OPTIMIZATION BASED ROBOT CONTROL
FINAL PROJECT

Alessandro Grassi 221224 Samuel Bortolin 221245

Introduction:

The aim of the project is to develop a neural network to evaluate the Q function of each control given the current continuous state.
To accomplish this goal we used the following techniques, based on the papers that used them to accomplish this goal, (Mnih et al., 2015) and (Roderick et al., 2017):

- ε-greedy: this strategy aims to have a tradeoff between exploration and exploitation, the concept is to choose a random control with a linearly decreasing probability ε and a greedy control with probability 1-ε.
- DQN: continuous state Q function approximation, it is a neural network that evaluates the state-control value function from a continuous state for each discrete control.
- Experience replay buffer: training a network with consecutive data breaks the assumption of i.i.d. samples, so the data of the episodes are stored in a buffer and sampled randomly in order to stabilize the training.
- Fixed Q targets: the neural network used to compute the TD-error is updated only after a fixed amount of episodes, without this at each optimization step we get closer to the target but we also move the target.
- Adam optimizer: an adaptive learning rate optimizer that downscales the model parameters accumulating an exponentially decaying average of the gradients, uses it for computing a velocity term and also uses bias correction terms to improve the convergence.

Code documentation:

- Pendulum Environment: Simulate the dynamics of a pendulum with continuous joint space and discrete control space.
    - nj: Number of joints
    - nu: Number of discretizations for joint torque
    - umax: Maximum torque

- Neural Network: the function approximator of the Q-value we used is a multilayer perceptron with the following structure:
    - input layer: nx units
    - layer 1: 16 units, nonlinearity: *selu* (Scaled Exponential Linear Unit)
    - layer 2: 32 units, nonlinearity: *selu*
    - layer 3: 32 units, nonlinearity: *selu*
    - layer 4: 16 units for 1 joint, 32 units for 2 joints when nu<9 and 64 units for nu=9, nonlinearity: *selu*
    - output layer: nu**nj units

- Update function:
    - Input:
        - **x_batch:** tensorflow tensor of dimension (*nx*,*batch_size,*1)
        - **u_batch:** tensorflow tensor of dimension (*nu*,*batch_size,*1)
        - **cost_batch:** tensorflow tensor of dimension (*batch_size,*1)
        - **x_next_batch:** tensorflow tensor of dimension (*nx*,*batch_size,*1)
        - **discount:** discount factor of the TD error
    - Return value: None
    - Description: performs the optimization step on the *Q* network.

- Experience Buffer: a class that represents the experience buffer where to store the tuples <state, control, cost, next_state>. The buffer is implemented using a deque (double ended queue) with a maximum length. It has two methods:
    - *append*: add an element at the end of the deque.
    - *sample*: choose elements from the buffer at randomly chosen indexes.

- Agent: a class that represents the controller. It chooses the control following an ε-greedy policy. Its principal method is *step*:
    - Input:
        - **exploration_prob:** the probability to choose a random control
    - Return value:
        - **cost:** the cost for following the chosen control at the given state
    - Description: a random number is sampled from a uniform distribution in the range [0,1]. If the number is less than *exploration_prob* a control is chosen randomly, otherwise the control chosen is the one that minimizes the Q-value according to the *Q* network. The environment is then updated with the reached state and the tuple <state, control, cost, next_state> is added to the *Experience* buffer.

- Train function:
  - Input:
    - **agent**: the agent that performs the simulation
    - **exploration_prob_start**: the starting probability of choosing a random control
    - **exploration_decreasing_decay**: how much the exploration probability is decreased at each step
    - **min_exploration_prob**: lower bound of exploration probability
    - **n_episodes**: number of episodes on which perform the training
    - **max_episode_length**: length of each episode
    - **discount**: discount factor of the TD error
    - **update_dqn_steps**: steps after which the *Q* network is optimized
    - **batch_size**: size of the training batch
    - **replay_start_size**: minimum number of elements in the *experience buffer* before starting the training
    - **sync_target_steps**: steps after which the *Q_target* network is synchronized with *Q* network
    - **n_print**: number of episodes after which print the status of the training
    - **use_evaluate_greedy_policy**: defines whether the *Q* network is evaluated after each update or not
  - Return value:
    - **h_ctg**: discounted cost to go history
    - **network_weights**: the last optimized weights or the best ones according to evaluate_greedy_policy if the relative flag is set to True
  - Description: performs the training procedure of the DQN. It is composed of two nested for loops, one for each episode and one for each step. At each step, the tuple <state, control, cost, next_state> is stored in the buffer. Every *update_dqn_steps* steps the *Q* network is updated according to the *Q_target* network by randomly sampling a batch of size *batch_size* from the *experience replay* buffer. Every *sync_target_steps* steps the *Q_target* network weights are updated with to the *Q* network weights.

- Evaluate Greedy Policy function:
  - Input:
    - **env:** the environment that performs the simulation
    - **max_episode_length**: max length of the episode
    - **discount:** discount factor used in the TD error
    - **angle_discretization:** how many discretizations of the joint angles for doing different simulations
  - Return value:
    - **evaluated cost to go:** the sum of the cost to go of the simulations
  - Description: initially the starting states are computed, each state is equidistant to each other. To optimize the simulation all the states are grouped together inside a batch and forwarded to the *Q* network, then the next states are computed according to the minimizations of the Q-values outputted from the *Q* network and forwarded again in the form of a batch. For each simulation, the cost to go is summed over the simulations and in the end returned.

- Render Greedy Policy function:
    - Input:
        - **env:** the environment that performs the simulation
        - **max_episode_length**: max length of the episode
        - **discount:** discount factor used in the TD error
        - **x0:** the initial state from which to simulate. If not specified the initial state is randomly chosen
    - Return value: None
    - Description: Renders on Geppetto UI a simulation following the policy according to the Q-values outputted from the *Q* network from state x0. This function is used just for testing purposes and showing results.

- Hyper-parameters and the choices we made:
    - NEPISODES = 20000 for two joints and = 10000 for one joint. For two joints there are more weights to be learned and more states to be explored, so it requires more training episodes.
    - MAX_EPISODE_LENGTH = 500 for two joints and = 200 for 1 joint. With two joints the swing up maneuver could take more than the 1 joint version because it would need to oscillate before rising to exploit the dynamics of the system.
    - DISCOUNT = 0.99, high discount factor in order to weigh a lot the future.
    - EXPLORATION_PROB_START = 1.0, it starts with 1.0 because we want to maximize exploration at the beginning of the training.
    - EXPLORATION_DECREASING_DECAY = 5e-7 for two joints and = 5e-6 for one joint. The exploration probability decays slower with two joints to let the agent explore more since there are more states and controls.
    - MIN_EXPLORATION_PROB = 0.001, we don't drop to 0 the exploration probability to let the network still explore some states.
    - REPLAY_SIZE = 50000, to ensure independence between samples we used a large replay size.
    - REPLAY_START_SIZE = 50000, we decided to wait for the buffer to be full before starting the training.
    - UPDATE_DQN_STEPS = 10, good trade-off between training frequency and data variability given by the buffer size.
    - BATCH_SIZE = 32, good trade-off between sample used and frequency of update.
    - LEARNING_RATE = 0.001, good value tuned for Adam optimizer.
    - SYNC_TARGET_STEPS = 1000, good value tuned for syncing targets.
    - umax = 4.0 for two joints and = 2.0 for one joint. The torque required by the two joint pendulum is higher than the one joint version.

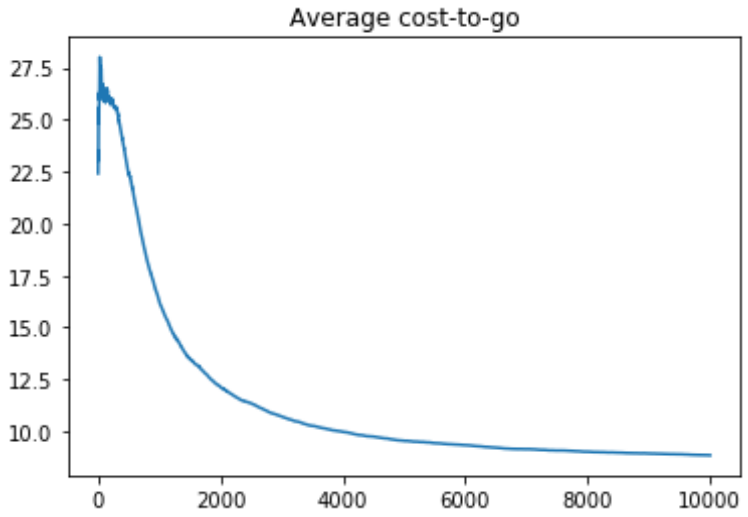
Note: We created the *Q* network as critic and *Q_target* network as target. We used them and the Adam optimizer as global variables in order to avoid problems passing around the tensorflow objects.

One Joint Solution:

We tried 2 main approaches:

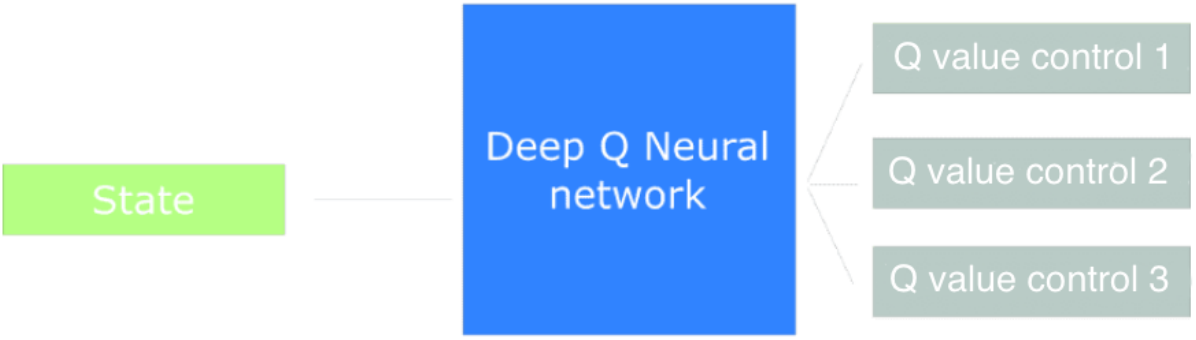
- A Deep Q network with in input the state of the robot (position and velocity for the joint) and also the one-hot encoded control and output the Q-value for the state-control pair: with this approach each state-control pair is forwarded through the network in order to retrieve its Q-value, this method showed little improvement with respect to the next method but it took a lot more time to train: linear with respect to the number of controls since in the update we had to do a forward into the net for each control.

Schema of the Deep Q network with single output for a state-control pair.

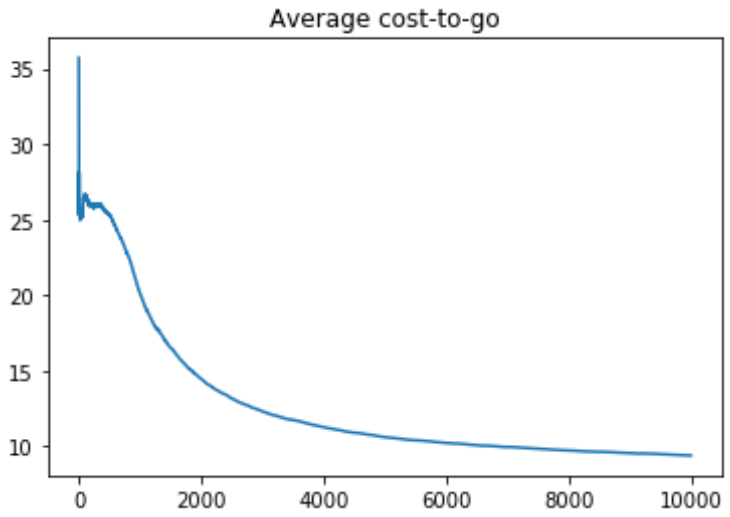


Average cost-to-go for Deep Q network with single output for a state-control pair applied to 1 join with nu=11.

- Deep Q network with input the state of the robot (position and velocity for the joint) and output Q-value for each control: this model takes as input just the state of the robot and outputs all the Q-values for all the controls, from them it is possible to choose the control with the minimum Q-value in just one forward into the network.



Schema of the Deep Q network with multiple outputs for a state.



Average cost-to-go for Deep Q network with multiple outputs for a state applied to 1 join with nu=11.

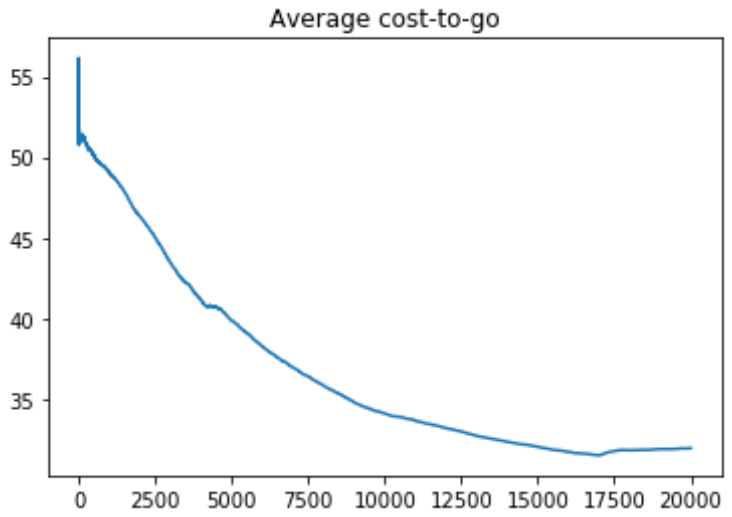
Two Joint Solution:

We keep the same Deep Q network with input the state of the robot (position and velocity for the 2 joints) and output Q-value for each control and we adapted the code in order to be able to work with multiple joints. So the output passes from *nu* to *nu**nj* in order to consider as possible controls all the combinations of the controls of the 2 joints. We have also doubled the maximum torque to *4.0* for the joints in order to make it work.

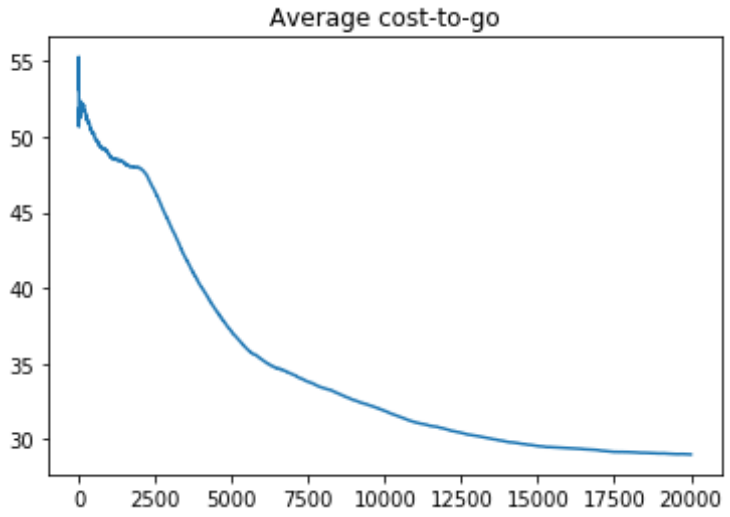
The training procedure requires a lot of time (about 10 hours without GPU) since we increased both the number of episodes and the maximum length of the episodes, but this is required in order to ensure an initial sufficiently large exploration and then a sufficiently large exploitation in order to improve the quality of the results. Probably with even more exploration and training the results could still improve, as suggested by the average cost-to-go that is continuously decreasing as the number of iterations increases.

As result we obtained that: with 5 control discretizations it tends to work almost always, with 7 works in most of the cases and with 9 only sometimes works. Probably since the outputs increase exponentially with the number of joints the network is not able to learn well with a higher discretization. Seeing the plots it can be noticed that with 7 control discretizations the average cost-to-go seems lower with respect to 5 and hypothetically could work better with a longer training even if the number of outputs is much bigger.

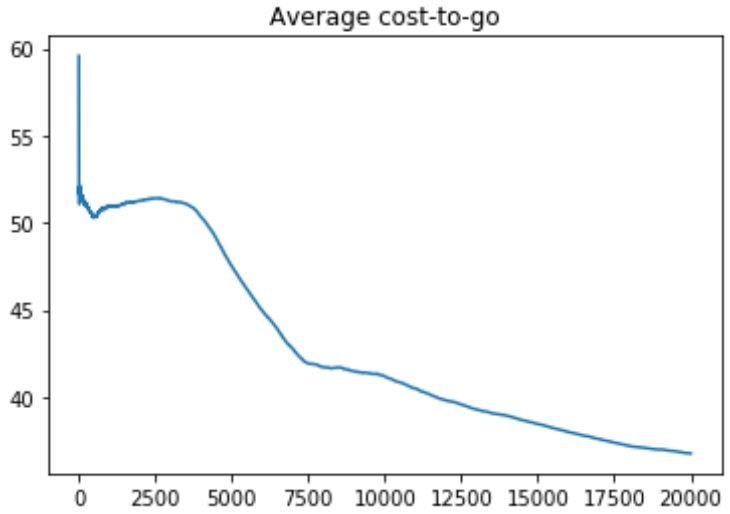
In the *videos* folder there are presented some video clips of results applying to the pendulum the Greedy policy given by the network after training.



Average cost-to-go for 2 joins with nu=5.



Average cost-to-go for 2 joins with nu=7.



Average cost-to-go for 2 joins with nu=9.

Tried Approaches:

This section contains all the approaches that we tried and then discarded due to poor performances.

- Double deep Q network: when training instead of taking the minimum among the Q-values outputted by the target network, the position of the Q-value to take from the target network is the one associated with the minimum among the Q-values outputted by the actor network. This solution promises to solve the moving target problem.

- Prioritized experience buffer: the idea is to add a higher priority to tuples in the *experience buffer* that have a higher error between the prediction and the target.

- Dueling Q network: this framework separates the state value *V* and the advantage *A* from the *Q*-value, we can get the actual *Q* by this relationship: *Q* = *V*+*A*. The agent is now able to evaluate a state without caring about the effect of each control from that state. Meaning that the features that determine whether a state is good or not are not necessarily the same as the features that evaluate a control. And it may not need to care for controls at all. It is not uncommon to have controls from a state that do not affect the environment at all. The mean advantage is used to be able to backpropagate.

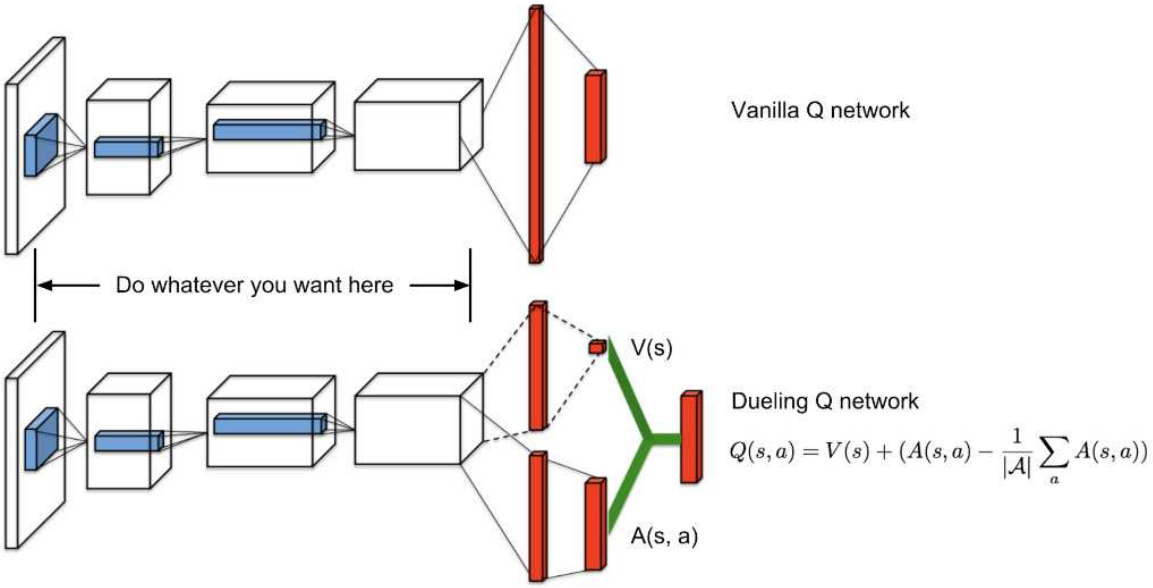


Image taken from (Wang et al., 2016).

Conclusions and Future Work:

We successfully implemented DQN and made it work for a 1 joint and 2 joints pendulum. The results obtained are quite good and the network is able to approximate in a good way the Q-value when the possible controls are restricted.
By looking at the videos it is clear that the pendulum never reaches the perfect vertical position, it always hangs a bit to the left or to the right. One possible cause of this problem is that TD(0) is biased, but not the only one since it is known that neural networks are biased and tend to overfit if not properly trained. A possible solution could be to implement a trade off between Monte Carlo and TD(0), e.g. TD(n) or a TD(λ) approach.
It is clear that the method implemented needs some optimization in order to reduce the training time, a possible solution could be to parallelize the episodes in order to optimize the training and reduce the time needed.

References

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015, 2 26). Human-level control through deep reinforcement learning. *Nature*. https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf

Roderick, M., MacGlashan, J., & Tellex, S. (2017, 11 20). Implementing the Deep Q-Network. *30th Conference on Neural Information Processing Systems (NIPS 2016)*. https://arxiv.org/pdf/1711.07478.pdf

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016, 2 5). Dueling Network Architectures for Deep Reinforcement Learning. *arXiv*. https://arxiv.org/abs/1511.06581