UNIVERSITÀ DI TRENTO
Department of Industrial Engineering

Master Degree in Mechatronics Engineering,
Electronics and Robotics

Course of Advanced optimisation-based robot control

ASSIGNMENT 03: Deep Q-Network

Surname and Name	Matriculation	Email address
Castioni Edoardo	232250	edoardo.castioni@studenti.unitn.it
Manfredi Simone	239337	simone.manfredi@studenti.unitn.it

Academic year 2022 / 2023

Contents

1 Introduction

Reinforcement learning is nowadays a spread field in both computer science and robotics. It is an automatic learning technique, which aims to create autonomous agents capable of choosing actions to be performed in order to achieve certain objectives through interaction with the environment in which they are immersed. Differently from other disciplines (e.g. optimal control), reinforcement learning tries to find the global optimum assuming the dynamic is unknown.
The goodness of an action is given by the reward, which should encourage the correct behaviors of the agent.
An important reinforcement learning method for unknown dynamics is Q-learning, an online learning algorithm briefly introduced in the following section.

1.1 Q-learning

The Q-learning is a model-free method that relies on the discretization of both the state and the action space. The core of the algorithm is the estimation of Q:

$$Q^{\pi} : S \times A \to \mathbb{R} \tag{1}$$

The action-value function Q^{π} defines the expected future discounted reward for taking action a in state s and then following policy π thereafter.
The new estimated Q is given by the previous one plus a learning rate multiplied by the difference between the estimated reward (i.e. the estimated optimal Q-function $Q^{*}(s,a)$) and the current value [5].

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha_t \Big[r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t) \Big] \tag{2}$$

In this way, the new Q is updated towards the optimal one for the future step. If the current Q is already optimal, it remains constant. The main parameters for the update are:

- α: Learning rate, defines how much we relay to the previous estimate.
- γ: Discount factor, defines how much we discount the future rewards with respect to the current one.

1.2 Deep Q-learning

When the state space becomes too large or infinite, the Q function has to be approximated instead of using a tabular function, e.g. using deep Q-learning.
The contributions of using a deep Q-learning approximation are:

- a deep convolutional neural network architecture for Q-function approximation;
- using mini-batches of random training data rather than single-step updates on the last experience;
- using older network parameters to estimate the Q-values of the next state and updating the parameters only every C steps.

Algorithm 1 Deep - Q-learning

```
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode 1, M do
    Initialize state $x_0$
    for $t = 1$, T do
        With probability $\varepsilon$ select a random action $u_t$
        otherwise select $u_t = \arg\max Q(x_t, u_t; \theta)$
        Execute action $u_t$ in the emulator and observe reward $r_t$ and state $x_{t+1}$
        Store experience $(x_t, u_t, r_t, x_{t+1})$ in D
        Set $x_{t+1} = x_t$
        Sample random mini-batch of experiences $(x_j, u_j, r_j, x_{j+1})$ from D

        $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{u'} \hat{Q}(x_{j+1}, u'; \theta^-) & \text{otherwise} \end{cases}$

        Perform a gradient descent step on $(y_j - Q(x_j, u_j; \theta))^2$ with respect to the
        weights $\theta$
        Every C steps reset $\hat{Q} = Q$                    ▷ or every few episode
    end for
    Decrease $\varepsilon$ exponentially
end for
```

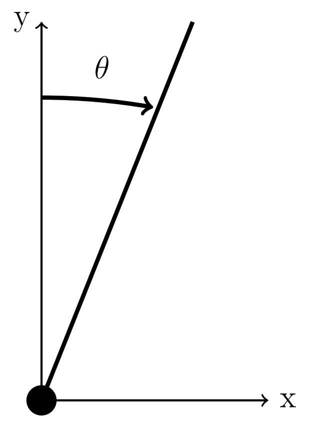
The Algorithm 1 shows the deep Q-learning steps.
First, a replay memory for the experience $(\phi_t, a_t, r_t, \phi_{t+1})$ is initialized, where the actual state, the actual action, the arrival state, and the reward received are stored. After having initialized the Q function and its weights (along with the target action-value function) and the state, an ε-greedy input is selected from the Q function, then a step is performed. Once the experience is stored in a batch, a sample is extracted from it. The future input u' is calculated as $\arg\max(Q_{target})$

so that the Q-associated target function calculated from the state-action pair is already the maximum among all the possible Q_{target}. It can be used to find the expected optimal Q function (y_i).
This is done for every experience in the `buffer_batch`, so that at the end there are n Q_{target} values and n $Q_{functions}$ to be interpolated, with $n =$ `buffer_batch`. Finally, a gradient descent step is performed in order to find the new weights that minimize the difference between the actual Q function and the optimal one. The new Q function is expected to increase, while the target function and the target weights are updated every C steps.

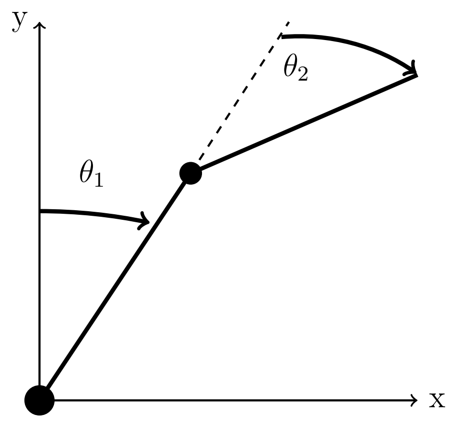
The usage of a mini-batch to store previous experiences and the idea of sampling randomly past experiences helps decorrelate the samples from the environment that otherwise can cause bias in the function approximation estimate. Furthermore, using older network parameters when estimating the Q-value for the next state in an experience and only updating the stale network parameters on discrete many-step intervals, provides a stable training target for the network function to fit, and gives it reasonable time (in the number of training samples) to do so. Consequently, the errors in the estimation are better controlled [5].
The reset of the target Q function is done every C steps so that the convergence is improved since the target is constant. If this was not the case, the Q function wouldn't be able to reach the moving Q target.
At the end of each episode the probability of choosing a random action is decreased exponentially, notice that the ratio of decreasing should be tuned accordingly to the number of episodes in order to have a good amount of "exploration". Finally, a minimum threshold of ϵ is set to 0.01.

2 Problem Statement

The goal of this work is to apply the deep Q-learning algorithm to find the optimal control policy to be fed to a single and a double pendulum in order to reach a desired configuration.
More in detail, the double pendulum has only one actuator on the first joint, and the desired configuration for both systems is the unstable equilibrium position (i.e. $\theta_i = 0$), as shown in Figure 1.



(a) *Single pendulum configuration.*



(b) *Double pendulum configuration.*

Figure 1

2.1 Environment

In reinforcement learning the system is represented with a Markov Decision Process (MDP), which is defined as a tuple of state, action, state transition probability, reward, and discount factor ($< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$). Considering the problem deterministic, the state transition $\mathcal{P}$ is uniquely defined by the current state and the current action ($x' = f(x, u)$).
The following formula reports the Euler integration scheme used for the dynamic integration to retrieve the joints' position from the accelerations found in the dynamics:

$$q = q_0 + (v + \frac{1}{2} \cdot a \cdot dt) \cdot dt \tag{3}$$

As introduced above the state-space is continuous, while the input u, used for computing the acceleration, is a finite set of values: the motor injects into the

system one value chosen by the discretized vector in the range $[-u_{max}, u_{max}]$.
Since the goal is to stabilize the pendulum around a defined configuration, it has been decided to choose a non-uniform input discretization, so that while the pendulum is approaching the vertical position the motor can act more precisely. Conversely when the pendulum is far away from the goal, the motor acts in a "coarse" way. To do that, the function "d2cu" is modified in the following way:

$$\texttt{iu} = clip(\texttt{iu}, 0, \texttt{dnu} - 1) - (\texttt{dnu} - 1)/2 \tag{4}$$

$$u = 2 * \frac{u_{max}}{\texttt{dnu}} * 2 * \frac{|\texttt{iu}|}{\texttt{dnu}} \tag{5}$$

where `dnu` is the number of discretizations for the input vector and `iu` is an arbitrary index to be converted. Notice that to have the possibility to inject zero torque, `dnu` must be odd.
Figure 2 compares two different scenarios, the blue one is a uniform discretization while the red is the one proposed in Equations 4 and 5:

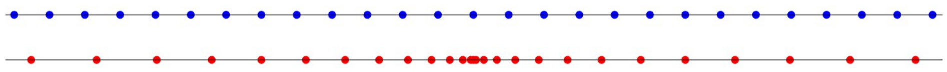


Figure 2: uniform and non uniform discretization for $\texttt{dnu} = 27$ and $u_{max} = 10$ Nm.

2.2 Cost function

As already mentioned, the Q function is computed to maximize the reward for the given problem. To better understand the mathematical model of $\mathcal{R}$, it is converted into a cost considering the following formula:

$$\max_{params} \; reward \rightarrow \min_{params} \; - cost \tag{6}$$

The cost chosen for the minimization is composed of three terms:

- `cost_q`: this is the squared sum of the joint positions (in the case of the single pendulum is only q_1)
- `cost_q_dot`: this is the squared sum of the joint velocities (in the case of the single pendulum is only $\dot{q_1}$)
- `cost_u`: this is the amplitude of the torque on the first joint

These terms are then summed with different weights to give more priority to a specific task. The final problem becomes:

$$\begin{aligned} \min_{x,u} \quad & \sigma \cdot \texttt{cost\_q} + \beta \cdot \texttt{cost\_q\_dot} + \delta \cdot \texttt{cost\_u} \\ \text{s.t.} \quad & \begin{cases} |u| < u_{max} \\ |\dot{q}| < \dot{q}_{max} \end{cases} \end{aligned} \tag{7}$$

Notice that the wights σ, β, δ are properly tuned for the single and double pendulum. More precisely in the second case, being more complex, β and δ were reduced to majorly reward the vertical position of the system.

2.3 Neural Network

The Q function approximation is managed by a Neural Network, implemented using the "keras" library in python [4]. In particular, the NN receives as input a state of the environment and the joint torque and returns a single value for Q.

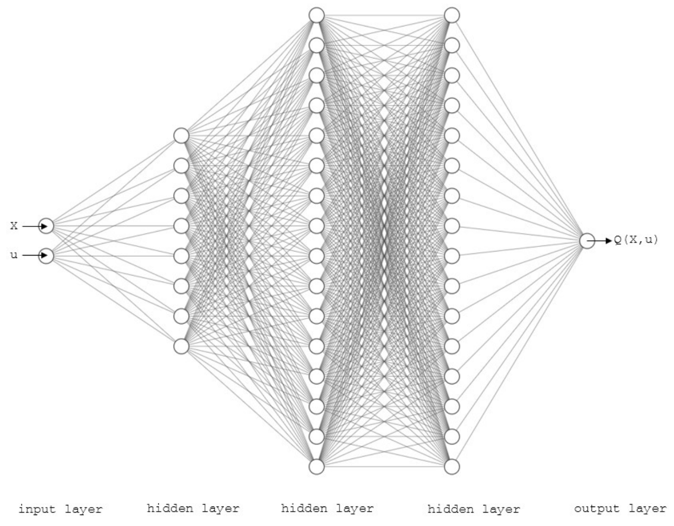


Figure 3: Neural Network synthetic representation

The number of layers and the nodes changes according to the performed tests: for the single pendulum, a simpler NN is chosen, while for the double pendulum, a more articulated structure is used to manage the more complex system. Each layer is dense and the ReLu activation function is used.

3 Tests and Results

In this section, the main tests and results are reported. In order to accomplish the goal of the project many attempts were performed. In particular, firstly the hyper-parameters of the "DQN" were tuned in order to achieve the swing-up maneuver. Then the same parameters have been modified to explore their effect on the overall result.
More in detail, attention has been paid to:

- Number of episode
- Size of mini-batch
- Maximum torque and its number of discretizations
- Neural network architecture
- Cost function weights

3.1 Single pendulum

3.1.1 Test with $\mathbf{u_{max}} = 1.5$ (Nm)

The parameters used for this test are those listed in Tables 1 and 2. With this configuration, the task is well achieved, with little oscillation and good precision in reaching the desired angle.
The cost used for this training is the following:

$$Cost = 1 \cdot \texttt{cost\_q} + 0.1 \cdot \texttt{cost\_q\_dot} + 0.001 \cdot \texttt{cost\_u} \tag{8}$$

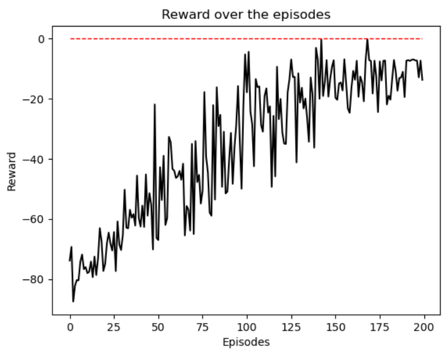
The different weights of the three components of the cost are mainly chosen in order to give more priority to the position with respect to the joint velocity or torque.
From Figures 4a and 4b is possible to see the increase of the reward during the training phase as a function of the episodes. In particular, the algorithm converges in around 130 episodes.

Table 1: Benchmark training parameters.

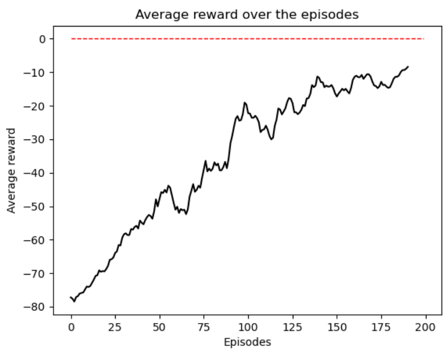
parameter	value
u_{max}	1.5 (Nm)
v_{max}	5 (rad/s)
dnu	21
NUM_EPISODE	200
LENGTH_EPISODE	100
BUFFER_SIZE	20000
MINI_BATCH_SIZE	128
C_UPDATE	300
DISCOUNT	0.99
LEARNING_RATE	0.001
ϵ DECAY	0.02
TRAINING_TIME	103 (m)

Table 2: Deep Neural Network.

layer	nodes
input	3
hidden_1	16
hidden_2	32
hidden_3	64
hidden_4	64
output	1

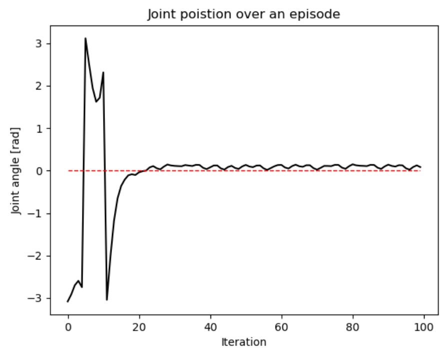


(a) *reward of the algorithm over the episodes.*

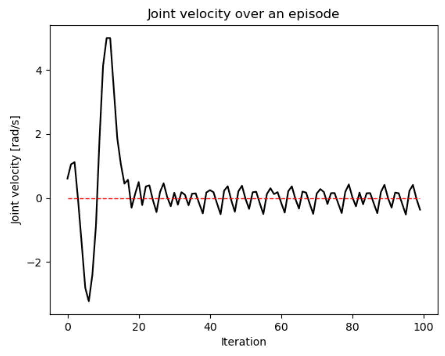


(b) *moving mean of the reward over a window of 10 samples.*

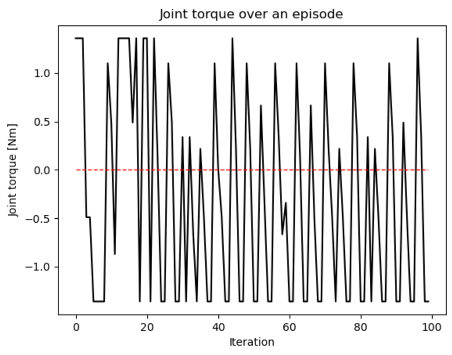
Figure 4: Reward trend during the training phase



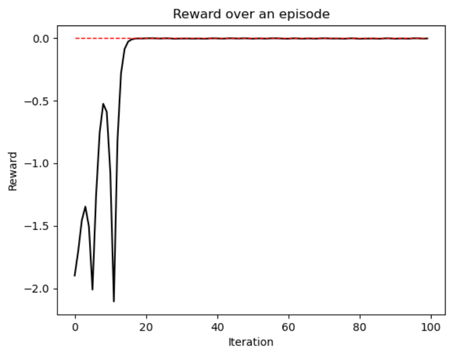
(a) *joint position*



(b) *joint velocity*



(c) *joint torque*

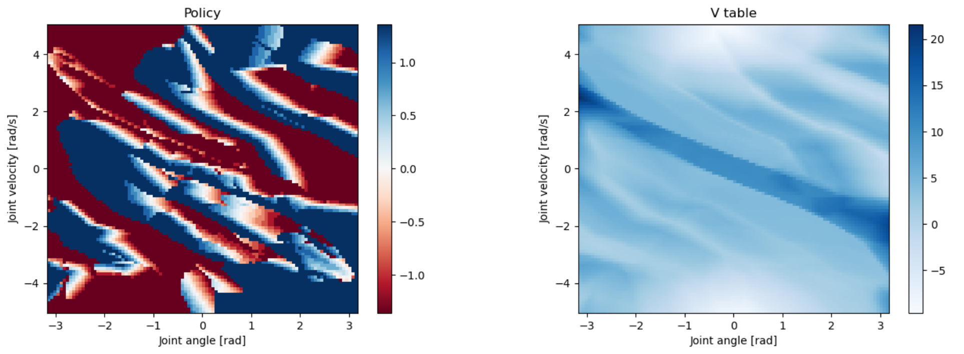


(d) *reward*

Figure 5: Simulation results

Once the model is trained, a simulation is performed in order to test its performance. The most important result, in Figure 5, is the convergence of the reward. This means that the goal is reached and, in fact, it is confirmed by the position plot. It has to be noticed that the joint angle doesn't converge directly to the desired configuration, but it initially oscillates: this is due to the fact that the maximum torque is quite low, so the pendulum exploits the gravity in order to increase its potential energy (this is clearer in the video posted in [2]).
Another interesting result is visible from the torque plot. This basically reports the best input to achieve the desired configuration trying to minimize the motor effort. This task is not well reached since the torque tends to saturate often to the maximum value, but being the u_{max} very low it is the only way to perform the swing-up maneuver. Notice also that the torque doesn't converge to zero, but it oscillates constantly in order to keep the pendulum in the vertical position.
Finally, the policy table and the V-table are computed from the trained Q. The policy table shows clearly a symmetric behavior. This is expected, being the dynamical model of the pendulum symmetric. From the V-table instead, it is possible

to retrieve the best reward as a function of the state, this is why in the central area (corresponding to small joint angles) the values are higher with respect to the other zones of the map.



(a) *policy table.* (b) *value table.*

Figure 6: Color-maps of policy and V-table given the trained Q-function

3.1.2 Test with different numbers of torque discretizations (dnu)

In order to analyze the effect of the number of discretizations on the torque, a comparison based on five training is reported.
In particular, a general set-up of the hyper-parameters was chosen (Tables 3 and 4), and only the value of "dnu" was changed between 3, 17, 35, 51, 91. A graphic visualization of the resultant torque vectors is reported in Figure 7.

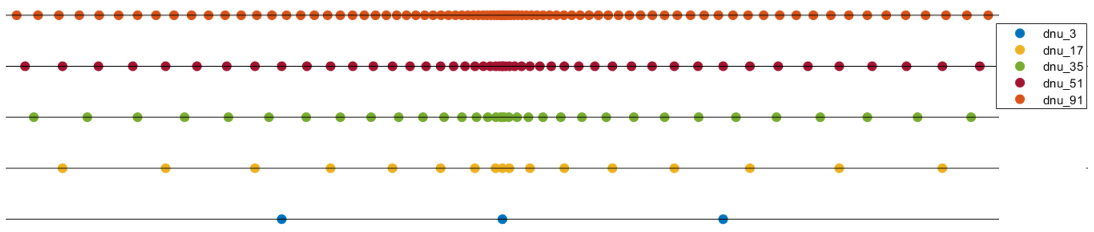
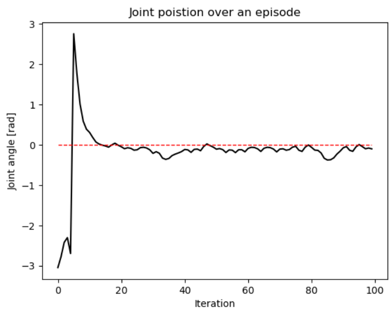
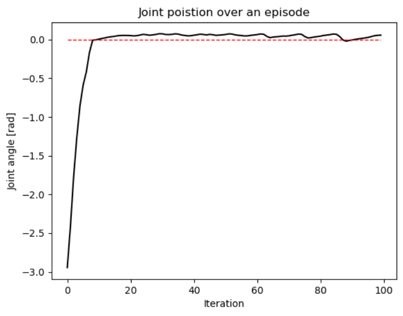


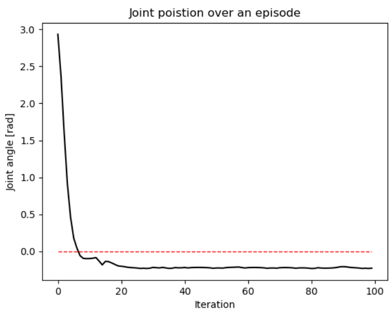
Figure 7: Vectors of discretized torque



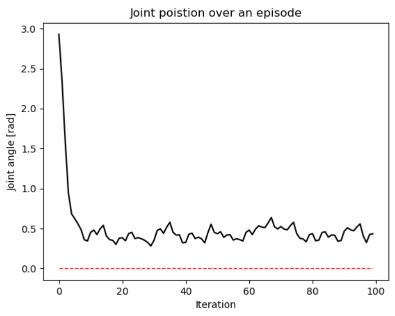
(a) *joint position (dnu* $= 3$*)*



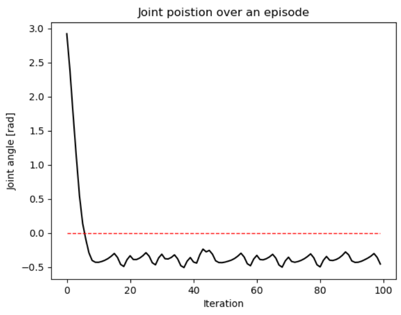
(b) *joint position (dnu* $= 17$*)*



(c) *joint position (dnu* $= 35$*)*



(d) *joint position (dnu* $= 51$*)*



(e) *joint position (dnu* $= 91$*)*

Figure 8: Joint position trend with different dnu

From Figure 8 it is possible to see that in general is better to keep a low dnu, in fact with 3 or 17 values of discretizations the joint positioning error is negligible. Conversely, by increasing this parameter the error grows progressively.
This is an interesting result because it seems counterintuitive. In fact, one might think that with a larger dnu the system can choose more precisely the correct torque to be fed to the motor.
This, however, does not happen probably because having too many choices the algorithm cannot retrieve the actual optimal policy from Q.

Maybe this aspect can be mitigated by increasing the number of episodes in order to make the NN able to explore enough all the possible conditions.
Furthermore, if the same tests are replicated using a smaller value of u_{max}, the oscillations are smaller for a higher number of discretizations. This may be due to the fact that, in the case of higher u_{max}, the smallest possible input fed into the controller is higher than the smallest possible input with a smaller u_{max}, so the pendulum is pushed more in the proximity of the target position. This behavior can be seen in the figures of the git repository [2].

Table 3: Training parameters.

parameter	value
u_{max}	5 (Nm)
v_{max}	5 (rad/s)
dnu	[3,17,35,51,91]
NUM_EPISODE	150
LENGTH_EPISODE	100
BUFFER_SIZE	20000
MINI_BATCH_SIZE	64
C_UPDATE	300
DISCOUNT	0.99
LEARNING_RATE	0.001
ϵ DECAY	0.03

Table 4: Deep Neural Network.

layer	nodes
input	3
hidden_1	16
hidden_2	32
hidden_3	64
hidden_4	64
output	1

The training time for each of the five tests was around 45 minutes.

3.1.3 Test with a mini-batch of 16 experiences

Another test was performed in order to analyze whether the algorithm was able to succeed in the goal even with a small mini-batch.
From the results, it is possible to see that indeed the pendulum reached the top vertical position but not very accurately (even if the torque was increased up to 5 Nm). In fact, looking at Figure 9a (or to the provided video) the joint position does not reach exactly zero but oscillates around 0.3 rad (≈ 17.2 °). This is probably due to the fact that with a too-small mini-batch, the algorithm cannot perform generalized training and has not sufficient data to rely on.
Another consequence is evident in the V-table. In fact differently from the one exposed in Section 3.1.1, the one obtained in this test has a wider central area. This worsens the performances because it is assigned a good reward even to combinations of states that are not the ones referred to the top vertical position.

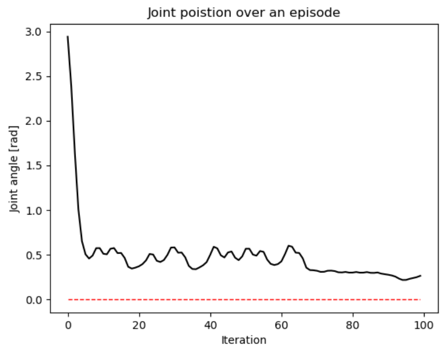
Below are reported the parameters used in the training and the simulation results.

Table 5: Training parameters.

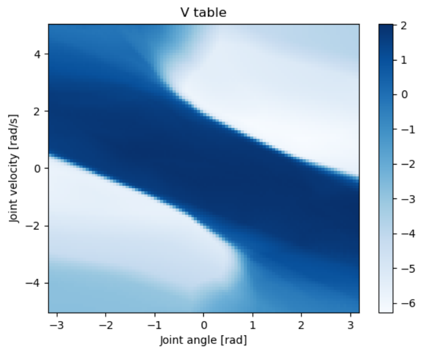
parameter	value
u_{max}	5 (Nm)
v_{max}	5 (rad/s)
dnu	21
NUM_EPISODE	150
LENGTH_EPISODE	100
BUFFER_SIZE	20000
MINI_BATCH_SIZE	16
C_UPDATE	300
DISCOUNT	0.99
LEARNING_RATE	0.001
ϵ DECAY	0.03
TRAINING_TIME	18 (m)

Table 6: Deep Neural Network.

layer	nodes
input	3
hidden_1	16
hidden_2	32
hidden_3	64
hidden_4	64
output	1



(a) *joint position.*



(b) *value table*

Figure 9: Results of the test with a small mini batch

As a final consideration, the training time is largely reduced having fewer episodes and a smaller mini-batch.

3.1.4 Other tests

During the development of the project, several other tests were performed to explore different aspects of the algorithm and of its hyper-parameters. In order to keep the report synthetic not all of them are reported but their results are available on the GitHub repository [2] or in the zip folder of the project.

3.2 Double Pendulum

The double pendulum was the most challenging part of the project. Initially, several tests were performed with the first NN used for the single pendulum. This attempt unfortunately did not give the desired results: in fact, the pendulum approached the vertical position but it fell down in a few instants.
Another problem was that the pendulum would have reached the desired configuration only if it started from a specific configuration so the trained model was not completely general.
In the end, a combination of changes brought to the goal. More in detail the cost was modified as follows:

$$Cost = 1 \cdot \text{cost\_q} + 1e^{-3} \cdot \text{cost\_q\_dot} + 1e^{-8} \cdot \text{cost\_u} \tag{9}$$

This choice aimed to give much less importance to the torque magnitude and to the joint velocity. This is equivalent to increasing the priority to the position task which is the main goal of the project.
The hyper-parameters of the algorithm and the used NN are shown in Tables 7 and 8:

Table 7: double pendulum training parameters.

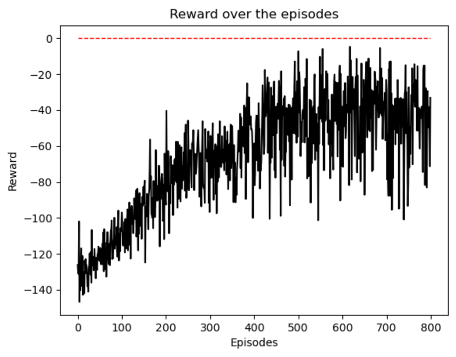
parameter	value
u_{max}	8 (Nm)
v_{max}	6 (rad/s)
dnu	27
NUM_EPISODE	800
LENGTH_EPISODE	120
BUFFER_SIZE	2000
MINI_BATCH_SIZE	64
C_UPDATE	150
DISCOUNT	0.99
LEARNING_RATE	0.001
ϵ DECAY	0.008
TRAINING_TIME	370 (m)

Table 8: Deep Neural Network.

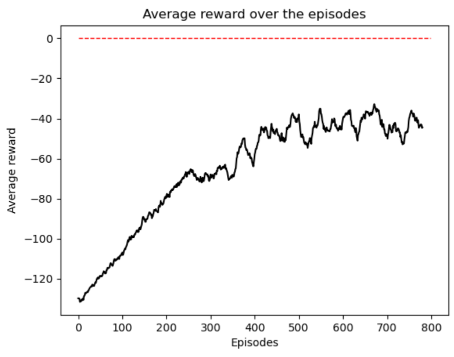
(layer	nodes
input	5
hidden_1	16
hidden_2	32
hidden_3	32
hidden_4	64
hidden_5	128
hidden_6	64
output	1

As it's possible to see from the two tables the whole algorithm was "upgraded". The number of layers and nodes was increased to have a better approximation of the Q function. In fact, by adding a new joint on the system the number of possible combinations between the states increases drastically. Then to give more power to the system the maximum torque and the maximum joint velocities were increased.

Finally, the number of episodes was increased up to 800, in order to better train the model. These modifications cause inevitably a larger training time which is around 6 hours.
The reward over the episodes is reported in Figure 10a, where is visible that the algorithm converges in around 600 episodes. Considering the same plot it is evident that the reward is highly episode-dependent (”noisy”) in the sense that from episode to episode it may change a lot regardless of the training phase, this is probably due to the large complexity of the problem.



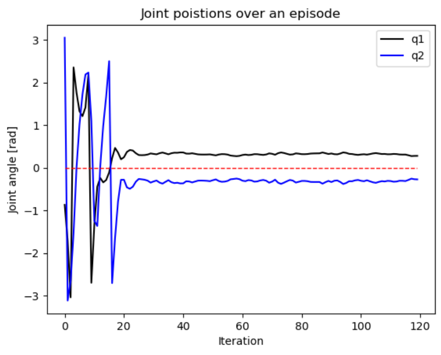
(a) *reward of the algorithm over the episodes.*



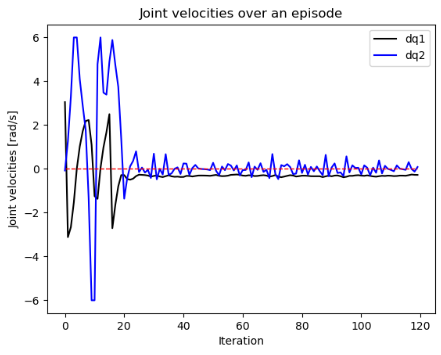
(b) *moving mean of the reward over a window of 20 samples.*

Figure 10: Reward trend during the training phase

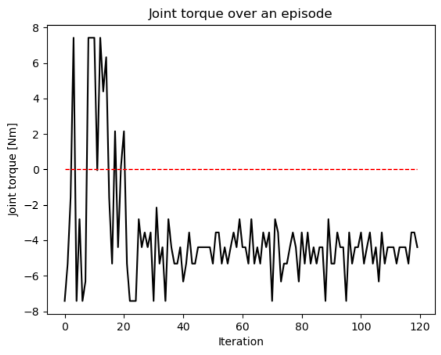
Once the model was trained a simulation was performed and its results are shown in Figure 11.
Firstly from the reward plot is possible to see that the system converges to zero in around 40 iterations. Similarly to what is exposed in Section 3.1.1 the double pendulum has a first phase of oscillations which are used to reach the vertical position, even in this case it is easier to look at the video in order to better understand the behavior of the system.
From the joints positions plot it is evident that the angles are not exactly zero after the initial transitory part: in fact, they are constantly biased with respect to the vertical axis. This is due to the fact that the second joint is not actuated so, in order to keep the second link of the pendulum vertically, the first joint has to move in the opposite way. This behavior, caused by the coupled dynamics of the system, is clearly visible in the second part of the plot where there is a sort of symmetric pattern with respect to the zero angle.
A final confirmation of this behavior is the velocity of the joints: while the first one moves very smoothly and in a controlled way, the second one responds with an abrupt behavior being an idle link.



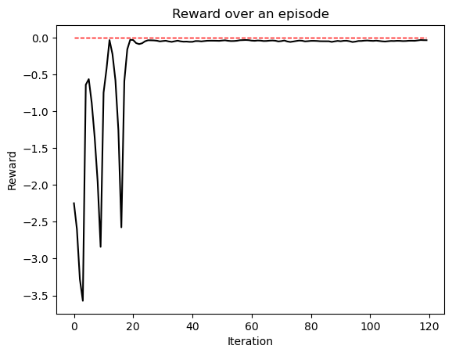
(a) *joints positions*



(b) *joints velocities*



(c) *joint torque*



(d) *reward*

Figure 11: Simulation results

4 Conclusion and improvements

In this report were explained the most important step and choices done to apply the DQN to the single and double-pendulum. In particular, the main effects of the hyper-parameters of the algorithm were explored.
Even if in both cases the system managed to realize the swing-up maneuver, different improvements may be performed. First of all, in the double pendulum, the training time is very long. This is probably due to the non-perfect tuning of the hyper-parameters. Surely performing other tests and changing a little bit the parameters it is possible to improve the whole performance of the algorithm.
Another consideration may be done on the architecture of the neural network. More in detail the one used in this project took as input a pair of states and input and gave as output a unique real value. As a possible improvement it can be interesting to analyze the performance of the algorithm if only the state is given as input and a vector of n values is returned, being n the number of possible joint torques.
An additional test may be done on the robustness of the model under external tangential forces applied to the pendulum. While in the case of the single one, the problem may be solved by increasing the motor torque, for the double pendulum it is not so trivial.
As a final possible improvement, the algorithm can be tested on a triple pendulum which increases drastically the difficulty. This idea comes from a video [6] found during the initial research phase of the project development.

5 Provided material

In the sent folder is present the full code developed for the project. In particular, there are python files written to implement the algorithm itself and a couple of Matlab scripts used to generate some figures.
Furthermore, in the "MODELS" folder there are all the performed tests and the corresponding results with figures and video. The ones cited in the report are in the folder "REPORTED_TESTS", while all the others are present in "OTHER_TESTS". The video cited in the report are also contained in the "VIDEO" folder.
The same material is present here: GitHub repository.

References

[1] *Deep Q Networks (DQN) in Python From Scratch by Using OpenAI Gym and TensorFlow- Reinforcement Learning Tutorial.* URL: https://aleksandarhaber.com/deep-q-networks-dqn-in-python-from-scratch-by-using-openai-gym-and-tensorflow-reinforcement-learning-tutorial/.

[2] *DQN PENDULUM PROJECT, GitHub repository developed by Castioni Edoardo and Manfredi Simone.* URL: https://github.com/edocas01/DQN_PENDULUM_PROJECT.

[3] "Human-level control through deep reinforcement learning". In: (2015).

[4] *Keras.* URL: https://keras.io/api/layers/core_layers/input/.

[5] James MacGlashan Melrose Roderick and Stefanie Tellex. "Implementing the Deep Q-Network". In: (2017).

[6] *Swing-up and Control of Linear Triple Inverted Pendulum , Stepan Ozana.* URL: https://www.youtube.com/watch?v=meMWfva-Jio.