Optimization Based Robot Control - Assignment 3

Marco Perini, *229203*
Erik Mischiatti, *233242*

***Abstract*—This paper explores the application of Deep Q Network (DQN), a combination of deep learning and reinforcement learning, to solve the classical control problems of the pendulum and double pendulum. Traditional control methods like PID have shown some success in these problems. However, DQN offers potential in handling high-dimensional inputs that are challenging for traditional methods. By training an agent using DQN, we aim to stabilize and optimize the control of the pendulums. The results of this study provide insights into the effectiveness of DQN compared to traditional control methods for pendulum systems.**

I. Introduction

Deep Reinforcement Learning, an extension of Reinforcement Learning (RL), combines the foundations of traditional RL with the power of deep neural networks. In Reinforcement Learning, problems are typically framed as Markov Decision Processes (MDP) consisting of states (S), actions (A), transition probabilities (P), rewards (R), and a discount factor (gamma). The Agent interacts with the environment, guided by a policy, to maximize the cumulative reward.
To solve optimization problems in Reinforcement Learning, the Bellman equation is widely used. It describes the optimal behavior of the Agent and forms the basis for dynamic programming techniques. Dynamic programming algorithms, such as value iteration or policy iteration, are employed to iteratively update the value function, which represents the expected cumulative reward in a particular state.
Deep Reinforcement Learning builds upon this framework by incorporating deep neural networks as function approximators. By leveraging the power of deep neural networks, Deep Reinforcement Learning algorithms, such as Deep Q-Networks (DQNs), can handle high-dimensional input spaces, such as images or raw sensor data. DQNs approximate the Q-function, estimating the expected cumulative reward for taking a specific action in a given state. Through techniques like experience replay and target networks, the DQN learns to accurately estimate optimal Q-values.
This integration of deep neural networks allows DQN algorithms to address complex and nonlinear relationships between states and actions, enabling them to tackle real-world problems effectively. Also the applications span various domains, including video game playing, robotics, and autonomous driving, where they have achieved impressive results.

II. Deep Q-Learning

General RL problem is formalized as a discrete time stochastic control process where an agent interacts with its environment, starting in a given state and taking an action at each time step. Actions are selected based on a policy, π. As a consequence of each action, the agent obtains a reward $r_t \in R$ and the state and actions are updated.
At this point, it is possible to define the transition probability of each possible next state $s_(t+1)$ as $P(s_(t+1)|s_t, a_t)$, with $s_(t+1)$ $s_t \in S$ and $a_t \in A$. This framework can be seen as a finite Markov Decision Process (MDP), defined in the Section II-B.

A key concept related to MDPs is the Q-function, Q^π, that defines the expected future discounted reward of taking action a in state s and following policy π thereafter.

$$Q^\pi(s,a) = E_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a] \tag{1}$$

where $\gamma \in [0,1]$ is the discount factor that determines the importance of future rewards. So from that, according with the Bellman equation, the optimal Q-function, Q^*, is defined as:

$$Q^*(s,a) = c(s,a) + \gamma min Q^*(s',a') \tag{2}$$

Obtained the optimal Q-function, the optimal policy π^* can be defined as:

$$\pi^*(s) = argmax_a Q^*(s,a) \tag{3}$$

Also, to improve the stability of the treaning and learning processes we use 2 different approaches: First of all the target Q-function, $\hat{Q}$, is introduced. It is defined as:

$$\hat{Q}(s,a) = r + \gamma max_{a'} \hat{Q}(s',a',\omega^-) \tag{4}$$

where r is the reward obtained by executing action a in state s and s' is the next state and ω^- are the weights of a target network that are updated less frequently than the weights ω of the Q-network. So the target action-value function is updated periodically, through a stochastic gradient descent (SGD), to match the current estimate of the action-value function.

$$V(\omega) = \mathbb{E}[(c(s,a) + \gamma min_{u'} \hat{Q}(s,a,\omega) - Q(s,a,\omega))] \tag{5}$$

The second help come from the experience replay. It is a technique that stores the agent's experiences at each time step in a data set called replay memory. The replay memory

contains tuples of the form (s_t, a_t, r_t, s_{t+1}) and is used to train the Q-network. The replay memory is updated at each time step by adding the latest experience tuple to the memory and removing the oldest tuple from the memory. The Q-network is trained by randomly sampling a batch of experience tuples from the replay memory and performing a stochastic gradient descent step on the loss function. These also help to decorellate the transitions and makes the learnig process more stable.

Another important feature related to the value function or action value function is the **Function Approximation**. It refers to the process of finding an approximate representation of a target function based on a limited set of input-output data or observations. The goal of function approximation is to construct a model or algorithm that can predict the output values for new input values within a certain range.
There are many function approximators, but for the DQN typically it's used a Neural Network.
The decision to employ a function approximator within the framework of DQN is motivated by the acknowledgement that the state-action-value function can be exceedingly complex to represent using a lookup tablel, cases where the discete space algorithms don't scale to large systems (e.g. 10^{170} states or continuous state space), and obtaining an analytical solution is often infeasible. Function approximators, such as artificial neural networks (ANNs), possess the ability to learn and approximate intricate functions while also exhibiting generalization capabilities to handle new data. The neural network structure employed in DQN is purposefully designed to cater to the specific requirements of the problem at hand. Typically, the network consists of multiple fully connected layers, with activation functions like rectified linear units (ReLU) incorporated to introduce non-linear properties to the model.

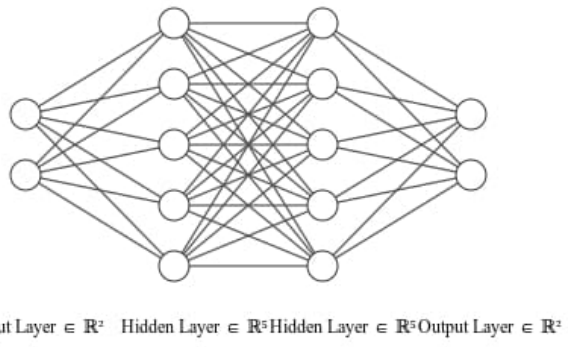

Fig. 1. Example of DNN structure

In the case of DQN, the input to the neural network is the state vector, and the output is a vector of Q-values corresponding to each action. The network is trained to minimize the loss function, which is the mean squared error between the predicted Q-values and the target Q-values.

A. Algorithm

Pseudo code of DQN is shown in Algorithm 1.

Algorithm 1 Deep Q-Learning

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights ω
Initialize target action-value function $\hat{Q}$ with weights $\omega^- = \omega$
for episode = 1, M **do**
  Initialize sequence $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for t = 1, T **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q(\phi(s_t), a; \omega)$
    Execute action a_t in 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 transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{for terminal} \phi_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \\ \hookrightarrow \text{for non terminal} \phi_{j+1} & \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \omega))^2$ with respect to the network parameters ω
    Every C steps reset $\hat{Q} = Q$
  end for
end for

B. Enviroments

In the context of reinforcement learning, the environment refers to the external system or domain in which an agent operates and interacts. It encapsulates the state of the system, the available actions that the agent can take, and the rules or dynamics that govern the transitions between states based on the agent's actions.
More formally, the environment is defined as a Markov Decision Process (MDP), characterized by a tuple (S, A, P, R, γ), where:

- S represents the set of possible states in the environment.
- A denotes the set of available actions that the agent can take.
- $P(s'|s, a)$ is the state transition probability function, defining the probability of transitioning to state s' from state s when action a is taken.
- $R(s, a, s')$ is the reward function, which provides the immediate reward obtained when transitioning from state s to state s' by taking action a.
- γ (gamma) is the discount factor that determines the importance of future rewards relative to immediate rewards.

The environment interacts with the agent in a sequential manner. At each time step, the agent observes the current state, selects an action, and receives feedback in the form of a reward signal and the next state. This process continues until a terminal state or a specified time horizon is reached.

The environment plays a crucial role in RL as it defines the task or problem that the agent aims to solve. It provides the agent with feedback to learn from and determines the consequences of its actions. The design and modeling of the environment are essential for setting up the reinforcement learning problem and formulating appropriate reward functions and state transition dynamics that align with the desired objectives.

In our case we have two different environments, both continuous in the states and discrete in the actions. Moreover they are both deterministic, that's because the dynamics of the system define the transition function.
The design of the enviroments are the following:

- Enviroment 1: Single Pendulum
- Enviroment 2: Double Pendulum (underactuated)

In the following tables, Table I and Table II, it's possible to see the parameters and the hyperparameters of the two enviroments.

TABLE I
PARAMETERS FOR PENDULUM AND DOUBLE PENDULUM

Parameter	**Pendulum**	**Double Pendulum**
State Space	$[\theta,\dot{\theta}]$	$[\theta_1,\theta_2,\dot{\theta}_1\dot{\theta}_2]$
Action Space	(31)$\rightarrow [a_0, a_1, ..., a_{30}]$	(101)$\rightarrow [a_0, a_1, ..., a_{100}]$
Space Range	$[-\pi, \pi]$	$[\pm\pi, \pm\pi]$
Velocity Range	[-8, +8]	$[\pm 8, \pm 8]$
Torque Range	[-2.0, 2.0]	[-15.0, 15.0]
Max Episode Steps	100	200

TABLE II
HYPERPARAMETER FOR PENDULUM AND DOUBLE PENDULUM

Hyperparameter	**Pendulum**	**Double Pendulum**
Train Episodes	1000	1000
Learning Rate (lr)	0.001	0.001
Discount Factor	0.99	0.99
Epsilon	1.0	1.0
Epsilon Decay	0.001	0.001
Epsilon Final	0.01	0.01
Replay Memory Buffer Size	1,000,000	100,000
Batch Size	128	128
Target Network Update	10 episodes	10 episodes

III. RESULTS

1) Single Pendulum: From the following plots it's possible to see the results of the training of the single pendulum. Figure 2 shows the cumulative reward during the training, computed considering 1000 episodes, Moreover, Figure 3 shows the mean squared error between the target and the prediction of the network. We can see that the loss is decreasing, which means that the network is learning, but it doesn't reach a consistent minimum. In the first 100 episodes the loss is decreasing very fast, then it shoots up and then start decreasing more slowly. This is probably due to the nature of the algorithm and to the parameters chosen.

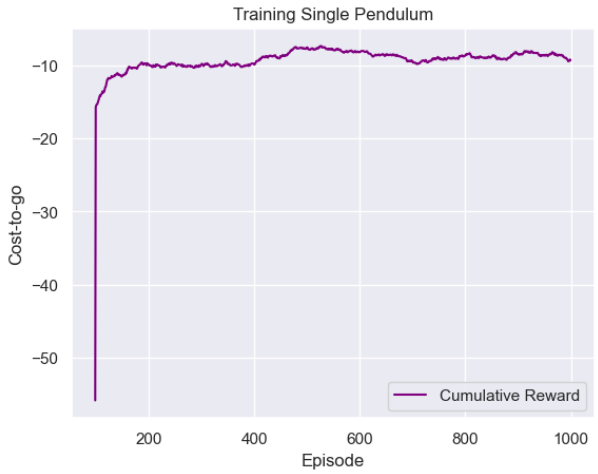

Fig. 2. Cost to go

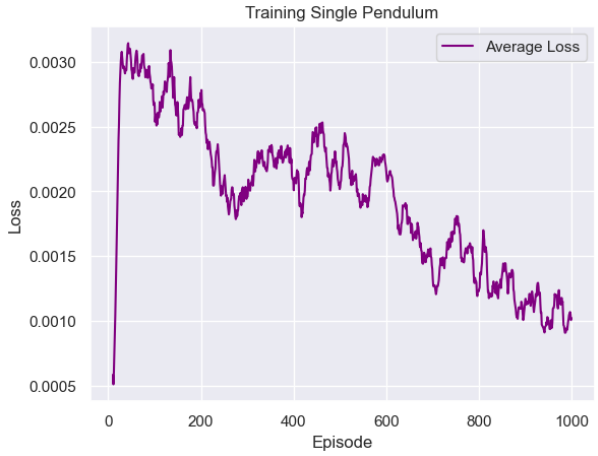

Fig. 3. Loss

The DQN agent has been trained for 1000 episodes and took ≈ 10000 seconds. The high value is due to the training perfomed in a low-end CPU, which could have been improved using a GPU with CUDA enabled
The policy is a description of the behaviour of the agent and tells the agent which actions should be selected for each possible state. The Value Function, in RL, is defined as the cost starting from state s. So for the optimal value function and the optimal policy we have:

$$V^*(s) = max_\pi V^\pi(s) = max_u Q^*(s,u) \tag{6}$$

$$\pi^*(s) = argmax_u Q^*(s,u) \tag{7}$$

The following plots (colormaps) show the value and the policy functions for the single pendulum. We notice a symmetry on the plots which is due to the dynamics of the system, in the attached video it is possible to see the 3D distribution of the functions.

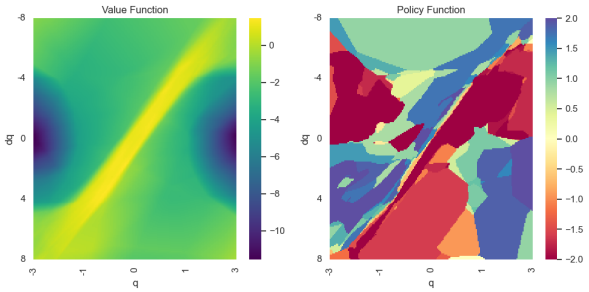

Fig. 4. Colormaps of optimal value and optimal policy functions

Through the following plots, Figure 5,6,7, it's possible to see the trajectories of the single pendulum during the training. The plots show the angle, the angular velocity and the torque applied to the pendulum during the training.

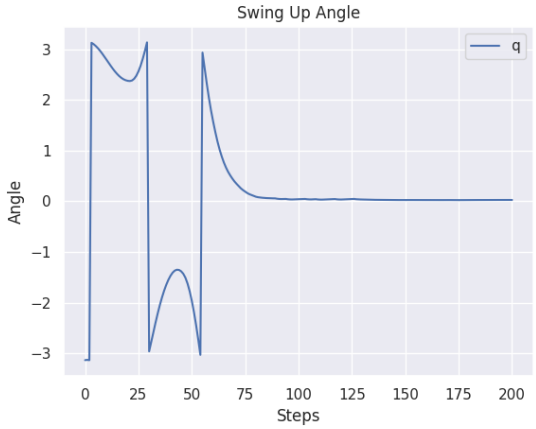

Fig. 5. Position

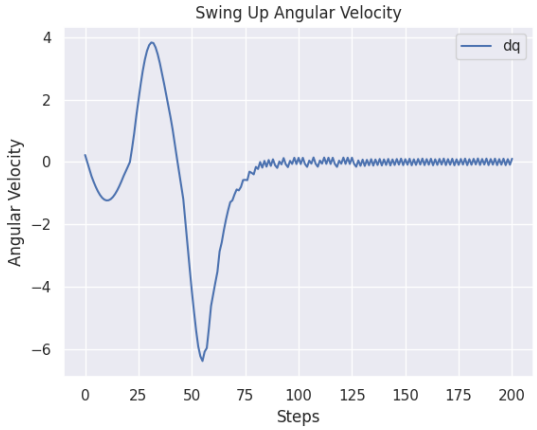

Fig. 6. Velocity

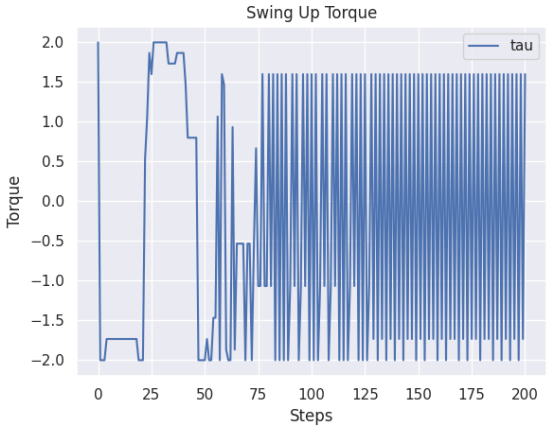

Fig. 7. Torque

2) Double Pendulum: As the previuous section, the following plots show the results of the training of the double pendulum. Figure 8 shows the cumulative reward during the training, computed considering 1000 episodes. The average loss during the episodes presents a better behavior respect to the single pendulum one, which shows a more rapid descent after 700 episodes. From the following plots it's intresting to see the behavior's difference between the 2 joints, the motorized one [1] and the passive one [2]. In the Figure 10 and 12, motor [2] present a pattern that try to follow that one of motor [1]. It is possible to notice that the pendulum does not stay perfectly vertical when balanced, which could be due to the small number of discretization steps of the action space. The training took around 40000 seconds.

From the following plots it's intresting to see the behavior's difference between the 2 joints, the motorized one [1] and the passive one [2]. In the Figure 10 and 12, motor [2] present a pattern that try to follow that one of motor [1]. Morover in Figure 11 the velocity

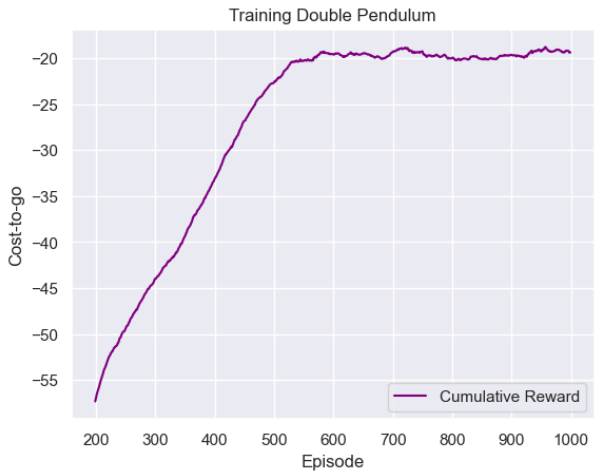

Fig. 8.

Fig. 9.

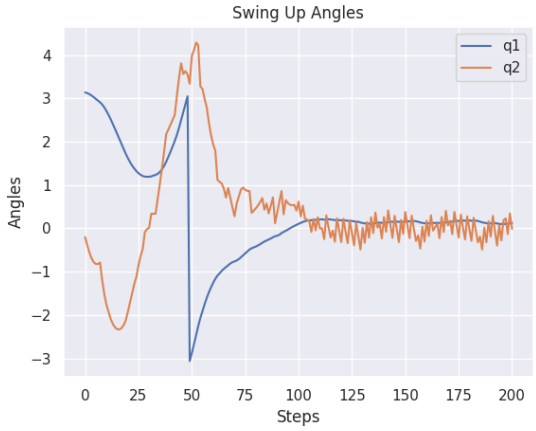

Fig. 10. Position

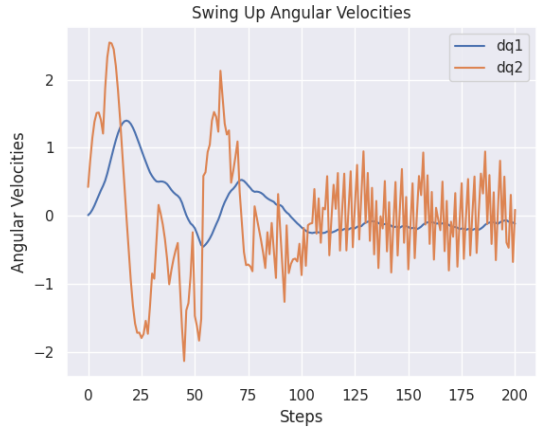

Fig. 11. Velocity

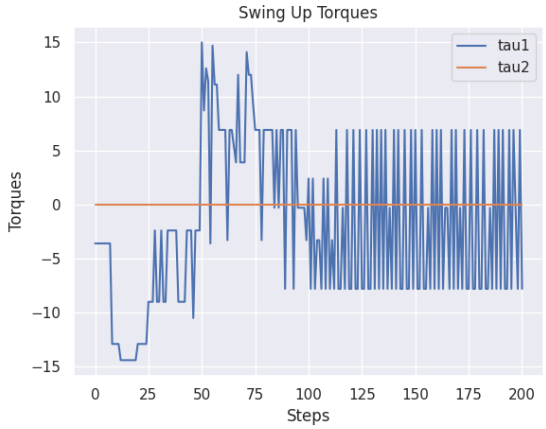

Fig. 12. Torque

IV. Potential Improvements

Overall, the DQN Agent has learnt to solve the swing-up problem both for the single and double pendulum. Many improvements consist of implementing more sophisticated and consolidated reinforcement learning algorithms, such as Double DQN, Dueling DQN, DDPG, and tuning all the parameters in a finer way. Moreover, it would be interesting to train a DQN Agent and then use the weights as soft start for a training of a real-life system.