DDPG Halfcheetah - v2
Implementation assignment

Pieri Sara, matricola 1799222
Santilli Sofia, matricola 1813509

February 2021

In this paper the DDPG algorithm is presented, discussing its main features and the decisions taken while implementing the code. The results obtained from the application of this algorithm on the HalfCheetah-v2 environment of Mujoco are shown.

1 Introduction

DDPG stands for '*Deep Deterministic Policy Gradient*'. It is a Reinforcement Learning off-policy algorithm that presents an actor-critic, model free algorithm based on the deterministic policy gradient for continuous action domains.
It combines ideas from DQN (*Deep Q-Network*) and DPG (*Deterministic Policy Gradient*). As the first, it uses the Experience Replay and slow-learning target networks. DDPG recalls DPG as it operates over continuous action spaces.

2 Algorithm

The fact that DDPG is adapted specifically for the environments with continuous action spaces is related to:

- how it computes the max over actions in $max_a\, Q^*(s,a)$, that can be approximated to $Q(s,\mu(s))$. In fact, given the continuous action space and presuming the function to be differentiable with respect to the action argument, DDPG uses an efficient, gradient-based learning rule for a policy.
- a specific way to interleave learning an approximator to $Q^*(s,a)$ with learning an approximator to $a^*(s)$.

Therefore, DDPG currently learns a Q-function and a policy. For this aim, DDPG uses the Actor-Critic architecture, composed of two neural networks: the Actor and the Critic.

2.1 Critic

The *Critic* Q is a Q-value network that takes a state and an action as input and outputs the Q-value for the state-action pair.
This neural network is the approximator to the optimal action-value function $Q^*(s,a)$; from now on it will be represented as $Q_\phi(s,a)$, where ϕ are the parameters of the network.
The goal is to have $Q_\phi(s,a)$ closely to satisfying the Bellman equation[1]. In order to do that, the mean-squared Bellman error (MSBE) function is used:

$$L(\phi, B) = E_{(s,a,r,s',d)\sim B}\left[(Q_\phi(s,a) - (r + \gamma(1-d)\, max_{a'}\, Q_\phi(s',a')))^2\right] \quad (2)$$

where a set B of transitions (s,a,r,s',d) from the buffer[2] is considered.
The term $r + \gamma(1-d)\, max_{a'}\, Q_\phi(s',a')$ is called target, as it is the value at which the Q-function has to tend when the MSBE loss is minimized. As this target depends on the same parameters ϕ that we want to train, in order to improve stability in learning avoiding divergence, the target Q network ($Q_{\phi_{targ}}$) is introduced.
A *target network* is a time-delayed copy of the original network and slowly tracks the learned network. The target weights are updated periodically, through soft updates, based on the comparison of target and main network:

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1-\rho)\phi \quad (3)$$

ρ is a hyperparameter between 0 and 1 (usually close to 1). This means that the target values are constrained to change slowly, greatly improving the stability of learning. However the presence of target networks may slow learning, since the target network delays the propagation of value estimations.
Combining all these informations, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, B) = E_{(s,a,r,s',d)\sim B}\left[(Q_\phi(s,a) - (r + \gamma(1-d)\, Q_{\phi_{targ}}(s', \mu_{\theta_{targ}})))^2\right] \quad (4)$$

where $\mu_{\theta_{targ}}$ is the target policy that will be introduced in the following paragraph.

2.2 Actor

The *Actor* μ, is a deterministic policy network, that uses the Q-function to learn the policy $\mu_\theta(s)$ that gives the action that maximizes $Q_\phi(s,a)$. In fact the actor takes a state as input and outputs an exact continuous action a*, instead of a probability distribution over actions across a discrete action space. In this it

[1]The Bellman equation describes the optimal action-value function Q*(s,a) as:

$$Q^*(s,a) = E_{s'\sim P}[r(s,a) + \gamma\, max_{a'}\, Q^*(s',a')] \quad (1)$$

where s' is sampled by the environment from a distribution P.
[2]The description of the buffer follows at paragraph 3.

consists the determinism.
The action space is continuous. So, as introduced before, assuming that the Q-fuction is differentiable w.r.t. actions, the gradient ascent is used in order to solve:

$$max_{\vartheta} E_{s\sim D}[Q_{\phi}(s, \mu_{\vartheta}(s))] \tag{5}$$

As anticipated at the end of the previous paragraph, in the case of the actor, a target policy network is used. It is found in the same way of the target Q-function and is subjected to soft updates:

$$\theta_{targ} \leftarrow \rho\theta_{targ} + (1-\rho)\theta \tag{6}$$

This target network is used to choose the next action. Its output is fed into the critic network to calculate the Q-value of a state during the calculation of the critic loss.

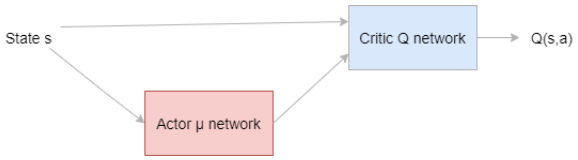

Figure 1: Actor and Critic operation scheme.

3 Replay Buffer

DDPG uses a Replay Buffer to sample previous experiences, later used to update neural network parameters.
The replay buffer has been implemented as a list of tuples. In fact during the training, every time an action is performed by the agent, the response of the environment is stored in the buffer as a tuple (state, action, reward, next_state, done[3]). To update the value and the policy networks, mini-batches of experience are randomly sampled from the buffer.
It should be noted that storing the data in a replay buffer and taking random batches satisfies the request of having the data to be independently distributed.

4 Exploration

Since the action space is continuous, exploration in DDPG is done by adding, at training time, noise sampled from a noise process N to the action determined by the policy μ:

$$\mu'(s) = \mu(s) + N \tag{7}$$

[3]If *done* is true, it means that the new state is a terminal state.

where N can be chosen to suit the environment.
In the experiments different types of noise with different parameter settings have been considered and the results obtained compared. It is an advantage of off policy algorithms: the problem of exploration can be treated independently from the learning algorithm.
Moreover, to improve the exploration at the beginning of the training, a number of *initial random steps* has been used. Therefore the agent, for the first few episodes, takes random actions from the environment.

4.0.1 Ornstein-Uhlenbeck Noise

The Ornstein-Uhlenbeck noise is commonly used in DDPG. It achieves temporally correlated exploration, generating at each step a noise that is correlated with the one at the previous step. The formula that describes the evolution of a general OU process and that was applied for the noise calculation is:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \tag{8}$$

where x_t is the state/noise, μ is the mean and $\theta > 0$ and $\sigma > 0$ are two parameters that required some tuning in the experiments.
The term $(\mu - x_t)$ is a *drift* term because it allows the mean to act as an equilibrium level for the process. In fact if the current value of the noise is less than the (long-term) mean, the drift will be positive; otherwise it will be negative. At the beginning of the training process the state, as the mean, is initialized to zero.
W_t denotes a Wiener process. It is a continuous-time stochastic process for $t \geq 0$ with $W(0) = 0$ and such that the increment $W(t) - W(s)$ is a Gaussian with mean 0 and variance $t - s$. In the code it was implemented by sampling random floats from the continuous uniform distribution over the half-open interval $[0.0, 1.0)$.

4.0.2 Gaussian Noise

The authors of the DDPG paper recommend time-correlated OU noise, but more recent results suggest that also uncorrelated, mean-zero Gaussian noise works well. Since the latter is simpler, it was included in the implementation and experiments.

4.0.3 Adaptive Gaussian Noise

The Adaptive Gaussian Noise was introduced to facilitate getting higher-quality training data by progressively reducing the scale of the noise over the course of training. This type of noise uses a maximum σ_{max} and a minimum σ_{min} value of the variance and the formula

$$\sigma = \sigma_{max} - (\sigma_{max} - \sigma_{min}) \times min\,(1, \frac{t}{d}) \tag{9}$$

As long as the number of the current step[4] is less than the fixed decay period, sigma is close to the maximum value. During the training it increases in the range $[-\sigma, \sigma]$. Otherwise when the step is greater than or equal to the decay period, sigma coincides with the established minimum sigma.

5 Environment

The environment taken into account is Halfcheetah-v2. The agent performs only continuous actions. It moves a 2D bipedal body, made of 8 articulated rigid links. The agent can observe the position and the angles of its joints, its linear positions and velocities (17D) and can act on the torques of its legs joints (6D). The reward is the sum of instantaneous linear velocities on the x axis $v_x(t)$ at each step, minus the norm of the action vector a:

$$r = \sum_t (v_x(t) - 0.1 \times |a(t)|^2) \tag{10}$$

The environment is reset after 1000 steps, without any termination criterion. So, every 1000 steps, *done* becomes true and the environment is reset. These 1000 steps correspond to one episode.

6 Experiments

This section discusses the experiments conducted, the results obtained and the best settings for the parameters.
The best result achieved is a reward of 3873. It was obtained setting: initial_steps=10 000, steps=50 000, buffer size = 10 000, batch_size=32, use of the Adam optimizer for actor and critic networks, discount factor $\gamma = 0.99$, $\rho = 5e-3$ and the OUNoise with $\theta = 1.0$ and $\sigma = 0.1$. Therefore, from now on, all the following experiments are intended to start from this configuration.

6.1 Networks Structure

In the implementation two separated networks for actor and critic are used. The actor network has three fully connected layers and three non-linearity functions, ReLU for hidden layers and tanh for the output layer. On the other hand, the critic network has three fully connected layers and it uses ReLu activation function for the hidden layers. While the input size of the actor network is the dimension of the state, the one of the critic network is the sum of the dimensions of state and action. One thing to note is that the final layer's weights are initialized so that they are uniformly distributed.
In order to set the weights of the hidden layers close to zero, without being too small, these weights are initialized in the interval $[-v, v]$ with $v = \frac{1}{\sqrt{n}}$, where n is the number of input for a certain neuron.

[4]Each *step* corresponds to a new action taken.

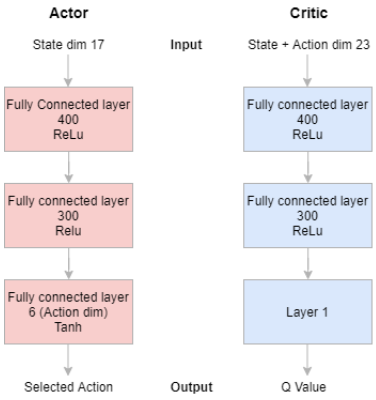

Figure 2: Actor and Critic network structure.

Some tests have been done using, for the actor and the critic, the most common optimizers. With the SGD optimizer the reward is -16, a very low result. On the contrary, applying the RMSProp optimizer the reward is 3263: it is a good result, but worse than the one obtained with the Adam optimizer (3873).

6.2 Buffer Implementation

6.2.1 Buffer Size

The replay buffer is a finite-sized cache. During the training, until the buffer is still not full, the new tuple is stored in the first available position; otherwise the oldest sample is discarded.
In order to have a stable behaviour for the algorithm, the replay buffer has to be large enough to contain a wide range of experiences, but not too much to keep every past experience. Using only the very-most recent data, we found a tendency to overfitting, while using too much experience, the training slows down and it is difficult to obtain good solutions.
These considerations on the buffer size are here supported by the tests made. Taking into account that the number of total steps done is 50 000, the best reward is obtained (3873) with a memory size of 10 000.

6.2.2 Batch Size

The batch size impacts how quickly a model learns and the stability of the learning process. A batch size with a lot of training examples results in a more accurate estimate of the error gradient, which makes more likely to adjust the network weights in a way that will improve the performance of the model. Nevertheless, the use of a small batch size with noisy updates can result in faster learning and a more robust model, offering a regularizing effect and lower generalization error. The results presented in the following table confirm that

using a batch size of 32 achieves the best training stability and generalization performance.

Buffer size	Result
100	341
1 000	653
5 000	1 953
8 000	3 063
10 000	3 873
20 000	2 419
50 000	919

Table 1: Comparison of the buffer size results.

Buffer Size	Result
8	-142
16	2 548
32	3 873
64	569
128	-208
256	970

Table 2: Comparison of the batch size results.

6.3 Total Steps and Initial Random Steps

The number of steps and initial random steps required some tuning in order to find the best combination.
Concerning the total number of steps, while for a small number the results are very low, increasing them beyond a certain value does not show a significant increase in performance. This is particularly evident when the training graphs are unstable and show peaks in the evolution of rewards and losses. While in the case of more uniform evolutions, the results tend not to deviate much from the best values obtained with a smaller number of steps. A good value for the total number of steps is 50 000. This number must be considered in relation to other parameters of our implementation such as the buffer size, the learning rates and the number of initial random steps. The Table 3 shows the results obtained in relation of the number of steps used.
As previously mentioned, the initial random steps are used in order to speed up the learning process. For values of the initial random steps smaller than the buffer size, the results obtained are low. This is due to the fact that until the buffer has been filled, updates are made using very similar batches. The best results are obtained when the number of initial random steps is equal to the size

Steps	Result
10 000	-1.63
20 000	-368
30 000	2 760
40 000	2 767
50 000	3 873
60 000	3 845
80 000	3 866
100 000	3 892
500 000	1 256

Table 3: Steps experiment results using OU-Noise with $\theta = 1$, $\sigma = 0.1$, 10 000 random steps and buffer size 10 000.

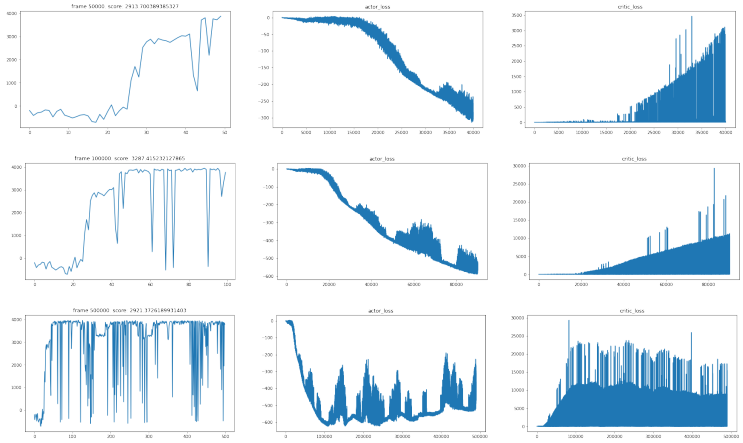

Figure 3: Comparison between two results obtained with OUNoise, $\theta = 1.0$, $\sigma = 0.1$ and 10.000 initial random steps. From above with 50.000, 100.000 and 500.000 steps.

of the buffer. In fact, in this case, the mini batch considered for the updates of the networks are sampled from a larger set of random actions.
Having the number of initial random steps greater than the buffer size, leads to a waste of actions; in fact the buffer is already filled with tuples obtained by taking random actions and no update of parameters is done.
The table below shows some examples of results obtained.

Random Steps	Result
0	-687
10	1 948
100	108
500	1 557
1 000	3 026
2 000	1 458
6 000	-106
8 000	3 774
10 000	3 873
15 000	2 737
20 000	1 623

Table 4: Initial Random Steps results comparison using OU-Noise with $\theta = 1$, $\sigma = 0.1$, 50.000 steps and buffer size 10.000.

6.4 Discount Factor γ

The discount factor previously met in the formulas (3) and (5) is a number within the range [0, 1]. It gives the best results when it tends to 1. In fact 3873 is reached with γ=0.99. Decreasing γ, also the reward decreases: already with γ=0.9, the reward is halved to 1555. When γ=0.3 the reward becomes negative (-85).
This result means that, when the agent is searching for the best action to do in order to maximize the reward, both the current and the future rewards have the same weight on its decision.

6.5 ρ in Soft Updates

Previously it was told that ρ is a hyperparameter between 0 and 1, usually close to 1. In the code implemented, the formulas (4) and (7) have been inverted:

$$x_{targ} \leftarrow \rho x + (1 - \rho) x_{targ} \tag{11}$$

So, in this implementation, the optimal ρ will be close to 0 (but not too much, otherwise the updates of the target network parameters would be too slow).

Rho	Result
5e-4	236
5e-3	3873
5e-2	2129
5e-1	-39
1	-600

Table 5: Results obtained with the *different* ρ.

6.6 Noise

The results obtained show how the type of noise chosen and the parameters set greatly influence the scores obtained.
Comparing the best result obtained for each noise, the test rewards with the Gaussian and Adaptive Gaussian are respectively 3398 e 2885, not very far from the one with the OU Noise (3873). However, the evolution plots have a less regular and much noisier trend in the case of the two Gaussians. This is also reflected in the half-cheetah's performance during the test. With the Gaussian Noise, the cheetah makes very long jumps in which it raises the front of the body, resulting sometimes in crashes and loss of balance. With the Adaptive Noise, on the other hand, the cheetah tends to overturn, especially when the speed increases. Moreover, in both cases, the starting moves are critical. On the contrary, in the case of the OU Noise, the trend is much more stable and the cheetah tends to always perform the same movements across the episode.
The best results obtained are highlighted in the following tables and a comparison of the the plots (reward, actor loss and critic loss) is shown below.

Mu	Sigma	Result
0.0	1.0	-16
0.0	0.1	2461
0.0	0.2	370
0.0	0.4	2527
0.0	0.05	3398
0.0	0.01	-296

Table 6: Results obtained with the *Gaussian Noise* for different values of μ and σ.

Sigma min	Sigma max	Decay_period	Result	Steps
1	5	100 000	-200	50 000
0	0.4	50 000	769	50 000
0	1.0	50 000	297	50 000
0	0.2	50 000	2885	50 000
0	0.05	50 000	2237	50 000
0	0.1	50 000	1325	50 000
0.2	1	50 000	772	50 000
0.2	1	50 000	1776	100 000
0	0.2	50 000	293	100 000
0	0.6	100 000	593	50 000

Table 7: Best results obtained with the *Adaptive Gaussian Noise* for different values of the parameters.

Theta	Sigma	Result	Steps
1.0	0.1	3 873	50 000
1.0	0.1	3 892	100 000
1.0	0.2	1 877	50 000
1.0	0.3	2 565	50 000
0.15	0.2	-595	50 000
0.5	0.1	3 448	50 000
0.5	0.1	2 145	100 000
0.7	0.1	2 544	50 000
2.0	0.1	-498	50 000

Table 8: Results obtained with the *OU-Noise* for different values of the parameters.

where N is the noise given by Ornstein-Uhlenbeck, correlated noise process.

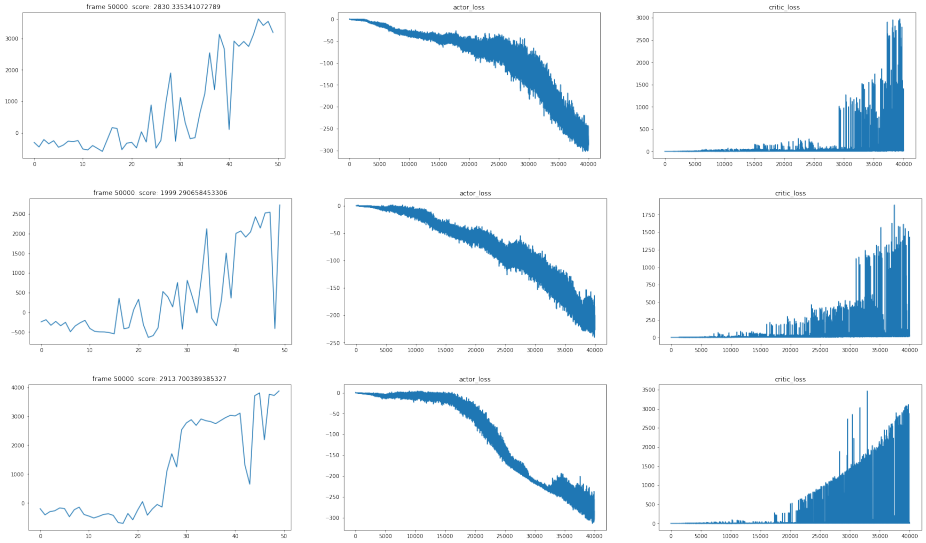

Figure 4: Comparison of the best results obtained by type of noise. From above, the plots of the reward, actor loss and critic loss for the Gaussian, Adaptive Gaussian and OU Noise.

7 Conclusions

In this paper has been presented the DDPG algorithm, describing its main features. The algorithm has been applied to the HalfCheetah-v2 environment. Different tests have been done changing the elements of actor and critic networks, the buffer structure, the type of noise applied and other parameters. The results obtained showed that the best score, along with the more stable behaviour, is obtained considering the OU Noise, all the rewards (both the current and the future ones) with almost the same weights and a number of initial random steps equal to the dimension of the buffer.

References

[1] "Continuous control with deep reinforcement learning", Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra - 5 Jul 2019

[2] https://spinningup.openai.com/en/latest/algorithms/ddpg.html

[3] https://towardsdatascience.com/three-aspects-of-deep-rl-noise-overestimation-and-exploration-122ffb4bb92b

[4] https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b