Third Assignment ORC
DQN Network

Bertelli Davide
mat. 223718
davide.bertelli-1@studenti.unitn.it

Hueller Jhonny
mat. 221254
jhonny.hueller@studenti.unitn.it

1 Abstract

The third assignment revolved around the task to implement a deep Q-Learning algorithm able to proficiently tackle the task of balancing a pendulum. This work focuses on the adaptation of the work in [1] to a simpler environment, along with many suggestion from [2] for implementing the details. The resulting deep neural network is able to accurately estimate the Q-function of the pendulum along with a greedy policy which together allow it to produce episodes with low cost, managing to balance the pendulum starting from any state.

2 Introduction

The pendulum is simulated having a continuous state space, which represents the position and velocity of its joint variables, and a discrete control space, which represents the joint torques requested by the controller. Given this circumstances it is not possible to execute traditional Q-learning and produce the Q-table necessary for estimating the Q-function, being the table incompatible with continuous state spaces. This is one of the reasons it might be preferable, if not required, to use V/Q-function approximation techniques. As instructed by [2], In this work the Q-function has been approximated using a deep neural network, which forced to implement several new components into the training in order to ensure a good convergence.

3 Components of Deep Q-Network

The deep Q-learning algorithms introduce 3 majaor components: a deep neural network used to approximate the Q function, the use of mini-batches and experience replay in order to aid the proper execution of stochastic gradient descent, and the use of old network parameters in order to estimate the Q target for the loss function[2]. Another important component, one which is not exclusive to deep Q-network, is the policy used for learning the Q-function.

4 Network

The network in use is composed of several fully connected layers which receive as input the state of the pendulum, described as the position in radiants and the angular velocity of each joint. The last output layer produces the predicted Q value for each possible control of the joints in the specific state. Specifically the network is composed

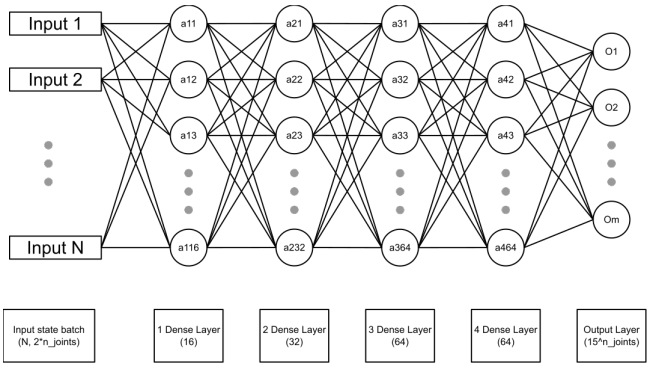

Figure 1: Network architecture in case of n_joints composing the pendulum and a control discretization over 15 steps.

of: input layer, 4 dense hidden layers having respectively 16, 32, 64 and 64 activation cells, and the output layer having dimension equal to the number of discrete controls. The number of controls is $|U| = resolution^{\#joints}$, which means that if $resolution = 15$ and $\#joints = 2$, $|U| = 15^2 = 225$. In Figure 1 is shown a graphical representation of the network in use.

5 System setup

The system provided as solution is composed of several objects:

- **Loss Function**: The training loss uses the common least squares method between the target Q^- values, produced using the current cost and the discounted minimum estimated cost of the next state using old network parameters, and the current Q value, produced by the up-to-date network parameters:

  $$L(Q^-, Q) = \sum_i (c_i + \gamma \min_{u'} Q^-(x_{i+1}, u') - Q(x_i, u_i))^2$$

  The necessary data for the loss function, like state and control, are obtained using experience replay.

- **Experience Replay**: Experience replay is a technique used in order to remove the correlation between sequences of data. Being (x_i, u_i, c_i, x_{i+1}) respectively the state, control, cost, and next state of the system at time i, the experience replay buffer is composed of the list of transitions $t_i = (x_i, u_i, c_i, x_{i+1}) \forall i \in [n, n+s)$ the system produced during training and from which we randomly sample a mini-batch to be used for stochastic gradiant descent. Since the experience replay buffer has a limited size s old transitions are

progressively removed as new transitions are added when the size limit is reached.

- **Policy**: The policy used during training is an $\epsilon - greedy$ one, similar to the one used in [1], meaning a policy which select a random action with probability ϵ and chooses the optimal action with probability $1-\epsilon$. This behaviour is desirable due to the necessity of exploration during training. In a similar way to [1] we have used an exponentially decreasing ϵ, from $max = 1$ to $min = 0.01$, following the equation:

$$\epsilon = max(\epsilon_{min}, e^{\frac{ln(\epsilon_{min})}{0.75N} \times n})$$

with n Episode index and N Number of episodes. The decaying has been conceived in order to adapt to the number of training episodes and reach the minimum epsilon value after executing 75% of the episodes.

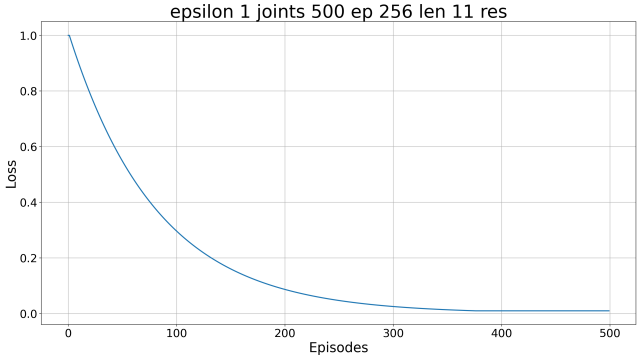

Figure 2: Epsilon graph over 500 training episodes.

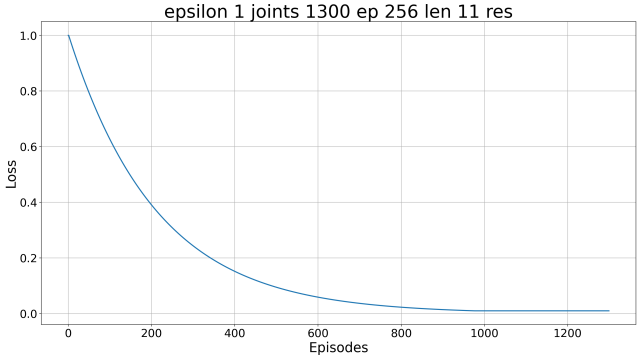

Figure 3: Epsilon graph over 1000 training episodes.

5.1 Actor Network Training

To run the training the user is required to provide two mandatory values: the number of episodes and their length, defined as number of steps. These two parameters define, respectively, the number of the random starting positions from which the robot will start the training and the length of the search executed for each episode. At each step, the current state of the pendulum, x_i, is processed by the policy, which produce the control u_i. The control is applied on the pendulum, producing the next state x_{i+1} and the cost c_i of the pair (x_i, u_i). This data are integrated as the transition $t_i = (x_i, u_i, c_i, x_{i+1})$ and added into the experience replay buffer. Once every k_1 steps we proceed to perform a gradient descent execution, updating the current Q-network's parameters by sampling a mini-batch of random transitions from the experience replay buffer and back-propagating the loss described previously using Adam as optimizer. After gradient descent, once every k_2 steps we update the Q-target network's parameters to match the current Q network ones.

6 Actor Network Testing

The testing of the actor network was carried out first varying the starting position of the robot randomly and secondly making it start from the lowest position to ensure the goodness of the results. For each configuration three episodes were run and the sum of costs gathered during their total length is used as the evaluation measure. The testing phase works by gathering the robot state, querying the policy for the next action, apply it and take the cost from the environment for all the length of the episode.

7 Experiments

The system has been tested over different configurations of the robot, specifically with 1 and 2 joints, 11 and 17 control resolution steps and different numbers of training episodes. The evaluation metrics used are the loss of the training, the training cost-to-go at the end of each training episodes and the cumulative cost in the testing phase.

8 Results

8.1 1 joint

Following are shown the results in case the robot has 1 joint.

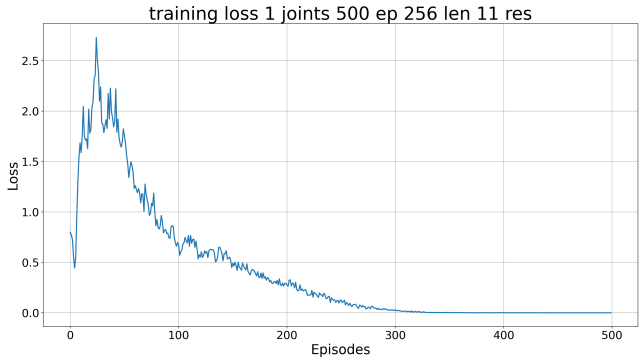

Figure 4: Training loss over 500 episodes lasted 256 iterations having 1 joint and 11 resolution steps.

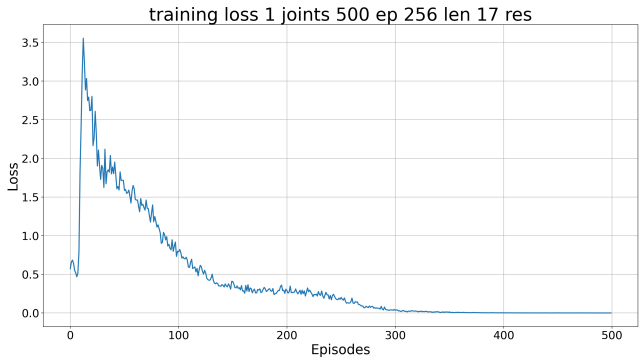

Figure 5: Training loss over 500 episodes lasted 256 iterations having 1 joint and 17 resolution steps.

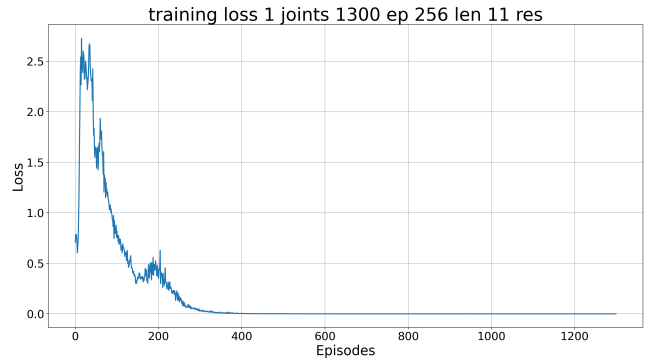

Figure 6: Training loss over 1300 episodes lasted 256 iterations having 1 joint and 11 resolution steps.

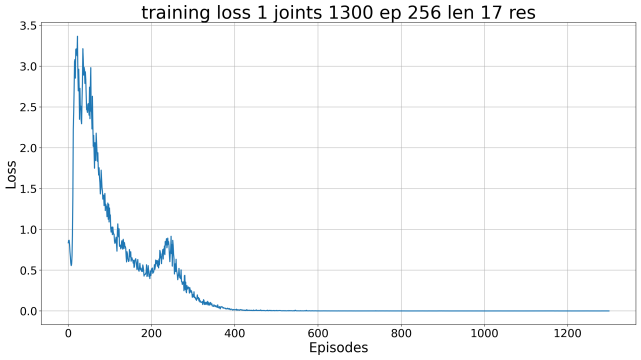

Figure 7: Training loss over 1300 episodes lasted 256 iterations having 1 joint and 17 resolution steps.

Together with the training loss, it is also important to observe the behaviour of the episodes cost-to-go at the end of their length.

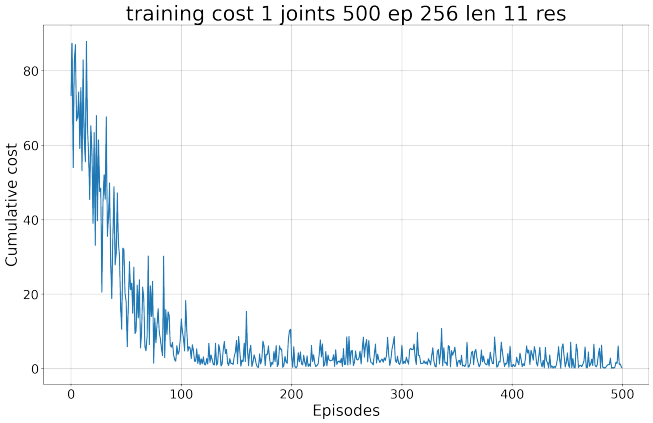

Figure 8: Training cost-to-go over 500 episodes lasted 256 iterations having 1 joint and 11 resolution steps.

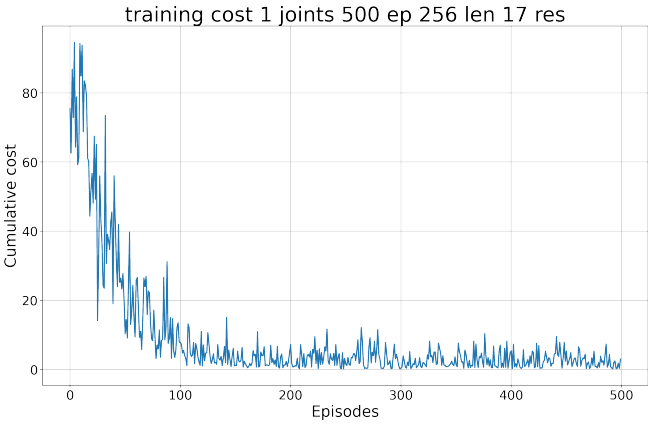

Figure 9: Training cost-to-go over 500 episodes lasted 256 iterations having 1 joint and 17 resolution steps.

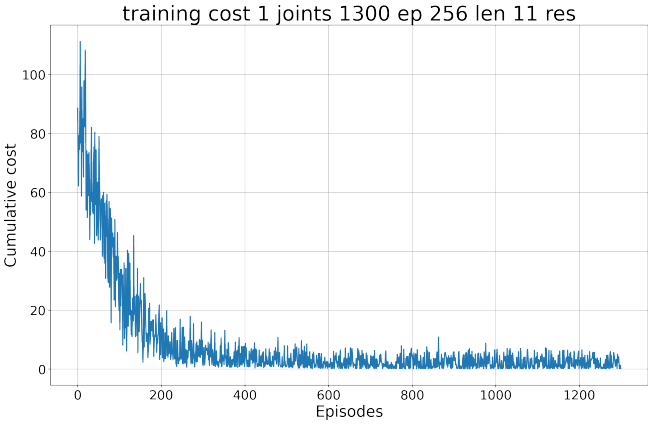

Figure 10: Training cost-to-go over 1300 episodes lasted 256 iterations having 1 joint and 11 resolution steps.

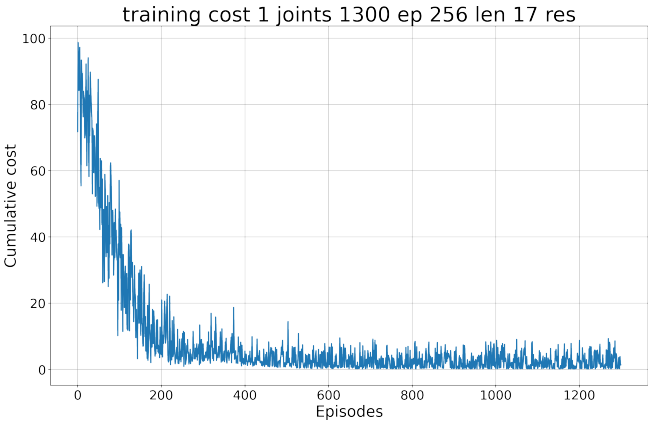

Figure 11: Training cost-to-go over 1300 episodes lasted 256 iterations having 1 joint and 17 resolution steps.

As shown in the graphs, the cost keeps decreasing until it reaches convergence in the latter episodes of the training, as seen in Figures 8 and 9 where it converges at around the 400^{th} episode, while using more training episodes also the convergence point is moved further in time as shown by Figures 10 and 11 where convergence is reached towards the 600^{th} episode.

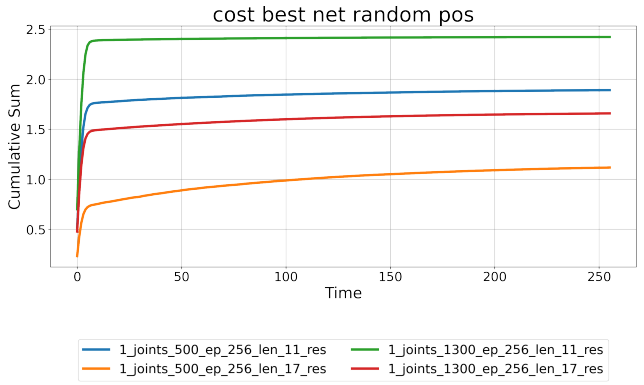

Figure 12: Total cost over the test episodes for the robot starting from random positions using the best performing network.

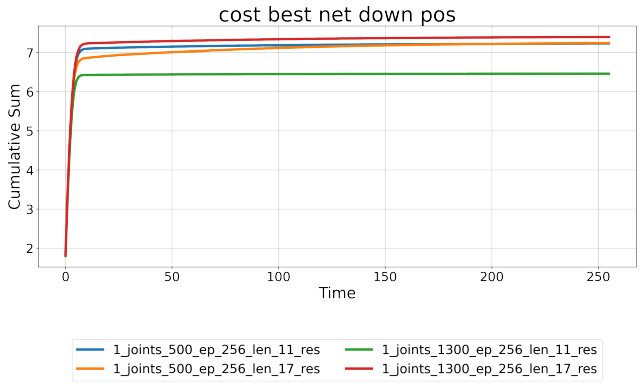

Figure 13: Total cost over the test episodes for the robot starting from down positions using the best performing network.

8.2 2 joints

Following are shown the results in case the robot has 2 joints.

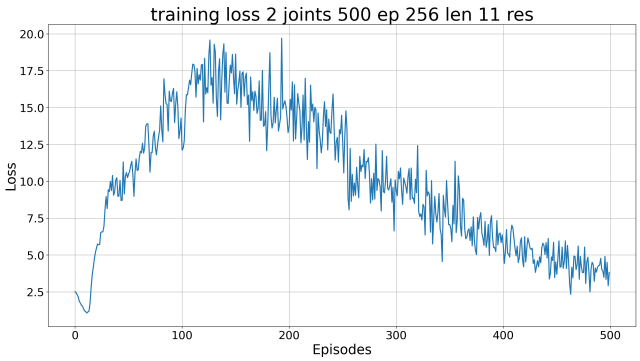

Figure 14: Training loss over 500 episodes lasted 256 iterations having 2 joint and 11 resolution steps.

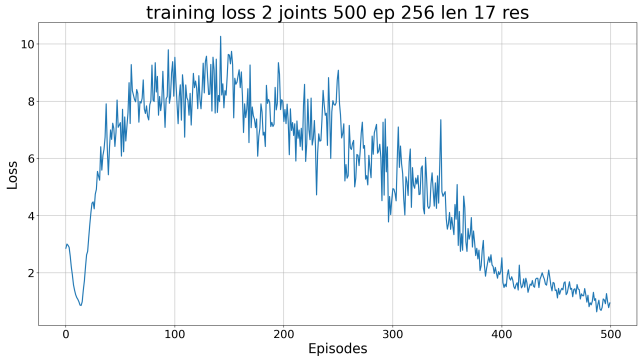

Figure 15: Training loss over 500 episodes lasted 256 iterations having 2 joint and 17 resolution steps.

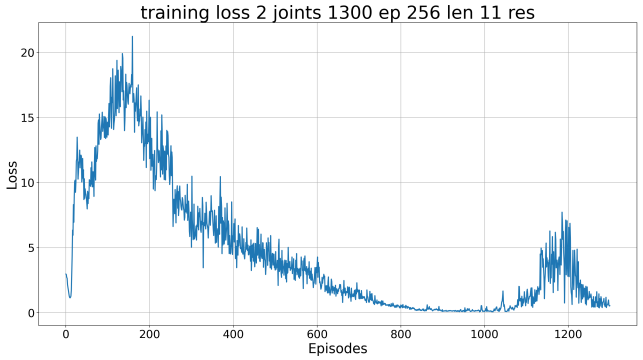

Figure 16: Training loss over 1300 episodes lasted 256 iterations having 2 joint and 11 resolution steps.

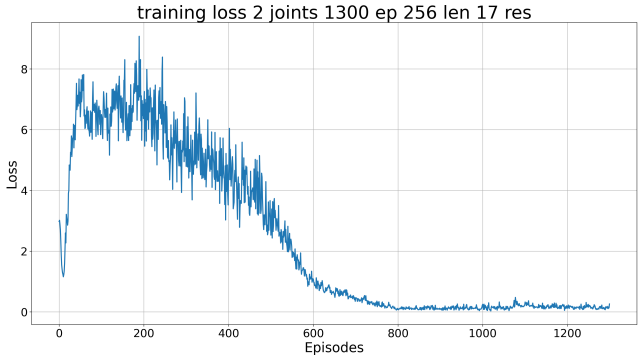

Figure 17: Training loss over 1300 episodes lasted 256 iterations having 2 joint and 17 resolution steps.

Following are the graphs showing the training cost-to-go, gathered from the training episodes.

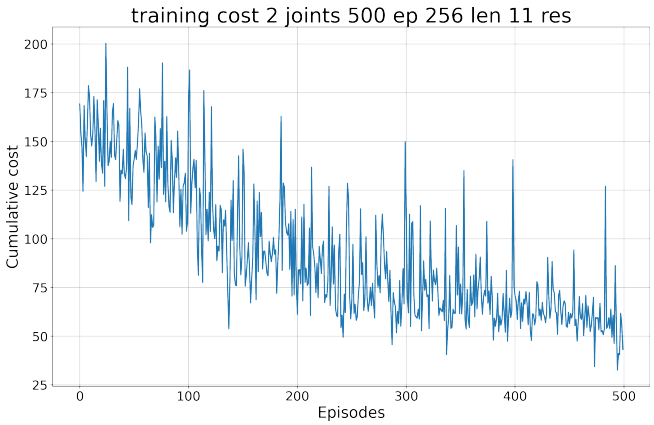

Figure 18: Training cost-to-go over 500 episodes lasted 256 iterations having 2 joint and 11 resolution steps.

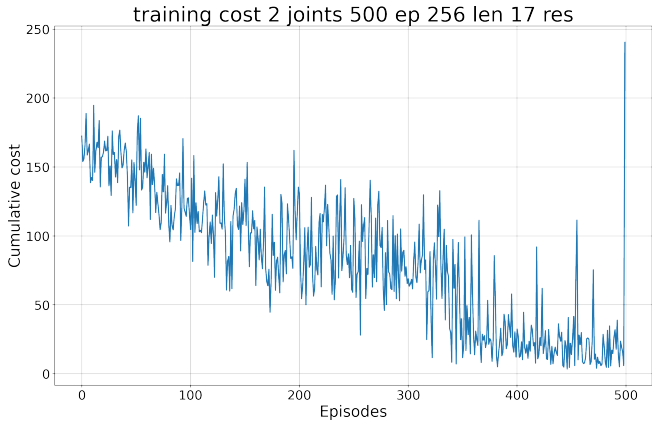

Figure 19: Training cost-to-go over 500 episodes lasted 256 iterations having 2 joint and 17 resolution steps.

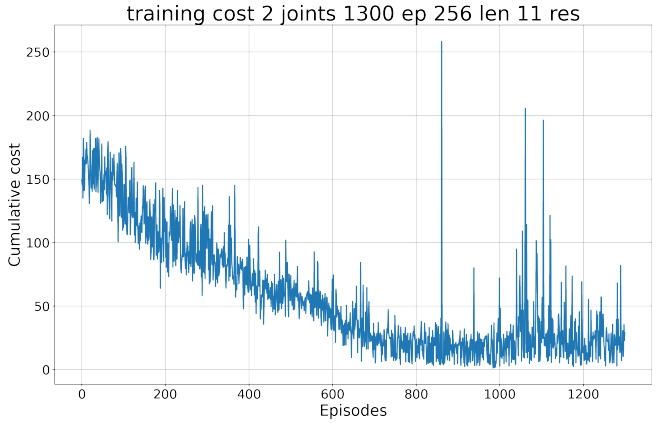

Figure 20: Training cost-to-go over 1300 episodes lasted 256 iterations having 2 joint and 11 resolution steps.

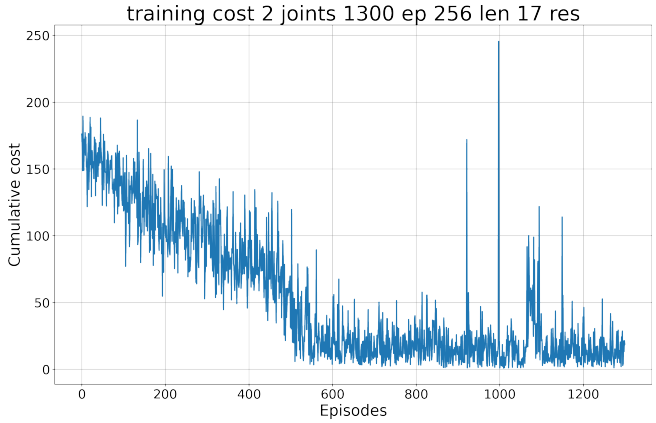

Figure 21: Training cost-to-go over 1300 episodes lasted 256 iterations having 2 joint and 17 resolution steps.

Differently from what said in the case of the 1 joint configuration, the 2 joints one presents strong peaks in the graphs of the training cost-to-go explained by the divergence of the network visible in the loss graphs represented in Figures 16.

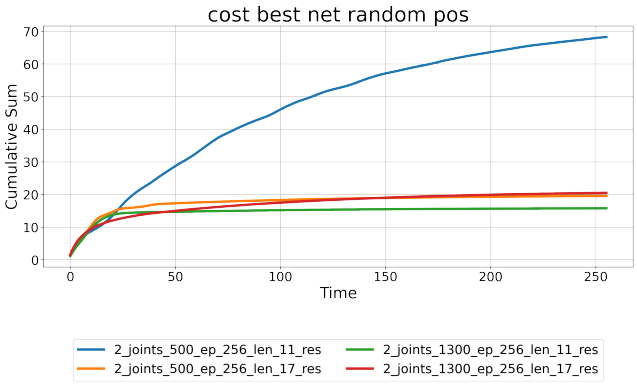

Figure 22: Total cost over the test episodes for the robot starting from random positions using the best performing network.

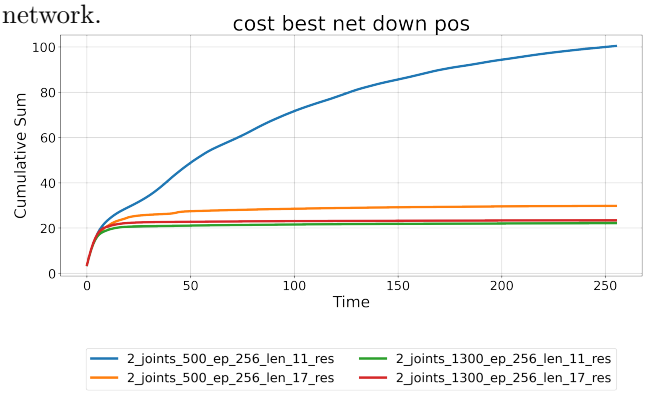

Figure 23: Total cost over the test episodes for the robot starting from down positions using the best performing network.

It is possible to observe how differently from the 1 joint case, the 2 joints implementation shows a training loss which decreases at around 800 episodes to then increase again. This behaviour may be due to the non-linear approximation which does not grant convergence. Furthermore, in the testing phase, the setting having 1300 episodes and 11 control resolution steps was the one leading to the lowest cumulative cost as shown in Figures 22 and 23.

9 Conclusion

The project correctly trained a system to complete the task of balancing the pendulum by using the deep Q learning algorithm shown in [1], both in the case of one single joint and in the case of two joints. Although the resulting motion is not perfect, and small continuous adjustments must be performed in order to maintain the pendulum in upright position, we judge the behaviour to be within acceptable bounds for a system ignorant of the exact dynamics of the pendulum.

9.1 Incorrect modelling of pendulum dynamics

During the development of the solution, we encountered a particular behaviour for the pendulum. Although we did not provided any torque to the joints it would move chaotically, not settling on the downright position that intuition would suggest. Such behaviour is show in the $error_test.py$

script, along with the solution found. By decreasing the value of the b variable in its *dynamics* function to one tenth of its original value we were able to recreate the correct behaviour that intuition would suggest, meaning that a pendulum not in its equilibrium state oscillates and slow down to its downward position. Although we corrected this behaviour, we still tried to apply the main deep Q learning algorithm with the original pendulum dynamics. The system was still able to balance the pendulum, with one joint, even in this circumstances. Though it is to be noted that the training took considerable more time and the final result was not as accurate.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[2] Roderick M., MacGlashan J., Tellex S. Implementing the Deep Q-Network. https://doi.org/10.48550/arXiv.1711.07478