16-662 Robot Autonomy - Homework 4

Q-Learning

Professor: Oliver Kroemer
TAs: Vibhakar Mohta (vmohta) and Abhinav Gupta (ag6)
Total Points: 100

1. Introduction

This homework will introduce you to Q-learning, which is a powerful off-policy model-free reinforcement learning algorithm. It is a fundamental algorithm in reinforcement learning that gained popularity after it beat human experts in three Atari 2600 games back in 2013 [1]. We will focus on applying Q-learning to a grid world, where our aim is to avoid hitting the red blocks and reach the gold-colored block, as shown in Figure 1. The agent is shown in blue.

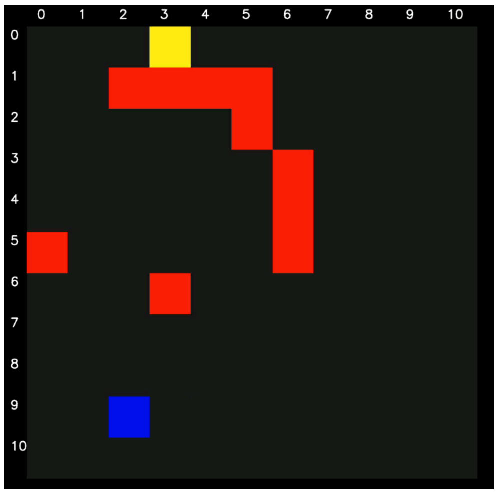


Figure 1. Gridworld Setup

2. Environment Setup

As reinforcement learning algorithms learn from experiences, we will start by building a simulator that, given an action, will return the next observation, the reward for the action, and whether the run terminated or not after executing this action. We define the reward as follows:

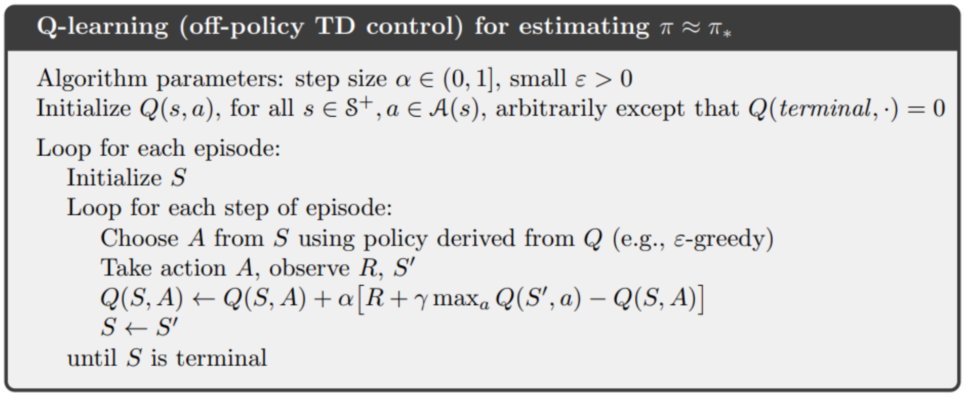
$$\text{Reward}(s, a) = \begin{cases} +500 & \text{if } s' = \text{GOLD} \\ -500 & \text{if } s' = \text{RED BLOCK} \\ -1 & \text{otherwise} \end{cases}$$

The action space here is just [“UP”, “DOWN”, “LEFT”, RIGHT”], and states are the coordinates of the robot in the gridworld. The next state **s’** is reached after applying action **a** at state **s.** The environment we work with is stochastic and executes a random action with probability ***rho*** and the given action with probability ***(1-rho).*** Fill out the TODOs in the class `GridWorld` in the file **environment.py**, defining the environment reward and transition function.

3. Defining Agents

Random Agent
To benchmark performance, we first define a random agent that executes possible actions with equal probability. Fill out the **get_action** function in the `RandomAgent` class in the file **agents.py**.

Q Agent
To implement the Q agent, we will use the vanilla Q learning updates from *S. Sutton & G. Barto* [2], as shown in Figure 2. Fill out the TODOs in the class `QAgent` class in the file **agents.py**.



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 2. Q-learning Algorithm

Some implementation guidelines:

- The Q table should be of size (N_States x N_Actions). You can use a dictionary
- As terminals are not known in advance generally, initialize all Q values randomly
- Perform Epsilon Greedy exploration in the **get_action** function if the explore flag is true (during training)

3. Training the Q-Agent

Implement the **train_q_agent** function in the file **q_learning.py.** Remember to call the **get_action** function with *explore=true* and update the q value table after each step. You should now be able to train the agent by running the **q_learning.py** file after setting the random seed.

4. Inference (100 points)

Now that we have everything in place, it is time to train the Q Agent and compare its performance with the random agent! Running **q_learning.py** should generate the following:

- An image **q_learning_training.png** that shows the average rewards while training
- An image **q_values.png** that plots the action with the highest Q value at each state
- In the visualizations/ folder, videos of three rollouts of the random agent and the Q agent (after training).

Deliverables:

1. **Answer the following questions:** [30 Points]
    a. While training, why do the rewards randomly dip to a low number?
    b. Do the arrows plotted in **q_values.png** make sense? What do they represent?
    c. Does the Q Agent always follow the optimal path with respect to Q values in the videos generated? Why might it not be doing so sometimes?
2. Attach **q_learning_training.png** [10 Points]
3. Attach **q_values.png** [10 Points]
4. Report the Average rewards obtained by the Random Agent and Q agent in the benchmark runs (this should be printed by default) [10 Points]
5. **Role of Hyperparameters:** [40 Points]
   How do the following hyperparameters empirically alter the performance of the Q Agent? Why do you think so?
    a. `Q_ALPHA`
    b. `Q_GAMMA`
    c. `Q_EPSILON`
    d. `ENV_RHO`

5. **Bonus:** Own Environment

Try setting up your own gridworld with

1. An easier map (fewer obstacles en route target)
2. A harder map (dense obstacles en route target)

Keeping hyperparameters fixed, how does the performance of the **Random Agent** and **Q Agent** vary in these environments? Does the difference in performance increase or decrease with an increase in the environment difficulty?

References

1. Playing Atari with Deep Reinforcement Learning (LINK)
2. Reinforcement Learning: An Introduction: Richard S. Sutton & Andrew G. Barto (LINK)