4180/5180: Reinforcement Learning and Sequential Decision Making (Fall 2022) Christopher Amato
Northeastern University Due Sep 14, 2022

Exercise 0: An Invitation to Reinforcement Learning

Please remember the following policies:

- Exercise due at **11:59 PM EST Sep 14, 2022**.
- Submissions should be made electronically on Canvas. Please ensure that your solutions for both the written and programming parts are present. You can upload multiple files in a single submission, or you can zip them into a single file. You can make as many submissions as you wish, but only the latest one will be considered.
- For **Written** questions, solutions may be handwritten or typeset. If you write your answers by hand and submit images/scans of them, please please ensure legibility and order them correctly in a single PDF file.
- The PDF file should also include the figures from the **Plot** questions.
- For both **Plot** and **Code** questions, submit your source code in Jupyter Notebook (.ipynb file) along with reasonable comments of your implementation. Please make sure the code runs correctly.
- You are welcome to discuss these problems with other students in the class, but you must understand and write up the solution and code yourself. Also, you *must* list the names of all those (if any) with whom you discussed your answers at the top of your PDF solutions page.
- Each exercise may be handed in up to two days late (24-hour period), penalized by 10% per day late. Submissions later than two days will not be accepted.
- Contact the teaching staff if there are medical or other extenuating circumstances that we should be aware of.

The purpose of this exercise is to understand the important basis of reinforcement learning that is *agent-environment interaction* (See the Figure below). In this exercise, you are asked to complete the implementation of the Four Rooms environment, implement an agent/policy, and run some empirical experiments.

Gym is a popular open source python library for developing and comparing reinforcement learning algorithms. It provides a standard template for agent-environment interaction. By following the standard API here, the implementation can be easily used by other researchers. In this exercise, we will consider a simple version of the API that contains only the following methods:

- `reset function`: Resets the environment to an initial state and returns the initial observation
- `step function`: Run one time step of the environment's dynamics.

We provide a scaffolding code in the **Jupyter Notebook**. Please read the code carefully and make sure you understand them before you start to implement the required components marked as ***"CODE HERE"***. Implement your solution in **Python 3** and include brief comments about your implementations.

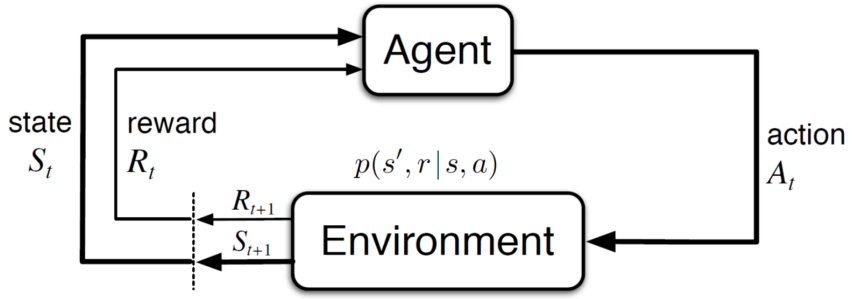
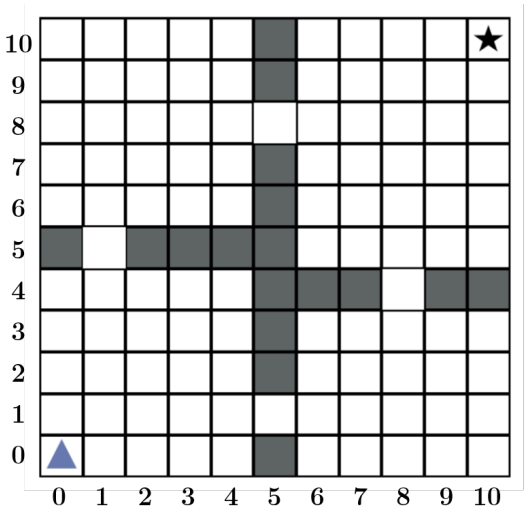


Figure 3.1: The agent–environment interaction in a Markov decision process.



1. **1 point. Code:** Now, we will implement the Four Rooms environment together. Please read the descriptions below carefully. Complete the implementation of the Four Rooms environment. In particular, you are responsible to implement the `step function` only.

   At every time step, the `step function` should take in the current state s and desired action a, and return the next state s' as well as the reward r. The specification of the domain is:

   - The state consists of integer (x, y) coordinates, where $x \in [0, 10]$ and $y \in [0, 10]$.
     The shaded cells are walls and the agent should never be in those cells.
   - The feasible actions are `LEFT`, `DOWN`, `RIGHT`, `UP`. All four actions may be taken at any valid state.
   - To determine the effect of taking an action in a given state:
     - The agent typically moves in the specified direction
       (i.e., taking `RIGHT` in $(2, 3)$ typically moves the agent to $(3, 3)$).
     - However, there is occasional noise in the domain (e.g., the agent slips).
       With probability 0.8, the agent moves in the correct direction. With probability 0.1 each, it moves in one of the two "perpendicular" directions.
       For example, taking `RIGHT` in $(2, 3)$ will take it to $(3, 3)$ with probability 0.8, to $(2, 4)$ with probability 0.1 (the effect of the `UP` action), and to $(2, 2)$ with probability 0.1 (the effect of the `DOWN` action).
     - If the result of the noisy action causes it to collide into a wall, or take the agent out of bounds, then the state does not change.
   - The agent starts in $(0, 0)$ (triangle in figure), and the goal state is $(10, 10)$ (star in figure).
   - Taking any action from the goal state $(10, 10)$ teleports the agent to the start state $(0, 0)$ (i.e., reset).
   - If the resulting state is the goal state, the reward returned is $+1$; otherwise, it is 0.
   - Check your understanding: Moving $a =$ `UP` from $s = (0, 10)$ will result in $s' = (0, 10)$ with probability 0.9 and $s' = (1, 10)$ with probability 0.1, and will return a reward of $r = 0$. In other words, taking the action `UP` from the state $(0, 1)$ should return `(0,10), 0` 90% of the time, and `(1,10), 0` 10% of the time.

2. **1 point.** **Code:** Implement a `manual` policy, and an `agent` that interacts with the simulator and the policy.

   - The *policy* is a state-action mapping $\pi : S \to A$, i.e., a function that produces an action given a state as input. Here, implement a policy that queries you to input an appropriate action for the given state. You may read from standard input.
   - The *agent*, as shown in the diagram on the previous page, is the entity that provides actions to the environment (simulator), and receives next states and rewards from the environment. Here, the agent should be in charge of getting the current state s, querying the policy for an appropriate action $a = \pi(s)$, calling the simulator with (s, a) to get (s', r), and repeating.

   Run the agent with you providing the action on each step. Check that the simulator working as expected, and that you are able to get to the goal reliably.

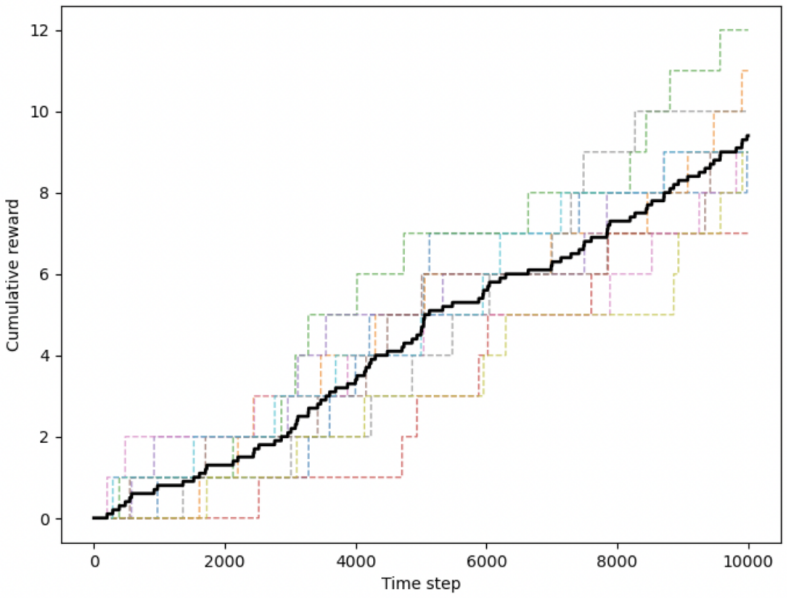
   You will probably get bored of interacting with the agent rather quickly! Let's try to automate this.

3. **1 point.** **Code:** Implement a `random` policy. A random policy outputs one of the four actions, uniformly at random. This is an example of a stochastic policy.

   How well does this policy perform? In reinforcement learning, the objective is to maximize the expected sum of rewards, i.e., a policy is "better" if, on average, it leads to greater cumulative reward. Since the environment and policy in this case are both stochastic, the obtained rewards (and the number of steps to obtain them) will be different every time we run them (unless we take care to use the same seed for pseudorandom number generation – a very good tool for debugging and controlling experiments). To get a good overall sense of performance, we should run our agents for many steps, over many trials.

   Plot: Run your random-policy agent for a total of 10 trials, 10^4 steps in each trial, and produce a cumulative reward plot similar to the one shown below. The dotted lines are the cumulative reward curves for each of 10 trials, and the thick solid black line is the average of those 10 dotted curves.
   We plotted this using `matplotlib.pyplot`. The plotting code is provided. You can use it or write your own one if you are interested.

   Written: How do you think this compares with your manual policy? (You do not have to run your manual policy for 10^4 steps!) What are some reasons for the difference in performance?



4. **1 point.** **Code:** Devise and implement at least two more policies, one of which should be generally worse than the random policy, and one better.
   Written: Describe the strategy each policy uses, and why that leads to generally worse/better performance.
   Plot: Show the cumulative reward curves of each, similar to Q3.

   You may use any algorithms you know about to do this, including reinforcement learning algorithms, but this should not be necessary. Explain you method.