Optimization Based Robot Control Assignmet 03

Alessandro Assirellli, mat. 231685

February 13, 2023

1 Introduction

Deep Q Network (DQN) is a reinforcement learning (RL) algorithm that is used to approximate the optimal action-value function in RL problems. It was first proposed by Google DeepMind in a 2015 paper titled "Human-level control through deep reinforcement learning".

DQN can learn a control policy for an agent interacting with an environment. It is an extension of the Q-learning algorithm, and it has been widely used in various RL tasks achieving impressive performance in many domains, including playing Atari games and controlling robots. This algorithm can be used when the state space of the system is huge or even continuous. Traditional Q-learning cannot be implemented for these problems, since it is not physically possible to store Q in memory. Contrary to Q-learning, which tries to fill a table containing (state, action, value) triples, DQN tries to approximate the Q function with a nonlinear approximator (Neural Network). This imply that DQN is able to generalize the action-value function Q even for states that have never been explored.

2 Algorithm

The basic idea behind DQN is to approximate the action-value function using a neural network, which is trained to predict the expected future cost to go for each action, given the current state of the environment. The optimal action value function $Q^*(s, a)$ is defined as the minimum future cost achievable by any policy starting from state s, after taking an action a. The bellman equation states that if the action-value function of the next state s' is known for every possible action a', then the optimal strategy is to take the action a' minimizing $Q^*(s', a')$:

$$Q^*(s, a) = c(s, a) + \gamma \min_{a'} Q^*(s', a') \tag{1}$$

While in traditional Q-learning the Bellman equation is directly used to compute an update for Q, in DQN at each iteration a neural network is trained to find the weights θ that reduce the mean squared error in the Bellman equation. To do so, the algorithm uses a technique called fixed Q-targets, which involves using a separate target action-value function $\hat{Q}(s, a, \theta^-)$ to compute the targets for the Bellman equation. The target action-value function is updated periodically to match the current estimate of the action-value function, which helps to stabilize the training. $Q(s, a, \theta)$ is updated based on stochastic gradient descent, where the cost function and gradient are:

$$\begin{aligned} L(\theta) &= \mathbb{E}\{[c(s, a) + \gamma \min_{a'} \hat{Q}(s', a', \theta^-) - Q(s, a, \theta)]^2\} \\ \nabla_\theta L(\theta) &= \mathbb{E}\{[c(s, a) + \gamma \min_{a'} \hat{Q}(s', a', \theta^-) - Q(s, a, \theta)] \nabla_\theta Q(s, a, \theta)\} \end{aligned} \tag{2}$$

To stabilize the learning process, DQN also uses a technique called experience replay, which involves storing transitions (state, action, cost, next state) in a replay memory and sampling mini-batches of transitions used to train the Q network. This helps to decorrelate the transitions and makes the learning process more stable.

Algorithm 1 reports the basic outline of the DQN training process. It consists of a loop over a fixed number of episodes, each of which consists of a loop over a series of time steps. At each time step, the algorithm selects an action using the current estimate of the action-value function by following an epsilon-greedy policy, executes the action, and stores the resulting transition in the replay memory. The algorithm then samples a mini-batch of transitions from the replay memory and uses them to update the action-value function using gradient descent.

Algorithm 1 DQN Algorithm (N,K,M,C)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- \leftarrow \theta$
for *episode* $\leftarrow 0$ **to** M **do**
  Initialize state s_1
  while episode is not ended **do**
    Select action using an epsilon-greedy scheme
    Execute action a_t and observe cost c_t and new state s_{t+1}
    Store transition (s_t, a_t, c_t, s_{t+1}) in replay memory D
    if $D > N$ **then**
      Remove D_1
    end if
    if $t \bmod K = 0$ **then**
      Sample random mini-batch of transitions (s_j, a_j, c_j, s_{j+1}) from D
      Set $\delta_j = c_j(s_j, a_j) + \gamma \min_{a_{j+1}} \hat{Q}(s_{j+1}, a_{j+1}; \theta^-) - Q(s_j, a_j; \theta)$
      Perform a gradient descent step on δ_j with respect to θ
    end if
  end while
  if *episode* $\bmod C = 0$ **then**
    $\hat{Q} \leftarrow Q$
  end if
end for
return Q

The results of training a Deep Q-Network depend on several factors, including the environment, the hyperparameters, and the quality of the experiences collected during training. In successful training runs, Q typically converges over time to a stable solution.

The algorithm hyperparameters have been hand tuned and they are reported in Table 1.

3 Environments

In a reinforcement learning problem, the environment is the system that the agent interacts with. The environment is typically modeled as a Markov Decision Process (MDP), which consists of a set of states, actions, and rewards or costs. The agent receives observations from the environment and takes actions based on these observations. The environment transitions to a new state and provides a cost to the agent based on the action taken. The goal of the agent is to learn a policy that minimizes the cost to go, which is defined as: $J = \sum_{t=0}^{T} \gamma^t c_t$. The environment can be discrete or continuous and stochastic or deterministic. The two environments under analysis are continuous in the state and discrete in the control. They are both deterministic, since the transition function is provided by the dynamics of the systems. Two different environments have been set, the first one (ENV1) is a simple pendulum system, while the second one is a double pendulum (ENV2). ENV1 has a state space of dimension two $[\theta, \dot{\theta}]$, where theta is the angle between the pendulum and the vertical axis. The control space is a discrete state space of dimension 51 corresponding to the 51 possible torque values that the motor can apply. The torque values are linearly spaced between $[-uMax, uMax]$. Note that by selcting an even number of steps, the value corresponding to 0 torque is not among the

Hyperparameter	ENV1	ENV2
Learning rate	1e-3	1e-3
Batch size	128	128
Memory buffer legth (N)	100000	100000
Minimum buffer to train	10000	5000
Min exploration probability	0.0	0.0
Exploration probability decay	1e-4	2e-5
Target network update frequency (C)	10	10
Number of training episodes (M)	1000	2000
Number of step before train (K)	4	4

Table 1: Hyperparameters

possible choices, so an odd number must be used. uMax has been set to 2 [Nm]. The cost function is shaped in order to provide zero cost if the pendulum at the current step is in an upright position with zero velocity and zero torque:

$$\mathrm{c} = 10\theta^2 + 0.1\dot{\theta}^2 + 0.01u^2 \tag{3}$$

ENV2 has a state space of dimension four $[\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$, where θ_1 is the angle between the first pendulum and the vertical axis, while θ_2 is the relative angle between the second pendulum and the first one. The robot is underactuated (only first joint is actuated), the control space is discrete and it consists of 201 possible torque values linearly spaced between $[-uMax, uMax]$. uMax has been set to 2[Nm]. The cost function is:

$$\mathrm{c} = 10\theta_1^2 + 0.1\dot{\theta}_1^2 + 10\theta_2^2 + 0.1\dot{\theta}_2^2 + 0.01u^2 \tag{4}$$

Which is zero when the robot is in its goal configuration.

Table 2 summarizes the environments setup.

	ENV1	ENV2
Observation space	Continuous (2) $\rightarrow [\theta, \dot{\theta}]$	Continuous (4) $\rightarrow [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$
Control space	Discrete (21) $\rightarrow [u_0, \ldots, u_{50}]$	Discrete (201) $\rightarrow [u_0, \ldots, u_{200}]$
Observation space lower bound	$[-\pi, -8]$	$[-\pi, -\pi, -10, -25]$
Observation space upper bound	$[\pi, 8]$	$[\pi, \pi, 10, 25]$
Control Space values	$[-2, -1.92, \ldots, 1.92, 2]$	$[-2, -1.98, \ldots, 1.98, 2]$
Episode length	200	200

Table 2: Environments summary

4 Function approximators

In Deep Q-Network, a function approximator is used to estimate the state action value function, $Q(s, a)$, representing the expected future reward for taking a specific action in a given state. There are various types of function approximators that can be used in RL, including linear functions, polynomial functions, and artificial neural networks (ANNs).

The use of a function approximator in DQN is motivated by the fact that the state-action value function can be too complex to be represented in a lookup table, and the analytical solution is often not available. Function approximators, such as artificial neural networks (ANNs), can learn to approximate complex functions and have the ability to generalize to new data.

The structure of the neural network used in DQN is designed to handle the specific requirements of the reinforcement learning problem, including the non-stationary nature of the environment and the high-dimensional state space. The network typically consists of multiple fully connected layers, with activation functions such as rectified linear units (ReLU) used to introduce non-linearity into the model.

In order to find an optimal control policy for ENV1 and ENV2, ANNs have been used. The ANN architectures are presented in Table 3. Both networks share the same input-output setting,

	ENV1	ENV2
Layer	Dense (32)	Dense (64)
Activation	ReLu	ReLu
Layer	Dense (32)	Dense(128)
Activation	ReLu	ReLu
Layer	Dense (dim_{action})	Dense(64)
Activation	-	ReLu
Layer	-	Dense (dim_{action})

Table 3: Neural networks architecture

which consits of dim_{state} inputs and dim_{action} outputs. In this setting the NN receives as input the state of the system, which is mapped to dim_{action} values, each of them representing the Q value for that state, taking the action corresponding to the index of the node. The output of the network is thus $Q(s, a_0)$, $Q(s, a_1)$, ..., $Q(s, a_{dim_{action}})$. The greedy action is then obtained by just one forward pass and selecting the action associated to the minimum Q.

Figure 1 shows an example of this setting in which a state space of dimension four is used, and mapped to the five possible actions. Each output node contains the action value $Q(s, a_i)$, which is the value of taking the action a_i in the state s. In this example the red bar indicates the value of $Q(s, a_i)$. For this specific case a_3 is clearly the best action to take, since it is minimizing Q.

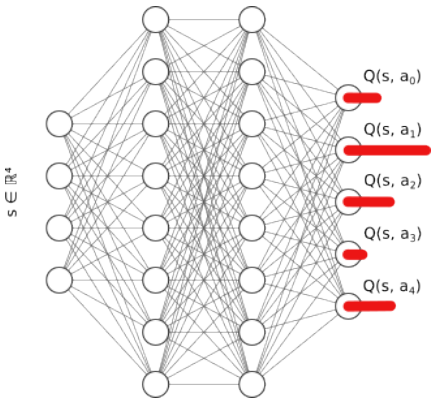


Figure 1: Neural network setting

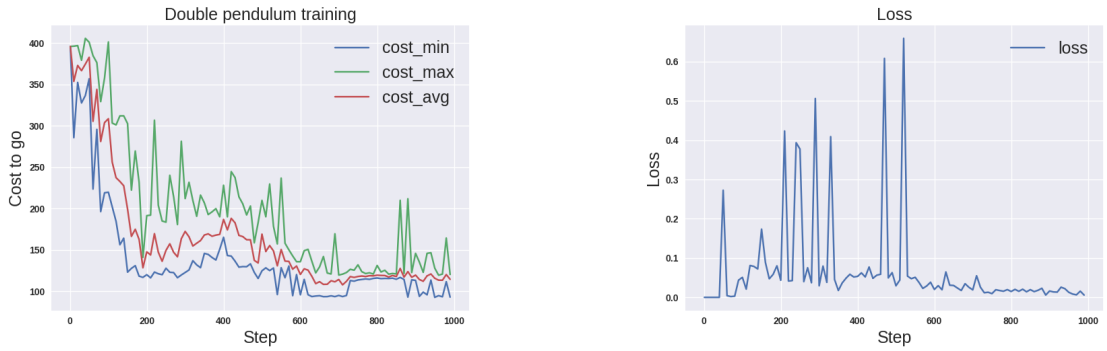
5 Results

The algorithm has been implemented for ENV1 and ENV2. In both cases the action space has been discretized in dim_{action} possible values. In Deep Q-Network algorithms, discretization of the action space can have a significant impact on the training process and the final performance of the agent. The number of discrete actions must be chosen carfully as it rules the expressivness of the final model. When the discretization is performed on a coarse grid the computational cost decreases, however a too grainy grid can result in a too coarse policy and lead to sub-optimal solutions, so a trede-off must be found. The discretization of ENV1 and ENV2 has been hand tuned, and it is described in Section 3.

The training performance of the algorithm is typically measured in terms of the cost payed by the agent in the environment it is trained on, or by loss function used to update the parameters of the algorithm. In this case, the training performance is measured by the cost to go during training. The reason is that, while the algorithm tries to learn $Q(s, a)$, what we are really interested in, is the optimal policy $\pi^*(s)$. Because $Q^*(s, a)$ is only a proxy for $\pi^*(s)$ it is possible that some improvement in estimating $Q^*(s, a)$ does not correspond to an improvement of estimating $\pi^*(s)$. At every training step of the network, which occurs with a frequency C (in algorithm 1), the model changes, as a stochastic gradient descent step is performed. In DQN there is no guarantee that the performances improve as the training proceeds. In order to keep track of the best policy, the value function for each policy is stored, then the weights of the model are saved every time a policy $\pi(s)$ with lower $V^{\pi}(s_0)$ is found.

5.1 ENV 1

Figure 2a shows the average cost to go during training for ENV1. The average is computed considering 10 episodes. As can be seen, the average cost decreases until around 650 iterations, where



(a) Cost to go

(b) Loss

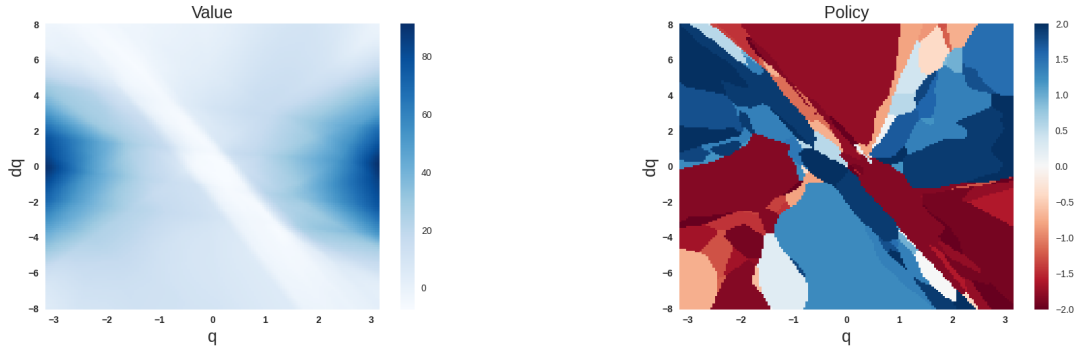
Figure 2: Training on ENV1

a plateau is reached. At this point the cost to go remains more or less constant, meaning that the algorithm has reached convergence and it cannot learn more. Figure 2b shows the mean squared error of the estimated Q function with respect to the Q target. Also the loss function reaches a plateau after around 650 iterations which confirms that after 650 iterations the Q value converged to the Q target. The algorithm has been trained for 1000 iterations and took ≈ 1600 seconds.

The optimal value function and optimal control policy can be retrieved as:

$$\begin{aligned} V^*(s) &= \min_a Q^*(s,a) \\ \pi^*(s) &= \operatorname*{argmin}_a Q^*(s,a) \end{aligned} \tag{5}$$

In order to plot $V^*(s)$ and $\pi^*(s)$ as colormaps the two functions have been sampled on a square grid 151X151, Figure 3 shows the results.

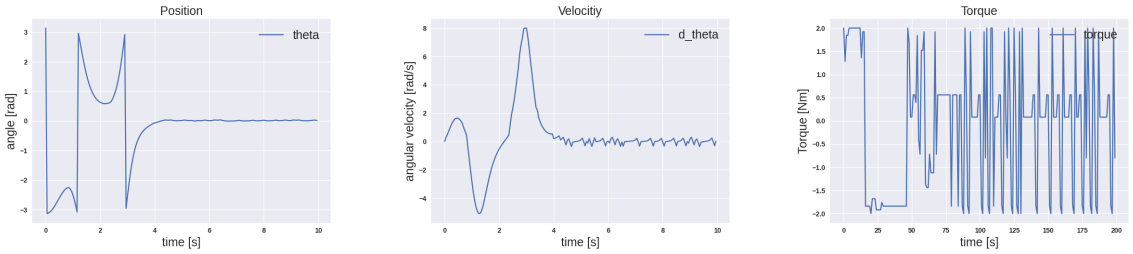


(a) Optimal value function (b) Optimal policy

Figure 3: Optimal value and policy functions

As can be seen the Value function present, with good approximation, an odd symmetry. This is reasonable, since the cost to go by starting rotating clockwise should be the same as starting counterclockwise and then behave oppositely. Also the optimal control policy presents a certain level of symmetry. In Figure 3b it is possible to recognize color blocks, whose distribution is fairly symmetric. The differences are the shades inside each block. The reason why shades are not that symmetric can be, as previously discussed, due to the fact that the algorithm is not trying to find directly $\pi^*(s)$, which would have been symmetric, but it finds it through an approximation of $Q^*(s,a)$.

Figure 4 shows the simulation results of the pendulum swing up.

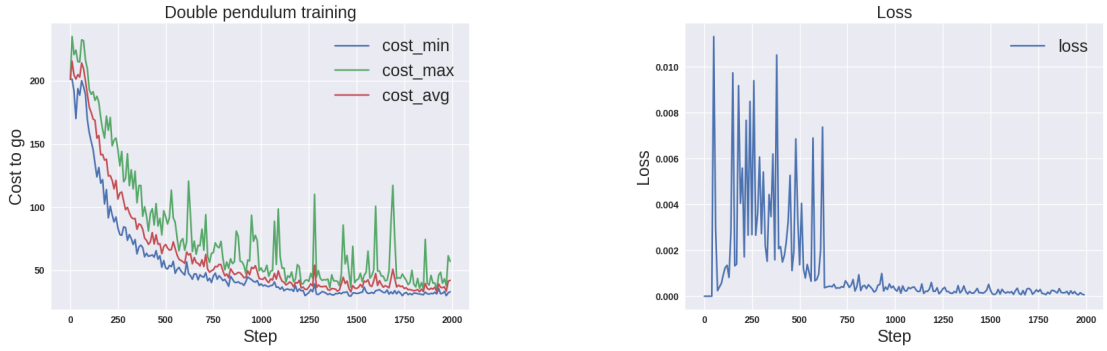


(a) Position (b) Velocity (c) Torque

Figure 4: Simulation of ENV1

5.2 ENV 2

As previously discussed, this environment consist of an underactuated double pendulum which has to perform a swing up maneuver. Figure 5a shows the cost to go during training.

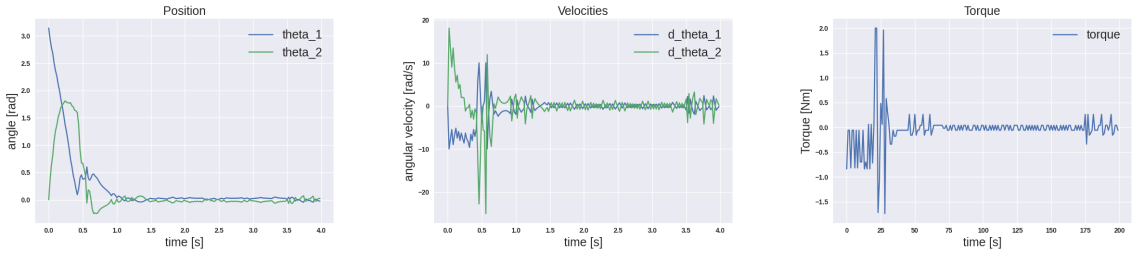


(a) Cost to go

(b) Loss

Figure 5: Training results

The average is computed considering 10 episodes. As can be seen, the cost decreases up to around 1250 iterations, where a plateau is reached. Also in this case the algorithm converged to the optimal action value function, as the mean squared error between the estimated Q and the Q target converges after around 1000 iteration. The algorithm has been trained for 2000 episodes and took $\approx$ 5500 seconds. Figure 6 shows the simulation results of the double pendulum swing up problem. As can be seen, the controller reaches very good accuracy in performing the task. The applied torque, after the first phase of swing, continues to jump between positive and negative values. The reason is that being the torque discrete, the agent doesn't have the freedom to select any torque value. Because of that, the available torque levels may not be fine enough to end up with 0 control torque. Overall the system performs really well and constantly applying positive and negative torque it's certainly the best thing the algorithm can do to stabilize the system.



(a) Positions

(b) Velocities

(c) Torque

Figure 6: Simulation of ENV2

6 Additional work

The DQN algorithm described in this report succeeded in controlling both the environments, as presented in Section 5. Over the years researchers have found many different techniques to improve the learning performances of DQN. An interesting solution which has not been tested, is the prioritized experience replay scheme, in which transitions with a high TD error are presented more frequently to the algorithm. The idea is that the agent can learn more from some transitions than from others. Some transitions may be surprising, while other redundant. Prioritized experience reply tries to speed up the learning by presenting less often the transitions that add little to the learning.
There are other possible solutions such as Duelling Network, Noisy Net and Actor Critic schemes which have not been tested.

I find quite attractive the idea behind Duelling Network, and it would have been interesting to see if it improves the learning performances in these two environments. Duelling Network is intended to facilitate the learning when there are states for which whatever action is taken, the cost would be very similar. In those cases there is no need to learn the value for every action. While I understand the advantages of this scheme in some environments, as the Enduro example provided by by Ziyu Wang et al. in *"Dueling Network Architectures for Deep Reinforcement Learning"*, it does not seem obvious to me that it should lead to better performanecs in the two environments under analysis.