Robot Learning
Exercise 3

November 30th, 2023

TAs: Gabriele Tiboni (*gabriele.tiboni@polito.it*), Andrea Protopapa (*andrea.protopapa@polito.it*), Elena Di Felice (*s303499@studenti.polito.it*)

1 Introduction

This exercise will ask the student to implement a simple reinforcement learning pipeline to solve the cartpole environment. In particular, we will consider a tabular version of temporal-difference methods to learn how to control the cartpole. In RL literature, "tabular" environments refer to MDPs with discrete state space and action spaces, namely value functions may be represented as a simple table-like structure (e.g. a numpy array).

To proceed with the assignment, clone the starting code from the repository at the dedicated webpage and follow the steps listed below in this document.

Your submission should consist of (1) a **PDF report** containing **a discussion of the results and answers to the questions** asked in these instructions, and (2) the **code** used for the exercise.

The **deadline to submit** the report and code through the Github Classroom is **December 14th, at 23:59.**

2 Preliminaries

This assignment asks for the implementation of Q-learning for solving the cartpole environment. A good preliminary knowledge of the theoretical aspects behind Q-learning is highly suggested before dealing with the assignment.
In particular, make sure you revise the course slides of Lecture 8c, and Chapters 3, 5, and 6 of [1].

3 Tabular Q-learning

Q-learning is a fundamental algorithm in the literature of Reinforcement Learning, which falls under the umbrella of off-policy, temporal-difference methods.

Recall the Cartpole environment introduced in Exercise 2a.

Task 3.1 Implement Q-learning for the Cartpole environment in the file `qlearning.py`. Here, we consider the tabular version of a Reinforcement Learning problem where states are discrete and actions are discrete. Deploy an epsilon-greedy policy as strategy to deal with the exploration-exploitation trade-off at training time. In particular, compare using a constant value of `epsilon=0.2`, to reducing the value of epsilon over time with GLIE (*greedy in limit with infinite exploration*). For implementing a GLIE schedule of epsilon, refer to the following formula

$$\epsilon_k = \frac{b}{b+k}$$

and tune `b` such that `epsilon` reaches `0.1` after `k=20000` episodes (round `b` to nearest integer).

***Hint**: the states in Cartpole are continuous, while tabular methods can only be applied to discrete state spaces. Work around this by discretizing Cartpole states to a finite grid (see starting code in qlearning.py).*

***Hint:** refer to slides 92-96 of Lecture 8c, and to Section 6.5 of [1], for details on how to implement the Q-learning update rule.*

Guiding questions:

- How do the performances differ between a fixed epsilon vs. the GLIE epsilon schedule?
- Can you plot the return obtained at each training episode, to view how the training process has progressed?

Task 3.2 Assuming a greedy policy, use the learned Q-function to compute the associated value function. Plot the heatmap of the computed value function in terms of x and θ, and therefore average the values over $\dot{x}$ and $\dot{\theta}$ for plotting.

***Hint**: for plotting the heatmap you can use Matplotlib `pyplot.imshow(array)` or Seaborn `seaborn.heatmap(array)` functions.*

***Note:** using a greedy policy for the final evaluation is a sensible choice as, if the learned Q-function converged to the optimal one, then a greedy policy would effectively be an optimal policy (revise Bellman Optimality Equations in slide 48 of Lecture 8a).*

Guiding questions:

- What do you think the heatmap would have looked like before the training, after a single episode, and half way through the training?
- Can you intuitively interpret why the heatmap looks the way it does? At least partially.

Task 3.3 Set epsilon to zero, i.e. making the policy greedy w.r.t. the current Q-function estimate. Therefore, run the code again with:

(a) keeping the initial Q-function values at zero;
(b) setting the initial Q-function values at 50 for all states and actions;

Guiding questions:

- In which case does the model perform better?
- Why is this the case? How does the initialization of Q values affect exploration?

Hint: try going through Sec. 2.6 of [1] as a further resource for explaining the behavior in task 3.3.

Question 3.4 Can Q-learning be used in environments with continuous state spaces, i.e. with function approximation?

Question 3.5 Can Q-learning be used in environments with continuous action spaces? If any, which step of the algorithm would be problematic to compute in the continuous action space case?

5 Extra: Q-learning with function approximation

Implement Q-learning with function approximation, i.e. without assuming a discrete state space. In the case of continuous state spaces, the learned Q-function must be an approximated, parameterized version of the true Q-function. This way, we interpolate the true Q-function from data and generalize to states that have not been previously explored.

To do this, implement a simple Neural Network in PyTorch, and use gradient descent to update your current estimate of the q-function.
Hint: perform batch updates for better performances, namely update your network with a gradient computed on a batch of recent transitions rather than a single timestep.

Refer to Chapter 11 of [1] for Off-policy methods with Function Approximation.

Note: this extra step might be considerably time-consuming. In order to implement Q-learning with function approximation, one must consider semi-gradients for the update rule—i.e. the gradients computed by PyTorch backpropagation algorithm should not propagate through the target (bootstrapped) return, despite its value comes from the same network we wish to update. In turn, this is often implemented with *target networks,* i.e. a copy of the current network that is frozen, used to compute the target values, and then updated with the most recent network after a few iterations. We suggest the students consider extra time to complete this step, or to even attempt it after the completion of the course.

References

[1] "Reinforcement Learning: An introduction (Second Edition)" by Richard S. Sutton and Andrew G. Barto, PDF