CS7180: Special Topic - Artificial Intelligence (Spring 2022) Christopher Amato
Northeastern University Due Feb 25, 2022

Exercise 4: Monte-Carlo Methods

Please remember:

- Exercise due at **11:59 PM EST Feb 25, 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 along with reasonable documentation.
- 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.

1. **1 point.** (RL2e 5.9, 5.4) *Incremental implementation of Monte-Carlo methods.*
   Written:
   (a) Modify the algorithm for first-visit MC policy evaluation (Section 5.1) to use the incremental implementation for sample averages described in Section 2.4.
   (b) The pseudocode for Monte Carlo ES is inefficient because, for each state-action pair, it maintains a list of all returns and repeatedly calculates their mean. It would be more efficient to use techniques similar to those explained in Section 2.4 to maintain just the mean and a count (for each state–action pair) and update them incrementally. Describe how the pseudocode would be altered to achieve this.

2. **2 point.** (RL2e 5.2, 5.5, 5.8) *First-visit vs. every-visit.*
   Written:
   (a) Suppose every-visit MC was used instead of first-visit MC on the blackjack task. Would you expect the results to be very different? Why or why not?
   (b) Consider an MDP with a single nonterminal state and a single action that transitions back to the nonterminal state with probability p and transitions to the terminal state with probability $1-p$. Let the reward be $+1$ on all transitions, and let $\gamma = 1$. Suppose you observe one episode that lasts 10 steps, with a return of 10. What are the first-visit and every-visit estimators of the value of the nonterminal state?
   (c) **[Extra credit.]** Read and understand example 5.5 first. The results with Example 5.5 and shown in Figure 5.4 used a first-visit MC method. Suppose that instead an every-visit MC method was used on the same problem. Would the variance of the estimator still be infinite? Why or why not?
   Code/plot: Implement Example 5.5 and reproduce Figure 5.4 to verify your answer.

3. **2 points.** *Blackjack.*
   Code/plot:
   (a) Implement first-visit Monte-Carlo policy evaluation (prediction).
   Apply it to the Blackjack environment for the "sticks only on 20 or 21" policy to reproduce Figure 5.1.
   (b) Implement first-visit Monte-Carlo control with exploring starts (Monte-Carlo ES).
   Apply it to the Blackjack environment to reproduce Figure 5.2.
   Note that the reset mechanism already selects all states initially with probability > 0, but you must ensure that all actions are also selected with probability > 0.

Useful tools for implementation:

- Instead of writing your own Blackjack environment, we recommend that you use the implementation provided by OpenAI Gym, or at least refer to it closely if you are re-implementing your own version. This would also be a good opportunity to start setting up and learning about the library.
- For installation instructions and a brief introduction: `https://gym.openai.com/docs/`
- Once you have installed Gym, you can instantiate the environment by calling:
```
import gym
env = gym.make("Blackjack-v1")
```
- For more specifics on the interface and implementation of Blackjack, see the source code at: `https://github.com/openai/gym/blob/master/gym/envs/toy_text/blackjack.py`
- To plot the value functions and policies, consider using: `matplotlib.pyplot.imshow`

4. **2 points.** *Four Rooms, re-visited.*
We are now finally ready to re-visit the Four Rooms domain from Ex0, now with better learning algorithms. First, we must make the domain episodic to apply Monte-Carlo methods. Take your implementation from Ex0, but instead of teleporting to $(0, 0)$ after you reach the goal, make the goal a terminal state (i.e., end of episode). Also, to encourage reaching the goal faster, we will use a discount factor of $\gamma = 0.99$.
In addition, consider adding a timeout to your episodes, i.e., an episode terminates after some maximum number of steps. For example, a timeout of $T = 459$ may be reasonable, since this is the threshold for which $\gamma^T < 0.01$ (i.e., even if the agent reaches the goal after T, it will experience < 0.01 return).

   (a) **Code:** Implement on-policy first-visit Monte-Carlo control (for ε-soft policies).
   Apply it to the original goal state of $(10, 10)$, as well as some other randomly chosen goal states.
   (To choose a random goal state, select a random goal state before any trial, instead of using $(10, 10)$. Ensure that this goal state should remains fixed throughout all trials.)
   Verify that it learns to reach these (unknown) goals.

   (b) **Code/plot:** Let us focus on the $(10, 10)$ goal, which is initially unknown to the agent.
   To verify the agent is learning, plot learning curves similar to those in Ex1.
   - The horizontal axis should be in episodes; the vertical axis should be each episode's discounted return.
   - Plot curves for $\varepsilon = 0.1, 0.01, 0$. For clear trends, running for 10 trials with 10^4 episodes within each trial is recommended, but if it is too time-consuming you may run less.
   - As in Ex1, provide $1.96\times$ (standard error) confidence bands, as well as an upper bound line.
   You can find an upper bound by solving for the optimal value using the known dynamics model, or estimate an upper bound based on the length of the shortest path.

   (c) **Written:** Explain how the results of the $\varepsilon = 0$ setting demonstrate the importance of doing exploring starts in Monte-Carlo ES.

5. **1 point.** (RL2e 5.10, 5.11) *Off-policy methods.*
Written:

(a) Derive the weighted-average update rule (Equation 5.8) from (Equation 5.7). Follow the pattern of the derivation of the unweighted rule (Equation 2.3).

(b) In the boxed algorithm for off-policy MC control, you may have been expecting the W update to have involved the importance-sampling ratio $\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$, but instead it involves $\frac{1}{b(A_t|S_t)}$. Why is this correct?

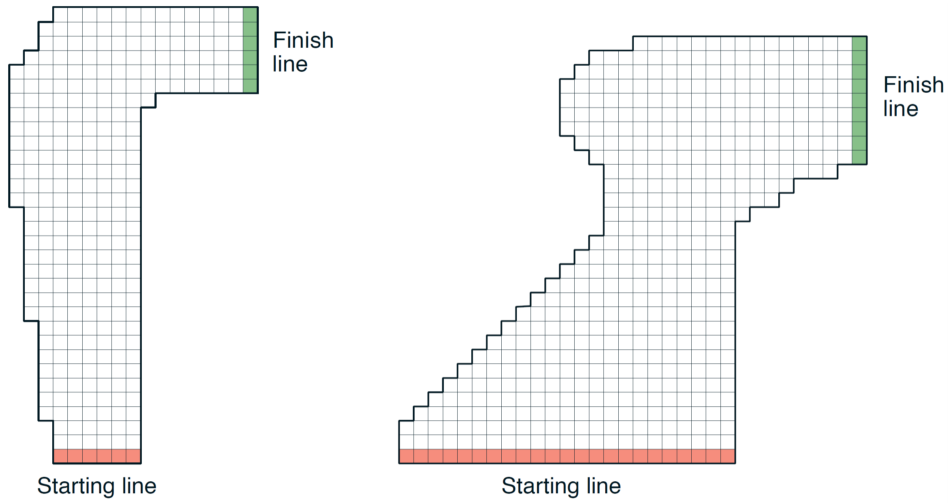

Figure 5.5: A couple of right turns for the racetrack task.

6. **2 points.** (RL2e 5.12) *Racetrack.*
Consider driving a race car around a turn like those shown in Figure 5.5. You want to go as fast as possible, but not so fast as to run off the track. In our simplified racetrack, the car is at one of a discrete set of grid positions, the cells in the diagram. The velocity is also discrete, a number of grid cells moved horizontally and vertically per time step. The actions are increments to the velocity components. Each may be changed by $+1$, -1, or 0 in each step, for a total of nine (3×3) actions. Both velocity components are restricted to be nonnegative and less than 5, and they cannot both be zero except at the starting line. Each episode begins in one of the randomly selected start states with both velocity components zero and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random position on the starting line, both velocity components are reduced to zero, and the episode continues. Before updating the car's location at each time step, check to see if the projected path of the car intersects the track boundary. If it intersects the finish line, the episode ends; if it intersects anywhere else, the car is considered to have hit the track boundary and is sent back to the starting line. To make the task more challenging, with probability 0.1 at each time step the velocity increments are both zero, independently of the intended increments.

   (a) **Code:** Implement the racetrack domain (both tracks). Apply on-policy first-visit Monte-Carlo control (for ε-soft policies), with $\varepsilon = 0.1$ – ideally, this would be a simple application of the code from Q4(a).
   Plot: For each racetrack, plot the learning curve (multiple trials with confidence bands), similar to Q4(b).

   (b) **Code:** Implement off-policy Monte-Carlo control and apply it to the racetrack domain (both tracks). For the behavior policy, use an ε-greedy action selection method, based on the latest estimate of $Q(s, a)$ – i.e., this is similar to on-policy Monte-Carlo control, except that the target policy is kept as a greedy policy.
   Plot: For each racetrack, plot the learning curve (multiple trials with confidence bands), similar to Q4(b). Show the performance of both the *behavior* and *target* policies; in the latter case, do this by collecting one rollout after each episode of training, which is collected solely for evaluation purposes. (The point of this is to inspect performance on the policy we actually care about, not the one used for data gathering.) Additionally, visualize several rollouts of the optimal policy; consider using: `matplotlib.pyplot.imshow`

   (c) **Written:** Do you observe any significant differences between the on-policy and off-policy methods? Are there any interesting differences between the two racetracks?

   Tip: You can find NumPy arrays containing the racetracks in `racetracks.py`.
   Think about which racetrack you expect is easier, and develop your methods in that domain.