UNIVERSITY OF TRENTO

Department of Industrial Engineering
Master's degree in Mechatronics Engineering

DEEP Q-NETWORK

Third assignment report

Mattia Pettene 239145
Gabriele Poggesi 233220

Course of Optimization-based robot control

Academic Year 2022 - 2023

Contents

Introduction

In this assignment we are asked to implement a Deep Q-Network (DQN) algorithm to find the optimal control policy for a single and a double pendulum swing-up problem. The double pendulum has only one actuated joint.

Being DQN a reinforcement learning problem, it is based on Markov Decision Process: the goal of planning a MDP is to find a policy $\pi : S \rightarrow A$, a mapping from states to actions, that minimizes the expected future discounted reward when the agent chooses actions according to π in the environment [1]. A policy that minimizes the expected future discounted reward is an optimal policy denoted by π^*. A key concept related to MDPs is the Q-function Q^π, that defines the expected future discounted reward for taking action a in state s and then following policy π thereafter. According to Bellman equation, Q^*, the Q-function for the optimal policy π^*, can be expressed as:

$$Q^*(x,u) \;=\; l(x,u) + \gamma \min_{u'} Q^*(f(x,u),u') \tag{1}$$

where γ is the discount factor that defines how valuable near-term rewards are compared to long-term ones. Given Q^*, the optimal policy π^* can be easily recovered by greedily selecting the action in the current state with the highest Q-value:

$$\pi^*(s) \;=\; \text{argmax}_a \; Q^*(s,a) \tag{2}$$

The DQN properties that characterize it with respect to the classic Q-learning algorithm are the following: a deep convolutional neural network architecture is used for Q-function approximation, mini-batches of random training are used rather then single-step update on the last experience, older network parameters are used to estimate the Q-values of the next state.

Algorithm implementation

The DQN algorithm to find the optimal control policy to solve the problem proposed has been implemented using Python language. As it is possible to notice looking at the code, the project is divided into several files. First of all, the dynamics of the problem is implemented in `pendulum.py`. This model was subsequently simplified in `dpendulum.py`, discretizing the joint torque in n steps between $-uMax$ and $uMax$, that is the maximum value of admissible torque. The most important part is implemented in `classes.py`:

it has been chosen to divide the used functions in classes, that are collected in this file. The core of the entire project is `main.py` that is lunched after setting up the parameters of the problem. The pseudo-code of the algorithm implemented is reported below.

Algorithm 1 Deep Q-Learning with experience replay

```
if Training $True$ then
    Load environment
    for Looping over $NEPISODES$ do
        Reset the environment
        Initialize action-value function $Q$
        Initialize the target action-value function $Q\_target$
        Initialize $replay\_buffer$ to store experiences (S.A.R.S)
        for Looping over $MAX\_EPISODE\_LENGTH$ do
            Loading states form environment
            Apply greedy strategy (prob = $\epsilon$) to find optimal control input:
            random control or take $\text{argmax}_u Q$
            Collect $next\_state$ and $cost$ from a step input simulation
            Store the results in $buffer\_replay$
            Collect randomly from $buffer\_replay$ some experiences
            Update weights of $Q$ with a Stochastic Gradient Descent method
            Update $Q_target$ weights every *c-step*
            Update cost-to-go
        end for
        Store cost-to-go of every episode
        Store Simulate system with the optimal control of this episode
    end for
else Neural network already trained
    Load saved model
end if
Evaluate V-function and policy form the resulting Q
```

1 Optimal control problem formulation

Optimal control problem are made up of a function that needs to be minimize over states and inputs and other equations to be satisfied. Those are related with the dynamic of the system, the initial conditions and some path constraints. In this assignment we are required to study a single and a double pendulum with DQN which is a composed by different methods developed in reinforcement learning environment. We are going to use both OC and RL terminology to be consistent with the papers and the material we used to develop the most part of the code.
The system dynamics are known for both the cases. In order to use neural network in a simplified way, their dynamics have to be discretized to obtain a finite set of possible states. The results can be expressed as:

$$x^+ = f(x)$$

The rewards from time t to ∞ are summed all together to obtain the total reward. The particularity is in the way they are added. The future rewards are discounted by the parameter γ smaller than 1. In this way the total reward will not reach infinite and it takes account of the uncertainty about the future.

$$G_t = l_t + \gamma\ l_{t+1} + \gamma^2\ l_{t+2} + \dots$$

This concept can be applied to all the states creating the definition of the value function V(x). This function can be interpreted as an idea of how good is to be in state x.
Markov Decision Processes can accept finite set of control inputs. If they are present, the total reward is no more only the results of the states but also of the controls. The action-value function Q(x,u) is introduced to take account of both the contribution. What this represents may be expressed as how good is to be in state x and take input u.
The last useful dimension is the optimal policy that can be directly obtained by minimizing the action value function $Q^*(x, u)$. The main issue is to get this last element. This problem is defined by Bellman optimality equations where the value function and the action-value function are related:

$$V^*(x) = max\ Q^*(x, u)$$
$$Q^*(x, u) = l(x, u) + \gamma\ V^*(f(x, u))$$

This formulation does not present a closed solution because it is a non linear problem due to the maximization.

In this assignment, the MDP is known and the goal is to find the optimal value function and the optimal policy, categorizing it as a dynamic programming control problem. This type of algorithm starts with a random policy π_0 and cycles in a loop where it evaluates the value function with the given policy. Then it improves the policy acting greedily with respect to the previous result ($\pi_{k+1}(x) = \underset{u \in U}{\operatorname{argmax}}\ l(x,u) + \gamma\ V^{\pi_k}(f(x,u))$).
A very important property is that this algorithm always converges to the optimal policy π^* thanks to the maximization. At every step it increases or, at least, stays as it is.
The code developed, instead of going directly to the solution, use ϵ-greedy strategy to try different input possibilities that may be worse in the first step but better in the final path. To do so, with probability ϵ a random control is chosen and for the remaining probability $(1-\epsilon)$, it is obtained from maximizing the action-value function. To optimize the problem and to avoid wasting time, the random research should be done at the beginning of the exploration because it is the moment where it does not know anything. Indeed, after many steps it might have already found a good sequence of actions and there are is no more need to continue the exploration task. Therefore, ϵ has been set with a big value and reduced up to a minimum threshold that is also given with a second parameters.
Having a lot of possible states to analyze makes discrete space algorithms very difficult to handle for their dimensions. To lighten the problem, value function approximation has been used. In particular it has been used a function that, given the states and the control, a singular scalar output is elaborated with some parameters. In this way, the work is translated on scalar values and on a vector of weights that are the results of the neural network training.
To change these parameters, the neural network approach usually operate with Stochastic Gradient Descent method. With this solution the actual and the next states are necessary with the control to calculate the Q_value and the $Q\_target\ values$ for all the experience considered. After computing the square of the differences, gradients with respect to the weights are computed and the results are used to update the neural network parameters.
The data utilized by this procedure upgrade the weights of the global approximating function, so they should came from different episodes in time. The batch learning approach, indeed, requires to collect various data and sample from them randomly, only once the system has stored enough data to lower the possibility of temporal correlation. When the experience replay buffer is full, at every step a new experience substitute an older one before sampling some of them to upgrade the weights of Q function. The Q_target weights,

instead, are updated every c_steps.

As reported above, to simplify the problem and to improve the convergence ability of the algorithm, the input torque is discretized: the range between the maximum and the minimum value of the input u is divided in $nDisc$ values of possible torque that can be applied to the actuated joint. Once the action is computed by the algorithm, it is necessary to "transform" it into a valid torque value, so the corresponding discrete step in which is divide the range, will be chosen. The action value returned by the algorithm is a positive number between zero and $nDisc$. That become an actual torque using the function `c2du`. A schematic example of this point is reported in figure 1:

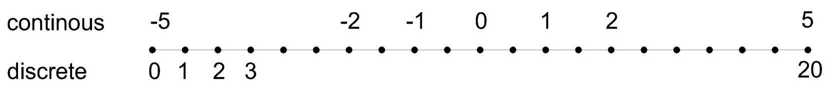


Figure 1: Schematic representation of the torque discretization.

Then, in order to simulate the system with the policy obtained by the algorithm, it is necessary to turn back to the continuous space using the inverse function `d2cu`.

All the methods exposed in this section, can be used for the single pendulum and also for the double pendulum thanks to the fact that it is under-actuated. Having the model of a full-actuated system, the solution that has been used involves the same analysis of the single pendulum but, when the system has to be evaluated, the second joint receives a null input. In this way, the second joint is always passive.

2 Description of the neural network

As shown above, one of the features of DQN algorithm is using a deep convolutional neural network architecture to approximate the Q-function. A neural network (NN) is a type of machine learning model inspired by the structure and function of the human brain. It consists of interconnected nodes or neurons that process and transmit information, allowing the network to learn and make predictions or decisions based on input data. An example of NN representation is shown in figure 2.

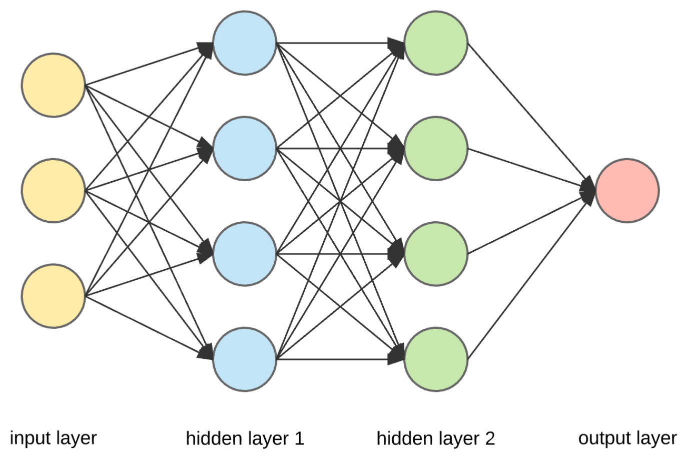


Figure 2: Schematic representation of a neural network.

In this project, the neural network is created using Keras interface, which is based on TensorFlow library. The model used has six layers: input, output and four hidden. In input there is a tensor of three elements containing the discrete states considered, position and velocity, with the control, torque input. Each layer is "dense": in NN terms it means that each node in a layer is connected to every nodes in the following layer, as sketched in figure 2. In this way each node in a layer receives input from every nodes of the previous one. The number of nodes for each layer is reported in table 1.

Layer	$N°$ **of nodes**
First hidden layer	16
Second hidden layer	32
Third hidden layer	64
Fourth hidden layer	64
Output layer	1

Table 1: Number of nodes for each NN layer.

Other important properties that must be defined in a neural network are the activation functions, mathematical functions that takes as input the output of a node and produces an output that will be used as input for the next layer. In our case the activation function is of the type ReLU, Rectifier Linear Unit, defined as:

$$f(x) = \max(0, x) = \begin{cases} x & if \;\; x > 0 \\ 0 & otherwise \end{cases}$$

where x is the output of a node. So this function returns the output itself if its value is positive, zero otherwise: in this way all the nodes are "activated" with non negative values.

3 Hyper and dynamics parameters

The main file begins with the definition of the random seed used to control the randomizations. Using time library it will be completely random but we set it equal to a value of 1000 in order to evaluate the performance of the problem using different hyper parameters.

The hyper parameters can be collected and explained in the table 2.

NEPISODES	Total number of training episodes
MAX_EPISODE_LENGTH	Length of every episodes
NPRINT	The number of episodes before printing what has been obtained up to that cycle
QVALUE_LEARNING_RATE	Alpha coefficient for Q-learning algorithm in upgrading the weights
DISCOUNT	Gamma value used to reduce the importance of future rewards
PLOT_LABEL	Flag to let the algorithm plot or not
EXPLORATION_PROB	Initial exploration probability for ϵ-greedy method
EXPL_DECREASING_DECAY	Used to regulate the exponential decreasing of the exploration probability
MIN_EXPLORATION_PROB	The minimum value that the exploration probability can reach
CAPACITY_BUFFER	Maximum number of experience that the replay buffer can store
BATCH_SIZE	How many experiences are collect from the replay buffer
MIN_BUFFER	Is the minimum number of experiences that have to be collected before starting to collect them randomly in order to update the weights
C_STEP	Number of step to be evaluated before upgrading the weights

Table 2: Hyper parameters defined in the code.

To discretize the system *ndcontrol* and *ndstates* are initialized as the number of how many times their ranges are divided. As a consequence, the evaluations are easier because the algorithm works with a finite set of possible values as explained before with the function approximation shown in the first section.
The last parameters to be defined before starting the neural network are related to some dimensions of the systems themselves used to obtain their dynamics in the discrete time domain. They are listed in the table 3:

nDisc	Number of discretization step for the states
nu	Number of discretization step for joint torque input
vMax	Module of the maximum velocity. In the pendulum it will have a range of [-vMax,vMax]
uMax	Module of the maximum torque. In the pendulum it will have a range of [-uMax,uMax]
noise_stddev	Standard deviation of input noise that can be simulated together with the input

Table 3: Systems parameters used to discretize the systems.

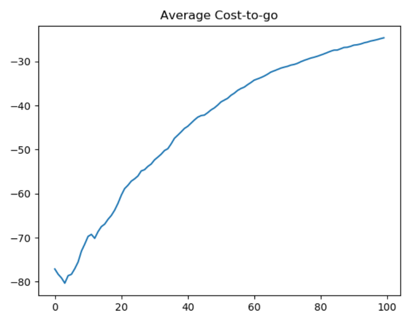
4 Simulations and training results

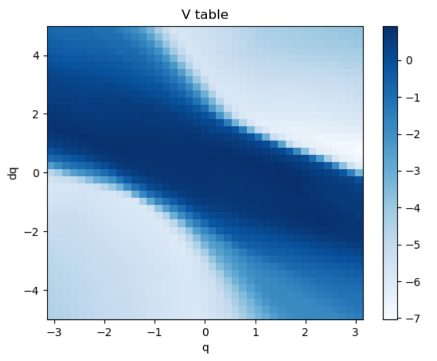
Several simulations have been carried out in order to analyze the contribution of each parameter that characterize the problems, for both the single and the double pendulum. As regards the dynamics, the parameters that have been tuned are the maximum torque $uMax$ $(N \cdot m)$ applied to the actuated joint, the maximum velocity of the joints $vMax$ (rad/s) and the number of discretized torque input $nDisc$. The hyper-parameters of the algorithm that have been changed to observe their contribution to the final behaviour are the batch size $BatchSize$ and the minimum buffer dimension $minBuffer$, whereas the number of episodes $nEpisodes$ and their length $EpisodeLength$ are kept constant.

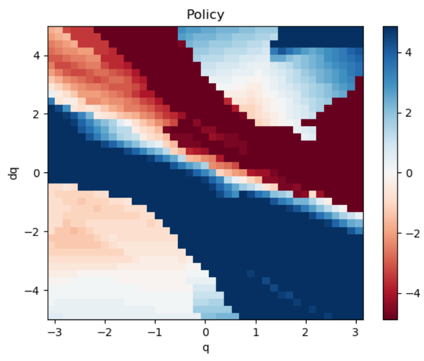
4.1 Single pendulum

Case 1: $uMax = 5$, $vMax = 5$, $nDisc = 42$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$.
The total time taken by the training was 21 minutes, 40 seconds.

This first attempt has been done using plausible values. This case will be used to compare the effect of modifying some parameters and for this reason all the plots are shown:







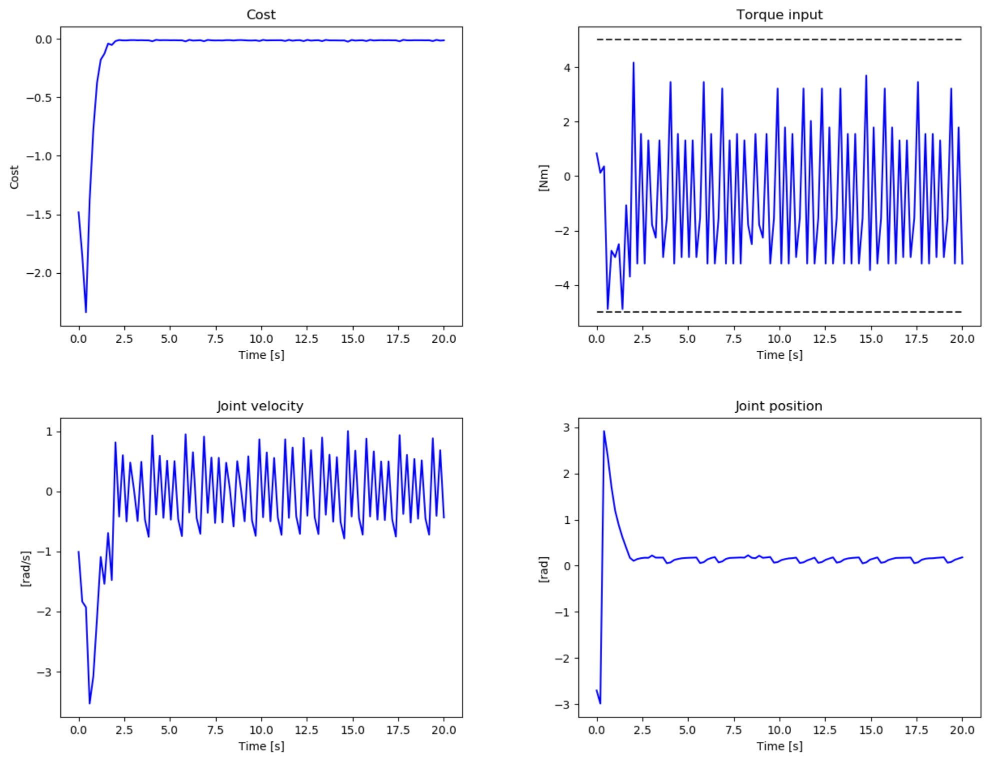


Figure 3: Results from simulating with the optimal control policy obtained with parameters of **case 1**

Case 2: $uMax = 5$, $vMax = 5$, $nDisc = 42$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 200$.
The total time taken by the training was 22 minutes, 18 seconds.

This case considers a different *minBuffer* which change doubly up to 200.

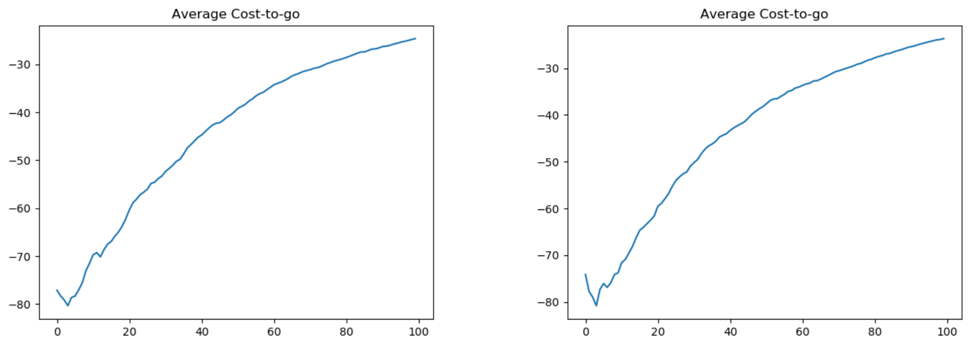


Figure 4: Average cost to go in **case 1**, left, and in **case 2**, right

The only different from the previous case is in the moment when the algorithm starts looking at the experiences to upgrade its weights. The final results, shown in figure 4, do not show this effect because after *nEpisods* of *EpisodeLenght*, they converge to the optimal values. It may be observed if the algorithm would have shown the action-value function after very few episodes. In this way, the delay of starting the updating would have results in a worsening of the situation because it had lost the first update.

Case 3: $uMax = 5$, $vMax = 5$, $nDisc = 42$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 64$, $minBuffer = 100$.
The total time taken by the training was 31 minutes, 38 seconds.

This case considers a different *BatchSize* which change doubly up to 64.

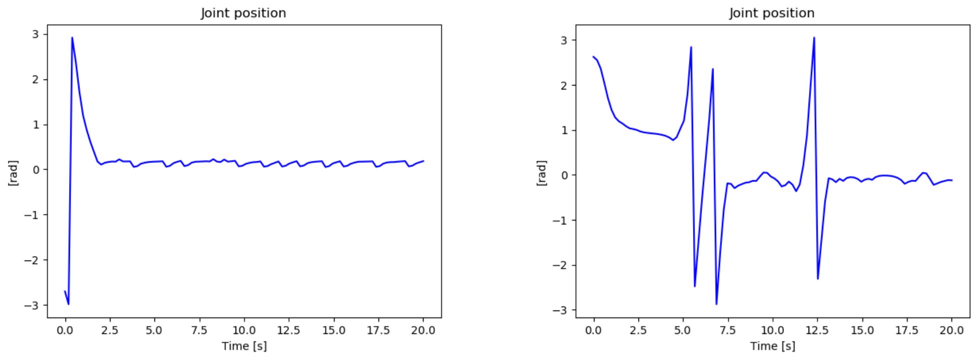


Figure 5: Pendulum position simulated in **case 1**, left, and in **case 3**, right

In this case, the pendulum reaches stability only after 13 seconds. This may have been caused by the temporal correlation between the experiences that the algorithm use to upgrade its weights. With respect to the previous cases, there is more probability to take similar data because the replay buffer contain the same quantity as before.
More data to be elaborated means also more computational effort. This is clearly visible in the computational time that increase from approximately twenty minutes to half an hour.

Case 4: $uMax = 5$, $vMax = 2$, $nDisc = 42$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$.
The total time taken by the training was 21 minutes, 38 seconds.

This case considers a different $vMax$ which is reduced to 2 rad/s.

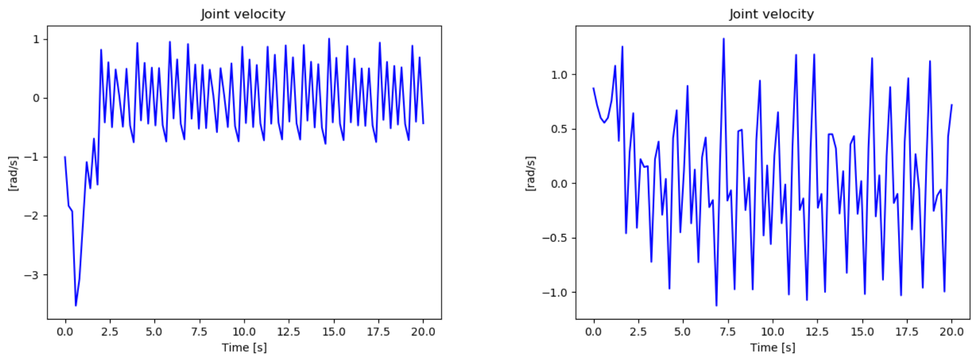


Figure 6: Pendulum velocity simulated in **case 1**, left, and in **case 4**, right

In addition to this analysis, a training with $vMax = 11$ has been launched and, as expected, no improvement has been recorded due to the fact that all the results up to this case have velocities that remain very small. Despite the fact that it can choose higher velocity, the optimal solution remain low. The following step has been to shrink the velocity range as described in this case. From the comparison with case 1 velocity history in figure 6, the pendulum goes directly to the target and remain there as shown in figure 7.

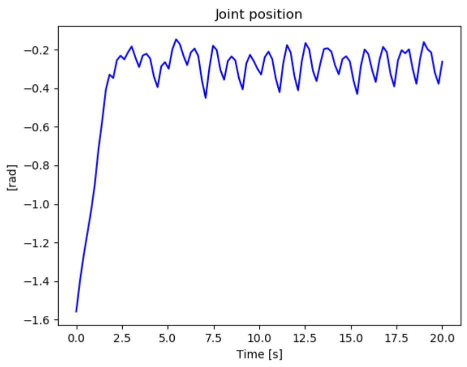


Figure 7: Pendulum position simulated in **case 4**

Although the possible values for the velocity are reduced, the algorithm has been able to find a good solution that bring the system to the target in a small period of time.

Case 5: $uMax = 5$, $vMax = 5$, $nDisc = 14$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$.
The total time taken by the training was 21 minutes, 43 seconds.

This case considers a different $nDisc$ which becomes 14 together with the discretization number for the states.

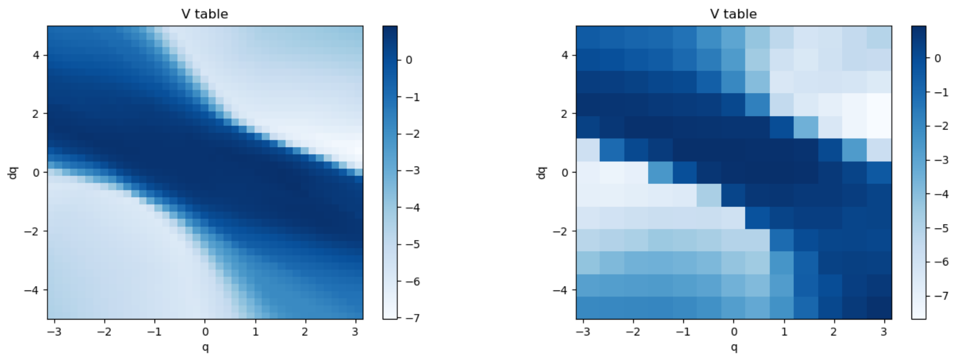


Figure 8: Color maps of the value functions in **case 1**, left, and in **case 5**, right

The effect of this variation is very evident. Dividing the states in less steps, they become lager and the effect in the value function map can be observed in figure 8. The same effect can be seen in the policy map that shares the same grid (this plot can be found in the Appendix).

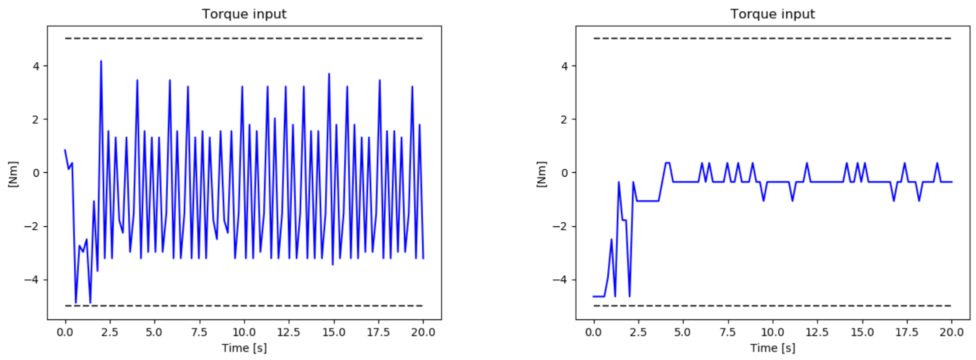


Figure 9: Torque history in **case 1**, left, and in **case 5**, right

Looking directly at the torque input history in figure 9, the $nDisc$ modification can be easily noted in the value that it assumes. It clearly moves between discrete states in this case while in the first plot the discretization is less evident due to the fact that the same range has been divided in 42 steps.

4.2 Double pendulum

Applying the same analysis seen above to a double pendulum with only one joint actuated, the results are not what we hoped for. As it is possible to observe in figure 10, the position of the pendulum does not stabilize in vertical position ($0\ rad$), but continues to oscillate or rotate around the joint. In this case the parameters chosen for the simulation are: $uMax = 5$, $vMax = 5$, $nDisc = 14$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$.
The total time taken by the training was 23 minutes and 52 seconds.

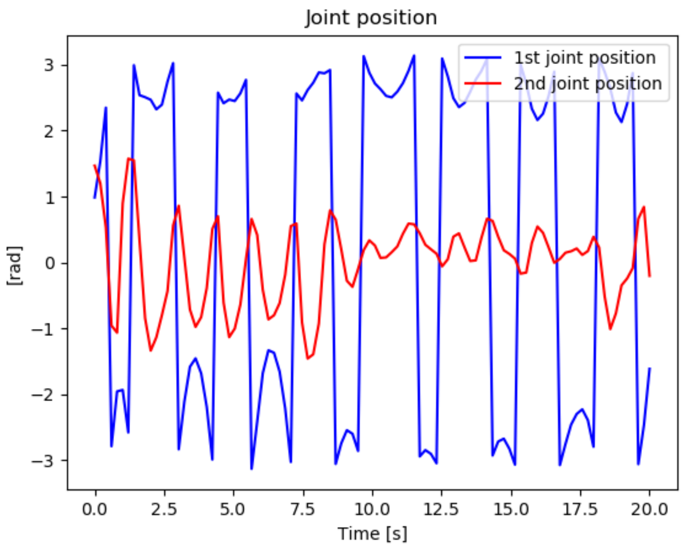


Figure 10: Position plot for a double pendulum.

Several attempts have been carried out trying to find an optimal policy able to perform the swing-up for a double pendulum. First of all, we changed the dynamics parameters ($uMax = 30$, $vMax = 30$), increasing the maximum input torque or the maximum velocity, thinking that, since in this case the first joint has to hold the weight of the second pendulum too, it is necessary to have a greater input torque, with respect to the case of the single pendulum.

Then, since this tuning had not effect, we tried to increase the number of episodes and their length ($nEpisodes = 300$, $EpisodeLength = 300$), thinking that being this problem more complex than the previous one, a greater number of episodes are necessary to train the model in a correct way.

Despite this attempts, we were unable to put the algorithm in a condition to find an optimal policy that is able to perform a swing-up for a double pendulum.

5 Potential improvements

First of all, since the dynamics of both a single and a double pendulum was studied in the second assignment too, we know that the values of maximum torque and velocity remain quiet limited. Based on this assumption we kept these parameters constant in most of the cases as regard the single pendulum, trying instead to tune the values related to the algorithm itself. In the case of the double pendulum, since it did not perform a swing-up, we assigned to it values of torque and velocity too high trying to find a solution: this was an useless attempt since it is known that such high values are not physically necessary to reach the goal.

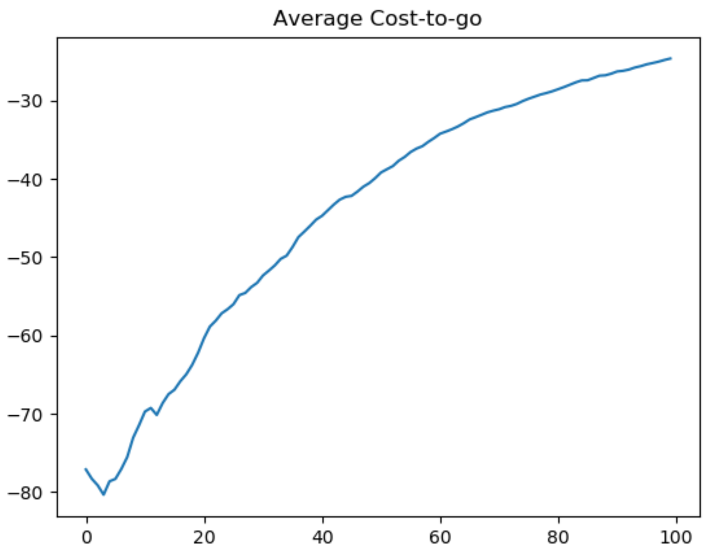
In order achieve good results with the double pendulum, it is probably necessary an higher computational effort with the evaluation of many other episodes and increasing their length. The available computational power could sustain such a workload but the training time would have last for a very long time. For this reason, we tried to use Google Colab to execute our code due to the fact that it is well suited for machine learning. After uploading the code and some libraries, we had problem importing Gepetto environment.

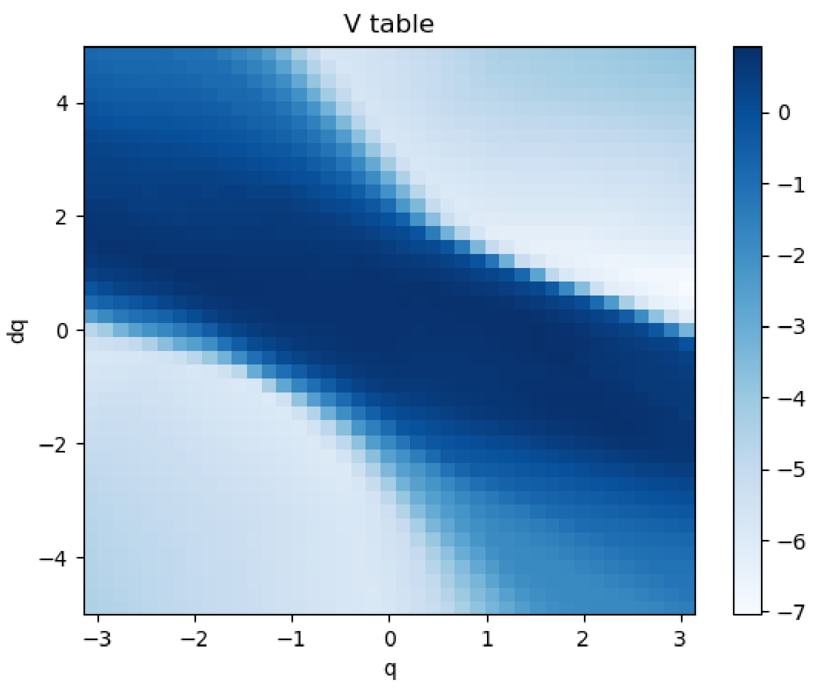
A future analysis could be performed used that Google product to train and save a model using a lot more episodes. In addition to that, we could have performed some training changing other hyper-parameters that are kept constant in all the results we present this report.

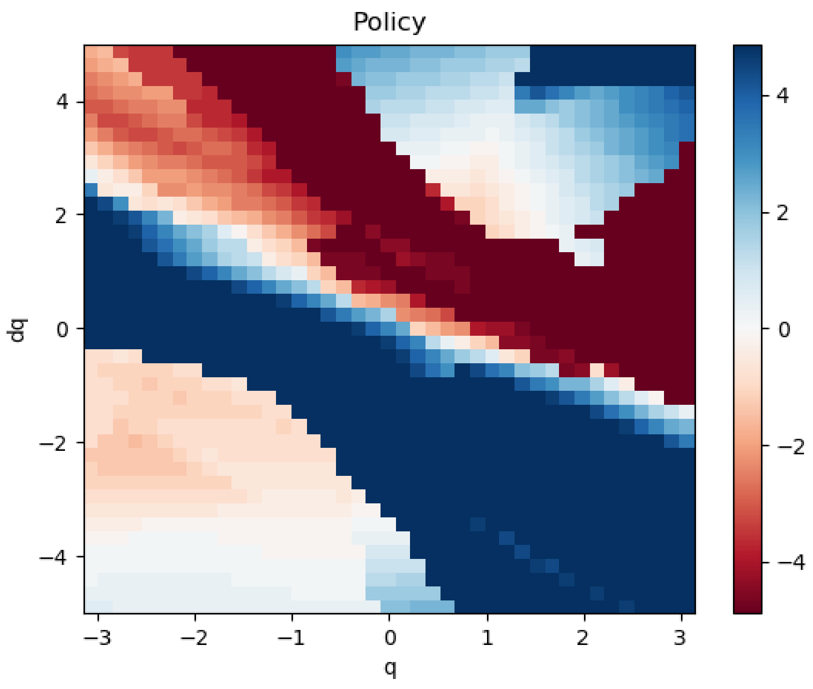
A Appendix

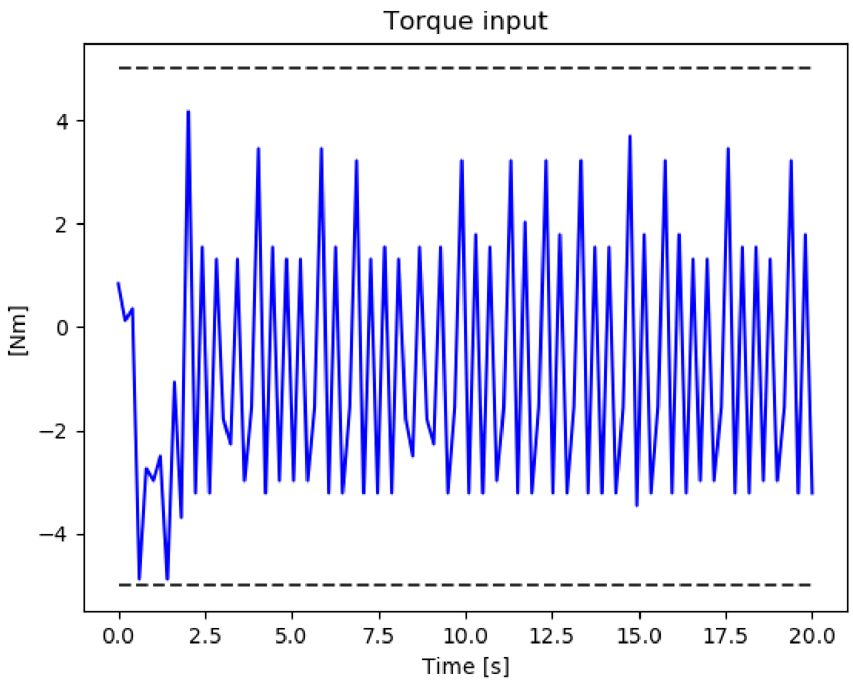
A.1 Single pendulum

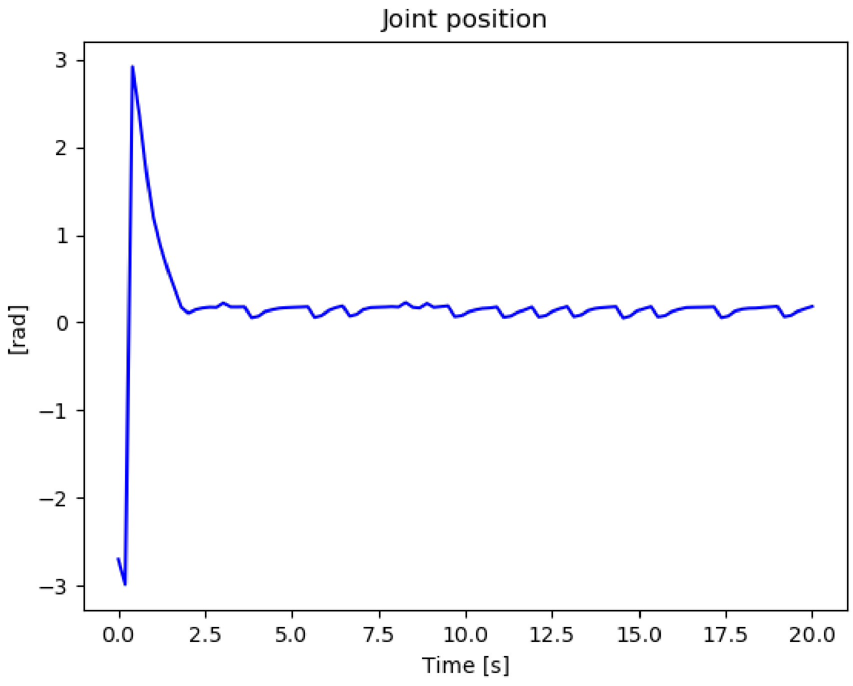
Case 1: $uMax = 5$, $vMax = 5$, $nDisc = 42$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$

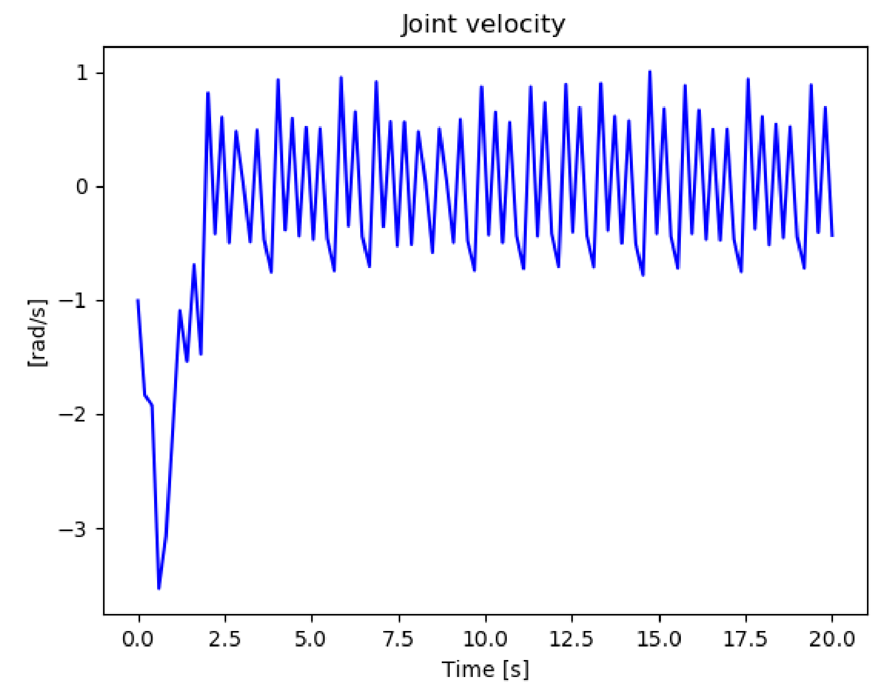


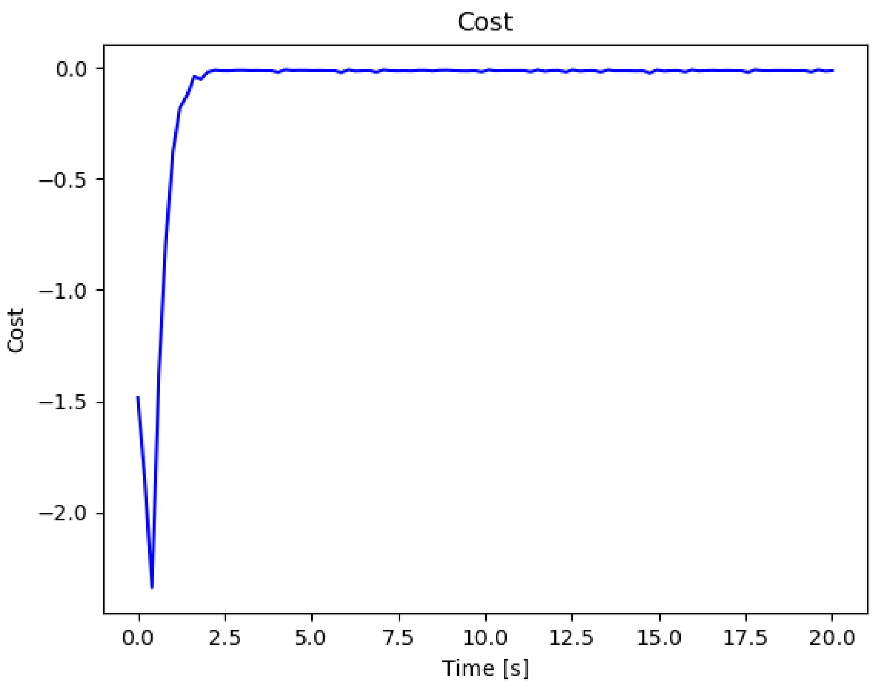

V table
dq
q
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
0
−1
−2
−3
−4
−5
−6
−7


Policy
dq
q
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
2
0
−2
−4

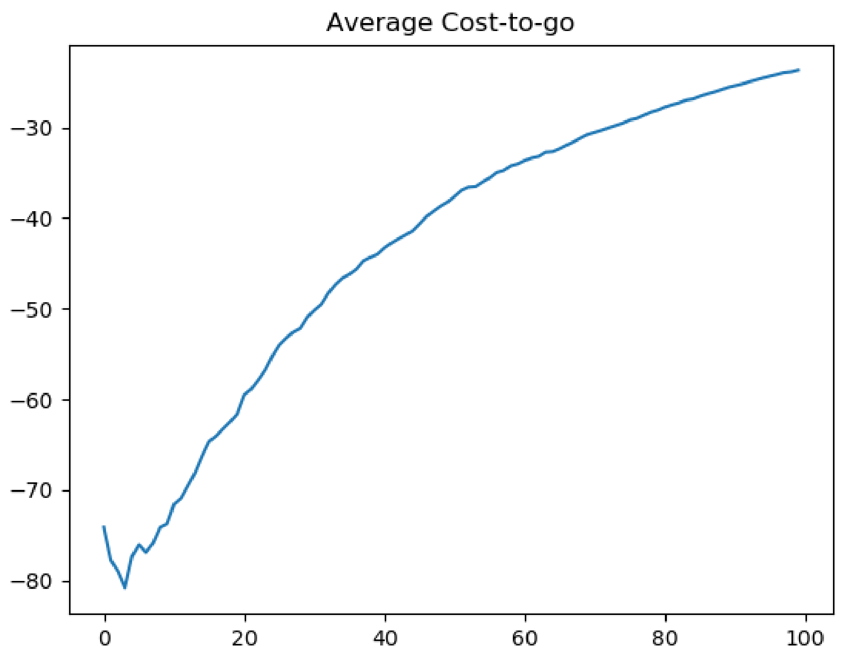

Torque input
[Nm]
Time [s]
4
2
0
−2
−4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

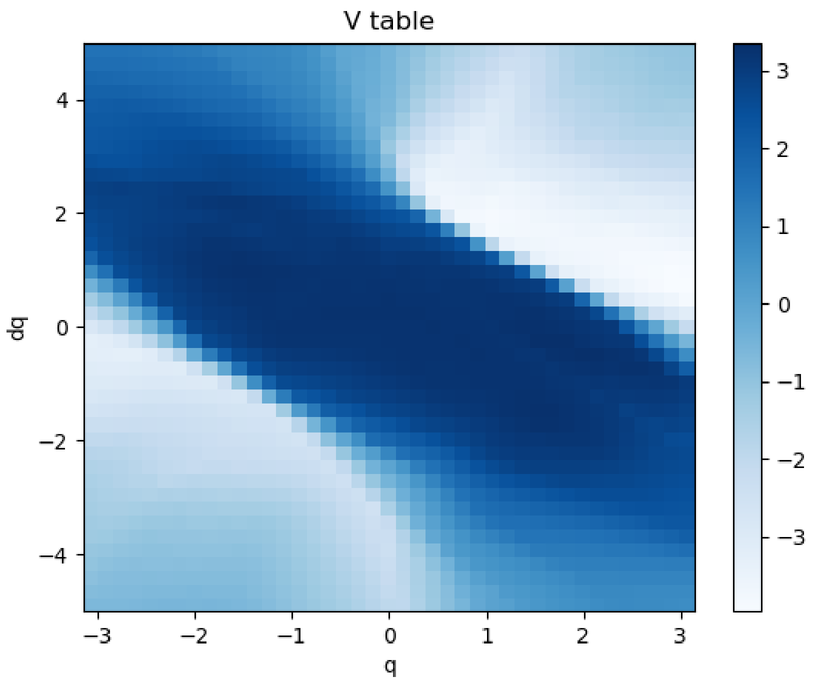

Joint position
[rad]
Time [s]
3
2
1
0
−1
−2
−3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

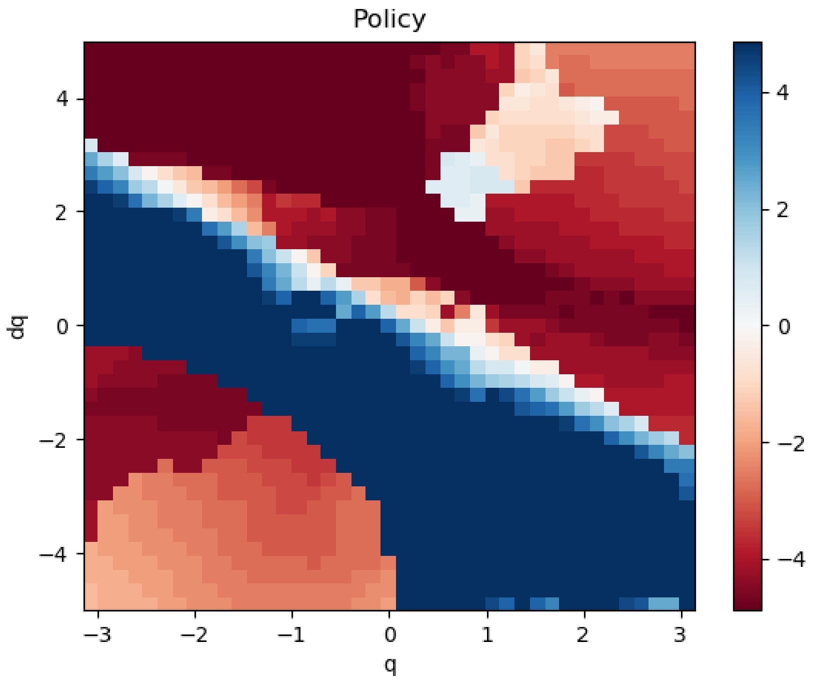

Joint velocity
[rad/s]
Time [s]
1
0
−1
−2
−3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

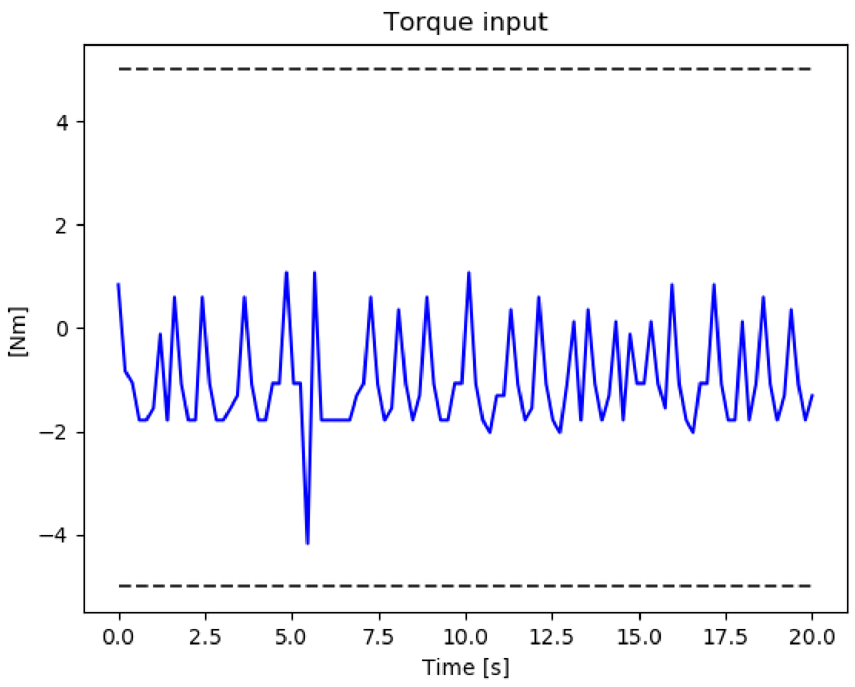

Cost
Cost
Time [s]
0.0
−0.5
−1.0
−1.5
−2.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Case 2: $uMax = 5$, $vMax = 5$, $nDisc = 42$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 200$

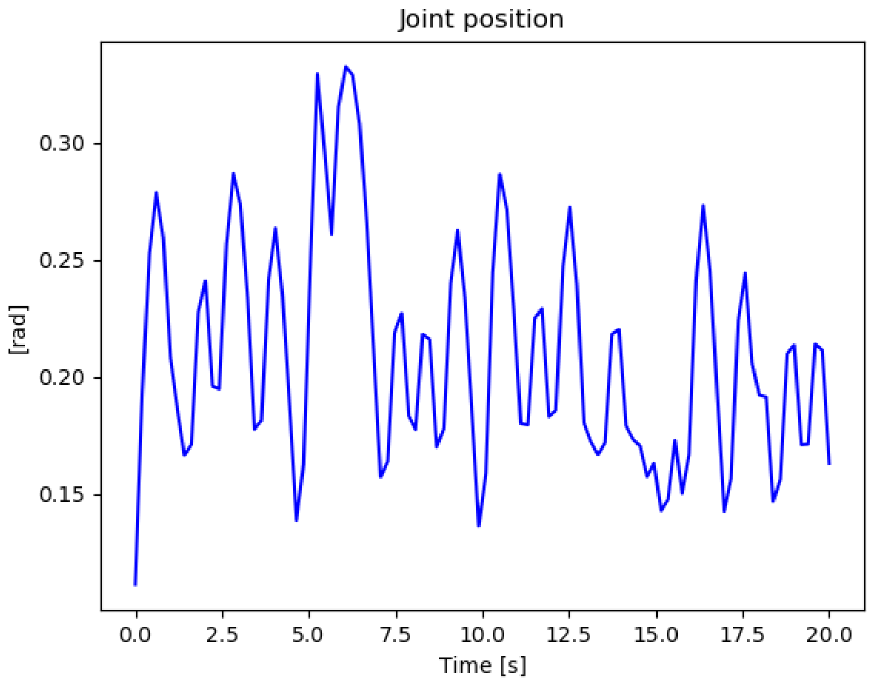



V table
dq
q
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
3
2
1
0
−1
−2
−3

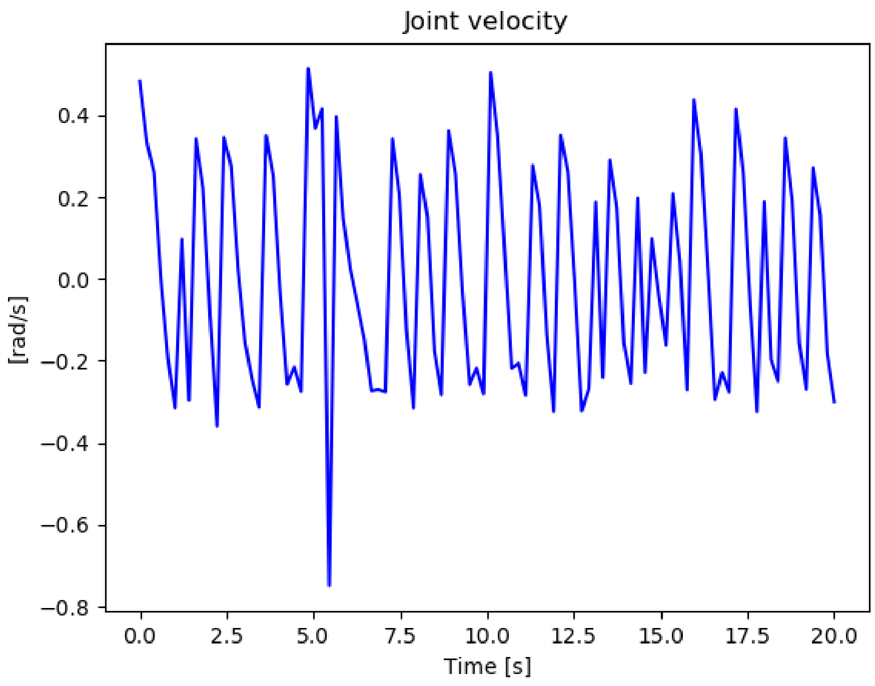

Policy
dq
q
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
2
0
−2
−4

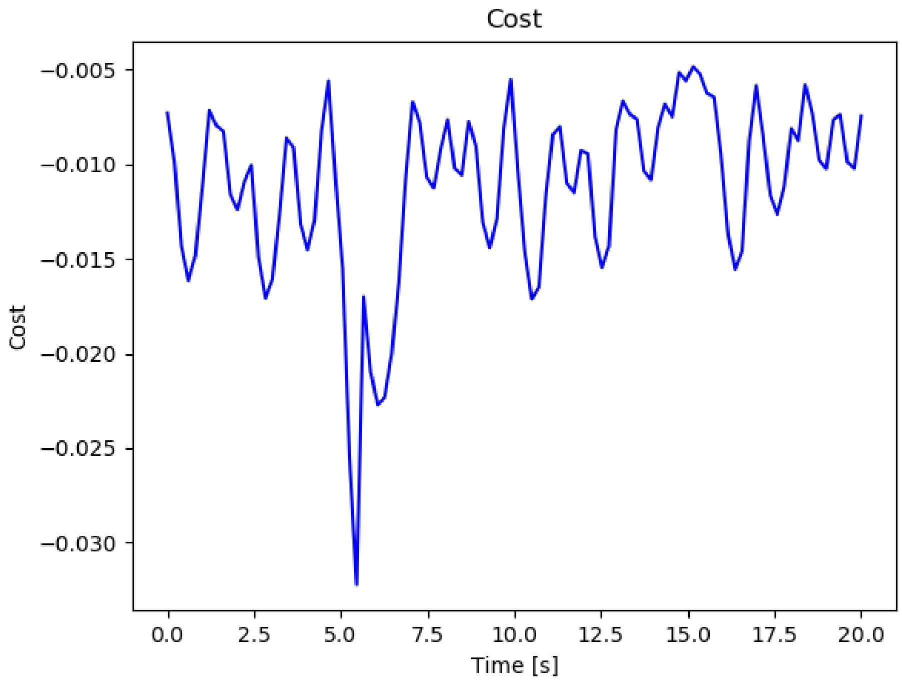


Torque input
4
2
0
−2
−4
[Nm]
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

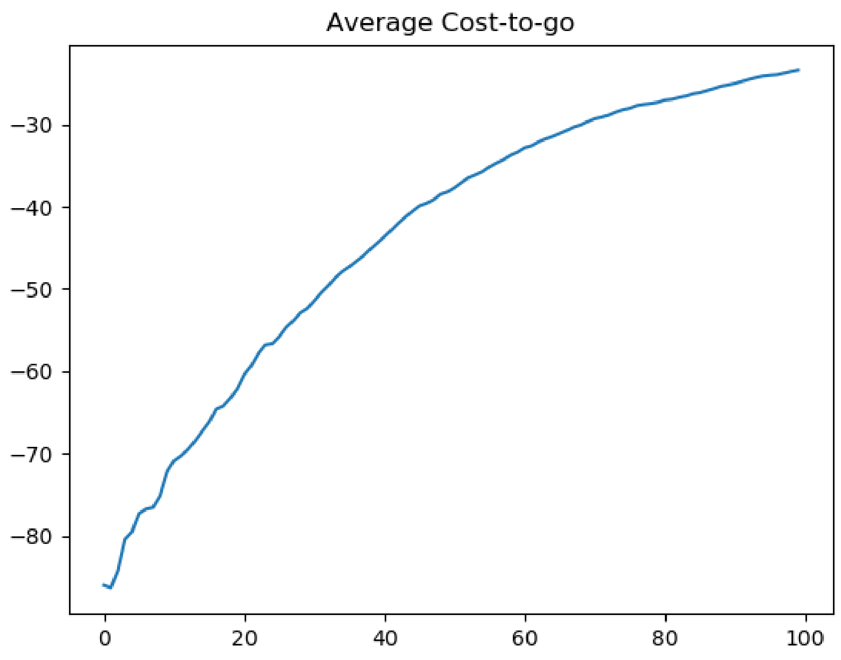


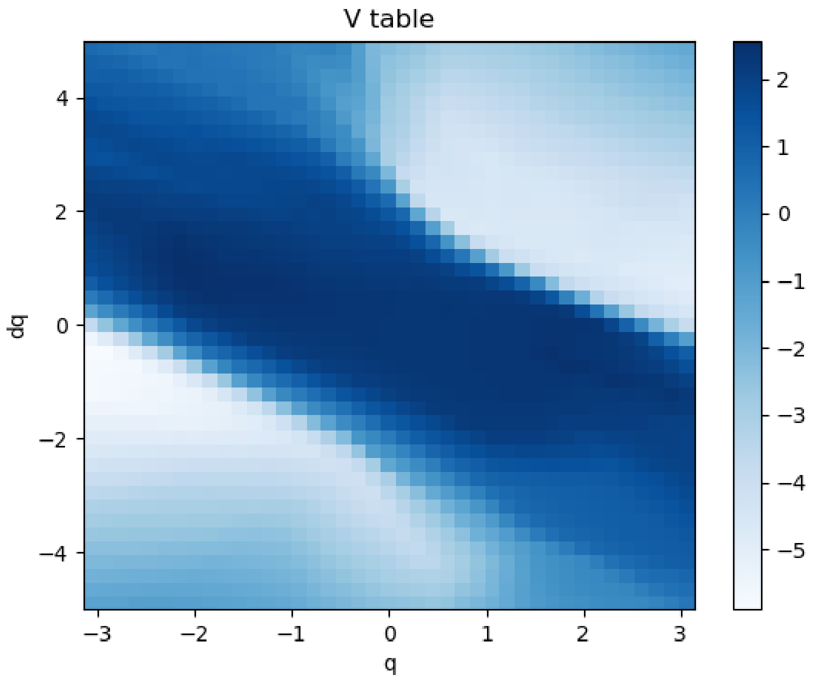
Joint position
0.30
0.25
0.20
0.15
[rad]
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

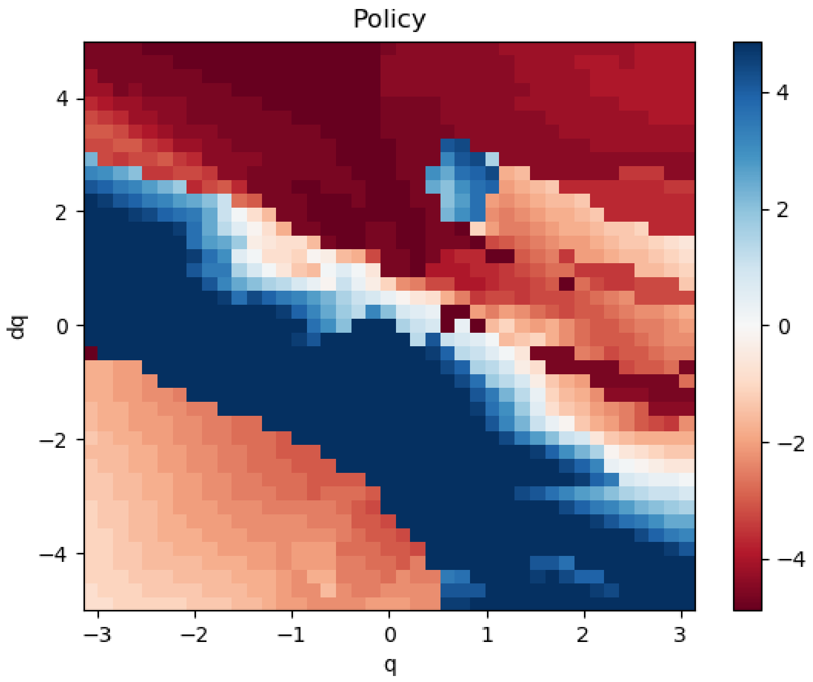

Joint velocity
[rad/s]
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

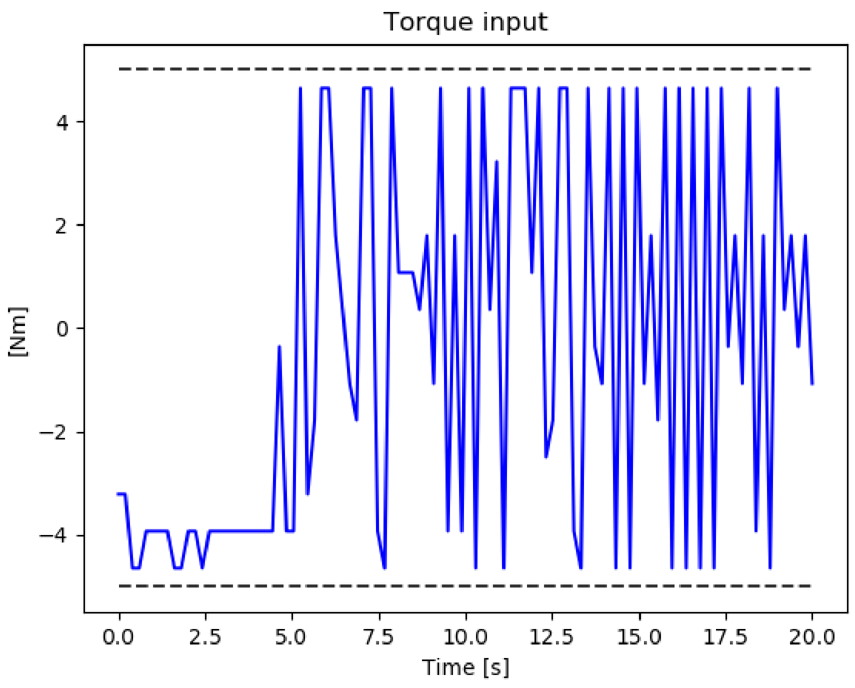

Cost
Cost
−0.005
−0.010
−0.015
−0.020
−0.025
−0.030
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

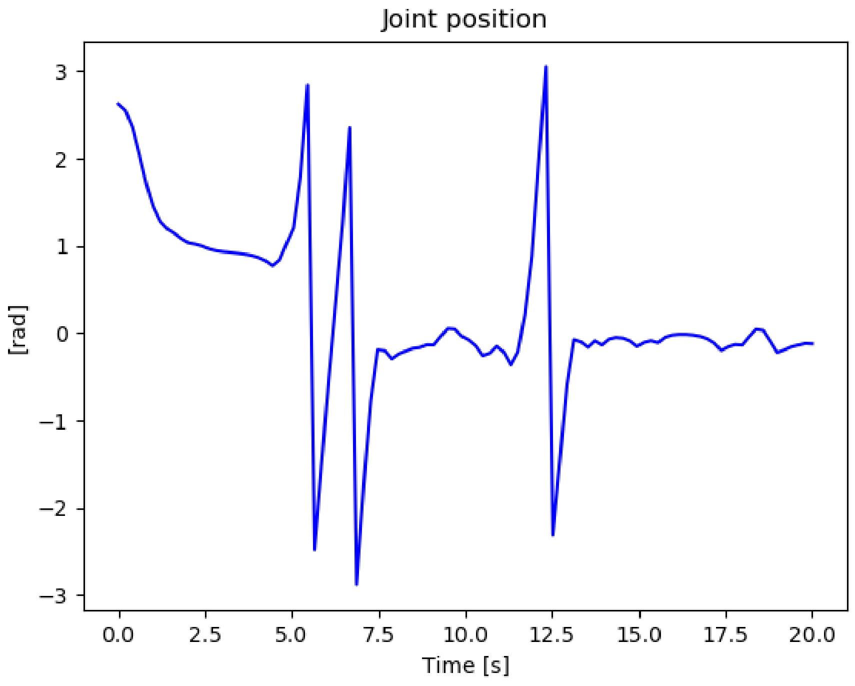
Case 3: $uMax = 5$, $vMax = 5$, $nDisc = 42$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 64$, $minBuffer = 100$

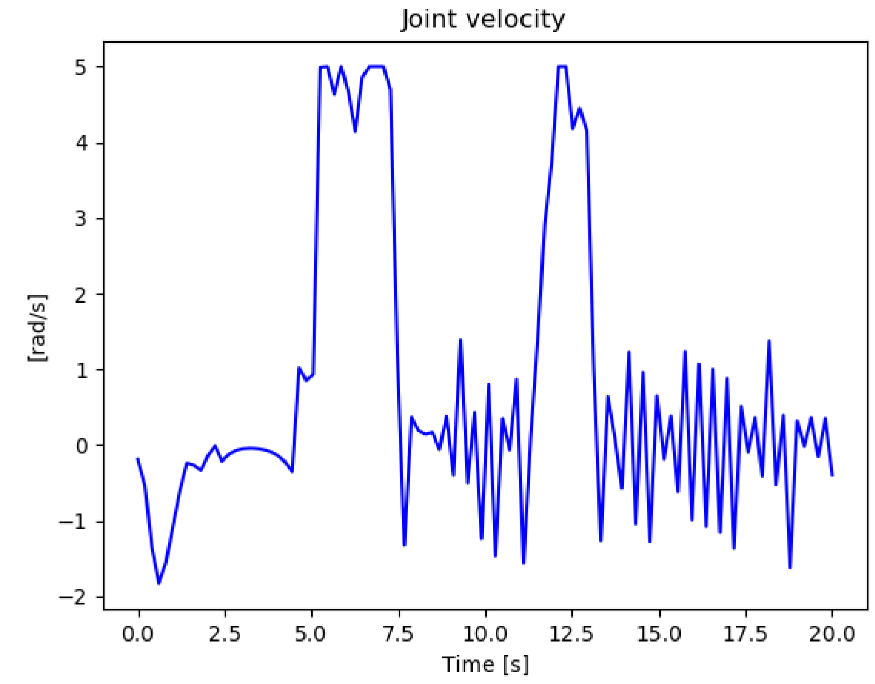


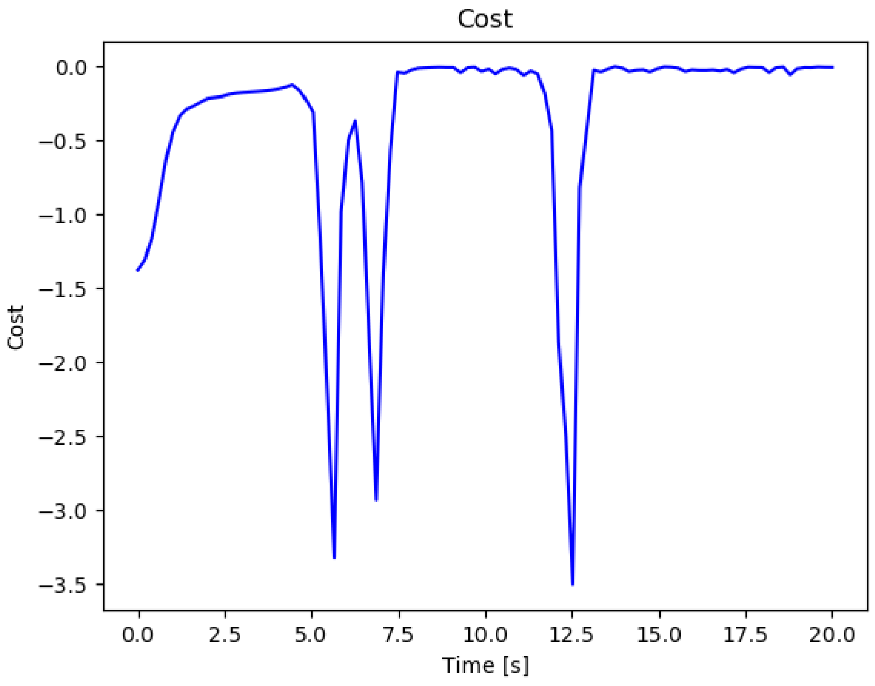

V table
dq
q
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
2
1
0
−1
−2
−3
−4
−5


Policy
dq
q
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
2
0
−2
−4

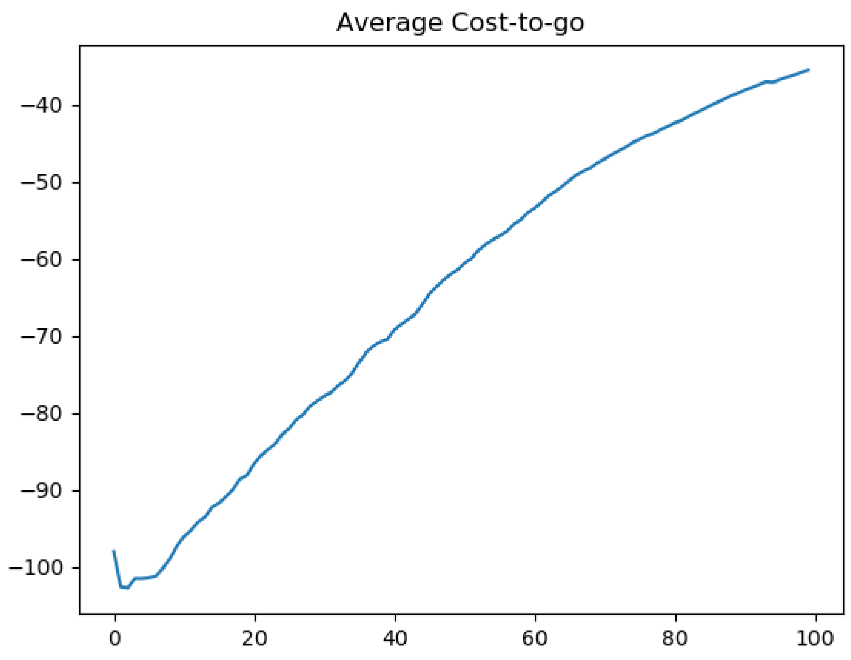

Torque input
4
2
0
−2
−4
[Nm]
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

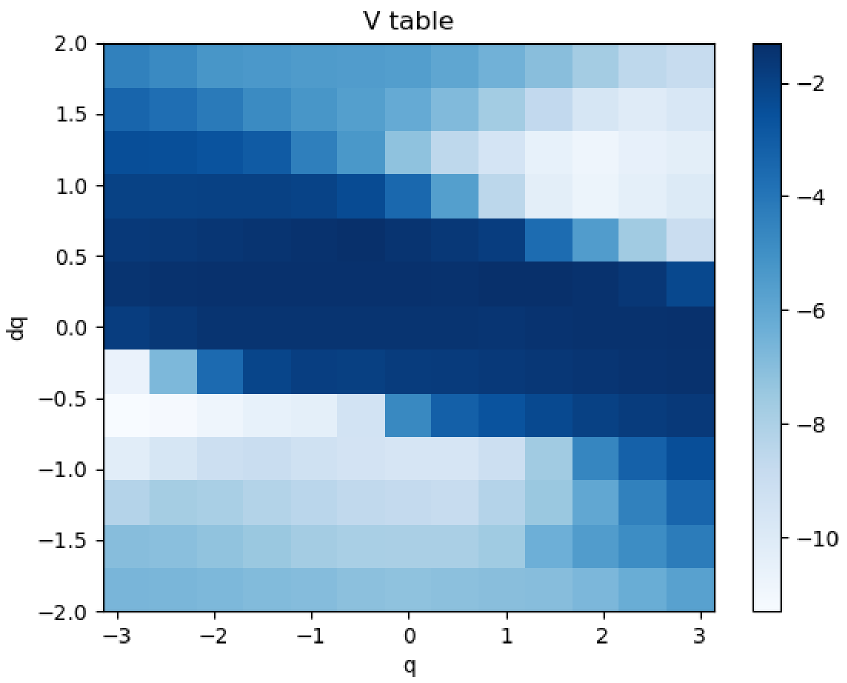

Joint position
3
2
1
0
−1
−2
−3
[rad]
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

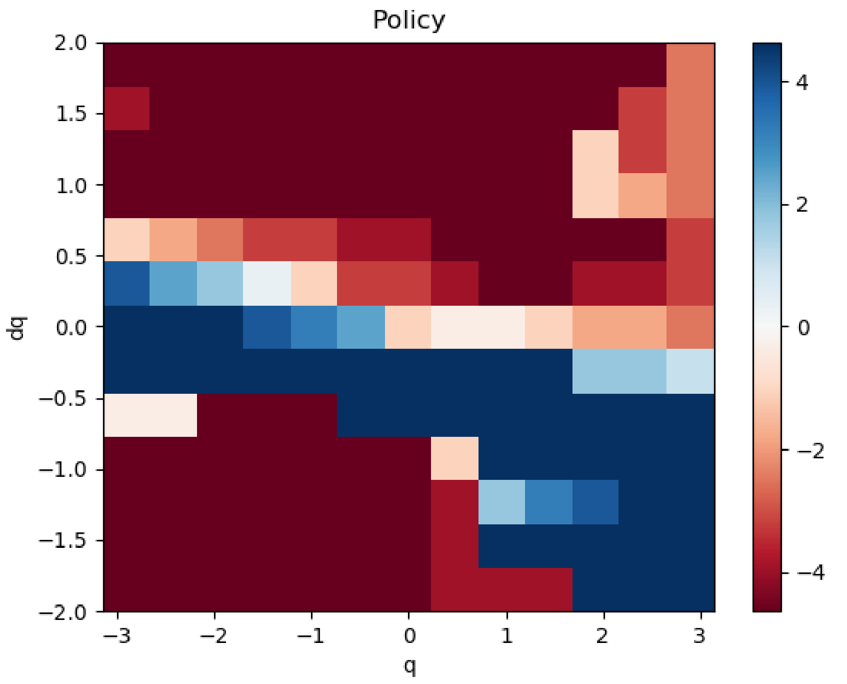

Joint velocity
[rad/s]
Time [s]
5
4
3
2
1
0
−1
−2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

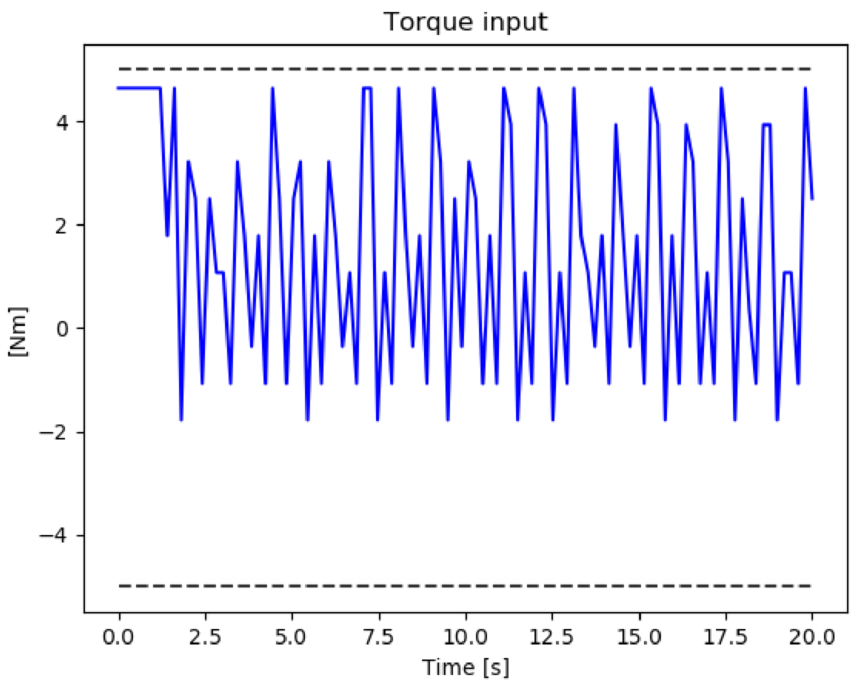

Cost
Cost
Time [s]
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

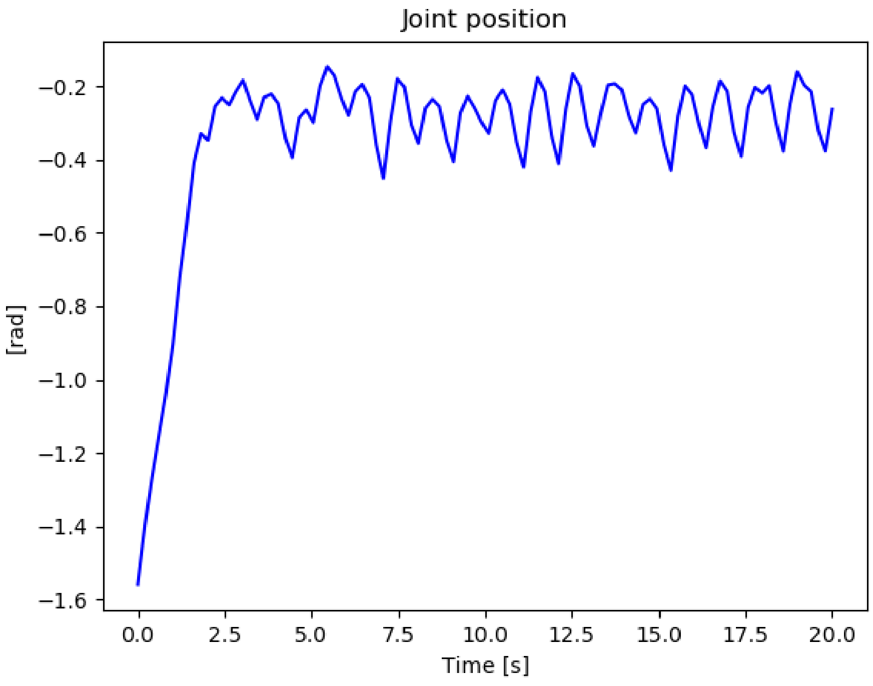
Case 4: $uMax = 5$, $vMax = 11$, $nDisc = 42$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$

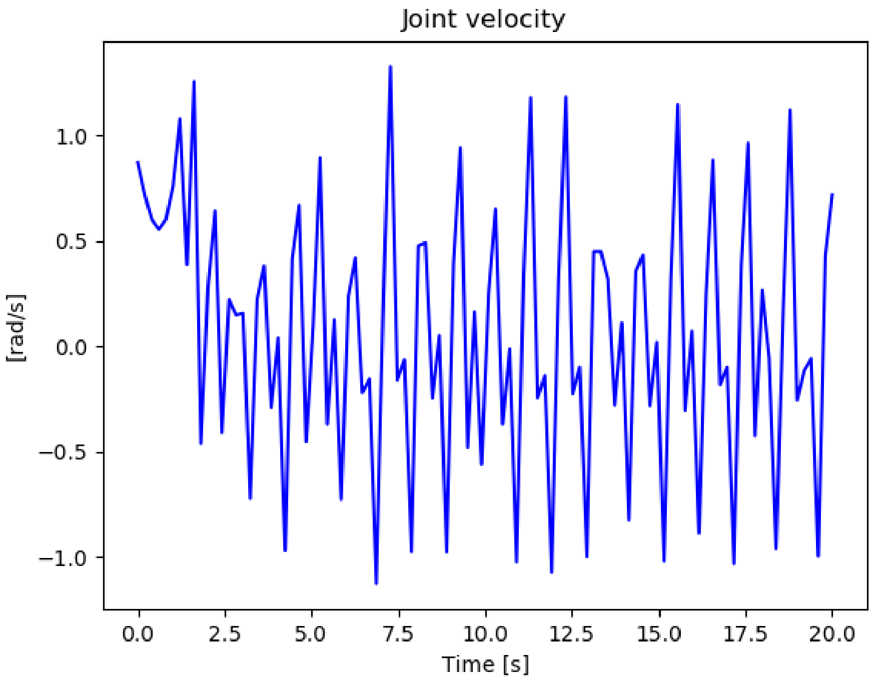


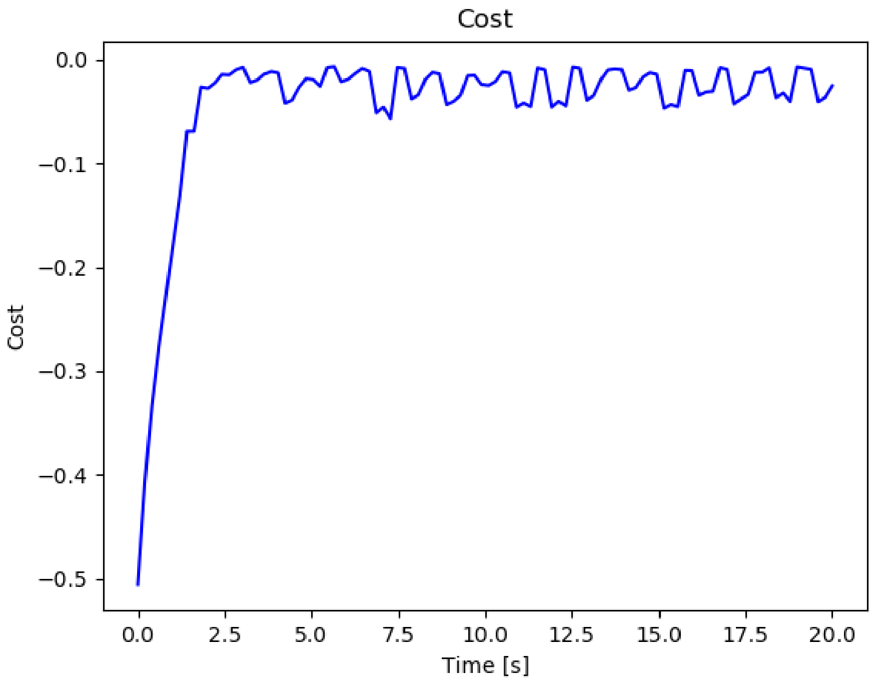

V table
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
dq
−3
−2
−1
0
1
2
3
q
−2
−4
−6
−8
−10


Policy
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
dq
−3
−2
−1
0
1
2
3
q
4
2
0
−2
−4

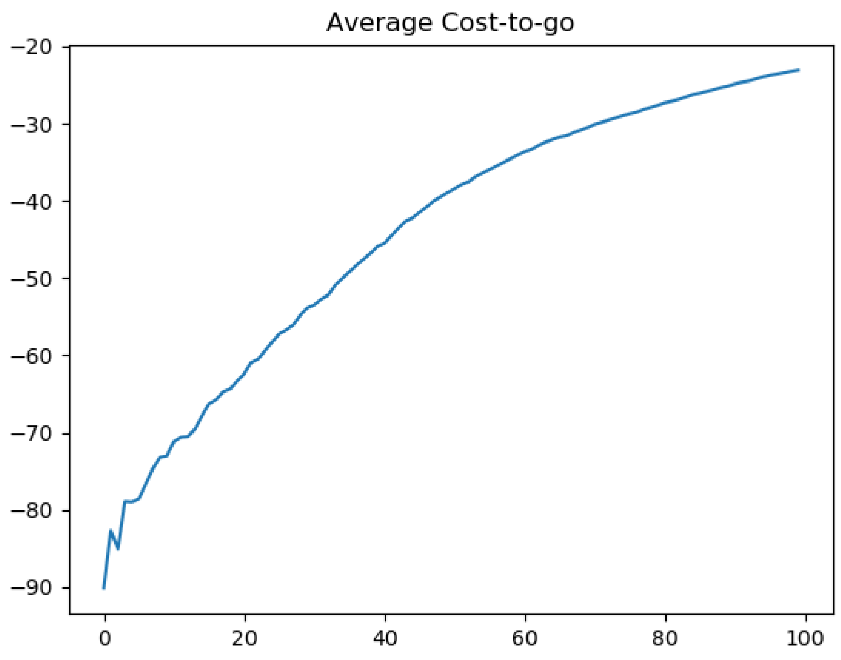

Torque input
[Nm]
4
2
0
−2
−4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

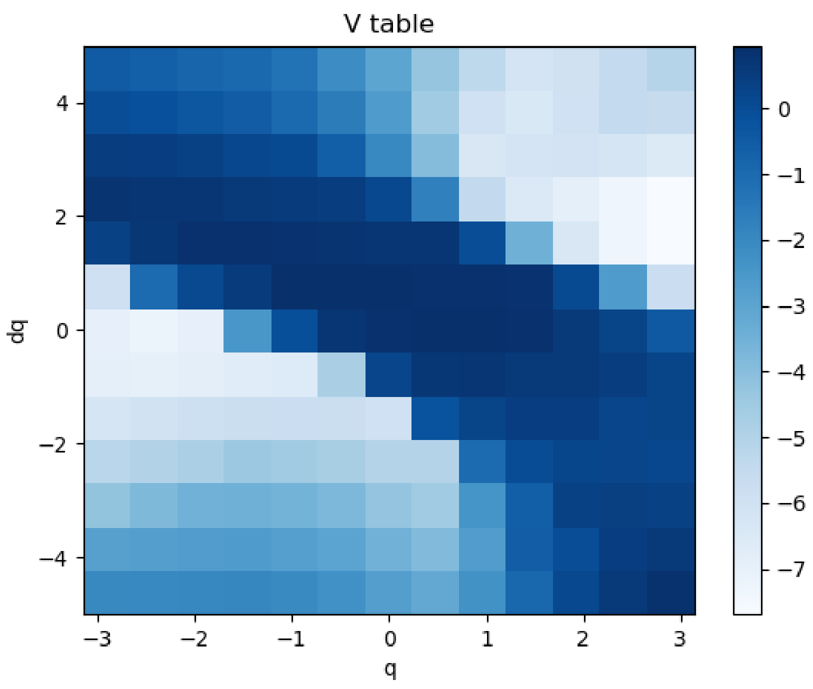

Joint position
[rad]
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
−1.4
−1.6
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

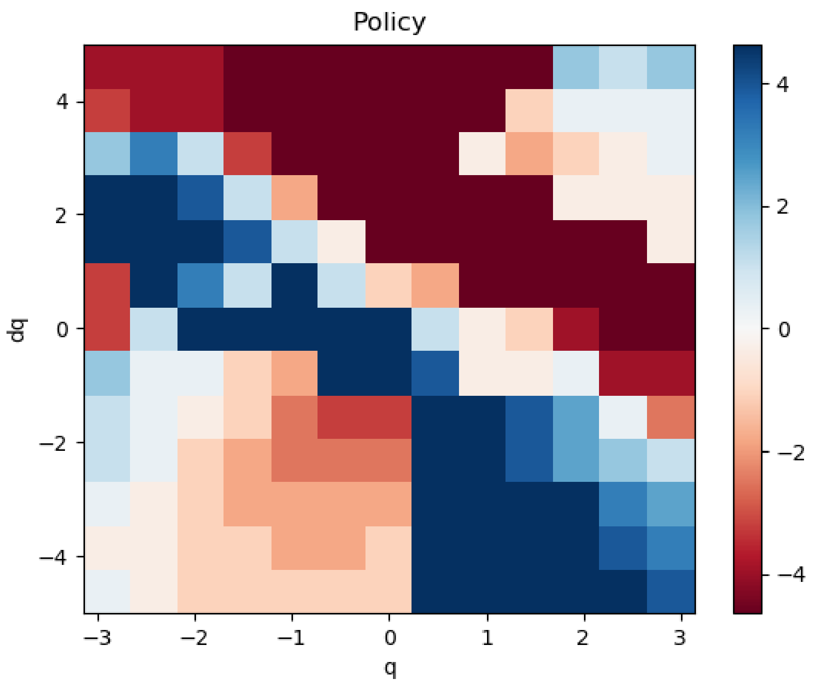

Joint velocity
[rad/s]
1.0
0.5
0.0
−0.5
−1.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

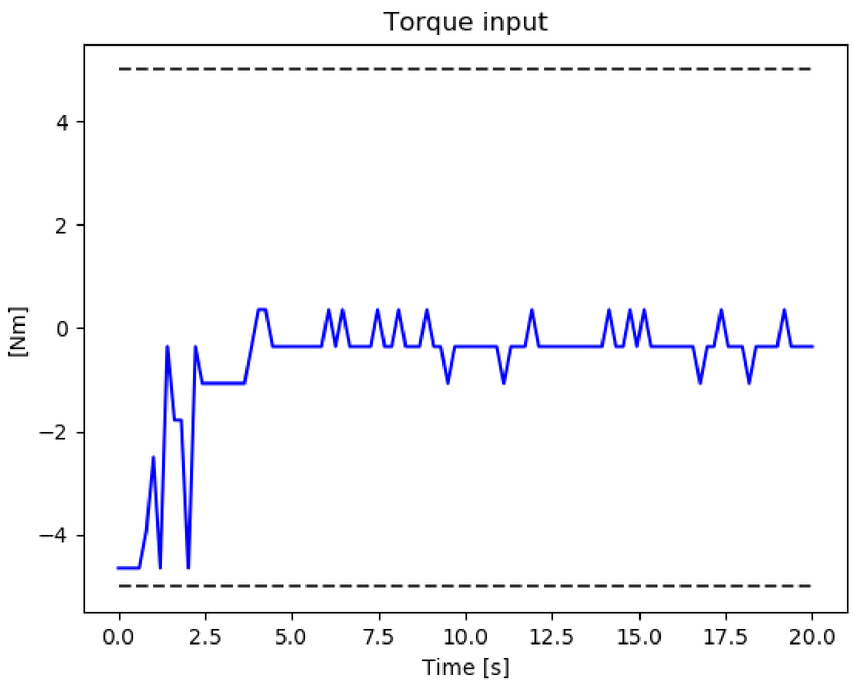

Cost
Cost
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

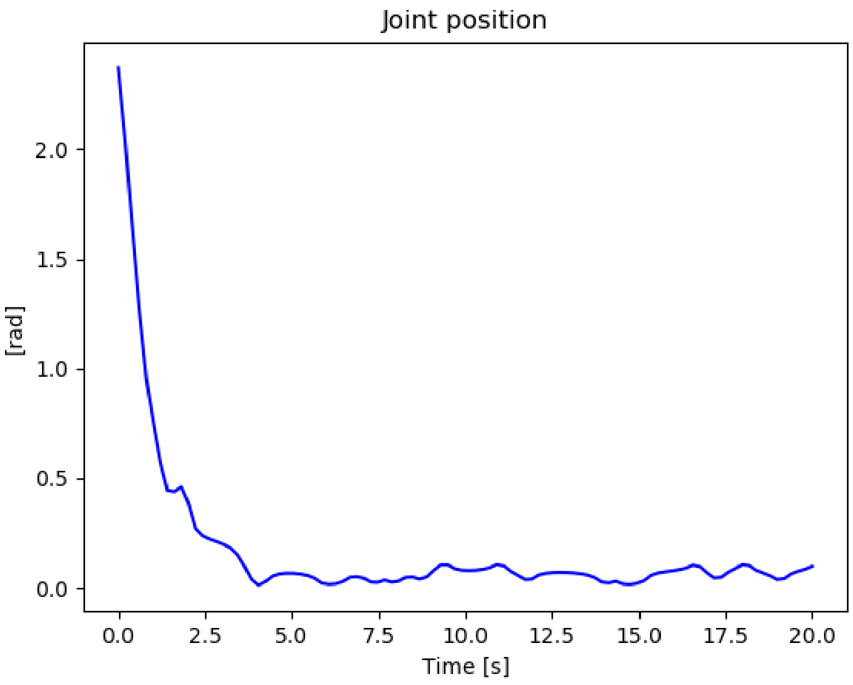
Case 5: $uMax = 5$, $vMax = 5$, $nDisc = 14$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$

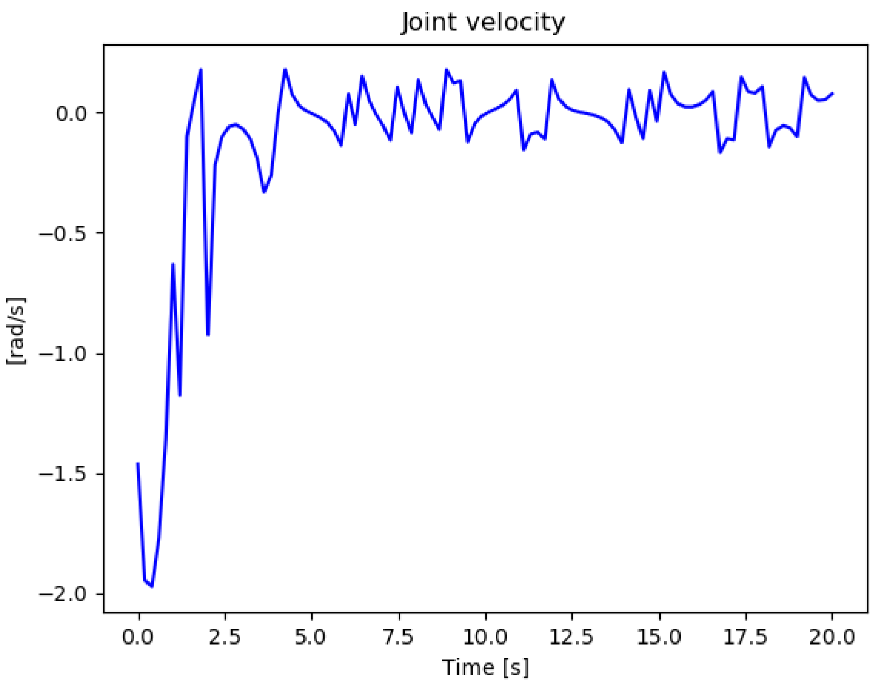


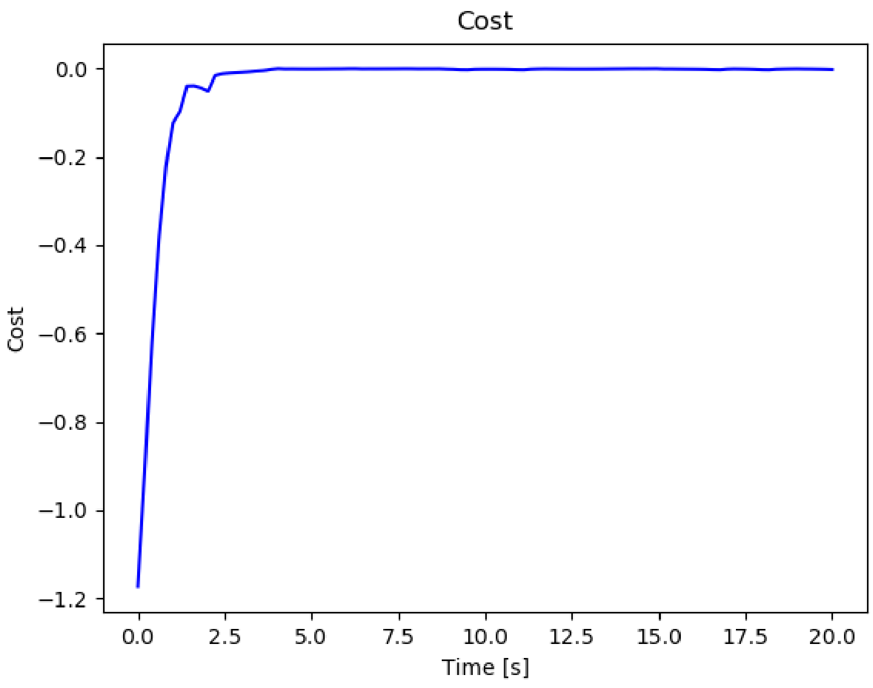

V table
dq
q
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
0
−1
−2
−3
−4
−5
−6
−7


Policy
dq
q
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
2
0
−2
−4


Torque input
[Nm]
4
2
0
−2
−4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

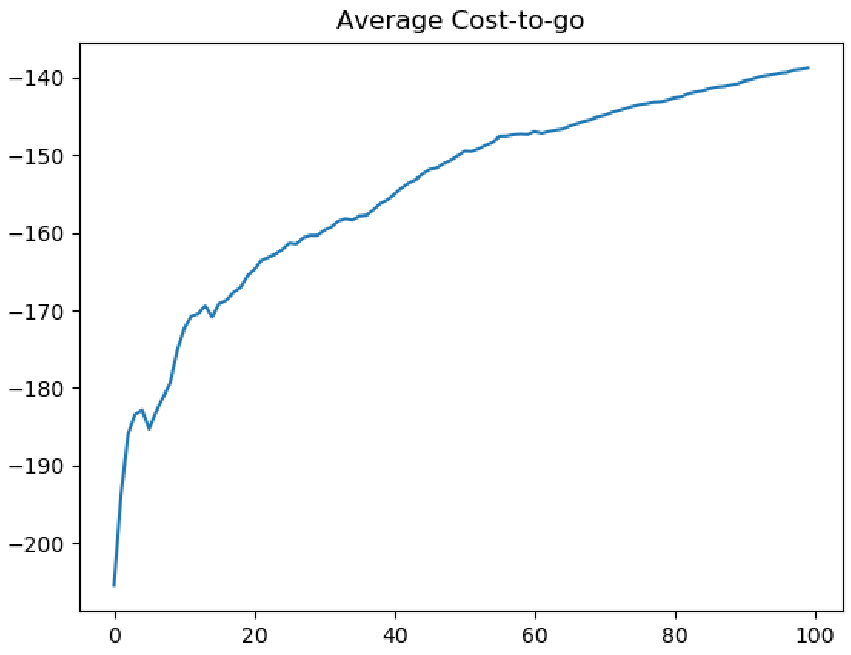

Joint position
[rad]
2.0
1.5
1.0
0.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

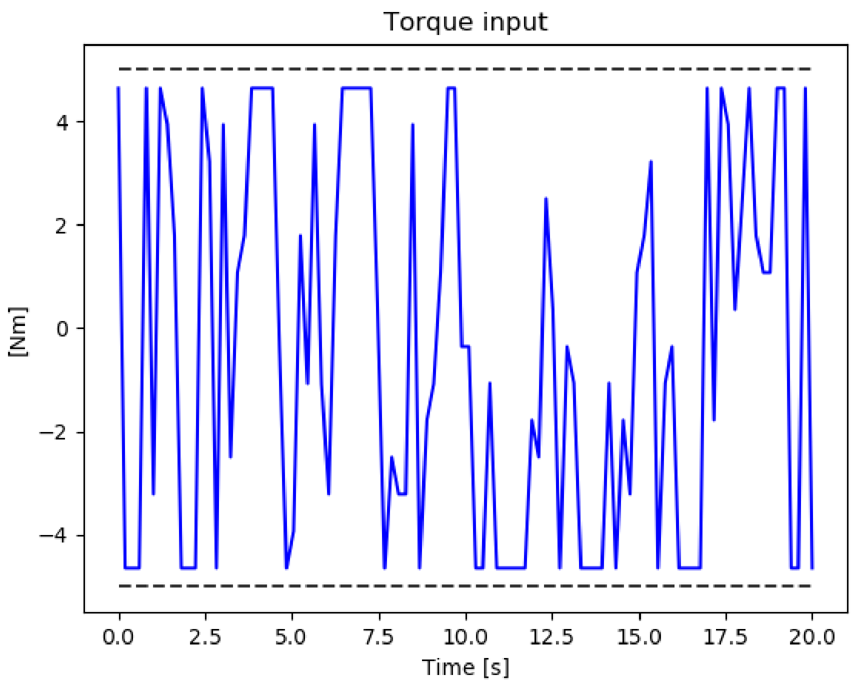

Joint velocity
[rad/s]
Time [s]
0.0
−0.5
−1.0
−1.5
−2.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

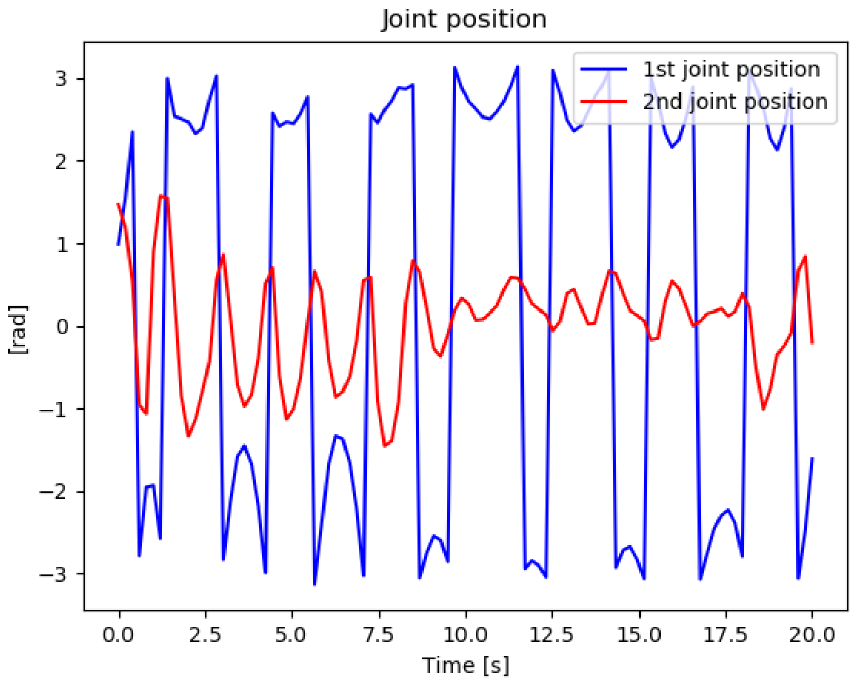

Cost
Cost
Time [s]
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

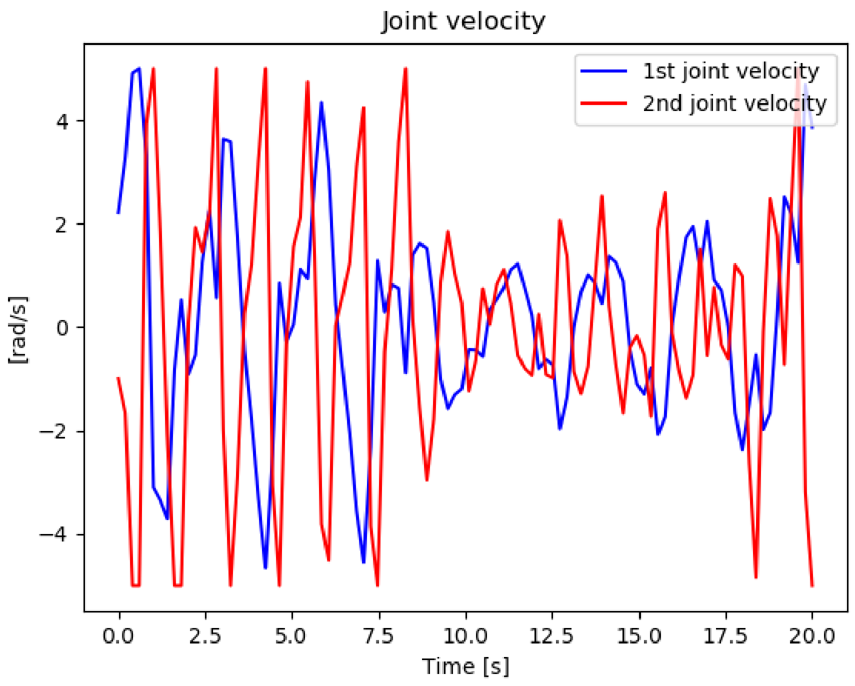
A.2 Double pendulum

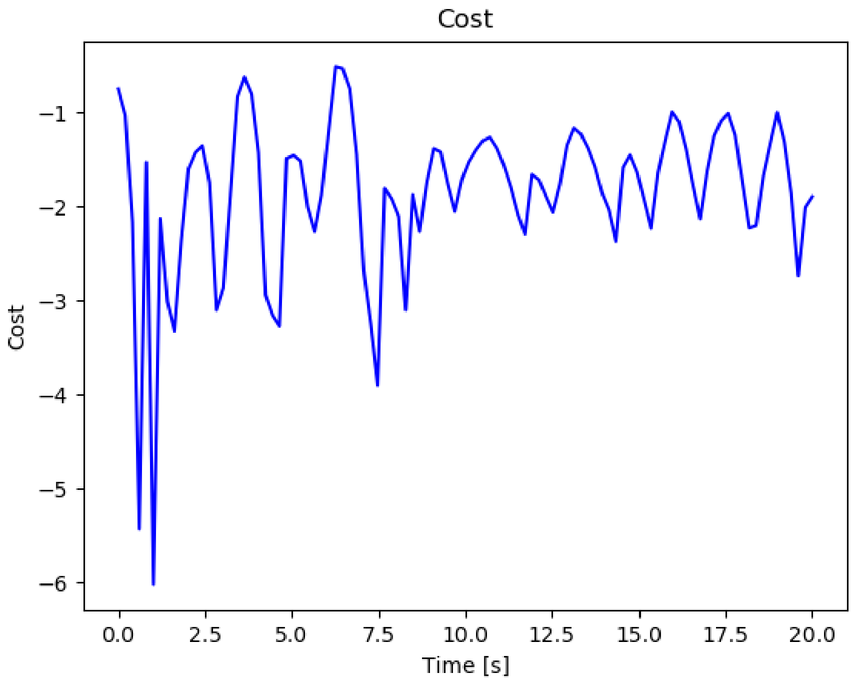
Case 1: $uMax = 5$, $vMax = 5$, $nDisc = 14$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$



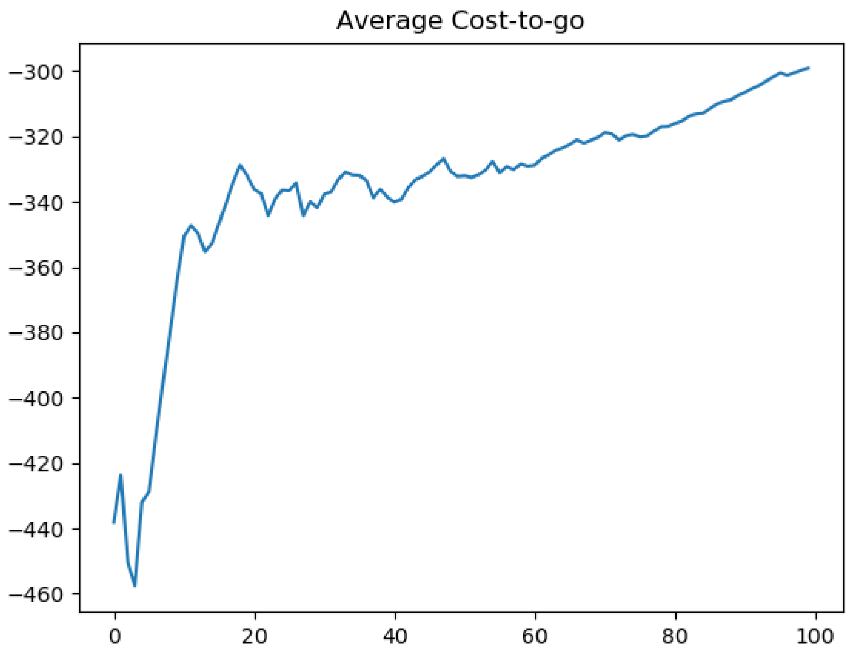

Torque input
[Nm]
Time [s]
4
2
0
−2
−4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

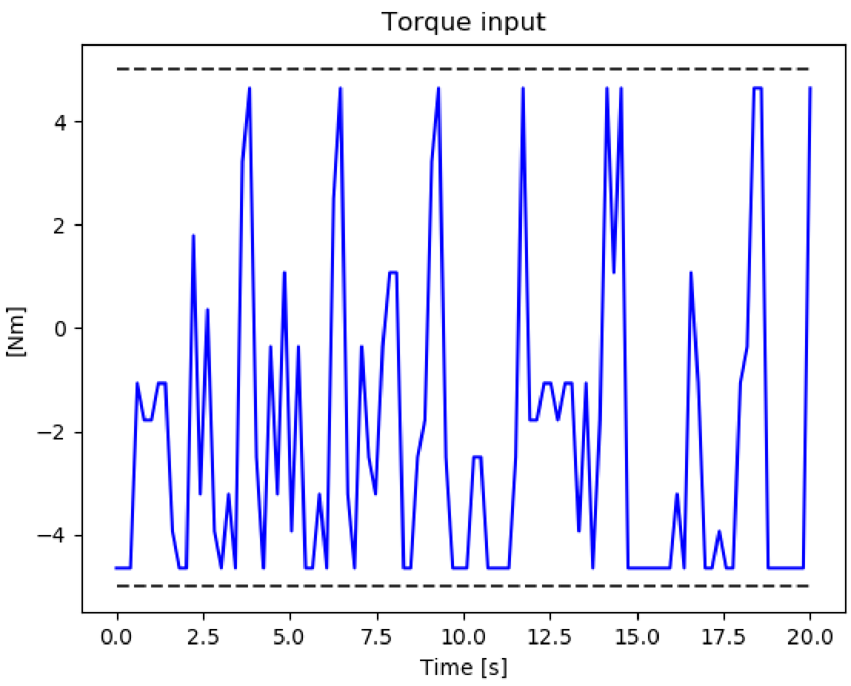

Joint position
1st joint position
2nd joint position
[rad]
Time [s]
3
2
1
0
−1
−2
−3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Joint velocity
1st joint velocity
2nd joint velocity
[rad/s]
4
2
0
−2
−4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

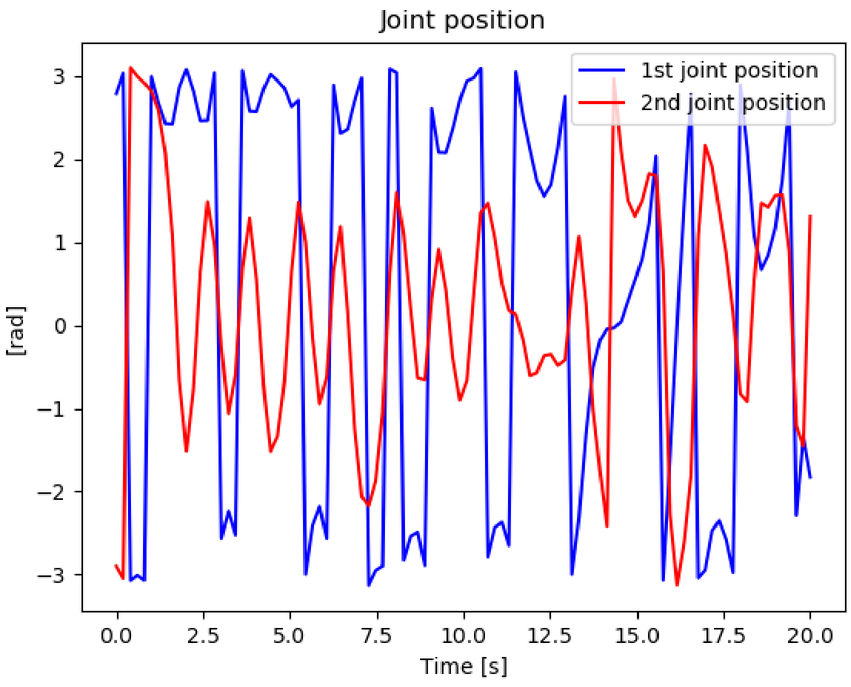

Cost
Cost
−1
−2
−3
−4
−5
−6
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

Case 2: $uMax = 5$, $vMax = 11$, $nDisc = 14$, $nEpisodes = 100$, $EpisodeLength = 100$, $BatchSize = 32$, $minBuffer = 100$

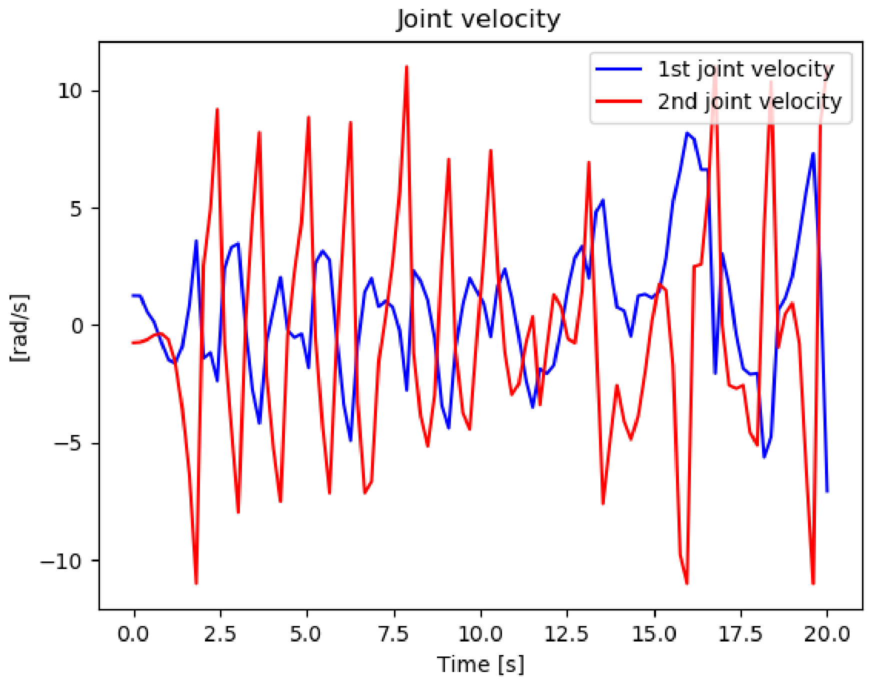




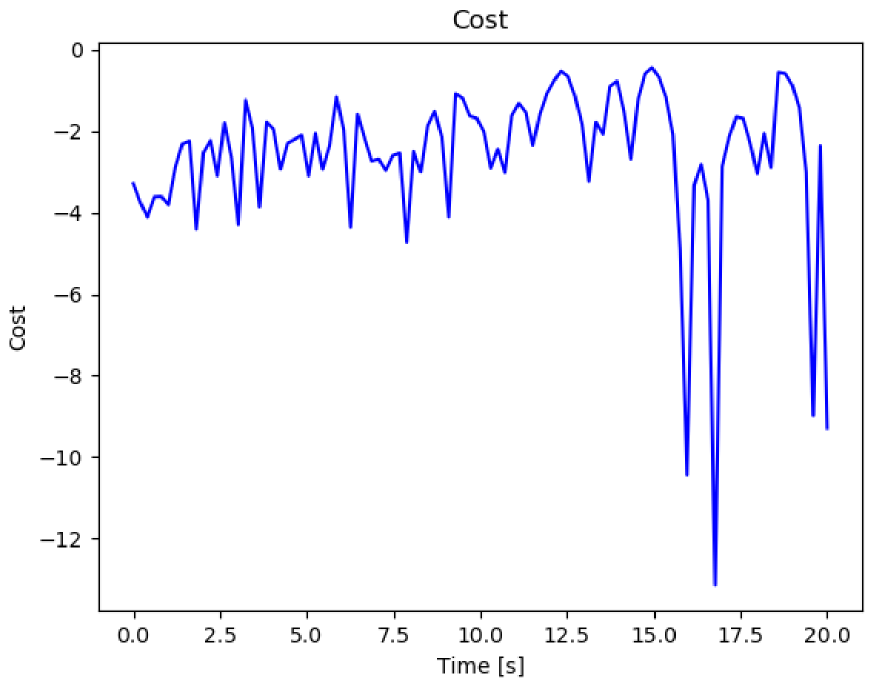
Torque input
[Nm]
Time [s]



Joint position
1st joint position
2nd joint position
[rad]
Time [s]



Joint velocity
1st joint velocity
2nd joint velocity
[rad/s]
10
5
0
−5
−10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]



Cost
Cost
0
−2
−4
−6
−8
−10
−12
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

References

[1] Melrose Roderick, James MacGlashan, and Stefanie Tellex. "Implementing the Deep Q-Network". In: *CoRR* abs/1711.07478 (2017). URL: `http://arxiv.org/abs/1711.07478`.