loss of the critics, they use the most pessimistic Q neural network (the one that returns the smaller long-term reward, implemented as the minimum Q_{θ_1} and Q_{θ_2}. Implement this improvement also in the `TD3.py` file.

3. **Delayed training of the actor:** Usually, in Actor Critic architectures, critic networks should learn faster than the actors because the actor is guided by the critic and, if we don't have good estimations of the critic, the actor will not learn well. This is usually implemented giving a larger learning rate to the critic than to the actor, but sometimes will not work. In TD3 they propose to update the critics at each iteration of the main loop and the actor only every two or three loops, delaying in this way the training of the actor and increasing the stability and precision in the learning of the actor. Implement also this changes.

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ
with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

Figure 5: TD3 Algorithm

4 Deliver your work

With all these changes implemented, run TD3 in the same environment and compare results with DDPG. Write a pdf file reporting a comparison of the performance of the two algorithms, and also write a section with your opinions and comments about the implementation and comparison. Finally, create a Zip file with your pdf document together with your solution for the TD3 and DDPG algorithms, and deliver them to the *racó*.

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *Iclr 2016*, 2016, pp. 1–14. [Online]. Available: http://arxiv.org/abs/1509.02971

[2] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," in *35th International Conference on Machine Learning*, feb 2018. [Online]. Available: http://arxiv.org/abs/1802.09477