Deep Q-Learning Neural Network for single and double pendulum control

Mario Tilocca & Zaida Brito Triana
Advanced Optimisation Based Robot Control
Professor Andrea Del Prete
Teacher assistant Gianluigi Grandesso
Department of Information Engineering and Computer Science
University of Trento
mario.tilocca@studenti.unitn.it
zaida.britotriana@studenti.unitn.it
Project Repository: Github page

Abstract—**Reinforcement learning (RL) offers reliable and powerful algorithms to identify optimal controllers to use with nonlinear systems where the dynamics is unknown.**
Within this framework, the aim of this project is to implement a Deep Q Network to control strategy for a single and a double pendulum, where the joints angle and velocities are continuous and torque is discretized. The pendulum environment was developed and controlled within an implementation made in *Python*.

Keywords—**Reinforcement Learning, Deep Q-Learning, Q-Learning, ϵ greedy policy, Stochastic Gradient Decent, Cost to go**

Acronyms - Reinforcement Learning (RL), Deep Q-Learning (DQN), Stochastic Gradient Decent (SGD)

I. INTRODUCTION

Despite the fact that Reinforcement Learning was born in the computer science community and not in the control engineering one, its methods have shown great success in recent years in various and diverse fields [1]
Reinforcement learning (RL) is a model-free framework for solving optimal control problems stated as Markov decision processes (MDPs) [2].
The goal of this project was to develop of an optimal policy which allowed a single and a double pendulums to achieve a stand-up pose. For this purpose, a Deep Q-Network was employed. The DQN algorithm was developed by DeepMind in 2015 using the already existing Q-Learning function as starting framework and approximating it into a NN to allow a greater versatility in a wider range of scenarios. In particular, the powerful concept of experience replay was one of the cornerstones in the development of the DQN. The DQN algorithm was first applied to the gaming industry and in particular it immediately showed promising results by being able to solve a wide range of Atari games, in several instances achieving superhuman results, by combining reinforcement learning and deep neural networks at scale. In this project a Deep Q Learning (DQN) approach was used to model the environment of a pendulum in a hybrid configuration. The states are continuous, while the control inputs supplied to the revolute are discretized.

A. Problem Formulation

With the above discussed premises, Reinforcement Learning (RL) is a mathematical framework that allows to solve optimal control problems.
In this paper, a DQN algorithm based strategy is used to control the torque inputs of a N-Pendulum with the goal of achieving a stand-up-pose. In this paper, different NN architectures and an $\epsilon -greedy$ based strategies are presented, thus a research question can be formulated as such:

Which NN architecture and strategy represent a reliable and yet time efficient approach to achieve a handstand maneuver in a N-Pendulum ?

II. METHODS

As discussed in the previous section, in Reinforcement Learning agents learn to perform actions in an environment so as to maximize a reward. However, as it will be explained in more detail below, the implementation of this algorithm has been carried out mirroring some of the optimal control terminology and approach, so the agent instead of maximizing a reward is seeking to minimize a cost solving a *non-convex* problem.
Thus an overview of the classic RL terminology and the terminology used in this paper can be found in Table I.

TABLE I: Classic RL terminology & terminology used in this project

Reinforcement Learning	Terminology used
State s	State s
Action a	Action a
Environment	Environment
Reward	Reward
Maximize	Minimize
Value function	Cost to go
Optimal Value function	Value function

Prior to the explanation of the proposed solutions approaches, it is important to clarify the mathematical concepts behind algorithm developed and Reinforcement Learning, whose definition can be explained as the following:

Markov decision processes

Markov decision processes gives a way to formalize sequential decision making, meaning, it gives the framework for RL problems, whose enviroment is assumed to be fully observable and follows **Markov property**. Markov property states that in a Markov process, *future is independent from the past given the present* as mathematically expressed in Eq.1. Thus meaning that in order for an agent to accurately predict the next state, it is sufficient to know the current state. This is particularly useful as it aids in reducing a problem complexity and in turn reducing the computational time when compared to more classic *look-up-table* approaches. The mathematical equation representing the above mentioned concept can be found below.

$$\mathbb{P}(x_{t+1} \mid x_t) = P(x_{t+1} \mid x_1 ... x_t) \tag{1}$$

For the subsequent explanations, is important to highlight that in classical Reinforcement Learning approaches it is assumed that the state space is finite, which implies that all **State Transition Probabilities** $\mathcal{P}_{xx'}$ can be gathered in a **State transition matrix** $\mathcal{P}$.
Since the system developed in this project is **deterministic** $\mathcal{P}$ only contains 0 and 1.
Before going into the details of the particular method employed in this project, it is useful to explain the basic operation of an MDP, which is a central part of the functioning of any RL approach.

In RL, the agent selects an action from a given state, which results in transitioning to a new state, and the agent receives rewards based on the chosen action. This process happens sequentially which creates a trajectory that represents the chain/sequence of states, actions, and rewards. The goal of an RL agent is to maximize the total amount of rewards or cumulative rewards that are received by taking actions in given states.

Bellman Optimality Equation

The Bellman Optimality Equation which is a concept strictly linked to *Dynamic-Programming* in optimal controlplays a crucial role in the field of RL, and in this case it is used to find the *optimal action-value function* $q*$, also called optimal Q-function. A detailed explanation of the equation can be found below.

$$\underbrace{\text{New}Q(s,a)}_{\text{New Q-Value}} = Q(s,a) + \underset{\text{step size value}}{\alpha} \Big[\underbrace{R(s,a)}_{\text{Reward}} + \underset{\text{Discount rate}}{\gamma} \overbrace{\max Q'(s',a')}^{\substack{\text{Maximum predicted reward, given} \\ \text{new state and all possible actions}}} - Q(s,a)\Big]$$

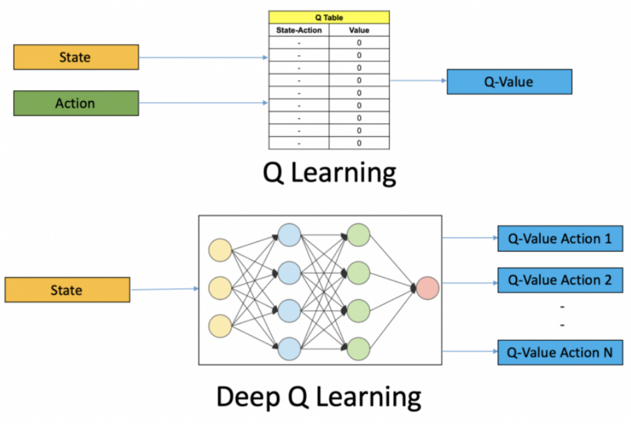

Fig. 1: Q-Learning vs DQN visual difference representation [3]

The aforementioned Bellman Optimality equation equation is used to determine the q∗ that gives the largest expected return achievable by any policy for each possible state-action pair. Being the *policy* π a function that maps a given state to probabilities of selecting each possible action from the given state.

Q-Learning

Q-Learning is strongly based on the just explained Q-function, thus it is a model-free based algorithm, meaning that the dynamics of the system are ignored, conversely from families of methods such as Reactive Control and Optimal Control where the knowledge of the dynamics of the robot is exploited in the majority of algorithms. Its main goal is to map for the agent the expected reward of each action at every step. This would create a so called "cheat sheet" for the agent, as the agent would be able to exactly know which action it should perform. Furthermore it is important to note that Q-learning is considered an off-policy algorithm. Therefore the agent learns from actions that are outside the current policy, such as taking random actions. However, one of the main drawbacks for Q-Learning is represented by the *scalability* of the algorithm. As shown in Fig. 1 the Q-Learning algorithm 'cheat-sheet' length is directly proportional to the total number of states and actions per state. Thus a significant computational challenge occurs when the number of states and actions increases.

Deep Q-learning

Deep Q-Learning exploits the advantages of NN to approximate the Q-value function. By contrast with Q-Learning as it is shown in Fig. 1 the sole input is the state and the Q-value of the possible actions is the output of the created NN. To be more precise this means that each output node represents an action and the the value inside it represents the actions Q-value. In the same fashion as for Q-Learning the agent employs the Bellman Optimality Equation to update the so called main and target networks accordingly. This is achieved using the concept of *experience replay*, where after a fixed number of steps the main network samples and is trained upon a batch of data representing the past experiences. Furthermore the main network weights are regularly passed to the target network. The concept of *experience replay* is the act of storing and replaying states such as for instance (the state, action, reward, next state) that the RL algorithm is able to learn from. This is particularly useful because it allows to exploit the previously mentioned concept of Off-Policy, which is a characteristic of Q-Learning. The pseudo-code of the algorithm of DQN whose logic was used in this project is depicted in Algorithm 1.

ϵ **greedy strategy**

One of the main challenges of RL is the so-called 'Exploration-Exploitation' dilemma. On one hand it is important that the agent is able to explore and improve the best cost to go as in this case we are solving a non-convex problem but conversely it should exploit the knowledge it has already gained. In Q-learning, the agent would ideally always choose the optimal action, however it is important not to stop exploration too soon as there might be better local minima. A good trade-off for this problem is represented by the epsilon-greedy action selection, where the agent uses both exploitations to take advantage of prior knowledge and exploration to look for new options. In the Epsilon-Greedy Exploration strategy, the agent chooses a random action with probability ϵ and exploits the best known action (cost to go) with probability $1 — \epsilon$. It is particularly important that the agent in particular during the initial episodes of the training has the possibility to explore, while the longer the NN is trained the more it should exploit the knowledge previously acquired. For this reason it is essential to implement an exponential decay of ϵ in order to achieve this.

A. System Model

The above mentioned DQN was used to train a NN to provide the control in the form of applying torque to the pendulum revolute joints. Several assumptions regarding the model had to be made. The pendulum has only 1 degree of freedom (DOF) and it is in free space. Furthermore it is assumed there is no Gaussian noise in the model. The environment of the pendulum has continuous states and discrete controls, thus a so called hybrid environment was created where the controls are discretized to a predefined number of steps, while the states are continuous. As it was previously mentioned, instead of maximizing for a reward, the system is modeled in order to minimize the cost.

B. Implementation details

The first part of the proposed solution is represented by the creation of pendulum environment where the control is discretized according to a fixed number of steps nu, while the state is continuous. Once this was achieved, the main effort was put into the creation of valid NN architecture and its relative hyper parameters. One of the main challenges in the creation a NN is defining a suitable hidden-layers architecture which would result in an accurate mapping of the Q-function without overfitting. In this case it was decided to opt for 2 types of NN architectures respectively, using a variable 3 and 4 hidden layers architecture. Their specific architetcure is presented in Tab. II. It was decided to keep the numbers of hidden layers low as increasing it would drastically negatively effect the computational time needed for the training.

TABLE II: NN architectures proposed in the solution

layers name	3 hidden layers	4 hidden layers
input	state dimension (nx)	state dimension (nx)
h1	64	16
h2	64	32
h3	64	64
h4	-	64
output	Joints * disc steps (nu)	Joints * disc steps (nu)

Furthermore, it is worth mentioning that the proposed solution employs a 2 NN pattern. One is used for computing the control inputs of the joints and one is used for computing the network targets with temporal difference *TD(0)*. Such targets are simply fixed values of the weights of the first NN and they are updated at a considerably lower rate compared to the first network. The reason for which this was done was that by keeping the target fixed for a longer period of time it allows the SGD to converge to the target before it is updated. This in turn results in a stabilization of the algorithm as the target the algorithm is aiming for is not moving too quickly. Moreover it is worth noting that every time a better cost to go is found the model is saved. As well as at regular intervals. In order to achieve a well designed NN, several hyper parameters need to be tuned following an heuristic fashion. The ones used in this project are presented in the table III.

TABLE III: Hyper-parameters

Symbol	Explanation
SAMPLING_STEPS	Steps to sample from replay buffer
BATCH_SIZE	Batch size sampled from replay buffer
REPLAY_BUFFER_SIZE	-
MIN_BUFFER_SIZE	Minimum buffer size to start training
UPDATE_Q_TARGET_STEPS	Steps number to update Q target
EPISODES	Number of training episodes
MAX_EPISODES_LENGTH	-
QVALUE_LEARNING_RATE	Learning rate of DQN
GAMMA	Discount factor
EPSILON	Initial exploration probability of eps-greedy policy
EPSILON_DECAY	Exploration decay in exponential decreasing
MIN_EPSILON	Minimum of exploration probability

Once the NN has been trained in order to visualize the results, a greedy policy based rederization function is used, where the inputs are the weights of the trained model.

The algorithm of Q learning found in [4] has been adapted to the matter of this project to show the approach taken in the system developed.

Algorithm 1 Deep Q-Learning implementation in the project

```
Given:
Pendulum environment with $N$ number of joints                    ▷ *
Hybrid Pendulum with discretized controller (torque)
Display environment                                               ▷ *
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialize sequence
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $u_t$
        Otherwise select $u_t = min_u Q^*(x(s_t), a; \theta)$
        Execute action $u_t$ in environment and observe cost $l_t$ and next state in the environment $x_{t+1}$
        Store transition $(x_t, u_t, l_t, x_{t+1})$ in $D$
        Sample random minibatch of transitions $(x_t, u_t, l_t, x_{t+1})$ from $D$
        Set
        $y_t = \begin{cases} \text{l}_t \text{ for final } x_{j+1} \\ \text{l}_t + \gamma\ min'_u Q^*(x(_{j+1}), a'; \theta) \text{ for} \neq \text{final } x_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(x_t, u_t, l_t, x_{t+1}))^2$
    end for
end for
```

III. RESULTS

The first trainings were executed on a 1-DOF pendulum. The hyper parameters tuning used for these trainings are presented in the second column of Table IV. As it can be depicted in Fig.(2) and Fig.(3) the initial exploration episodes yielded significantly different *best cost to go*, this has to be addressed by the high stochasticity of the ϵ greedy strategy at the early stages of the training. However it can be observed how the *best cost to go* converges to similar values and this behavior is mirrored on the average cost to go, as in the later stages of the training the ϵ greedy strategy behaves more deterministically.

TABLE IV: Hyper-parameters Tuning by trial and error

Parameter	V. 1	V. 2	V. 3
SAMPLING_STEPS	4	4	4
BATCH_SIZE	64	64	64
REPLAY_BUFFER_SIZE	50000	50000	50000
MIN_BUFFER_SIZE	5000	5000	5000
UPDATE_Q_TARGET_STEPS	60	200	50
EPISODES	750	5000	1000
MAX_EPISODE_LENGTH	100	100	100
QVALUE_LEARNING_RATE	0.05	0.9	0.09
GAMMA	0.9	0.9	0.9
EPSILON	1	1	1
EPSILON_DECAY	0.01	0.01	0.01
MIN_EPSILON	0.01	0.01	0.01

Testing the behavior of the system using the weights obtained, for 3 hand 4 hidden layers architectures resulted in a successful and prolonged hand-stand maneuver by the 1-DOF pendulum as depicted in Fig(4). It is important to note that in

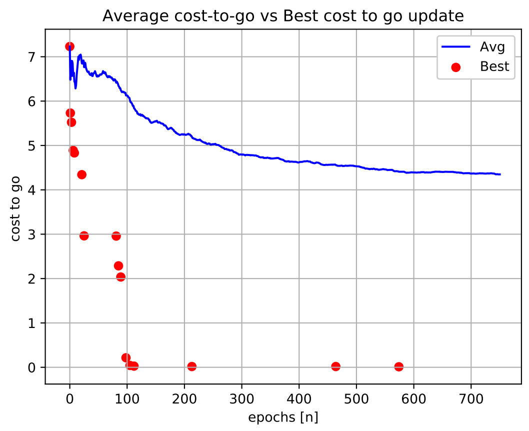

Fig. 2: 1-DOF pendulum 4 hidden layers training results

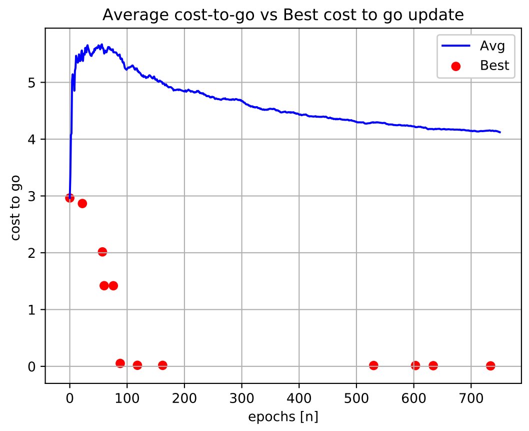

Fig. 3: 1-DOF pendulum 3 hidden layers training results

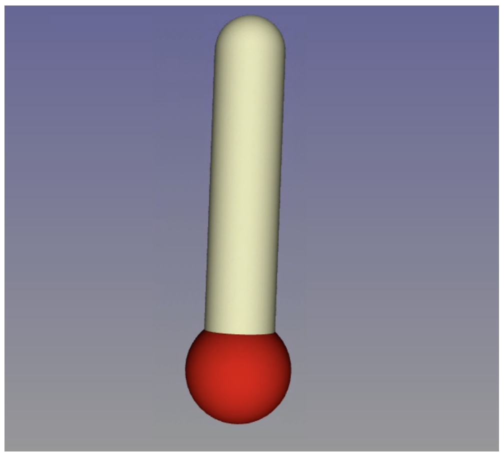
Fig. 4: 1-DOF pendulum handstand maneuver

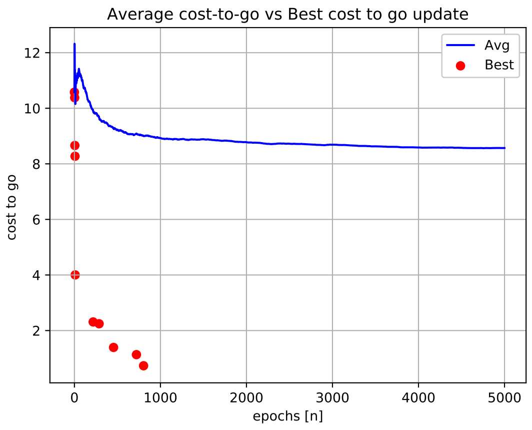

Fig. 5: 2-DOF pendulum 4 hidden layers training results trial 1

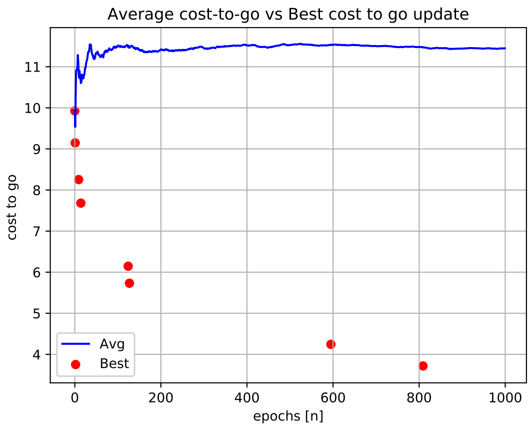

Fig. 6: 2-DOF pendulum 4 hidden layers training results trial 2

this case, the training time for the DQN was around 1.5 hours. Consequently the training was focused on a 2-DOF pendulum. The first trial was implementing a 3 hidden layers DQN using the hyper-parameters which were successful for the 1-DOF pendulum, except for the episodes which were drastically increased to 5000. This resulted in an increased training time of 9 hours circa. The results obtained resulted in the pendulum being able to achieve an handstand maneuver, however for a very limited amount of time. Therefore it was decided to tune the hyperparameters according to the thrid column of Table IV and use a 4 hidden layers architecture. The training time was significant in this case with almost 8 hours. However despite the fact that during the later stages of the training the average cost to go decreased towards 8 as depicted in Fig(5), which was a similar value and behavior with the one obtained with the 3 hidden layers, the results were unsatisfactory as the pendulum was not able to perform a successful handstand maneuver. Therefore it was decided to adjust the hyperparameters following the scheme presented in fourth column of Table IV. The main difference was that the number of episodes, which here are defined as epochs in the figures, was reduced and the learning rate was increased. As it can be seen in Fig(6) the results were not satisfactory. Hyperparameters tuning represents a difficult challenge in particular when the training times of the system are high as in these cases. Another factor to take into account is that large learning rates results might result unstable training and tiny rates result in a failure to train, thus finding a trade-off is essential. Moreover it needs to be considered that the best training results might not be in the last episodes per se, but for instance at 80% or 70% of the total episodes. Therefore given the results obtained it might have been beneficial to add additional hyperparameters, such as revolute joints velocities thresholds or joints angle thresholds. Based on those the experience replay could be optimized. Thus, the initial 3 hidden layers model, despite not being able to achieve a prolonged handstand was the only one which was able to partly achieve it as it is shown in Fig.(7).

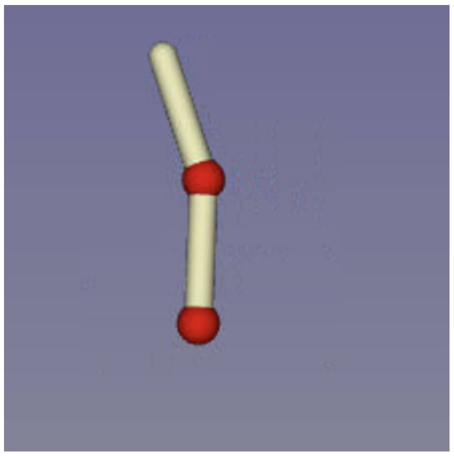

Fig. 7: 2-DOF pendulum handstand maneuver

IV. Conclusions

In this paper, a Deep Q Network approach to develop an optimal solution to achieve a stand-up pose for both a single and double pendulum has been presented. As it has been shown in the previous sections, finely tuning the hyper-parameters is crucial for obtaining successful results. Furthermore the computational time required by the 2-DOF pendulum represented a challenge in finely tuning the hyper-parameters. However when considering the 1-DOF pendulum, both of the NN architecture tested yielded similar results and the same has observed for the 2-DOF pendulum. While the most important part for achieving successful model weights is a combination of finely tuning several hyper-parameters, most notably the learning rate of the Q-function, the episodes length, the Q target update rate and the random exploration probability ϵ decay. Moreover in case of future research, the results highlighted how the introduction of additional hyper-parameters might result in beneficial results.

References

[1] Reinforcement learning for control: Performance, stability, and deep approximators
[2] Markov Decision Processes: Discrete Stochastic Dynamic Programming
[3] Deep Q-Learning
[4] Jafari, R., Javidi, M. Kuchaki Rafsanjani, M. Using deep reinforcement learning approach for solving the multiple sequence alignment problem. SN Appl. Sci. 1, 592 (2019).
[5] Notes and materials provided in the class.Advanced Optimisation Based Robot Control. Professor Andrea Del Prete. Teacher assistant Gianluigi Grandesso