Master of Science in Mechatronics Engineering

Deep Q-Learning

Professor:
Prof. Andrea Del Prete

Students:
Cardone Andrea 224836

Department of Industrial Engineering
Academic Year 2021 - 2022

1 Hybrid pendulum environment

The pendulum state is constituted by a vector containing the angle q in the first position and the velocity v in the second. The angle q is continuous between $[-\pi, \pi]$, while velocity v is continuous between [-5,5], it is clipped if larger.
The torque at the joint is discretized with 11 values, therefore the joint torque is discretized with 11 steps between +5 and -5.
The function `step(u)` is made by three steps:

1. Compute the continuous torque from the discretized one with the function `d2cu(u)`
2. Call the `dyanmics(x,u)` function that simulates the dynamics of the system.
3. Compute the cost/reward, since the state is continuous the reward is given if $|q| < 0.1$ and $|v| < 0.1$.

The function `reset()` reset the state to a random value between $+\pi$ and $-\pi$ for q and random value between $+5$ and -5 for v.

2 DQL algorithm

2.1 Neural network

The neural network is constituted by the input layer with size equal to the length of the state. Then there are three layer, respectively made by 32, 64 and 512 neurons. This configuration, found after some tentatives, should ensure a good representation of the Q function. The output layer is constituted by eleven neurons, which represent the Q function for every action taken, at the state x. The loss function is the mean squared error and the chosen optimizer is Adam.

2.2 Algorithm

In the following section is resumed the DQL algorithm implementation following the provided documentation. The `done_condition` is simply the maximum number of steps allowed for each episode.

```
Initialize environment
Initialize $Q(\theta)$
Initialize $Q_{target}(\theta_t)$
Set $\theta_t = \theta$
Initialize replay_memory
Initialize secondary elements
for episode in 0:nepisodes do
    if total_steps > max_exploration then
        Increment training_episodes
    end if
    Reset environment
    Initialize secondary elements
    while not done do
        action = get_action(Q, state, epsilon)            ▷ $\epsilon - greedy\,action$
        state_next, cost = env.step(action)                 ▷ simulate step
        Update secondary elements
        Save [state, action, cost, state_next] in replay_memory
        state = state_next                                   ▷ update state
        if update_Q condition then
            Initialize batch_memory
            Get random indexes i from replay_memory
            Fill batch_memory with replay_memory[i]
            Compute target
            y[j] = cost_batch[j]                 ▷ if episode terminates at step j+1
            y[j] = cost_batch[j] + $\gamma \min_{u'} Q_{target}$(state_next_batch, $\theta_t$)
            masks(action[j])          ▷ create masks to select the correct Q_value
            q_value = $Q$(state_batch, $\theta$)
            q_action = q_value$\cdot$ masks
            loss = (y − q_action)$^2$
            Compute SGD with respect to weights $\theta_t$
        end if
        if update_$Q_{target}$ condition then
            Set $\theta_t = \theta$
        end if
        if total_steps > max_mem_size then
            Delete first row of replay_memory
        end if
        Update done condition
    end while
    Update $\epsilon$ exploration probability
    Update secondary elements
```

2.3 Parameters

The main difficulty encountered in making the algorithm work correctly concerns the choice of the parameters values. The parameters used will be reported below and the choice of their value will be explained.

- **n_episodes**: the number of episode is important to give enough trials to the algorithm learning. It is strictly correlated to the dimension of the state and to the learning rate of the algorithm. For the single pendulum `nepisodes` is set to 5000.
- **max_episode_length**: the episode length is very important because it can heavily affect the final result, in fact a too low number of steps could make it impossible for the agent to find some long-term strategies to reach the goal (such as swinging the pendulum repeatedly), even up to never being able to reach the goal . On the other hand, a large number of steps considerably increases the total learning time, with the same number of episodes.
  So for the single pendulum this value was chosen equal to 100.
- **gamma**: the discount factor is important to ensure that the agent prefers short-term or long-term rewards, given that we are interested in performing the swing-up maneuver and keeping the pendulum in balance, the gamma value equal to 0.98 has been chosen.
- **exp_decreasing_decay**: this value is important to ensure a good matching between the number of episodes and the `epsilon` value for the epsilon greedy policy improvement. With 5000 episodes a value of 0.001 should ensure a good trade-off between exploring and exploiting.
- **update_Q**: this parameter indicates every how many steps the SGD is performed. This is useful for speeding up the total time of the algorithm, as the last step is added in the replay memory, and between one step and another the elements that can be chosen from the replay memory are practically identical. So it was chosen to perform the SGD every 4 steps.
- **update_Q_t**: this parameter indicates how many steps the target network is updated. Too low a value would make learning ineffective and make it very unstable, while too high a value would make it very slow. After a few tests, the value of 10000 was found to be quite effective.
- **batch_size**: this value indicates how many samples are taken from the replay memory to perform the SGD, it has been chosen equal to 32, in order to have enough samples to see enough simulations, but at the same time not to make the SGD too slow.
- **max_exploration**: this parameter is used to indicate how many steps to take by choosing a random action before starting the epsilon greedy policy, this is used to fill the replay memory with random experiences at the beginning. This ensures good initial exploration. In this case I tried to fill all the replay memory, so this parameter is equal to 10000.
- **learning_rate**: adjusting this parameter well is important to obtain a good performance of the neural network. Too high a value could cause the network to

diverge, while too low a value would make learning too slow. So this parameter has been set with a value equal to 0.001.

- **replay_memory_size**: the correct size of the replay memory is important because if it is too small the agent is based exclusively on recent experiment samples and this could cause overfitting or instability. On the contrary, if it is too large it can cause a great slowdown in learning, as the samples are taken with a random index inside the memory, in fact it seems that too large a memory makes the epsilon greedy policy less effective. A replay memory size equal to 10000 was adopted.

2.4 Results

Below is reported the cost-to-go and average cost-to-go for each episode, with the algorithm parameters set as described above, as well as the value function obtained.

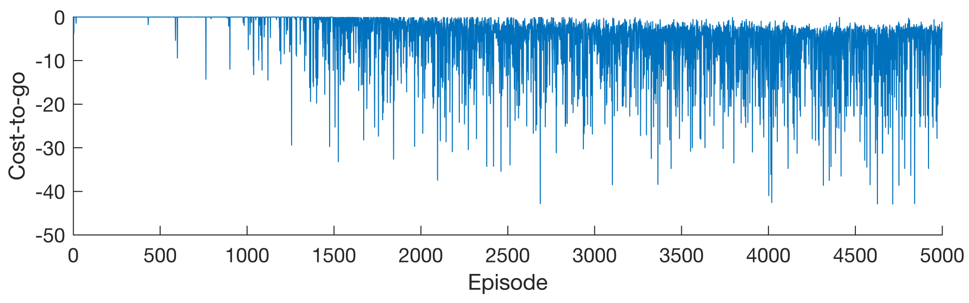

Figure 1: Cost-to-go for each episode

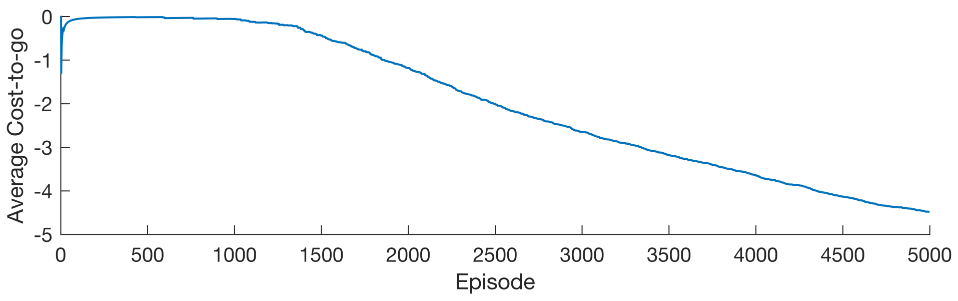

Figure 2: Average Cost-to-go

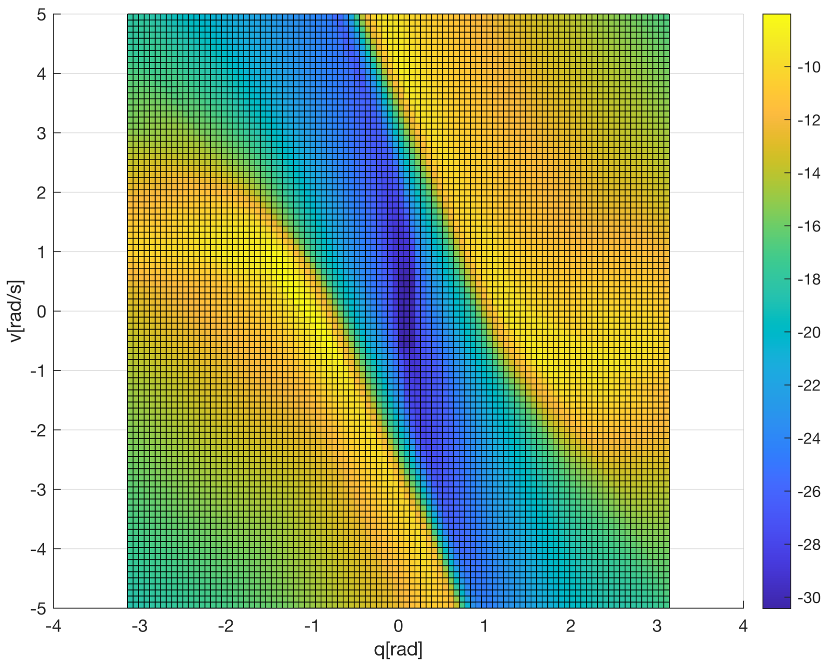

Figure 3: Value function

2.5 Considerations

The agent learns successfully to achieve the task, in figure 1 we can see that for some episodes the agent achieve a very low cost, other times the cost is higher. This is due to the fact that every episode the pendulum starts at a random state. Figure 2 shows better the learning progress: the initial exploring phase and then the following exploiting phase. As we can see from figure 2 after 5000 episodes the learning is still not completely finished, as we expect the average cost to go to stabilize on a certain value. As we can see the value function obtained with DQL algorithm is qualitatively pretty similar to that obtained with simple Q-Learning algorithm.
Despite this we obtained a very good approximation of the Q function in continuous state space, in addition the agent successfully achieve the swing-up maneuver from a random position and it is able to stabilize the pendulum in the goal position.

The main problem encountered for this algorithm is the time need for the whole training process. Obviously this problem is closely related to the computational power of the machine, but there are also other parameters that has a big influence, such as the neural network complexity, with the same number of episodes and steps per episode.
This problem was the main reason why it was not possible to carry out numerous tests in order to observe the algorithm performance variations as the parameters changed.
It must be said that the parameters reported have been chosen after some tests that have shown little if any results.

Another interesting consideration concerns the choice of the cost/reward function. For the pendulum it was chosen to give the reward only when it reaches the goal,

so for many episodes the agent does not receive any information about the environment, until after numerous explorations he manages to find the goal state. Another alternative is to give a cost to each state, such as a quadratic cost of position and velocity. Obviously in the second case the algorithm is faster, as in a few hundred episodes it reaches good performances, as we can see in figure 4.In this case the cost is defined as $cost := q^2 + 0.1v^2 + 0.01u^2$. In addition it require less replay memory and it is less sensitive to the parameters discussed before.
This is interesting because it allows us to observe the difference between fully supervised learning and semi-supervised learning. In particular, the difficulty, on the one hand, of elaborating a function that describes the cost is, on the other hand, compensated by the large number of exploration steps required.

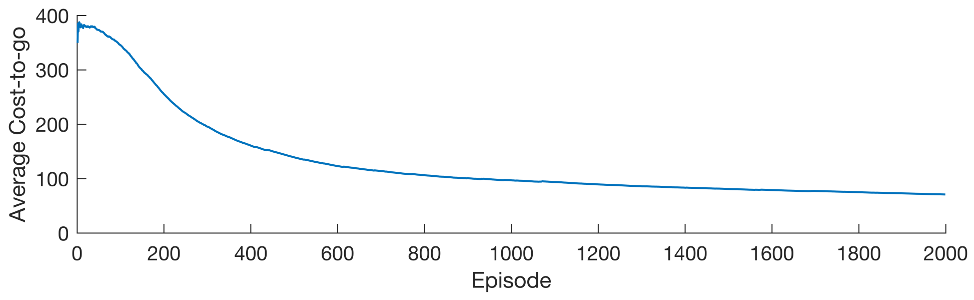

Figure 4: Average Cost-to-go fully supervised learning

2.6 Comparisons

The algorithm was run by varying the following parameters: learning rate of 0.05, a batch size of 64 samples and a discount factor equal to 0.9. The following images show the average cost-to-go and the value function obtained.

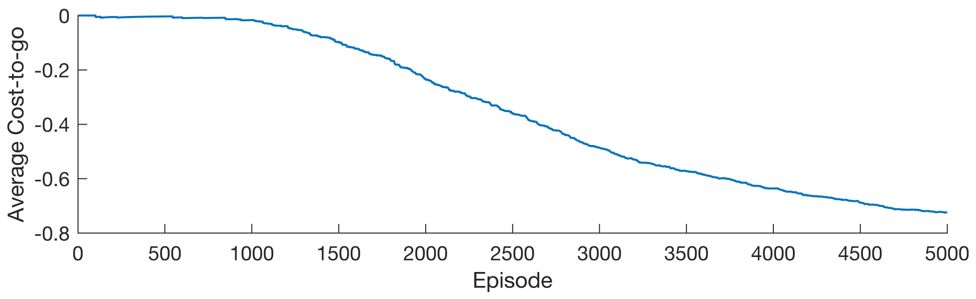

Figure 5: Average Cost-to-go

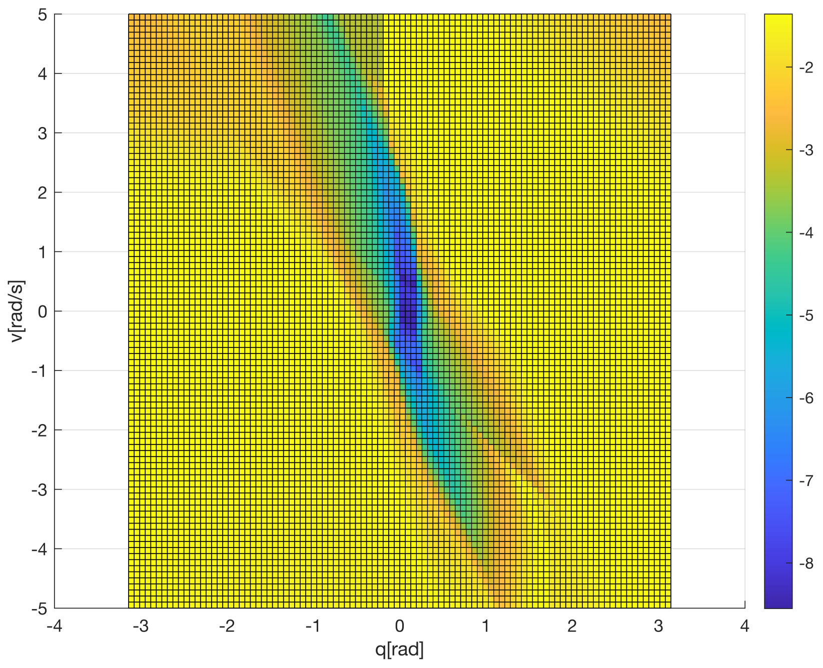

Figure 6: Value function

As we can see from the value function, the environment seems to be less explored, this is probably due to the smaller discount factor rather than the difference of the batch size or the learning rate of the network. This can be explained considering that with a lower discount factor, the agent is less encouraged to take long-term strategies. This is also suggested in the following image, where are shown the averaged cumulative sum of the terminal costs of the first try and the second try. In this image we can see that in the second test the agent reach less frequently the goal state in the final episodes.

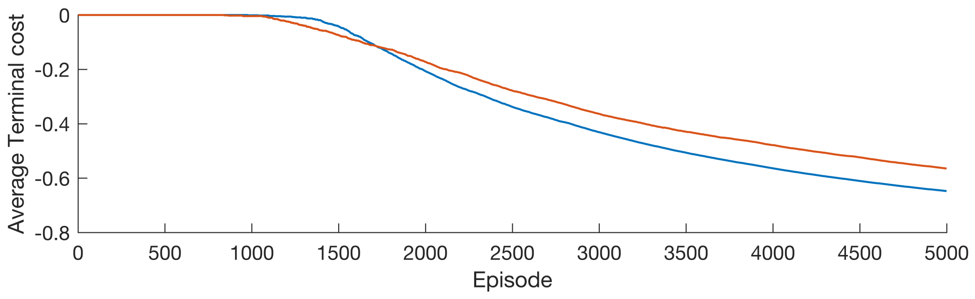

Figure 7: Average terminal cost, blue first test, red second test

Reducing a lot the replay memory size and the rate at which we update Q_{target} can completely reduce the performance of the algorithm, in the following image it is shown the cost-to-go obtained for each episode with a replay memory size of 2000, Q_{target} is updated every 1000 steps and the discount factor is equal to 0.98, as we can see the agent isn't able to learn effectively.

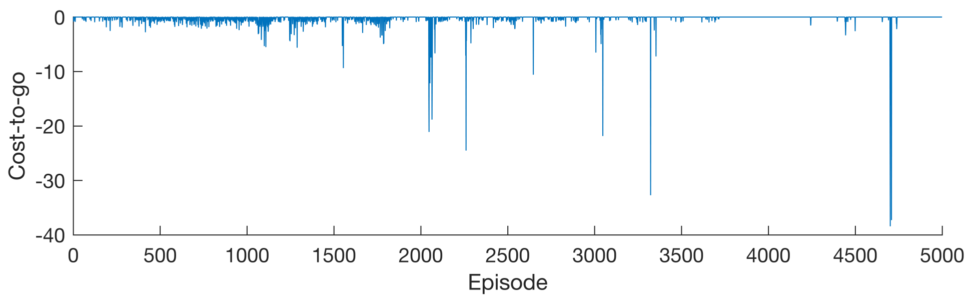

Figure 8: Cost-to-go

3 Double pendulum implementation

First of all the number of steps per episode is increased to 200 to give enough time to the agent to do the swing-up. In addition the actuator torque is increased to a maximum of 15 as well as the maximum joint velocity to 8 in the simulation. The purpose of these changes is to try to reduce the time for the swing-up maneuver. The main difficulty concerns the fact that the agent manages to reach the goal very rarely, thus requiring a large number of episodes dedicated to exploration. Since this would take a long time, the idea adopted is to make the goal limits less restrictive. So the goal is defined as: $|q_1| < 0.3, |q_2| < 0.5, |v_1| < 0.3, |v_2| < 0.5$. In this test the learning rate is set to 0.005, the batch size is equal to 64 and the discount factor is equal to 0.99. In this case it is used a neural network with two layers, respectively of 32 and 512 neurons, with the aim of making the network lighter and therefore the computation time of the training lower. The remaining parameters are the same used in the test for the single pendulum. In the following images are shown the results.

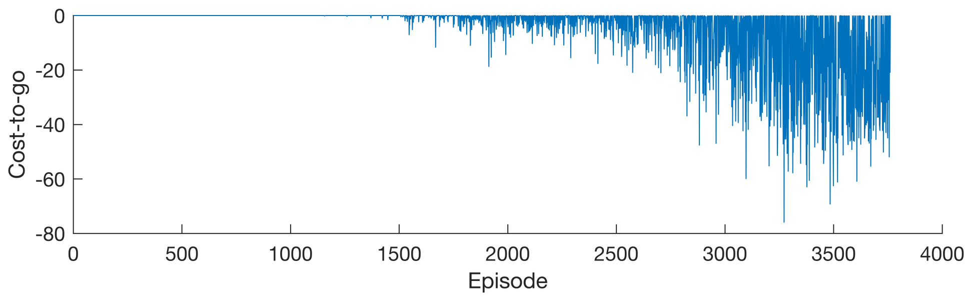

Figure 9: Cost-to-go double pendulum test

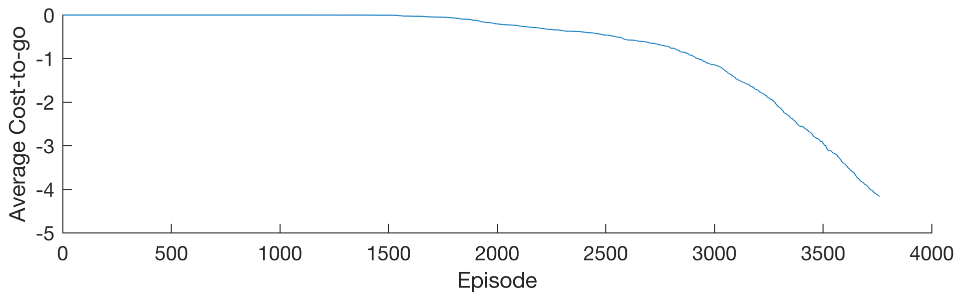

Figure 10: Average Cost-to-go double pendulum test

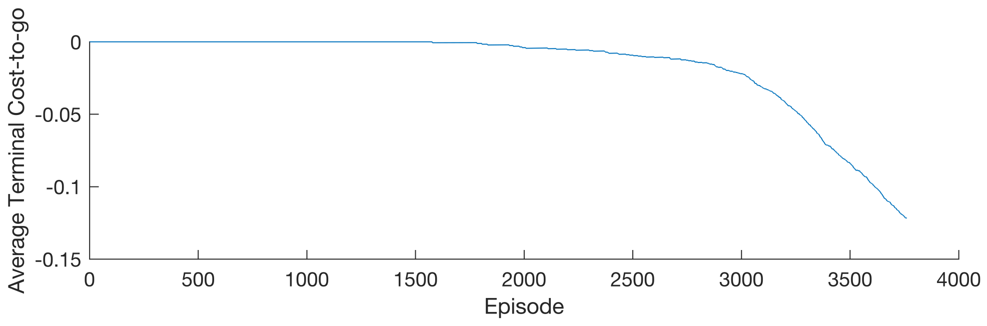

Figure 11: Average Cost-to-go double pendulum test

As we can see from this images the training works well, but the algorithm needs a greater number of episodes, as we can see how after just under 4000 episodes it is still in an initial learning phase. As expected in this case the algorithm needs much more episodes than the single pendulum case, we can see how only after 3000 episodes the agent starts to manage to reach the goal state at the end of the episodes.