Assignment 3: Deep Q-Network

145873 Optimisation-based Robot Control

Simone Peroni
230406
simone.peroni@unitn.it

Marco Peressutti
230403
marco.peressutti@unitn.it

October 24, 2023

Contents

1 Introduction

Contrary to the Q-learning algorithm, Deep Q-Network (DQN) exploits Deep Learning techniques to control more complex systems via function approximators.
In fact, the Q learning approach requires to, first and foremost, discretize both state and action space, and later keep track of the Q function value associated with each state and action pair in a table. This approach does not scale well with complex systems such as the UR5 robot, and its use cases are limited to problems involving a small amount of states and actions (such as the single pendulum).

Hence, by using the DQN approach, one can extend Reinforcement Learning techniques to control complex problems using a Neural Network to model the Quality function. By doing so, there is no more need to discretize the state of the system since it will be provided as input to the Neural Network.

Moreover, the Q function is uniquely defined by the weights and biases of the Neural Network, which are proved to be less aggressive in terms of memory consumption when compared to storing each value of the Q function in a table.

In this report, we will:

1. Give a brief overview of the DQN algorithm and discuss its implementation in Python using the PyTorch library (section 2)
2. Discuss the results obtained by applying the algorithm to the single and double pendulum swing-up manoeuvre (section 3 and 4)
3. Draw conclusion about the experience gathered in the two simulations

2 Deep Q-Network (DQN)

The basic elements of our implementation of the Deep Q-Network algorithm are:

1. The Replay buffer, using a double ended queue (a.k.a. *deque*) containing the experience of each time step t in the form of a tuple (s_t, a_t, r_t, s_{t+1}), namely state, action, reward and next state.
2. The action-value function and the target action-value function; both modeled by a Neural Network. Both NNs have the same architecture with 4 hidden layers of 16, 32, 64 and 64 neurons respectively activated via a ReLU function.
   The output of the Neural Networks are the Q-function associated with the input state and each discretized value of the action space.
   Hence, the output dimension of the Neural Networks is equal to the number of discretization steps of the joint torque.

   A broad representation of the Neural Network architecture is displayed in figure 2.
3. The environment model, in our case the single and double pendulum, which have been modified accordingly to have continuous state space and discrete action space.
4. An epsilon-greedy strategy, that balances the exploration-exploitation trade-off.

   As displayed in figure 1, the epsilon decay follows the law:

$$\varepsilon_n = end + (start - end) \cdot \exp\left(-\frac{n}{decay}\right)$$

where

$$start = 1.0 \quad ; \quad end = 0.001 \quad ; \quad decay = 1000$$

n is the episode

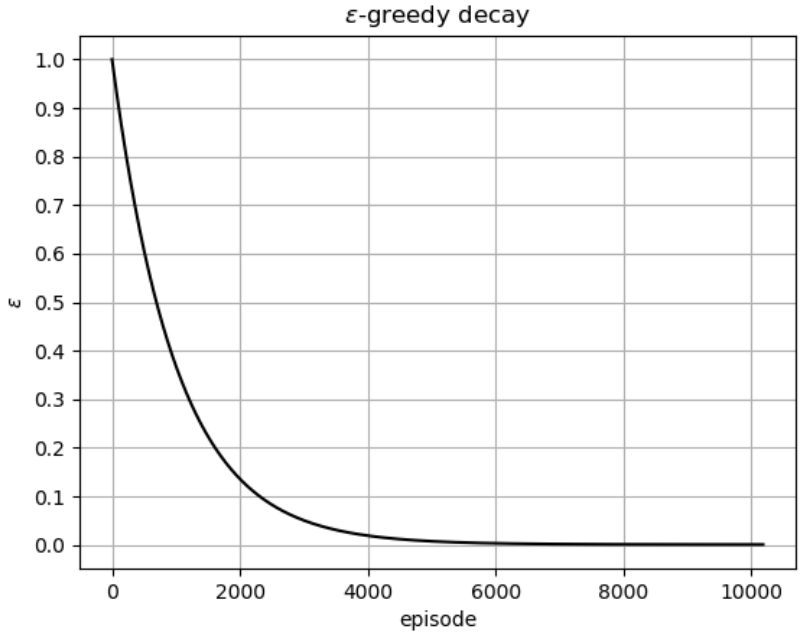

Figure 1: Epsilon decay used in the simulation

2.1 Training

The Deep Q-Network is trained to satisfy the Bellman equation

$$Q^{\pi}(s, a) = r + \gamma Q^{\pi}(s', \pi(s')) \tag{1}$$

or, in other words, to minimize the temporal difference error e:

$$e = Q^{\pi}(s, a) - (r + \gamma Q^{\pi}(s', \pi(s'))) \tag{2}$$

where π is the greedy policy. The reward, in our scenario, will always have negative values, as at each transition it will account for the value of the cost function of the new state.

The pseudocode of the Deep Q-Network training can be found in Algorithm (1).

The initialization of the environment at each episode puts the pendulum in a random state of configuration and velocity. In this way, a wider variety of states will be visited in simulation and will allow the neural network to better generalize the Q-function. The general performance of the algorithm, however, is measured in a standard setting every fixed amount of episodes, as the main

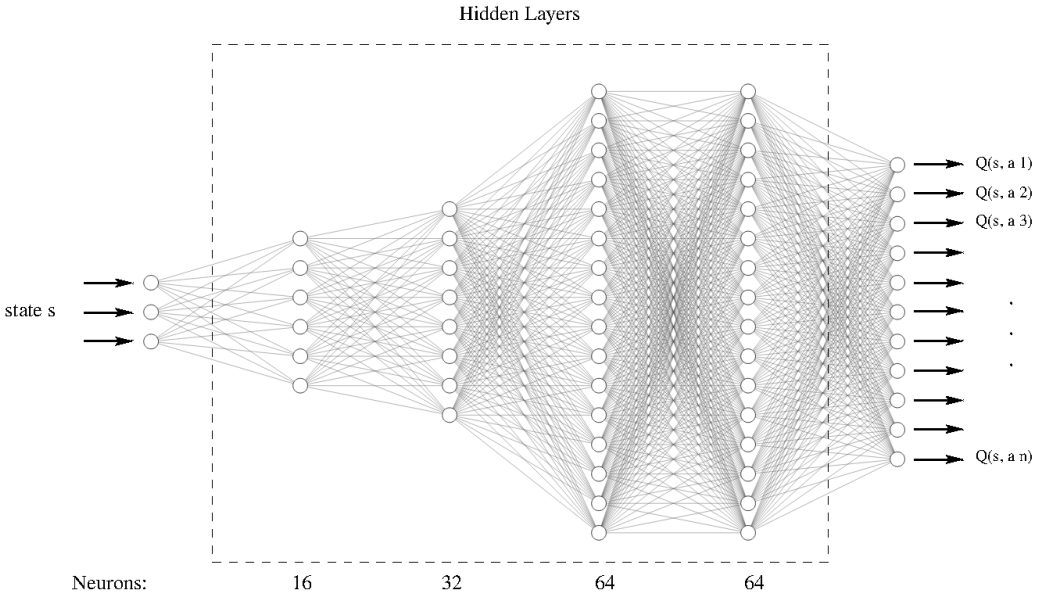

Figure 2: Visual representation of the neural network architecture used in the simulation.

variable taken into account is the cost-to-go, which is dependent on the initial state. The chosen initial state is the stable equilibrium point of the pendulum.

A few tricks were used to make the training process more stable and robust with respect to noisy value estimation. First of all, as mentioned before, we made use of two networks with identical architecture. The first one is the one that goes through the optimization loop, while the second one is used to estimate $Q^{\pi}(s', \pi(s'))$ and is updated every once in a while with the weights of the first one. Secondly, we used Huber loss for training, which is less sensitive to outliers compared to the common MSE. It behaves just the same as MSE for small values, but as MAE for bigger values. Lastly, before performing a gradient descent step, the computed gradients were clipped to the range $[-1, 1]$ to prevent exploding values.

Algorithm 1 Deep Q Network training

Initialize replay memory D to capacity N
Initialize action-value function $\mathcal{Q}$ with random weights ϑ
Initialize target action-value function $\hat{\mathcal{Q}}$ with weights $\hat{\vartheta} = \vartheta$
for episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$
    for t = 0, N **do**
        With probability ε select a random action a_t
        otherwise select $a_t = \text{argmax}_a \, \mathcal{Q}(s_t, a|\vartheta)$
        Execute action a_t and observe reward r_t and next state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        $s_t \leftarrow s_{t+1}$
        Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
        $y_i \leftarrow \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{\mathcal{Q}}(s_{j+1}, a'|\hat{\vartheta}) & \text{otherwise} \end{cases}$
        $L \leftarrow \mathcal{L}(y_i, \mathcal{Q}(s_j, a_j|\vartheta))$
        Perform a gradient descent step on L w.r.t. the network parameters ϑ
        Every C steps update target weights $\hat{\vartheta} \leftarrow \vartheta$
    end for
    $\varepsilon \leftarrow \varepsilon_{end} + (\varepsilon_{start} - \varepsilon_{end}) e^{\frac{episode}{\varepsilon_{decay}}}$
end for

3 Single Pendulum swing-up manoeuvre

The single pendulum swing-up manoeuvre is a classical problem that requires a pendulum, starting at its stable equilibrium point, to find the correct succession of joint torques (i.e. actions) in order to reach and maintain an upright position.

At each episode, the pendulum environment is reset to a random state, i.e. a random pair of angle and angular velocity, and the algorithm will try to learn the correct action to take by means of exploration of the state and action space and exploitation of the previously gathered experience.

The intent of having the algorithm to reset the initial state of the pendulum to a random one at each episode, is to explore the entire state space and find the correct response (in terms of control input provided to the joint motor).
By doing so no matter if the pendulum starts at a random state or in its stable equilibrium point, it will choose the action, at each step of the dynamics of the system, that maximizes its expected reward.

This aspect can be clearly seen in figure 3, in which the V function is represented at each value of the angle of the pendulum, w.r.t. to the upright position, and its angular velocity.

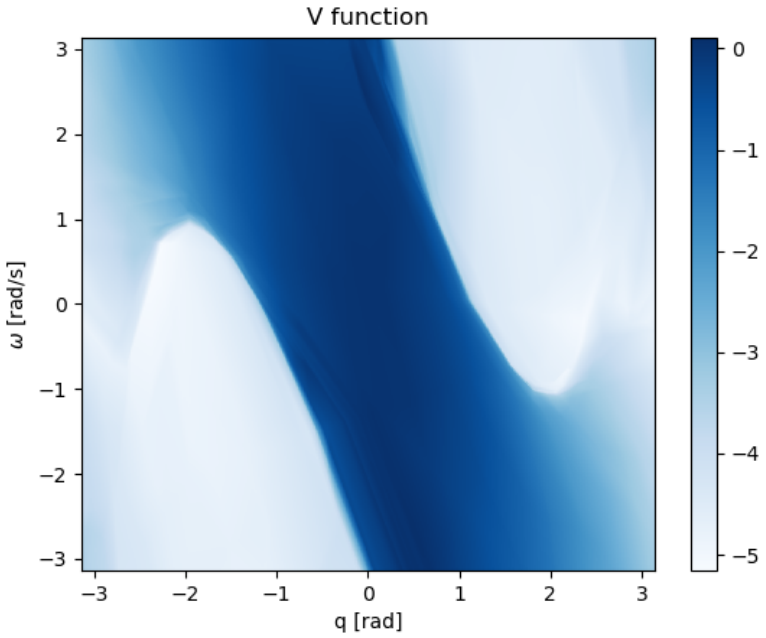

Figure 3: Value function of the single pendulum swing-up manoeuvre

Even more interesting are the results of figure 4, in which the greedy policy after training is displayed as a function of, once again, the angle of the pendulum, w.r.t. to the upright position, and its angular velocity. In the proposed figure, we can notice that the algorithm in order to maximize the expected reward, tries to reach the upright position in the shortest time possible by applying one of the two extreme torques to its motor (lighter and darker areas in the plot).

Once the pendulum has reached the desired position it will maintain it by applying zero torque to its motor and occasionally choose alternative torques in order to make minor adjustments.

The hyper-parameters used during the training are summarized in table 1.

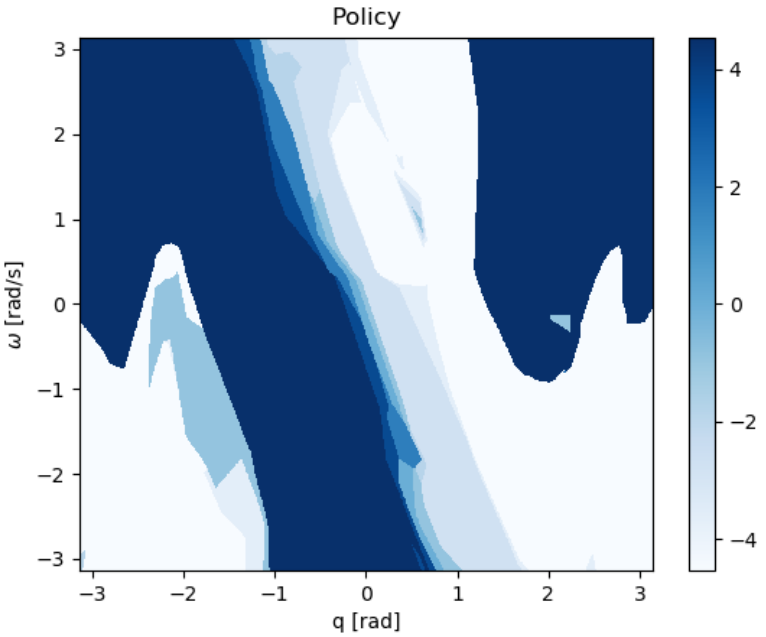

Figure 4: Action selected by the single pendulum for each pair of angle and angular velocity.

Component	Parameter Name	Value
Replay Buffer	MINIBATCH_SIZE	64
	REPLAY_START_SIZE	128
	REPLAY_MEMORY_SIZE	50.000
	REPLAY_SAMPLE_STEP	1
Training	GAMMA	0.95
	LEARNING_RATE	0.001
	MAX_EPISODE_LENGTH	150
	NETWORK_RESET	32
Epsilon Greedy	INITIAL_EXPLORATION	1.0
	EXPLORATION_DECAY	1000
	FINAL_EXPLORATION	0.001
Pendulum environment	N_JOINTS	1
	NU	11
	UMAX	5.0
	DT	0.1
	NDT	10
	NOISE	0.
	WITHSinCos	False

Table 1: Single Pendulum Hyper-parameters

4 Double Pendulum swing-up manoeuvre

The double pendulum environment provides a case of underactuation. In this case, the agent will learn to balance both links in an upright position by only acting on the torque of the joint connected to the main frame.

Due to the high dimensionality of the state representation, it is impossible to provide a graphical representation of the obtained V function for the double pendulum case.

On the other hand, we are reporting the plot of two values that show the evolution of the agent during training (Figure 5). Even though the cost-to-go measured in order to assess the performance of the training steps happens to be quite noisy, the general trend appears to be decreasing, signaling a steady improvement of the model. The model we are providing is the one that resulted after training over 5300 episodes, which from the plots appear to be a point of local minimum for both the cost-to-go and the temporal difference error, after the general decreasing phase of the cost-to-go.

The hyper-parameters used during the training are summarized in table 2.

Component	Parameter Name	Value
Replay Buffer	MINIBATCH_SIZE	64
	REPLAY_START_SIZE	512
	REPLAY_MEMORY_SIZE	100.000
	REPLAY_SAMPLE_STEP	3
Training	GAMMA	0.999
	LEARNING_RATE	0.001
	MAX_EPISODE_LENGTH	300
	NETWORK_RESET	1500
Epsilon Greedy	INITIAL_EXPLORATION	1.0
	EXPLORATION_DECAY	1000
	FINAL_EXPLORATION	0.001
Pendulum environment	N_JOINTS	2
	NU	21
	UMAX	5.0
	DT	0.1
	NDT	10
	NOISE	0.
	WITHSinCos	True

Table 2: Double Pendulum Hyper-parameters

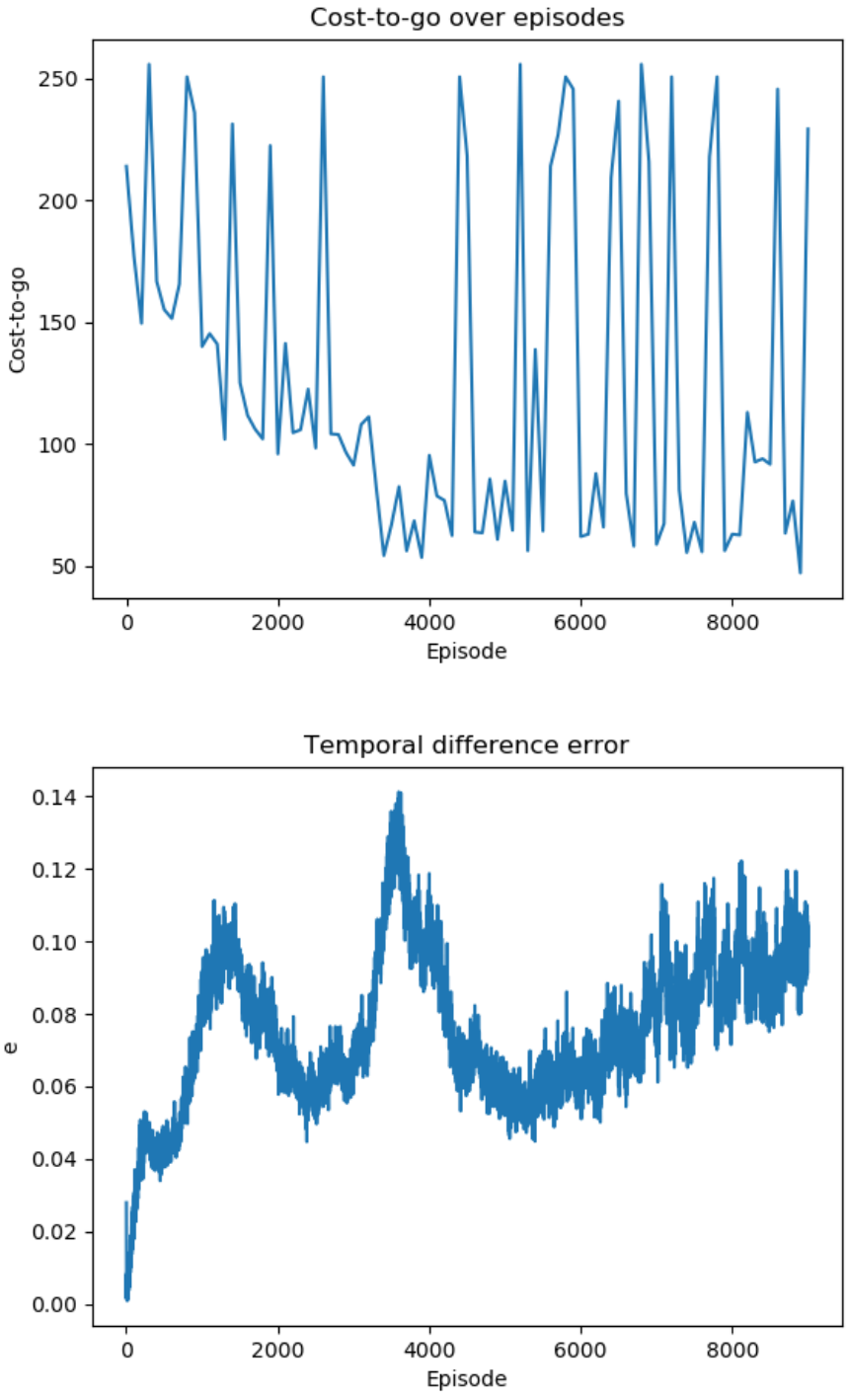

Figure 5: Cost-to-go improvements of the double pendulum training (top) and changes in the training loss (bottom)

5 Conclusion

With the experience gathered from the application of the DQN algorithm to the single and double pendulum, we could clearly notice the advantages brought by this type of approach in the simulation of complex systems:

1. first and foremost, the compactness of the Q function represented as a Neural Network allowed us to infer and display the single and double pendulum motion during the execution of the script, without compromising the training step.
   This aspect turned out to be of key importance to track the model improvement as well as debug the algorithm implementation.

2. Moreover, by keeping the state continuous, the use of neural networks had a smaller impact on memory consumption when compared to the Q learning approach, opening the possibility to update the weights and biases of the action-value function in a memory friendly way.
   This is of key importance from an Operating System stand point given that storing the Q function in a table might result in continuous memory swapping and, as a consequence, in prolonged execution time.

3. The lack of specific guidelines on how to tune the hyper-parameters turned out to be the most time consuming task. Since the state of the art of Deep Reinforcement Learning is still at its early stages, most of the parameters used where derived by matter of trial and error.

4. Both models (i.e. for the single and double pendulum) were able to reach and remain in the upright position. The training process for both the system lasted less than 8000 episodes, which translated in obtaining a solution in less than 30 minutes of computation time using CPU.

5. The close resemblance of the V function obtained in the single pendulum training to the one obtained in the exercise session served as a reference point to verify the correctness of the obtained solution.

6. It has been seen that training the model was possible both by using the joint angles in the state representation and by using the sine and cosine of said angles. However, models trained with the angle representation presented more noise around the initial configuration because of a discontinuity in the angle values. Due to the angle being given in the range $(-\pi, \pi]$, the angle switches from positive to negative and viceversa very often, around the limit values. For this reason, it is not trivial for the model to learn that $\pi = -\pi$ and to model the continuity of the Q-function around those states. In many cases, this leads to a problem that sees the pendulum stuck oscillating with low amplitude around the initial pose. The use of sine and cosine of the joint angles in the state representation helps overcoming this problem, since the discontinuity in the elements of the state vector is no longer present. Sine-cosine representation was used for the double pendulum experiment.

7. Cost-to-go evaluated by using a greedy policy from fixed initial condition was used as a metric to compare the performance of models at training time. However, we believe this might not be the best way of doing such measurement. In fact, improvements in the cost-to-go during training were only noticeable for very high values of γ, such as 0.999. In any other case, one could barely see any trend in the resulting plot. Anyway, since this parameter was not the direct objective of minimization during training, it is only natural that it does not show a consistent decrease over time.

A Navigating the submitted files

In the submission you can find the following files:

```
/
├── env
│   ├── cdPendulum.py
│   ├── conf.py
│   ├── display.py
│   ├── Network.py
│   └── pendulum.py
├── DQNAgent.py
└── test.py
```

A.1 DQNAgent.py

This is the main script, where training of the model is performed. Here you can find the implementation of the high-level pseudocode (1), presented in section 2.

A.2 test.py

This script allows you to load a trained model and see how it performs. In the case of a simple pendulum, the script additionally plots the state value function and the policy function.

A.3 conf.py

This file allows you to choose which environment to use (single or double pendulum) and the hyper-parameters for training and testing. When loading a trained model in the `test.py` script, make sure you use the correct settings.

A.4 Network.py

This file contains the definition of the Neural Network architecture (class `Network`) used to model the Q-function. You can also find the implementation of the Replay Buffer (class `ReplayMemory`) wrapping a `deque` structure.

A.5 cdPendulum.py

This file implements a wrapper of the Pendulum environment (found in `pendulum.py`) discretizing the input torque to a finite action space, but still allowing for continuous state representation (angular position and velocity).

B Hyper Parameters

Component	Parameter Name	Description
Replay Buffer	MINIBATCH_SIZE	Number of Replay Memory samples over which the Q Network is trained
	REPLAY_START_SIZE	Minimum size of Replay Buffer needed for training
	REPLAY_MEMORY_SIZE	Maximum size of Replay Buffer, after that samples are overwritten
	REPLAY_SAMPLE_STEP	Number defining how often to sample from the Replay Buffer
Training	GAMMA	Discount factor used in the Q-Learning update
	LEARNING_RATE	Learning rate
	MAX_EPISODE_LENGTH	Maximum length of an episode before it terminates
	NETWORK_RESET	Number defining how often to reset the target action-value function network
Epsilon Greedy	INITIAL_EXPLORATION	Initial value of epsilon (exploration)
	EXPLORATION_DECAY	Decay factor 2
	FINAL_EXPLORATION	Final value of epsilon (exploitation)
Pendulum environment	N_JOINTS	Number of joints of the pendulum
	NU	Number of discretization steps for the joint torque
	UMAX	Maximum value of torque in absolute value
	DT	Time step
	NDT	Number of Euler steps per integration
	NOISE	Standard deviation of Gaussian noise
	WITHSinCos	If set to True, the state is represented by the tuple $(\cos q, \sin q, \dot{q})$