University of Liège

Inverted Double Pendulum

Optimal decision making for complex problems

Maxime Meurisse (s161278)
François Rozet (s161024)

Academic year 2020-2021

Introduction

In this project, we consider the *Double Inverted Pendulum* control problem. We used this source code[1] as a reference implementation of the environment.

The Double Inverted Pendulum consists of two arms connected to each other by a hinge at their ends. One of the two arms is connected to a motorised cart, via a hinge, at its other end (Figure 1).

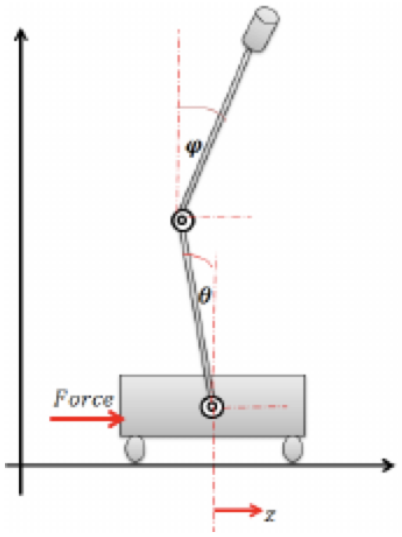


Figure 1 – The Double Inverted Pendulum environment.

This system is described by the displacement z of the cart on a rail, along a single horizontal axis, and the position of each arm, respectively θ for the "first arm" and φ for the "second arm", relatively to their vertical position. The only input of the system is a force F applied on the cart to move it along the rail.

The goal is to keep the system, as long as possible, in a vertical position close, or similar, to the equilibrium position, *i.e.* a situation where $\theta = \varphi = 0$.

This system is under-actuated since the number of actuators is lower that the number of links: a single input force (F) in used to control the 3 degrees of freedom (z, θ and φ). Being unstable and chaotic, the Double Inverted Pendulum is among the most difficult problems to control. [1, 2]

In this project, we try to stabilize the Double Inverted Pendulum using *Reinforcement Learning* algorithms. The rest of this document is organized as follows: Section 1 will formalize and characterize the environment and Section 2 will explained our Reinforcement Learning algorithm (*Deep Deterministic Policy Gradient* [3, 4]), evaluate its performance and compare it to a well-known algorithm (*Fitted-Q-Iteration with Extremely Randomized Trees* [5]) seen during the course.

[1] We made a very little fix about camera settings. Our modified version of the library is provided in our archive. This modification does not affect the dynamics.

1 Domain

In this section, we define the dynamics of the system and give formalization and characterization of the environment.

1.1 Dynamics

The Double Inverted Pendulum as a continuous-time dynamics [5]. Since it is very hard to infer the exact dynamics based on the reference source code we use, we referred to the dynamics described by Bogdanov [6] instead.

In the implementation we used, the dynamics has been discretized with the time between t and $t+1$ defined equal to 0.0165 s. An integration time step of 0.01 s has been fixed.

1.2 Formalization and characterization

The system is *deterministic*: when a force F is applied to the cart, the latter and all other components react, without any randomness, according to their dynamics.

To formalize the environment, we provide *state space*, *action space* and *reward signal*.

1.2.1 State space

As mentioned in the introduction, the environment has three degrees of freedom: z, θ and φ. It can then be fully described by a state $\boldsymbol{x} \in \mathbb{R}^6$,

$$\boldsymbol{x} = \begin{pmatrix} z & \dot{z} & \theta & \dot{\theta} & \varphi & \dot{\varphi} \end{pmatrix}^T \tag{1}$$

where $\dot{\square}$ denotes the derivative of $\square$. The derivatives give velocities, respectively of the cart, the first arm and the second arm, which are necessary to describe the state without ambiguity. Indeed, for a given position, an element can be moving to the left, to the right or stay still.

The displacement z is a continuous value belonging to the interval $[-1, +1]$ (with -1 corresponding to the left most position, and $+1$ the right most one) representing the displacement of the cart on a rail, relative to the center position 0.

Each angle can take any value between $-\pi$ and $+\pi$. However, one should note that, in practice, a wide range of angle values can never be reached by the system since the simulation is stopped when it reaches a terminal state (defined later in this Section).

In the implementation of the environment we used, the agent doesn't have direct access to these values. Instead, the agent receives an *observation* $o(\boldsymbol{x}) \in \mathbb{R}^9$ of the state,

$$o(\boldsymbol{x}) = \begin{pmatrix} z & \dot{z} & p_x & \sin\theta & \cos\theta & \dot{\theta} & \sin\varphi & \cos\varphi & \dot{\varphi} \end{pmatrix}^T \tag{2}$$

where p_x is the the displacement of the second arm's center of mass, in the same coordinate system as z.

The system starts in an *initial state* defined by setting the displacement z to 0 and the position of each arm's center of mass, along the horizontal axis, to a value drawn from $\mathcal{U}([-0.1, 0.1])$. The velocity (and torque) of each arm is initially set to 0.

The simulation stops when a *terminal state* is reached. A state is considered as terminal when p_y, the second arm's center of mass position along the vertical axis, is lower or equal to a fixed threshold $\tau = 0.7$.

1.2.2 Action space

The system only has one action u that correspond to the force F applied on the cart.

The action u is a continuous value belonging to the interval $U = [-1, 1]$. At each step of the simulation, the value of u is injected in the system dynamics to move the cart.

Depending of the sign of u, the cart accelerates to the left (negative value) or to the right (positive value).

1.2.3 Reward signal

The *reward signal* r is defined as a sum of three terms,

$$r(\boldsymbol{x}) = R_a - r_d(\boldsymbol{x}) - R_v \tag{3}$$

where

- R_a is a constant reward equal to 10 that the agent receives at each step for being still alive;
- $r_d(\boldsymbol{x})$ is defined as
$$r_d(\boldsymbol{x}) = \frac{1}{100} p_x^2 + (p_y - 1.7)^2 \tag{4}$$
with $p = (p_x, p_y)$ the position of the second arm's center of mass;
- R_v is a constant reward equal to 0.

Note. Normally, the term R_v is computed using velocity of each hinge so that the higher the velocity of each hinge, the more the agent is penalised. However, in the implementation we used, this term is manually set to 0 due to an issue in the code. Therefore, there is no penalty for having arms that move too fast.

Therefore, at each step of the simulation, the agent receives a reward that depends mainly on the position of its second arm: the closer the arm is to its vertical equilibrium position, the higher the reward will be.

2 Policy Search Techniques

In this section, we first implement the *Deep Deterministic Policy Gradient* (DDPG) algorithm [4] as a policy search technique.

Then, we implement a well-known algorithm seen during the course: the *Fitted-Q-Iteration with Extremely Randomized Trees* (FQI with XRT) [5].

Finally, we evaluate and compare policies obtained in terms of *expected discounted cumulative reward* obtained at the end of each episode.

Note. For the sake of simplicity, in the rest of this Section, the observation $o(\boldsymbol{x})$ of the agent will be simply noted as x and be qualified as its "state".

2.1 Deep Deterministic Policy Gradient

The explanations provided in this section are inspired by [3, 4, 7–10].

While *Q-learning* and *Deep Q-learning* (DQL) were particularly well suited for our previous assignments with discrete action spaces, they can not be used in the Double Inverted Pendulum problem since the action space is continuous. For this reason, we decide to work with the DDPG algorithm, which is a kind of adaptation of DQL to continuous action spaces.

The DDPG algorithm is shown in Algorithm 1. The main steps is explained in the following.

2.1.1 Neural networks

The DDPG algorithm uses four neural networks: a deterministic policy function network θ, a Q-network ϕ and two target networks associated to, respectively, θ and ϕ. Networks θ and ϕ acts like *Advantage Actor-Critic* [11]: the actor (θ) chooses an action at each time step and the critic (ϕ) evaluates the quality of the Q-value.

Target networks are time-delayed copies of original networks. Their use helps to improve stability during training, as already explained in our Assignment 3.

Each network is a shallow *Multi-Layer Perceptron* (MLP) with an input layer, one hidden layer composed of 256 neurons and an output layer. All activation functions are ReLU. The output of the actor network passes trough a tanh activation function.

For both networks, we use an Adam [12] optimizer with learning rates set to, respectively, 10^{-4} and 10^{-3}.

Target networks are updated using *Polyak averaging* [13] at the end of each update step (*cf.* equations 8 and 9). We fixed the value of ρ to 10^{-2}.

2.1.2 Action selection

Each action is selected using the policy network (line 4 of the Algorithm). Exploration is done by adding a noise ε directly to the action selected.

To generate the noise ε, we use the same process as Lillicrap et al. [4]: the *Ornstein–Uhlenbeck process*. The latter generates a sequence of noises ε_i which are correlated. The exact re-

Algorithm 1 Deep Deterministic Policy Gradient [4, 7]

1 Initialize policy parameter θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
2 Set target parameters equal to main parameters: $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3 **repeat**
4 Observe x and select $u = \min\left\{\max\left\{\mu_\theta(x) + \varepsilon, u_{Low}\right\}, u_{High}\right\}$
5 Execute u
6 Observe next state x', reward r and done signal d
7 Store (x, u, r, x', d) in $\mathcal{D}$
8 **if** x' is terminal **then**
9 Reset environment state
10 **if** it's time to update **then**
11 **for** however many updates **do**
12 Randomly sample a batch B of transitions from $\mathcal{D}$
13 Compute targets

$$y(r, x', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(x', \mu_{\theta_{\text{targ}}}(x')) \tag{5}$$

14 Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(x,u,r,x',d)\in B} (Q_\phi(x, u) - y(r, x', d))^2 \tag{6}$$

15 Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{x\in B} Q_\phi(x, \mu_\theta(x)) \tag{7}$$

16 Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi \tag{8}$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta \tag{9}$$

17 **until** convergence

lation is

$$\varepsilon_{i+1} = \varepsilon_i + a(b - \varepsilon) + c\mathcal{N}(0, 1) \tag{10}$$

with ε_0 drawn from $\mathcal{N}(0, 1)$. We set a, b and c to, respectively, 0.15, 0 and 0.2.

2.1.3 Replay buffer

As for Deep Q-learning implemented in our previous Assignment 3, we use a replay buffer $\mathcal{D}$ of capacity 65 536. We fill the buffer by adding transitions collected during episode simulations (line 5, 6 and 7 of the Algorithm). A transition is a tuple (x, u, r, x', d) where x is the state of the agent, u the action played, r the reward, x' the new state and d, the *done signal*, a binary value indicating whether the agent has reach a terminal state (1) or not (0).

When the buffer contains enough elements, we sample batches of 32 elements (line 12

of the Algorithm) to train the networks. An update of the networks (line 10 of the Algorithm) is done after each step of an episode (line 11 of the Algorithm), when the buffer contains enough elements.

2.1.4 Discrete action space

DDPG algorithm is, by default, no suitable for discrete action space. To adapt it, we use the *Wolpertinger policy* [14].

This policy can be seen as additional steps added to the action selection of the Algorithm 1 (line 4). Firstly, we define a discrete action space U_d. Secondly, we use the Actor network to get a (continuous) action u. Instead of directly using u, we give it to a k *nearest neighbors* model that outputs k nearest discrete actions $u_{d_i} \in U_d$ (with $i = 1, \dots, k$). Finally, we create all (x, u_{d_i}) pairs, with x being the current state of the agent, and keep the action associated to the pair that maximize the Q-value approximated by the Critic network.

In our project, we set k to half the number of discrete actions. It is important to note that, in the case of discrete action space, we do not add noise ε to the selected discrete action.

Note. As mentioned in the original paper [14], the Wolpertinger policy has been designed for *large* discrete action spaces. In our project, working with k nearest neighbors of a continuous action or with all discretized actions doesn't make much difference because U_d is very small. Still, we choose to work with the Wolpertinger policy to have a more general algorithm.

Moreover, to get the nearest neighbors, we work with a k nearest neighbors model from the scikit-learn library. Our action space being one-dimensional, this model is overly complex for the task, but, once again, we keep it in order to have a more general implementation.

2.2 Fitted-Q-Iteration with Extremely Randomized Trees

The content of this Section is inspired from our previous Assignment 2.

We choose to compare performance of the DDPG algorithm with the FQI with XRT algorithm seen during the course.

The goal of the FQI algorithm is to train a sequence of supervised regressors $\hat{Q}_1$, $\hat{Q}_2$, …, $\hat{Q}_N$ estimating respectively the functions Q_1, Q_2, …, Q_N. To do so, N training sets are built. Let h_i $(i = 1, \dots, N)$ be sets of one-step transitions $\{(x_k, u_k, r_k, x'_k) \mid k = 1, \dots, t\}$. Then, the training sets are

$$\begin{aligned} \mathrm{TS}_1 &= \{((x, u), r) \mid (x, u, r, x') \in h\} \\ \mathrm{TS}_i &= \left\{ \left((x, u), r + \gamma \max_{u' \in U} \hat{Q}_{i-1}(x', u') \right) \mid (x, u, r, x') \in h_i \right\} \quad \forall i \geq 2 \end{aligned}$$

and $\hat{Q}_i$ is trained on TS_i.

We use our implementation of Assignment 2 and adapt it to the Double Inverted Pendulum control problem. More precisely,

1. We compute N using
$$N = \left\lceil \log_\gamma \left(\frac{\epsilon}{2B_r}(1-\gamma)^2 \right) \right\rceil . \tag{11}$$
By fixing γ to 0.95, ϵ to 1[2] and B_r to 10 (maximum possible value of (3)), we get $N = 176$.

2. We create h_1 by generating and storing 3000 transitions using a random policy.

3. For all values i from 1 to N,
   (a) We train the model (XRT) using all stored transitions (h_i) to get $\hat{Q}_i$.
   (b) We play 20 episodes to evaluate $\hat{Q}_i$ and save all new transitions obtained.
   (c) We create a new training set by appending all new transitions to current transitions stored. More formally, let h_e be the transitions collected during evaluation. The new training set h_{i+1} is defined as
   $$h_{i+1} = h_i \cup h_e \tag{12}$$
   with h_1 defined in step 2 of the algorithm.

2.3 Performances and comparisons

In this section, we compare performances of our algorithms. To do this, at the end of each optimization step, we play some trajectories (50 for DDPG, 20[3] for FQI, *cf.* step 3b of the algorithm) and compute, for each trajectory i, the *discounted cumulative reward* C using
$$C_i = \sum_{t=0}^{T-1} \gamma^t r_{i,t} \tag{13}$$
where T is the (maximum) number of steps in a trajectory. We then compute the *expected discounted cumulative reward* J^μ by computing the mean of all C_i, *i.e.*
$$J^\mu = \frac{1}{n} \sum_{i=0}^{n-1} C_i \tag{14}$$
with n being the number of trajectories played. We also compute the standard deviation σ of all C_i. In the following, each graph presented shows J^μ as a thick blue line and $J^\mu \pm \sigma$ as a lighter blue area around the line.

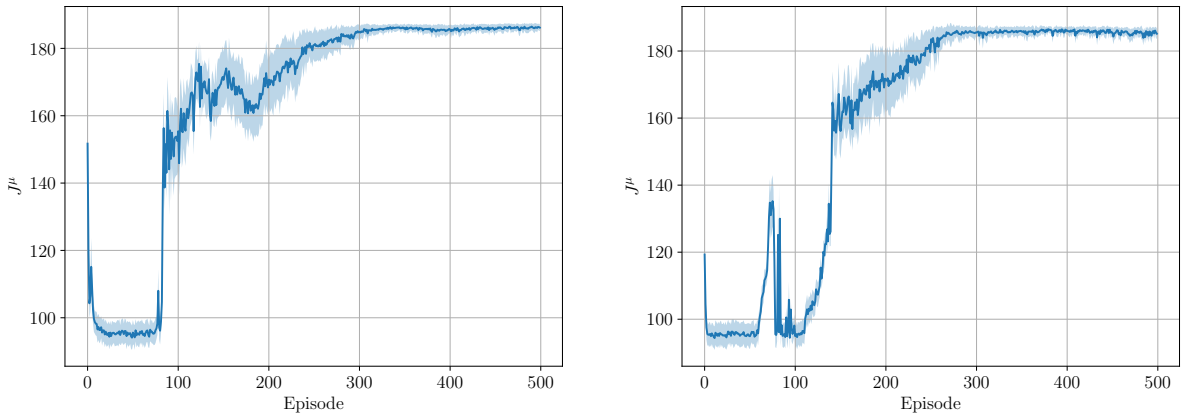
For both algorithms, we set γ to 0.95.

2.3.1 DDPG

We played 500 episodes with a maximum number of steps fixed to $T = 1000$. In the discrete action space case, we use 11 discretized actions that are linearly spaced in the interval $[-1, 1]$. We choose an odd number of actions to be sure that the action "0" (no force is applied) is available. The obtained results are shown in Figure 2.

[2]We considered this precision to be sufficient as it is numerically insignificant with respect to the cumulative rewards, *cf.* Section 2.3.

[3]We do not play 50 trajectories for FQI because it adds too much new transitions and therefore greatly increases the training time of the model.



(a) Continuous action space.

(b) Discrete action space.

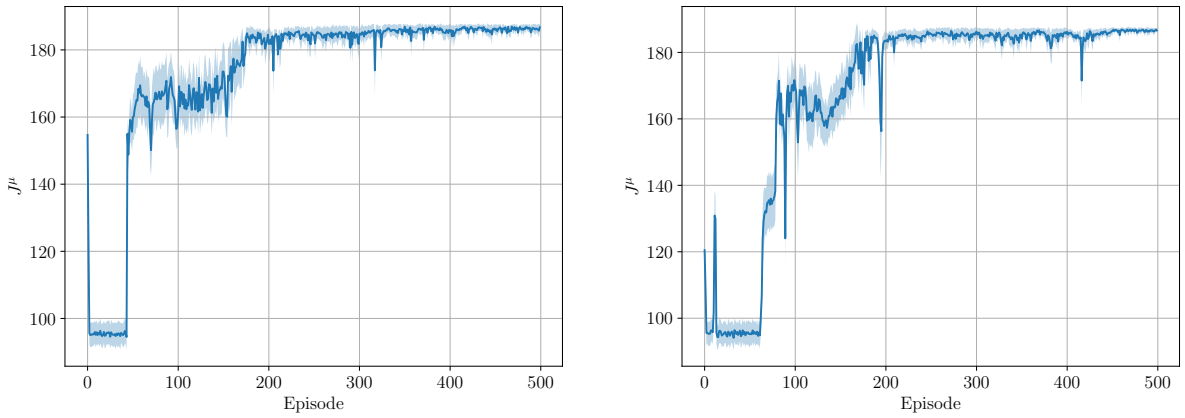
Figure 2 – Expected discounted cumulative rewards of DDPG algorithm.

We observe that, for both continuous and discrete action spaces, the algorithms converge, after approximately 300 episodes, to an expected discounted cumulative reward of approximately 190. Once the algorithms have converged, it seems to stabilize and do not evolve anymore over time.

Except at the beginning, we do not note significant differences in results obtained between continuous and discrete action spaces.

We also judge the quality of our policy by visually checking the simulation. Although the agent does not remain in equilibrium for long, it clearly improves over time, moving from an anarchic policy to one that allows it to remain in equilibrium for a short period of time.

Neural networks depths By default, we use very shallow neural networks for Actor and Critic. We try to use deeper ones to see the influence. We use, for both Actor and Critic, 8 hidden layers of 256 neurons and ELU [15] activation function. The obtained results are shown in Figure 3.



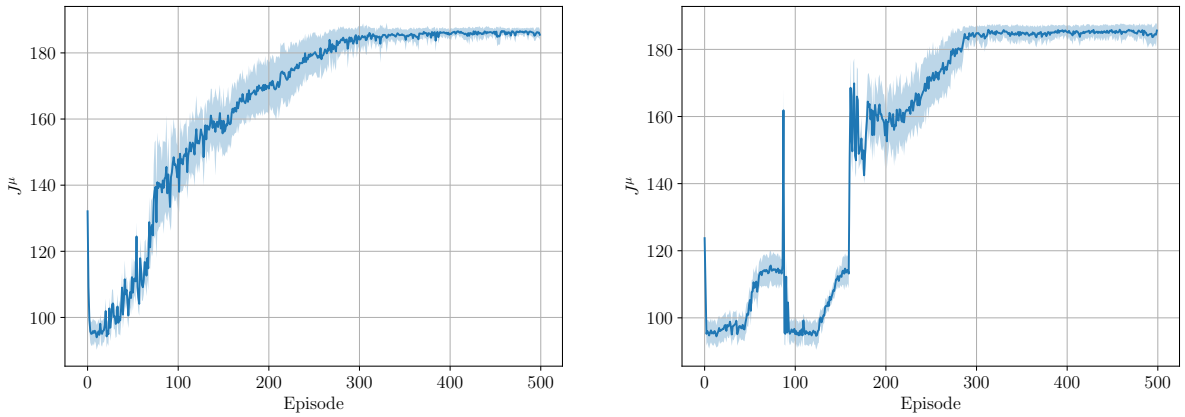
(a) Continuous action space.

(b) Discrete action space.

Figure 3 – Expected discounted cumulative rewards of DDPG algorithm using deeper neural networks.

We observe that the algorithms converge to the same J^μ, but faster: only 200 episodes is needed to reach such a value.

Discrete action space size We try to reduce the number of discretized actions in the case of a discrete action space. We choose two odd numbers (7 and 5) generated in the same way as previously. The results are shown in Figure 4.



(a) 7 discretized actions.

(b) 5 discretized actions.

Figure 4 – Expected discounted cumulative rewards of DDPG algorithm with different action space sizes.

We observe that, in both cases, the agent still learns a good policy. However, the convergence of J^μ with 7 actions is smoother, and slightly faster, than the convergence with 5 actions. It can be explained by the fact that, in view of the complexity of the problem, the fewer actions available, the more difficult it will be for the agent to learn. To verify this fact, we train the agent with 3 discretized actions (*cf.* Figure 5).

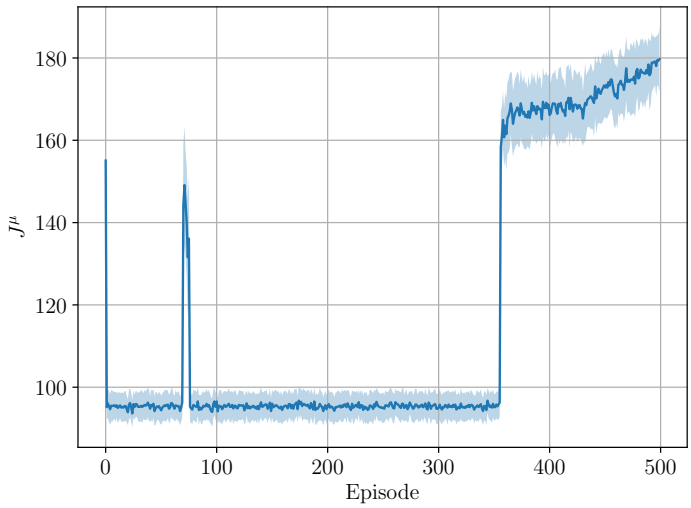


Figure 5 – Expected discounted cumulative rewards of DDPG algorithm with 3 discretized actions.

We observe that, as expected, the agent has more difficulty to learn a good policy. The values of J^μ start to improve after 350 episodes and only reaches 180 after 500 episodes.

2.3.2 FQI with XRT

We train our model (XRT with 20 estimators) until $N = 176$. We fixed the maximum number of steps to $T = 1000$. We use 11 discretized actions linearly spaced in the interval $[-1, 1]$. The results are shown in Figure 6.

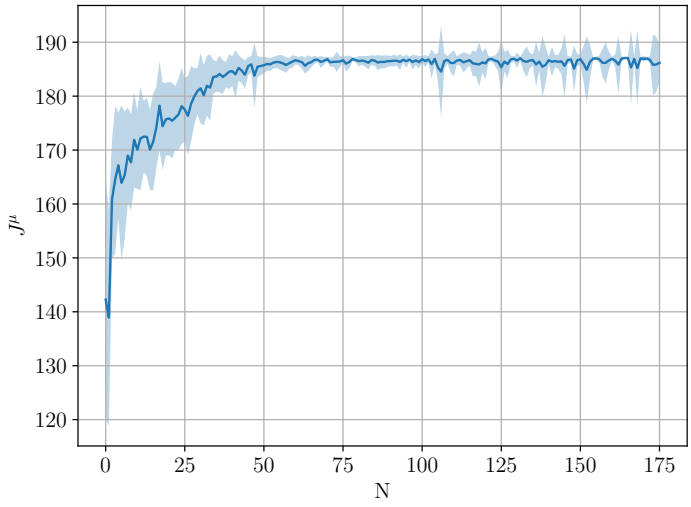


Figure 6 – Expected discounted cumulative reward of FQI with XRT algorithm.

We observe that the algorithm converge, after $N \approx 47$, to a J^μ slightly below 190, as for DDPG algorithm.

2.3.3 Comparison

All algorithms implemented (DDPG in continuous and discrete action spaces and FQI with XRT) seem to converge and learn a policy that allows to stabilize, for a short amount of time, the pendulum. While FQI with XRT seems to converge faster than DDPG, the latter is faster to train. It can be explained by the fact that FQI is trained on all transitions stored (that grows during the training. To give an idea, at the last optimization step, the transition list contains 601 237 transitions.) while DDPG is trained with random batches of the same size regardless of how many episodes have passed.

2.4 Possible improvements

Our algorithms could be improved on several aspects.

Concerning the DDPG algorithm, we could make the sampling of an action more complex. For the moment, in our implementation of the Ornstein-Uhlenbeck process, *i.e.* (10), the variance of $\mathcal{N}$ is fixed at 1. We could, for example, modify the Actor neural network so that it returns an action u and a variance σ^2 and uses the latter in the noise generation process.

We also used, in our implementation, a replay buffer that randomly samples batches of data at each optimization step. An interesting work by Hou et al. [16] has shown that working with a *Prioritized Replay Buffer* can improve results in term of training time, robustness and final performance.

Other improvements to the DDPG algorithm are also possible [17]: *Asynchronous Advantage Actor-Critic* [18] that takes advantage of multiple Actor networks trained in parallel, or *Distributed Distributional Deterministic Policy Gradients* [19] that works with multiple independent Actors, prioritized replay buffer and distributional Critic.

Concerning the FQI algorithm, we could reduce the number of transitions used to train the model and thereby reduce the training time. As the transitions generated at the beginning progressively lose their relevance as the training progresses, we could, for example, work with a cyclic buffer of fixed capacity. Thus, when the maximum capacity is reached, the oldest transitions are replaced by new ones and the dataset size remains constant.

References

[1] Slavka Jadlovska and Jan Sarnovsky. "Classical double inverted pendulum—A complex overview of a system". In: *2012 IEEE 10th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*. IEEE. 2012, pp. 103–108 (page 1).

[2] Hong-Yue Zhang. "Fault Detection, Supervision and Safety of Technical Processes 2006: A Proceedings Volume from the 6th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes". Elsevier, 2007 (page 1).

[3] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. "Deterministic policy gradient algorithms". In: *International conference on machine learning*. PMLR. 2014, pp. 387–395 (pages 1, 4).

[4] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015) (pages 1, 4, 5).

[5] Damien Ernst, Pierre Geurts, and Louis Wehenkel. "Tree-based batch mode reinforcement learning". In: *Journal of Machine Learning Research* 6 (2005), pp. 503–556 (pages 1, 2, 4).

[6] Alexander Bogdanov. "Optimal control of a double inverted pendulum on a cart". In: *Oregon Health and Science University, Tech. Rep. CSE-04-006, OGI School of Science and Engineering, Beaverton, OR* (2004) (page 2).

[7] OpenAI Spinning Up. "Deep Deterministic Policy Gradient". URL: https://spinningup.openai.com/en/latest/algorithms/ddpg.html (pages 4, 5).

[8] Chris Yoon. "Deep Deterministic Policy Gradients Explained". URL: https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b (page 4).

[9] Adrien Bolland. "Gradient-based techniques for reinforcement learning in continuous domains (finite time control)". URL: http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2021/03/rl_course_1.pdf (page 4).

[10] Adrien Bolland. "Gradient-based techniques for reinforcement learning in continuous domains (infinite time control)". URL: http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2021/03/pg-part-2.pdf (page 4).

[11] Richard S Sutton and Andrew G Barto. "Reinforcement learning: An introduction". MIT press, 2018 (page 4).

[12] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014) (page 4).

[13] Boris T Polyak and Anatoli B Juditsky. "Acceleration of stochastic approximation by averaging". In: *SIAM journal on control and optimization* 30.4 (1992), pp. 838–855 (page 4).

[14] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. "Deep reinforcement learning in large discrete action spaces". In: *arXiv preprint arXiv:1512.07679* (2015) (page 6).

[15] Djork-Arne Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015) (page 8).

[16] Yuenan Hou and Yi Zhang. “Improving DDPG via Prioritized Experience Replay”. In: *no. May* (2019) (page 11).

[17] KT2713. “Improvements”. URL: https://medium.com/robotic-arm-control-using-deep-reinforcement/improvements-b2de257617dc (page 11).

[18] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. “Asynchronous methods for deep reinforcement learning”. In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937 (page 11).

[19] Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva Tb, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. “Distributed distributional deterministic policy gradients”. In: *arXiv preprint arXiv:1804.08617* (2018) (page 11).