Report

Report Assignment 03

(Optimization Based Robot Control)

Student: *Edoardo Barba 231592*

Student: *Matteo Brugnera 232670*

Course: *Optimization Based Robot Control*

18/06/2023

Contents

1. Introduction

In this report, we delve into the problem of optimal control and stabilization of both the **pendulum** and **double pendulum** systems. The pendulum system, consisting of a mass attached to a fixed pivot point, exhibits complex dynamics that require sophisticated control strategies to achieve desired objectives. In this specific scenario, our objective is to bring the pendulum to an upright position, as shown in the following image [1]. We achieved this result by utilizing the powerful **Deep Q-Network (DQN) algorithm**. The *DQN* algorithm has gained substantial popularity due to its ability to learn *control policies* directly from raw sensory inputs, making it an ideal choice for addressing complex control problems such as this one. To gain a comprehensive understanding of the problem, we introduce two systems: the pendulum and the double pendulum.

2. Problem Description

As we said previously, our objective is to solve the pendulum swing-up problem. But before doing that, we first formalize the setting.

2.1 Pendulum State

In our problem the state of the system can be represented by the angle θ (measured from the vertical upward position) and the angular velocity $\dot{\theta}$ of the pendulum. Thus, the state can be denoted as s = $[\theta,\dot{\theta}]$. In which θ is the vector of joint angles (length equal to 1 for single pendulum and length equal to 2 for double one).

2.2 Control

For this particular setting, we consider a scenario where only the first joint is *actuated*. So in case of the double pendulum, the second joint is actuated. So the torque can be represented as a vector: u=$[u_1$, 0]

2.2.1 Control discretization

To accommodate the continuous state and **discrete control** nature of the environment, we needed to define a *range of torque values* for the pendulum systems. Specifically, we determined the maximum and minimum torque values for both the single and double pendulum scenarios. For the single pendulum, we set the maximum torque value to 2 and the minimum torque value to -2. Similarly, for the double pendulum, we defined the maximum torque value as 4 and the minimum torque value as -4. This wider range of torques accounts for the additional complexity and degree of freedom in the double pendulum system.
To **discretize** the range of torques, we opted to use 21 *equally spaced values* between the minimum and maximum torque values for both the single and double pendulum scenarios.

2.3 Dynamics of the system

In this following subsections, we present their mathematical models and derive the **system dynamics** to capture their behavior accurately.

2.3.1 Pendulum System

The pendulum system features a single mass attached to a rigid rod or string, permitting oscillatory motion under the influence of gravity. The *dynamics* of the pendulum can be described by a *second-order ordinary differential equation* known as the **pendulum equation**. It can be mathematically represented as follows:

$$\theta^{''}(t) + (g/L) * \sin\theta(t) = 0; \tag{1}$$

where $\theta(t)$ represents the *angular displacement* of the pendulum from its equilibrium position at time t, $\theta^{''}(t)$ denotes the second derivative of θ with respect to time (*angular acceleration*), g represents the acceleration due to gravity, and L denotes the length of the pendulum.

2.3.2 Double-Pendulum System

The double pendulum system represents a more complex dynamic system, consisting of two pendulums connected in series. The system exhibits a richer and more intricate behavior with respect to a simple pendulum.

2.4 Objective function

In this section, we describe the objective function used to evaluate the quality of states within the environments. We will refere to the functions as costs but in both single and double pendulum, rather than framing it as a cost, we chose to view it as a reward and therefore used the negative of the cost:

$$reward = -cost$$

2.4.1 Cost functions - Single pendulum Scenario

To assess the quality of states within the environments, we employed two different cost functions. The first cost function is based on the square of the joint angle and is computed as follows:

$$\text{cost} = \Delta q = \theta^2 \tag{2}$$

where θ represents the joint angle. The second cost function we used considers also the **angle changing rate** (velocity) and is represented as:

$$\text{cost} = W_q \Delta q + W_v \exp\left(-\frac{\Delta q}{2}\right) v. \tag{3}$$

where Δq is computed as seen previously, W_q and W_v are weights (respectively 0.7 and 0.3), and v represents the velocity. What we are trying to achieve with this cost is to exponentially penalize the velocity as the position of the joint approaches the upright position. This is done in order to make the pendulum learn to **"brake"** when near the final position.

2.4.2 Cost functions - Double pendulum Scenario

For the double pendulum scenario, we utilized two distinct cost functions to evaluate the quality of states. The first cost function is defined as follows:

$$\text{cost} = W_1 \Delta q_1 + W_2 \Delta q_2, \tag{4}$$

where Δq_1 and Δq_2 are computed as seen previously and they refer to the first and second joint respectively. The same goes for the weights W_q and W_v. The angle of the second joint is measured with respect to the first link of the pendulum, and it is zero when the two links are aligned.
The second reward function we used takes also into account the angle formed by the second joint with respect to the environment (called $q_{verical}$) rather than just considering the angle with respect to the first link. In this case, $q_{verical}$ will be equal to zero only when the second link is in the upright position.

$$\text{cost} = W_1 \Delta q_1 + W_2 \Delta q_2 + W_3 \Delta q_{vertical}, \tag{5}$$

In this scenario, we used as weight $W_1 = 0.7$, $W_2 = W_3 = 0.15$.

3. DQN

DQN combines the principles of **reinforcement learning (RL)** and **deep neural networks** to enable agents to learn optimal policies through interaction with their environments. Traditional *RL* algorithms face challenges when dealing with high-dimensional state spaces, making it difficult to approximate the optimal action-value function accurately. *DQN* addresses this limitation by utilizing deep neural networks as *function approximators*, allowing for more effective representation of state-action value functions.
At the heart of *DQN* is the concept of **Q-learning**, which aims to learn the **optimal action-value function**, denoted as $Q(s, a)$. In *DQN*, this function is approximated using a deep neural network, denoted as $Q(s, a; \theta)$, parameterized by θ which represents the weights of the network. As we will see in the next subsections, the network is trained through an iterative process, where it learns from experiences stored in a **replay memory**, aiming to minimize the difference between its *predicted Q-values* and the observed *target Q-values.* The **loss function** for training the Q-network is given by:

$$L(\theta) = E[(y - Q(s, a; \theta))^2]; \tag{6}$$

where θ represents the parameters of the Q-network, *Q(s, a; θ)* is the predicted Q-value for state-action pair (s, a), and y is the target Q-value.

3.1 Structure

DQN introduces several key innovations that have significantly advanced the field of *RL*. One of the major contributions is the integration of **experience replay** into the learning process, thanks to which both the stability and efficiency of the *DQN* are enhanced. By storing previous experiences in a *replay memory* and employing random sampling during training, *DQN* effectively breaks the correlation between consecutive samples and achieves *independent and identically distributed (i.i.d.)* samples. This technique promotes more efficient learning by providing a diverse range of experiences and reducing the bias introduced by sequential data. As a result, *DQN* can learn from a wider range of scenarios and achieve better generalization, leading to improved performance in solving complex control and decision-making problems. Furthermore, *DQN* introduces an extra network, known as **target network**, which is a periodically updated copy of the Q-network. This helps to stabilize the learning process by providing a consistent target for the Q-value updates. In contrast to the initial mention in the *DQN* introduction where both the state-action pair (s, a) was stated as the input, we have decided to utilize only the state as input. As a result, the system's output will consist of a collection of Q-values, with each Q-value corresponding to a potential action that can be taken. Furthermore, we imposed a limitation on the torque τ range, restricting it to a certain range depending on the scenario. Following this constraint, we **discretized** the torque range by dividing it based on the desired number of control values we actually want. It is worth noting that the specific number of control values may vary depending on the particular scenario we are addressing. When it comes to the structure of the neural network (as we can see in the following image [2]), we have:

- an input layer that takes the state as input, and its size varies depending on the scenario at hand. For the single pendulum, the input layer size is set to 2 as we only have the angle and its derivative. On the other hand, for the double pendulum, the input layer size is increased to 4 to accommodate the additional state variables, including the angles of both pendulums and their respective angular velocities;
- after that we have four dense layers, each with a specific size: 16, 32, and the last two layers with a size of 64. The **ReLU** activation function is utilized for all of these layers;
- the final layer of the neural network is a fully-connected layer that directly produces the set of Q-values as its output. Each Q-value in this set corresponds to a specific potential action that can be taken in the given scenario.

4. Training and testing

4.1 Training

To facilitate the implementation of the training phase, we adopted the pseudocode outlined in the paper [1][8]. In the subsequent sections, we will thoroughly analyze each step of this process, providing detailed insights into the methodology employed. Throughout this examination, we will refer to the specific parameter values utilized, which are conveniently summarized in *table* [1] and *table* [2]. This comprehensive review will offer a clear understanding of the training methodology employed.

4.1.1 Training implementation

The training process begins by initializing the *replay memory* with *random transitions*. To efficiently utilize the memory, we employ an index that functions as a **circular queue**. This enables us to continuously fill the replay memory by removing the oldest transition whenever a new one is added. Once the replay memory reaches its maximum capacity, it operates in a cyclical manner, ensuring that the most recent transitions are always stored for training purposes.

After that, we proceed to initialize the *Q* and *Q_target* network. Both of them are initialized with *random weights*, ensuring an initial exploration of different action-value estimates. At the start of each episode, we perform the *initialization* of the state. To do so, we decided to experiment with two different approaches: **random initialization** and **rest position initialization**. For *random initialization*, we set a random *angle* for the joint (or joints in the case of the double pendulum) and random *velocities*. This approach allows for a wide range of initial conditions, introducing variability into the learning process. In contrast, *rest position initialization* involves setting the pendulum(s) in a stable equilibrium state with no movement. This means that the pendulum(s) remain motionless and in a state of balance at the beginning of the episode. As we will observe later on, this particular method faces challenges in generalizing the completion of the task. In fact, instead of acquiring a broader understanding of the problem, the method tends to focus on learning how to achieve the task from a specific initial state. In other words, the learned policy may not effectively adapt to different starting conditions, limiting its ability to generalize across the entire state space. This insight highlights a potential limitation of the method, underscoring the importance of further exploration and potential enhancements to overcome this constraint. By implementing both random and rest position initialization methods, we can compare and analyze the differences between them during the training and testing phases. This examination provides insights into the impact of the initial conditions on the learning process and the resulting performance of the trained agent.

In our implementation, we employed different values for the *number of time steps* in each episode, depending on whether we were simulating the single or double pendulum. For the single pendulum scenario, we utilized 100 time steps per episode. This means that the simulation of the pendulum's dynamics and the agent's decision-making process occurred over a span of 100 discrete time intervals within each episode. In contrast, for the double pendulum scenario, we increased the number of time steps to 400 per episode. This extended time frame allowed for a more detailed simulation of the complex dynamics of the double pendulum system and provided the agent with additional decision-making opportunities throughout the episode.

Furthermore, our algorithm incorporates an **ϵ-greedy strategy** with **exponential decay** to determine the action selection at each time step. Initially, we initialize the **exploration probability**, denoted as ϵ, with a value equal to 1. This encourages extensive exploration during the early stages of training and promotes a thorough exploration of the state-action space to gather valuable information about the environment. As the algorithm progresses through episodes, the exploration probability ϵ gradually decreases. This exponential decay schedule reduces the emphasis on ex-

ploration and shifts the focus towards **exploitation** in later episodes. By *exploiting* the learned knowledge and experience gained from previous episodes, the algorithm becomes more adept at selecting actions that minimize costs.
Upon selecting an action, we execute it within the environment and we observe the associated cost and the resulting next state. Thanks to this, we create a new tuple *(x(t), u(t), c(t), x(t+1))* where *x(t)* represents the current state, *u(t)* denotes the chosen action, *c(t)* signifies the observed cost, and *x(t+1)* represents the resulting next state. This tuple is then stored in the replay memory, preserving the sequential order of the experienced transitions.

Following the aforementioned steps, we extract a **batch of transitions** from the replay memory every 4 steps. With this batch, we proceed to calculate the **target Q-value**, denoted as y, as we can see in the procedure outlined in the provided pseudocode [8]. In our case, the output of the neural network is a vector of values, with each value corresponding to a specific action that can be taken in the given scenario. Once we have obtained the *target Q-value*, we initiate the *gradient descent step*, utilizing the previously introduced **loss function** which quantifies the discrepancy between the predicted *Q-values* and the *target Q-values*. In addition to updating the *Q network*, we also update the weights of the *Q_target network* periodically. Specifically, we synchronize the weights of the *Q_target network* with those of the *Q network* every 2500 steps.

4.2 Testing & Evaluation of the System

After the training phase, we proceed to the **testing** and **evaluation** phase to assess the performance of our system. During this phase, we evaluate the trained model's ability to bring the pendulum to an upright position across *different scenarios* and *starting states*. We analyze various metrics such as *average cost-to-go* incurred during the task as well as the *state* and *control trajectories*. The different configurations, which capture different settings and parameters used in the training and evaluation process, are presented and summarized in table [2].

Upon analyzing the results of **run number 1** and **run number 2**, both conducted on a single pendulum starting from the *rest position* but with different cost functions, we observed a lot of similarities in the obtained outcomes. This observation is supported by the evidence presented in the videos 1 and 2 and graphs labeled as 3 and 4.
In both cases, we observed that the pendulum successfully reached the upright position by employing a strategy of utilizing momentum to propel itself (as we can clearly see in the videos). However, upon closer examination, it became evident that the policy implemented by the system was unable to precisely maintain the pendulum at a *0-degree angle*. Instead, there was a slight tilt. When it comes to the average cost-to-go of this two methodologies, they behave in a similar way even though the overall costs that we have are, for obvious reasons, different.

Upon examining the **third run** and **fourth run** (involving a single pendulum starting from a *random position*), it becomes evident from the videos that the pendulum consistently achieves the upright position, regardless of its initial position. If we compare it with the two previous runs, the main difference is the fact the pendulum is able to better maintain its upright position by constantly applying torques to counteract the gravity forces (as shown in the state trajectories in images [5][6]). This was not achieved in the previous runs as well as we did here. This effect may be attributed to the fact that training the pendulum with random initial positions allowed for faster convergence and provided more opportunities to refine the strategy during the final steps.

The double pendulum in **run 5** exhibits good results even thought it didn't converge to the optimal solution. Upon examining the average cost-to-go achieved (image [7]), a consistent increase trend is observed. However, due to time constraints, we were unable to conduct extensive testing for this scenario. Consequently, the double pendulum is able to propel itself with some manoeuvres to the upright position; nevertheless, it has yet to acquire the ability to maintain stability in the upright position and it keeps spinning. We think that, if we keep on training the structure for more episodes, the system will be able to achieve the desired result.

5. Conclusions & Future improvements

To summarize, in this project we have discussed about the problem of bringing a pendulum to an upright position by using the *Deep Q-Network* algorithm. We introduced the system dynamics and highlighted the differences between the single and double pendulum scenarios. Moreover, we analyzed the performance of this system, we obtained good results in single pendulum scenario where we were always able to achieve the desired objective, whereas for the double pendulum, the system wasn't able to reach the optimal policy. However we think that if we extended the training time even further we should be able to achieve such results.

When considering potential enhancements to improve the overall performance of the system, an idea worth exploring is the introduction of a final cost to the objective. This addition would further penalize the system if, at the final time step, it fails to reach the desired goal position. Another approach that could be implemented is the incorporation of a penalty to discourage excessive usage of high torque by the system. This penalty would serve as a constraint, limiting the system's tendency to rely heavily on large torques.

Bibliography

[1] S. T. Melrose Roderick, James MacGlashan, "Implementing the deep q-network," *arXiv:1711.07478v1 [cs.LG] 20 Nov 2017.*

6. Images & Graphs

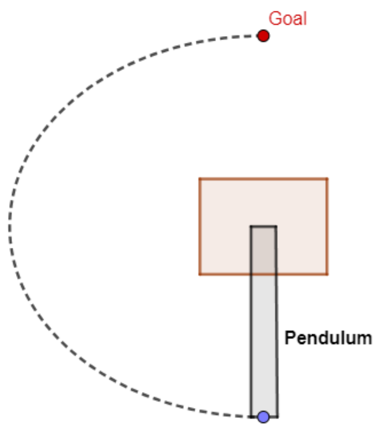

Figure 1: Representation of the swing-up problem for the pendulum.

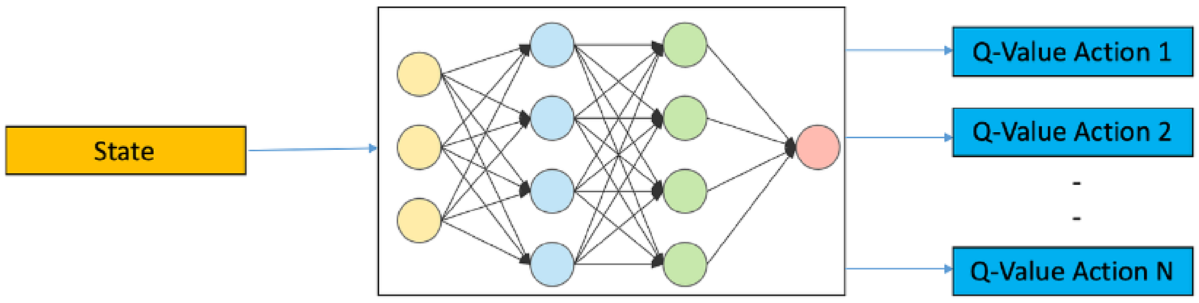

Figure 2: Representation of the structure of the *DQN*.

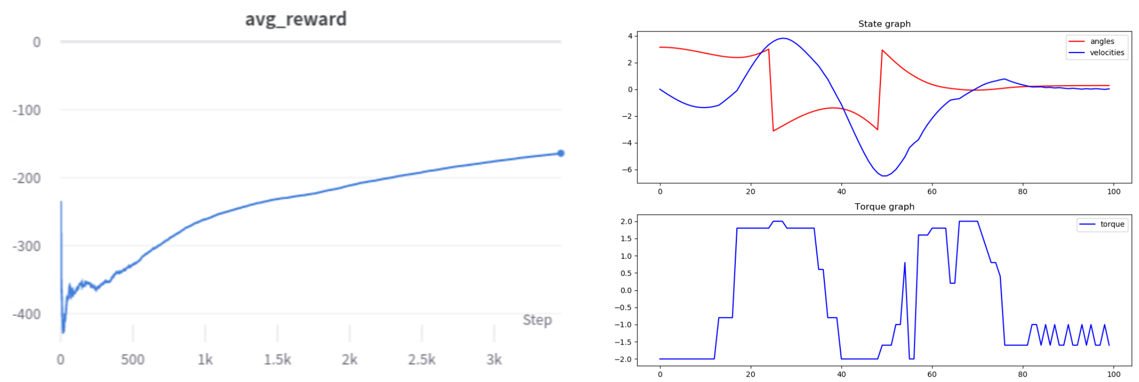

Figure 3: Graphs of the average cost-to-go, state and control of single pendulum using the first cost function and starting from rest position (run 1).

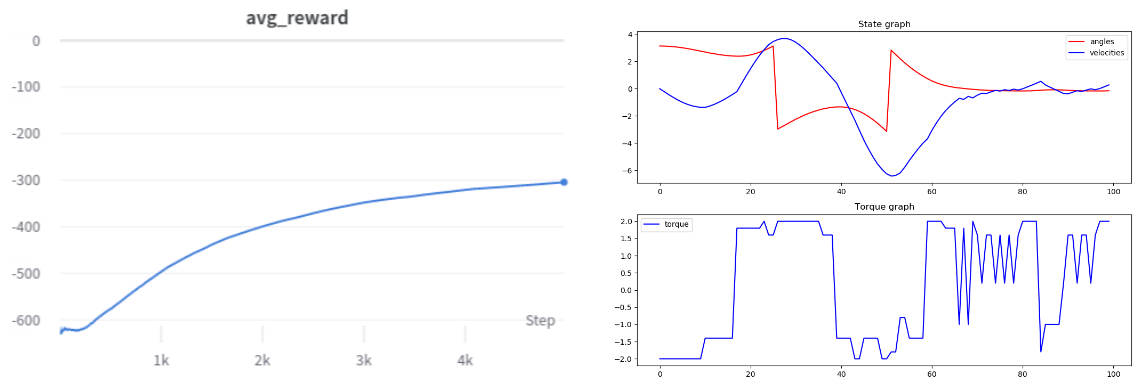

Figure 4: Graphs of the average cost-to-go, state and control of single pendulum using the second cost function and starting from rest position (run 2).

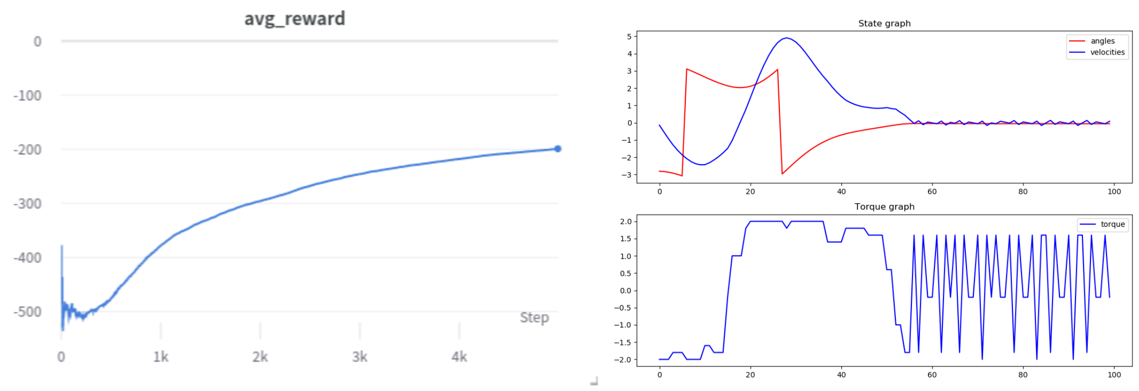

Figure 5: Graphs of the average cost-to-go, state and control of single pendulum using the first cost function and starting from random position (run 3).

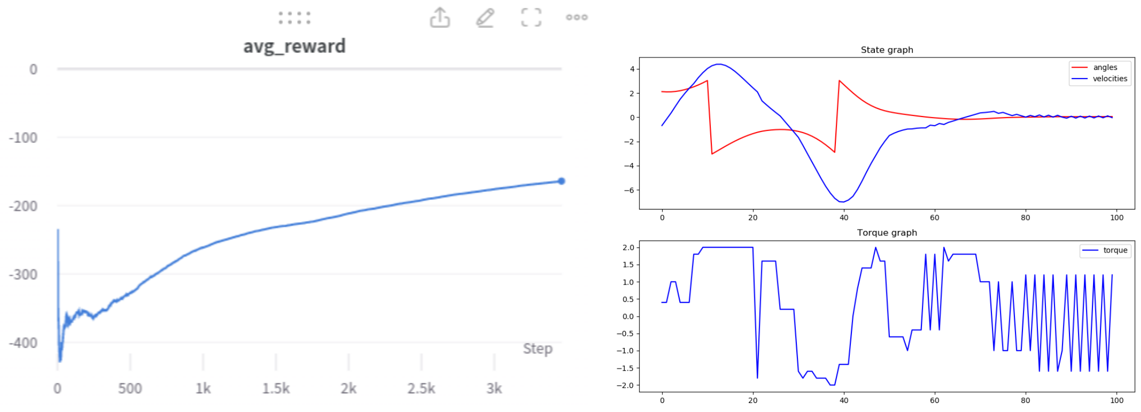

Figure 6: Graphs of the average cost-to-go, state and control of single pendulum using the second cost function and starting from random position (run 4).

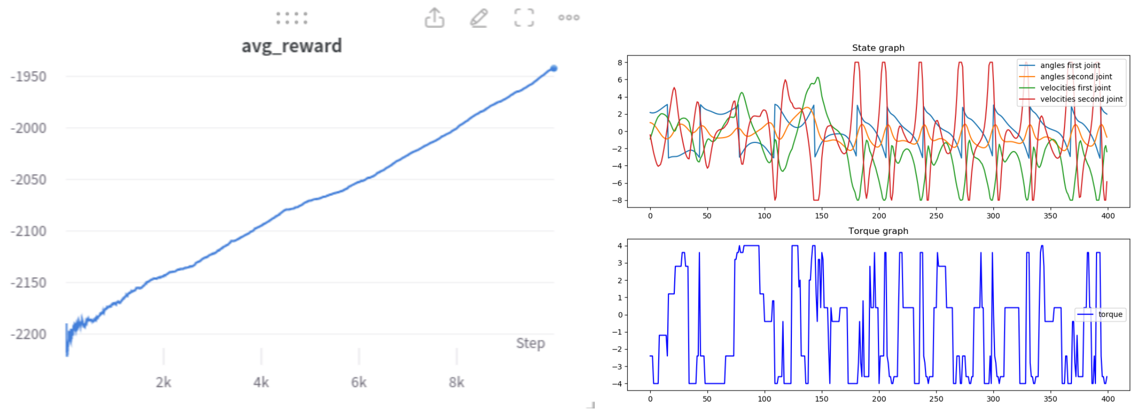

Figure 7: Graphs of the average cost-to-go, state and control of double pendulum using the first cost function and starting from random position.

Table 1: Parameters table

Parameter	**Values Tested**
Replay Memory Size	$10000, 100000$
Learning Rate	0.001
Batch Size	64
Episodes	$5000, 10000$
Episode Length	$100, 400$
Initial ϵ	1
Final ϵ	0.1
ϵ-decay	$0.001, 0.0001$ per episode
Discount rate	0.99
Q-target Update ratio	every 2500 steps
Initial Position	random - rest

Algorithm 1 Deep Q-learning with experience replay

```
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 sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in the emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the weights $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    end for
end for
```

Figure 8: Representation of the pseudocode to train the DQN.

Table 2: Run configuration table

#run	Pendulum Type	# Episodes	Reward	Initial Position	Training Time
1	Single	5000	First	Rest position	200 minutes
2	Single	5000	Second	Rest position	200 minutes
3	Single	5000	First	Random position	200 minutes
4	Single	5000	Second	Random position	200 minutes
5	Double	10000	First	Random position	1400 minutes