Optimisation Based Robot Control
Final project

Nicola Farina
ID 229296
Dept. of Information Engineering and Computer Science
nicola.farina@studenti.unitn.it

Paolo Furia
ID 239451
Dept. of Industrial Engineering
paolo.furia@studenti.unitn.it

Abstract

In this project, we successfully design and implement a Deep-Q-Learning environment to train two kinds of robot pendulums to reach and hold their inverted state using a swing-up maneuver: a single pendulum, and a more complex double pendulum with no actuation for the second joint. Our code is available on GitHub at this link.

I. Introduction

The goal of this project is to use the Deep Q-Learning algorithm to compute an optimal policy for the control of a pendulum swing-up motion. In particular, we compute two policies: one for a single pendulum; and one for an under-actuated double pendulum, i.e. a double pendulum with no actuator on the second joint.

In Section II, we introduce the main mathematical concepts of Q-Learning and Deep-Q-Learning and formalize our problem. In Section III, we present our implementation choices. In Section IV, we evaluate our results. Finally, in Section V we make some final remarks and discuss possible improvements on our work.

II. Problem formalization

Q-Learning (QL) and Deep-Q-Learning (DQL) are reinforcement learning algorithms. As usual in a reinforcement learning setting, these are the main building blocks of the framework of the problem:

- an *agent*, the model which we try to design, which interacts with an environment;
- an *environment* on which the agent operates, which changes as the agent performs actions;
- a set of states $\mathcal{X}$, which describe the possible states of the environment;
- a set of actions or controls $\mathcal{U}$, which describe what the agent can do;
- a cost function $l : \mathcal{X}, \mathcal{U} \to \mathbb{R}$, which measures the cost of reaching state x with control u;
- a goal, which the agent tries to achieve with minimum cost.
- a transition function $f : \mathcal{X}, \mathcal{U} \to \mathcal{X}$, which yields the resulting state x' from applying control u in state x.

The goal of this optimal control problem is to find a globally optimal policy $\pi^* : \mathcal{S} \to \mathcal{U}$, i.e. a sequence of controls, for each state, that brings the agent to the goal with minimum total cost. To achieve this, QL and DQL use the concept of a Q-function $Q : \mathcal{X}, \mathcal{U} \to \mathbb{R}$, which gives, for each state-control pair, the overall future cost:

$$Q(x, u) = l(x, u) + \gamma \cdot min_{u'} Q(f(x, u), u')$$

where γ is the discount factor: the higher it is, the more important immediate costs are w.r.t. future costs. This recursive formulation is a form of the Bellman equation and derives from Bellman's principle of optimality. Having the optimal Q-function Q^*, an optimal policy can be easily derived:

$$\pi^*(x) = argmin_u Q^*(x, u)$$

In QL, this Q-function is iteratively improved from a starting estimate with temporal difference (TD):

$$\delta(Q(x, u)) = l(x, u) + \gamma \cdot \underbrace{min_{u'} Q(x', u')}_{\text{estimate of optimal future value}} - \underbrace{Q(x, u)}_{\text{current value}}$$

$$Q_{t+1}(x, u) \leftarrow Q_t(x, u) + \alpha \cdot \delta(Q_t)$$

where α, the learning rate, measures how much the estimate is updated. This algorithm converges to the true Q-function only after an infinite number of iterations and given that every state is explored an infinite number of times. This means that *exploration* is important, and it is achieved by choosing actions using an ϵ-greedy policy: with probability ϵ choose a random action, otherwise select the action that minimizes the Q-value.

While QL stores the Q-function in a Q-table of size $X \times U$, meaning the state and action space must be discrete, DQL approximates the Q-function through a neural network $Q(x, u, w)$, called Q-network, with weights w. This allows to have a continuous state x as input to the neural network (the action space remains discrete), and this is necessary in high-dimensional state spaces like those of modern robots. The algorithm is thoroughly explained in the paper by Mnih et al. [1]. Its main ideas are:

- using an *experience replay* buffer, which stores a number of *experiences* $(x, u, f(x), l(x, u))$. The parameters of the model are updated by sampling a random batch of experiences from this buffer, breaking the correlation between adjacent experiences that would disrupt training;
- using a separate target network $\hat{Q}$ to compute the estimate of the optimal future value. This network is frozen for a number of steps before it is updated with the same weights as the main network. This solves the moving-targets problem, since the main network is able to learn to reproduce the target values before they change.

A pseudocode of the Deep Q-learning algorithm used in this project is shown below in Algorithm 1.

Algorithm 1: Deep Q-Learning algorithm in the project

```
Given: Pendulum environment with N number of joints with discretized controls (torque)
Initialize: replay memory $D$ to capacity replay-size
Initialize: action-value main network $Q$ with random weights $w$
Initialize: action-value target network $\hat{Q}$ with weights $w' = w$
for episode = 1, max-episodes do
    initialize environment to a random state;
    for $t$ =1, max-steps do
        with probability $\epsilon$ select a random action $u_t$ otherwise select $u_t = min_u Q(x(s_t), u_t; w)$;
        execute action $u_t$ in environment and observe cost $l_t$ and next state $x_{t+1}$;
        store experience $(x_t, u_t, l_t, x_{t+1})$ in $D$;
        if $t >$ replay-start then
            sample random mini-batch of size batch-size from $D$;
            set: $y_t = \begin{cases} l_t & \text{if } x_{t+1} \text{ is goal} \\ l_t + \gamma \cdot min_u \hat{Q}(x_{t+1}, u; \hat{w}) & \text{otherwise} \end{cases}$;
            perform a gradient descent step on $(y_i - Q(x_t, u_t))^2$ with respect to the weights $w$
        end
        every sync-target steps: copy main network $Q$ weights $w$ to target network $\hat{Q}$ weights $w'$;
        if $x_{t+1}$ is goal then end episode;
    end
end
```

Having introduced the setting and its mathematical concepts, we report in Table I the formalization of our problem, where `num_controls` is the number of discretization steps of the torque (i.e. the control space size), `max_vel` is the maximum velocity that each joint can reach, and `max_torque` is the maximum amount of torque that can be applied to the first joint actuator. We also report in Table II all the hyper-parameters of our implementation of the Deep-Q-Learning algorithm.

Agent	Single pendulum, double pendulum
Environment	Simulation environment
States	All combinations of joint angles $[-\pi, \pi]$ and velocities [`-max_vel`, `max_vel`]
Actions	A set of `num_controls` actions which discretize the torque range [`-max_torque`, `max_torque`] for the first joint actuator
Cost	A custom cost for applying torque u and reaching state x'
Goal	All joints of the pendulum in the upright position ($x = 0$)
Transition function	The deterministic dynamics of the pendulum $x' = f(x, u)$

TABLE I: Formalization of the problem

Hyper parameter	**Explanation**
`replay_size`	maximum experience replay buffer size
`replay_start`	number of steps to start replay training
`discount`	cost discount factor
`max_episodes`	maximum number of episodes
`max_steps`	maximum number of steps per episode
`sync_target`	rate of syncing target with main network (steps)
`eps_start`	starting value of ϵ
`eps_decay`	decay of ϵ ($\epsilon \leftarrow \epsilon \cdot \texttt{eps\_decay}$)
`eps_min`	minimum value of ϵ
`batch_size`	size of an experience batch
`lr`	initial learning rate

TABLE II: Hyper parameters of DQL

III. Implementation

Implementing the code almost from scratch was not trivial, but the final result allowed us to easily run experiments with different architectures for the Q-network, cost functions, and combinations of hyper-parameters. In practice, the tuning of those inputs of the algorithm is what determines the final policy and performance of the pendulum.

What follows is a presentation of the main elements that we had to tune to achieve our final results, along with implementation choices that are shared between the single and double pendulum. In Sections III-A and III-B we go into the details for each pendulum.

a) Q-network architecture: The neural network architectures that we tested were all multi-layer perceptrons (MLP), i.e. a series of fully-connected hidden layers connecting an input to an output layer. This differs from the original DQN paper, where they used a convolutional neural network, because we do not deal with images but with numeric values. MLPs are usually designed in a semi-symmetric manner: an increasing number of neurons for each layer starting from the input to the middle of the network, followed by a decreasing number of neurons towards the output. The more hidden layers and neurons per layer, the more *learning capacity* the network gains; but at the same time, the number of weights to train exponentially increases and can slow down or even harm training for simpler problems.

The number of neurons (size) of the input layer is 2 for the single pendulum (joint angle and joint velocity), and double that for the double pendulum. The output size corresponds to the number of possible controls `num_controls` and is a tunable parameter: conceptually, the higher it is, the more granularity there is in the torque applied by the robot; but at the same time, it could be more difficult to learn what each action does.

b) Loss and optimizer: For the loss, we used a standard mean squared error (MSE) loss. For the optimizer, whose main goal is to adapt the learning rate during training, we chose the most widely used Adam optimizer. We did not experiment with the optimizers since it was not the main focus of this project.

c) Hyperparameters: Tuning all possible combinations of hyperparameters was not feasible for us due to constraints on time and resources, so we used default values which generally work good in a DQL setting. We still tried to experiment with them, like decreasing how much ϵ is decayed to improve exploration, increasing/decreasing the experience buffer size, and changing how often the target network is updated based on the value of the training loss. However, we do not cover these experiments because they did not bring any major improvement to the final result.

d) Pendulum parameters: Parameters like `max_vel` and `max_torque` play a crucial role in determining the final policy: usually, having harder constraints on their values results in more stable and better-performing robots, but makes the training process much harder.

e) Cost function: The design of the cost function is crucial, because it directly drives training by telling the pendulum when it is doing good or bad.

A. Single pendulum

Since the single pendulum is not a complex problem, we had the resources to run and evaluate three main experiments. In practice we ran a lot more experiments, but their results are not worth being presented here. We call those three experiments: `[SmNet, HighU]`, `[LgNet, LowU]`, `[SmNet, LowU]`, where `LgNet` and `SmNet` stand for *large* and *small network* respectively; and `HighU` and `LowU` stand for *high* and *low torque*.

The network architectures are shown in Table III, while the training parameters are shown in Table IV.

	[SmNet, HighU]	[LgNet, LowU]	[SmNet, LowU]
Input	2	2	2
Hidden 1	12	24	12
Hidden 2	32	48	32
Hidden 3	24	48	24
Hidden 4	-	24	-
Output	11	21	11

TABLE III: Q-network architectures for the single pendulum, with the size of each layer. Every layer is followed by ReLU

Hyperparameters	
replay_size	10000
replay_start	1000
discount	0.99
max_episodes	150
max_steps	500
sync_target	1000
eps_start	1.0
eps_decay	0.999
eps_min	0.005
batch_size	128
lr	0.001

Pendulum limits			
	[SmNet, HighU]	[LgNet, LowU]	[SmNet, LowU]
max_vel [rad/s]	8.0	8.0	8.0
max_torque [Nm]	5.0	2.0	2.0

TABLE IV: Parameters used for the single pendulum

The cost function is a standard one that penalizes joint angle, velocity and torque with different weights:

$$l(q, v, u) = q^2 + 10^{-1} \cdot v^2 + 10^{-3} \cdot u^2 \tag{1}$$

which is zero when the pendulum is in the upright position ($q = 0$), with zero velocity and no torque applied.

B. Under-actuated double pendulum

This is a much more complex problem to solve, so we had to run the training procedure for longer. For this reason, we present only one architecture out of the experiments we ran, since we did not have enough resources to train a lot of models for many episodes.

The architecture of the network is the same as the LgNet structure for the single pendulum, minus the input size which is now 4. Also the hyperparameters remain the same, minus max_episodes which we set to 1700. For the limits, we set max_torque to 2.0 and increased max_vel to 13.0. This is a high velocity, but we saw that lower values never reached the goal.

Finally, greater attention has been paid to the choice of the cost function. We came up with the following:

$$l(q, v) = (q_1 + q_2)^2 + 0.85 \cdot q_1^2 + 10^{-1} \cdot q_2^2 + 10^{-2} \cdot v_1^2 \tag{2}$$

where q_1 and q_2 are angles of the joints (q_2 relative to q_1), and v_1 is the velocity of the first joint.

The idea is to give the highest weight to the task of bringing the tip of the second joint on top, and a slightly lower weight for bringing the first joint on top. We thought this would work because, in order to balance the tip, the first joint needs to be free to move left and right. We also assigned a weight to the task of having the two joints aligned ($q_2 = 0$). Finally, we added a penalization for the velocity of the first joint only, since the other joint would need to move fast to be swinged into the upright position. We left out a penalization of the torque since it made learning too difficult and thus slow, but a small penalization to it would be preferable.

IV. Evaluation

All the models presented in the previous section managed not only to learn a policy to perform the swing-up manoeuvre, but also to reach the inverted state from (almost) any initial state. Videos to demonstrate this are available on the GitHub, and in Figures 1 you can see snapshots of the states of both pendulums after reaching the top.

(a) Single pendulum

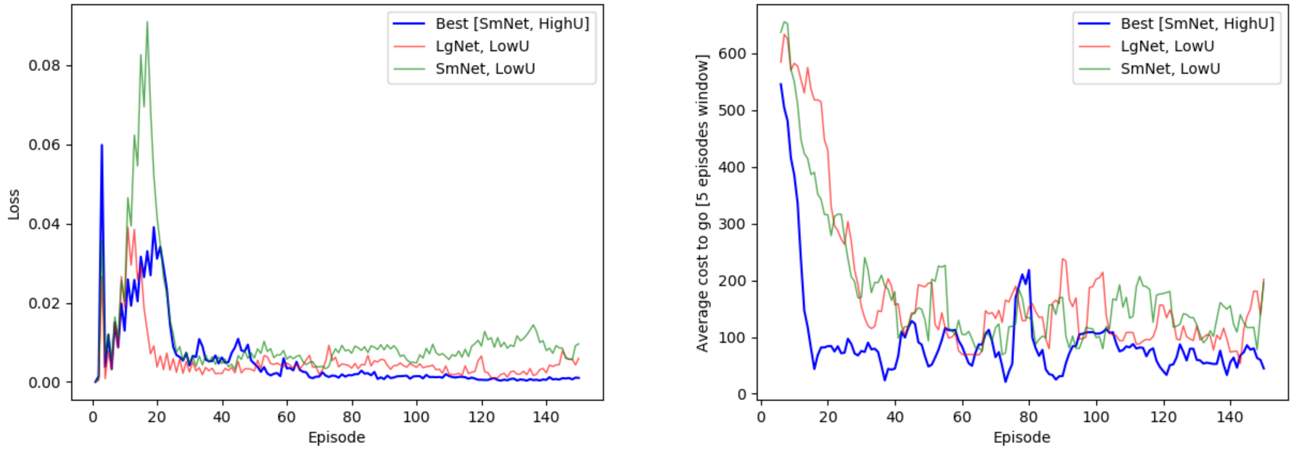
(b) Double pendulum

Fig. 1: Simulated pendulums after having reached the top with the learnt policy

A. Single pendulum

We trained each model for a total of 150 episodes. The average training time for one episode was around 18 seconds for all models, making the total training time about 45 minutes. The `LgNet` and `SmNet` achieved the same training time because the former is really not much bigger than the latter.

After training, we plotted the average loss and cost to go for each episode to find the best weights for each model, since in DQL the last weights are not necessarily the best. The plots can be seen in Figure 2. In particular, for the average cost to go, we decided to average it over a window of 5 episodes. We did this because the cost to go depends on the initial state, which is random at each episode: episodes, where the pendulum starts near the goal, will have a lower cost to go, even if the underlying policy is not perfect. This also helps in smoothing the plot, which shows a clear downward trend. The same happens for the loss: the spike at the beginning is when training begins after a number of experiences are stored, and then it decreases steadily, with smaller spikes when the target network is updated. `SmNet, HighU` achieves the best results in both metrics, and from now on we will refer to it as the best model.



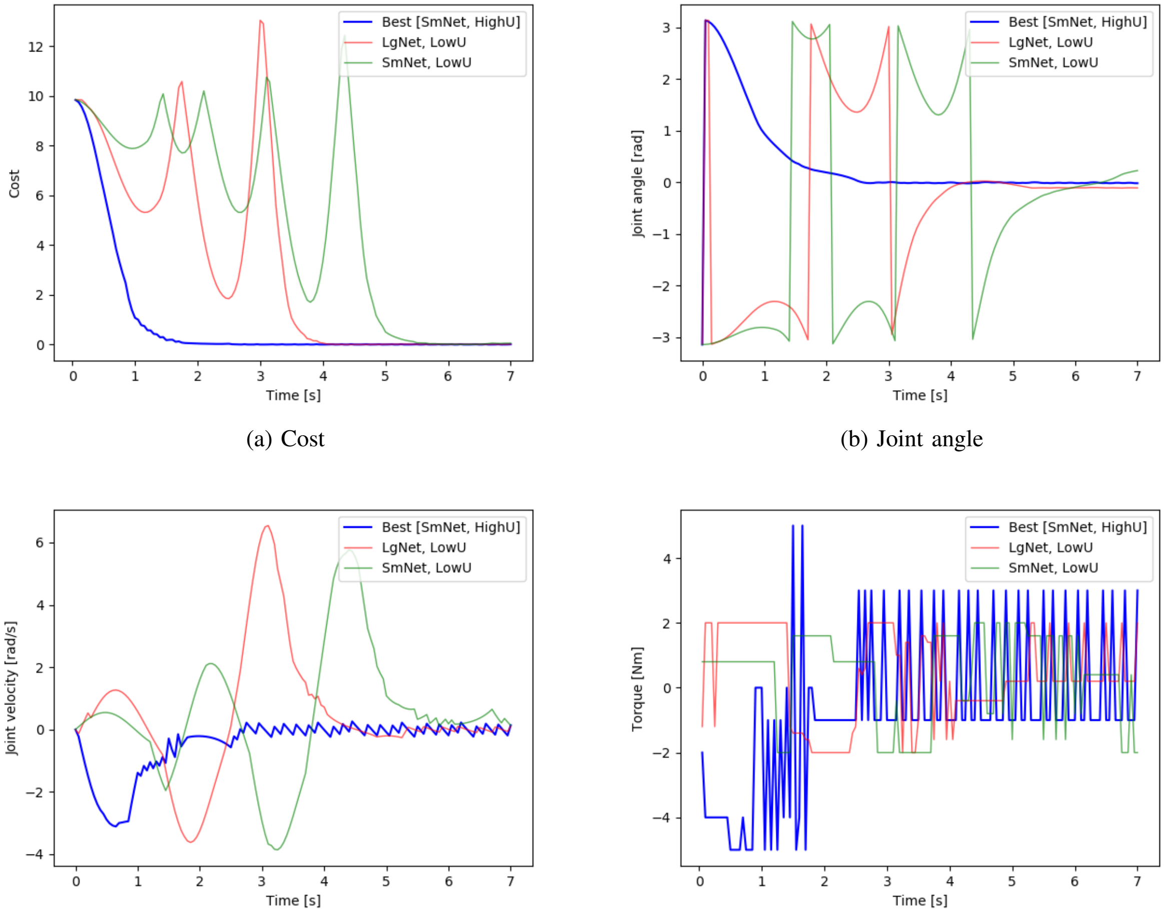
(a) Loss

(b) Cost to go

Fig. 2: Training loss and cost to go for each training episode of the single pendulum

Using these plots and running some simulations, we identified the best weights for each model: the best model at episode 130, `[LgNet, LowU]` at episode 100, `[SmNet, LowU]` at episode 100. We loaded these models, ran a simulation starting from the bottom state with zero velocity, and obtained the plots in Figure 3. The cost trajectory in Figure 3a shows how the best model reduces the cost it takes much quicker than the others. The joint angle trajectory shows that the networks trained with a limited torque learn to swing and use gravity to speed up. The best model, instead, has learnt to use just the torque to swing to the top in a single movement. Depending on the situation, one behaviour may be preferable to the other. We have chosen the best model also based on this: we prefer torque over velocity. It is important to notice that the torque trajectory is very jagged, which is not desirable in a robot, but all models showed this behaviour. This could perhaps be mitigated by increasing the number of possible actions to make the control space much more granular.

The worst performing model is `[SmNet, LowU]`, which struggles to get to the top and, once there, stops in a non-vertical position. We observed this behaviour in many of the other models that we trained. Instead, `[LgNet, LowU]` achieves performance comparable to the best model, even though it is not perfectly vertical; this shows that a slightly larger net may be needed to learn to use gravity in absence of torque.



(a) Cost

(b) Joint angle

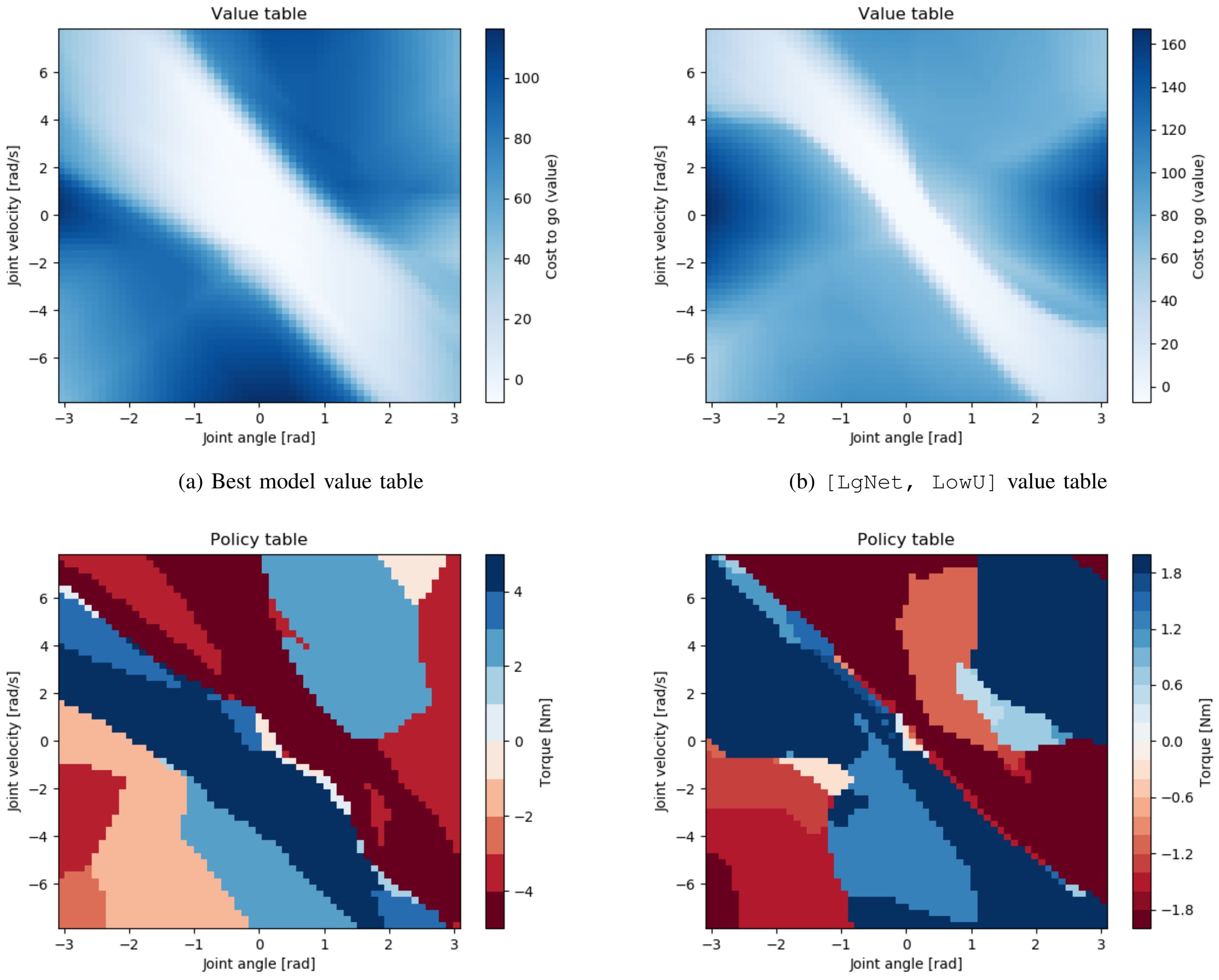
(c) Joint velocity

(d) Torque

Fig. 3: Trajectories of the simulation for each single pendulum model

For a more in-depth comparison between the best model and `[LgNet, LowU]`, we created a Q-table from each Q-network by discretizing the states and computed the value and policy tables shown in Figure 4. They both reflect the expected behaviour: the cost to go (from the value table) is lower on the diagonal, i.e. when the pendulum is slow and at the top, or fast and far from the top; and the policy tables show that torque is applied in almost all states in the right direction, except at the top and when the model is already swinging. All tables are fairly symmetrical, which is expected since the problem itself is symmetrical around the zero-state. `[LgNet, LowU]` tables look slightly better than the ones of our chosen model in terms of symmetry, and can be explained by the larger network being better at learning, even though the final joint angle is not always precise as stated previously. We also noticed that the policy uses a lot of extreme actions, especially around the goal: this is why the torque trajectory is very jagged. It is possible that increasing the number of possible actions may push the agent into using also the less extreme ones, simply by allowing more granular choices of torque.

In the end, we chose the `[SmNet, HighU]` model as our best because it reached the goal more consistently and in less time than the other models. `[LgNet, LowU]` comes at a close second, and it is interesting to see how it learnt to use gravity to bypass the lack of torque. Either of the models can be chosen depending on the preference of achieving higher torques or higher velocities.



(a) Best model value table

(b) `[LgNet, LowU]` value table

(c) Best model policy table

(d) `[LgNet, LowU]` policy table

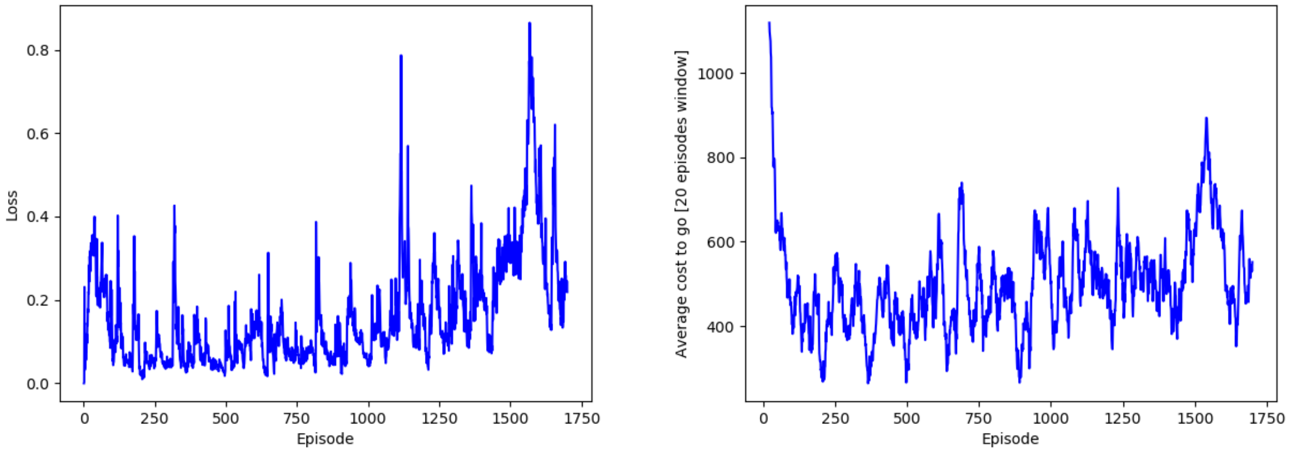
Fig. 4: Value and policy tables of the single pendulum models

B. Double pendulum

We trained the model for 1750 episodes. The average training time for one episode was around 18 seconds like the single pendulums, as we used the same parameters and architecture. The total training time was around 9h.

As with the single pendulum, we looked at the average loss and cost to go plots, increasing the averaging window of the latter to 20 episodes. Figure 5 shows that the curves are very jagged, showing some instability during training, especially during the later episodes: this is not surprising in a DQL setting, especially for a complex problem like this one. We identified the best model at episode 910, ran a simulation and obtained the trajectories shown in Figure 6. While the angle trajectories look really good, the velocity and torque trajectories are incredibly jagged, and wouldn't be suitable to real world simulations. We tried to limit the velocities and torque both in the cost and by setting `max_vel` and `max_torque`, but we could never achieve a reliable swing up motion. For this reason, it is not surprising to see such a jagged torque trajectory: simply because we did not penalize it.

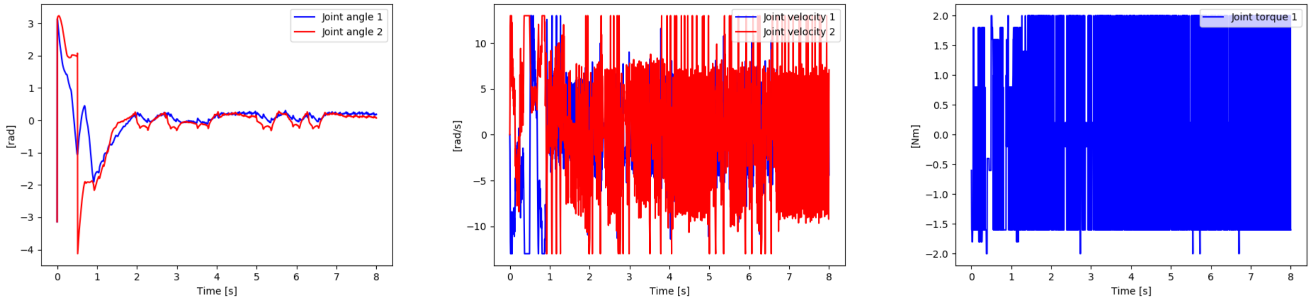
We do not report the value and policy tables because discretizing the states of a double pendulum and visualizing them in a 2D table is not as trivial as the single pendulum case.



(a) Loss

(b) Cost to go

Fig. 5: Training loss and cost to go for each training episode of the double pendulum



(a) Joint angle

(b) Joint velocity

(c) Torque

Fig. 6: Trajectories of the simulation for the best double pendulum model

V. Conclusions and possible improvements

In this project, we implemented from scratch a Deep Q-Network framework that successfully computes an optimal policy to perform a swing-up manoeuvre for a single and under-actuated double pendulum. The crucial points to achieve this result were the tuning of the hyperparameters and the design of the cost functions.

For the single pendulum, we obtained a good policy that focuses on using the torque and always reaches the inverted state with feasible controls and very good precision. We also obtained a second policy that, instead of using torque, uses gravity to swing to its inverted state. As the best model, we chose the first one, as it was more precise and reached the goal faster.

For the double pendulum, we obtained a policy that achieves the desired goal, although with extremely rapidly-changing controls and velocity that would not be suitable for real world simulations. It is, nonetheless, a very good proof of how DQL can achieve complex results.

The main bottleneck of the project was the high computational cost, in terms of time needed to train the various models, which did not allow us to examine additional combinations of the neural network parameters. More complex architectures and larger models could be tried. Furthermore, different cost functions could be tested, especially for the double pendulum, by changing different weight combinations in order to obtain an ideal result, or designing cost functions that take into account the position of the pendulum's frames. We are convinced that finding a way to penalize the velocities, torques and accelerations of the double pendulum while still achieving the goal can be achieved with enough trial and error.

With more time, we would also re-run at least the double pendulum experiments with a greater number of time steps. The double pendulum runs the dynamics with a 0.005 seconds time step, ten times faster than the single pendulum. With the same `max_steps` value of 500, each episode lasts only 5 seconds, which is less than ideal for the model to learn complex motions to swing up (even though such behaviours are shown in the final policy). We realized this only after running the long training, and could not re-train the whole system. A similar thing also applies to the single pendulum: it runs with a time step of 0.05 seconds, making each episode last 25 seconds. We believe this is more time than needed, but this does not impact training as much as having episodes which are too short.

As for the coding part of the project, we developed a good starting point for a possible framework in which various robotic agents can be trained to perform different tasks using reinforcement learning. One thing we considered was introducing other variations of deep reinforcement learning algorithms, for example Double Deep Q-Networks (DDQN) [2], but we did not do it because it was not in the direct scope of this project.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[2] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.