UNIVERSITY OF COLORADO - BOULDER
Robotics Program

ROBO 5302 (CSCI 4302/5302) - Advanced Robotics

Homework #3 (Assigned: Fri 9/26, Due: Fri 10/24 11:59pm on *Canvas*)
Markov Decision Processes

Instructions

- Submission will take the form of a link to your GitHub branch that contains the work for this assignment. This branch needs to contain all of your code and your writeup.
- Your writeup should be well organized and must contain the relevant figures that your code generated (such as value functions and policies) in additional to you documenting your approach to the problems, issues encountered and how you solved them. The writeup should also contain your answers to the written problems posed in this document.
- The provided skeleton code clearly shows where students need to add code. You are advised against changing the core functionalily of the code that is provided (except for debugging purposes).

Part 1: Setup

First, you will need to install matplotlib, numpy, scipy, and the Gymnasium Environment for Python: (if you have used gym before, gymnasium is the new supported version. It has some minor changes, but should feel mostly the same.)

```
pip install numpy
pip install gymnasium
pip install matplotlib
pip install scipy
```

or

```
conda install numpy
conda install gymnasium
conda install matplotlib
conda install scipy
```

Next, you need to install the HW4-RL python package. Make sure you are in the correct directory:

```
cd /path/to/your/code/HW4-RL
```

Install the package using pip:

```
pip install .
```

For more installation and usage instructions see README.md.

Part 2: Gridworld Environment

For the first part of this assignment, you will be implementing tabular solutions (policy and value iteration) for a basic gridworld environment. The environment consists of a 8x8 grid of cells where each cell represents a state. The two environments you are required to test your algorithms on can be found in `/envs/gym_gridworld/envs` in files `plan0.txt` and plan1.txt. In these files, the environment is defined according to the following definitions:

- 0 - walls that the agent cannot enter (or exit if entered).
- 1 - space that the agent is able to explore freely. No reward is given for entering these cells.
- 2 - the starting location for the agent.
- 3 - a terminal state with +10 reward upon entry.
- 4 - a terminal state with -10 reward upon entry.
- 5 - entry into this state receives a -1 reward upon entry.

The agent always has 5 possible actions all of which are deterministic (unless prohibited by walls or terminal states):

- 0 - stay in place (0,0)
- 1 - move up (-1,0)
- 2 - move down (1,0)
- 3 - move left (0,-1)
- 4 - move right (0,1)

Please consult the file `gridworld_env.py` for further details on the gridworld environment for this problem.

2.1 Policy Iteration [30 points]

Next, you will implement the policy iteration algorithm for the tabular case. You will need to fill the code in `code/tabular_solution.py` below the line`if self.policy_type == 'deterministic_pi'`. Run the script for the both gridworld domains (`plan0.txt` and `plan1.txt`) and turn in three performance graphs for each environment - policy, value and return as a function of iteration. Note: functions to generate these plots have been provided.

Written Question:

1. Try a few different values of γ for your policy iteration algorithm ($\gamma = 0.5$, 0.75, 0.9 and 0.99 may be good choices). Does your policy change based on the value of γ? Why is this the case? Why not?

Bonus Question [10 points]: This problem can also be solved exactly (instead of iteratively) by casting the unknown state values as a set of linear equations. Implement this solution method and show in your writeup that you get the same results as for your policy iteration above. Also comment on the runtime between the exact and iterative approach.

2.2 Value Iteration [20 points]

Implement the value iteration algorithm for the tabular case. You must fill the code in `code/tabular_solution.py` below the lines `if self.policy type == 'deterministic_vi'`. Run the script for the both grid-world domains (`plan0.txt` and `plan1.txt`) and turn in three performance graphs for each environment - policy, value and return as a function of iteration. Note: functions to generate these plots have been provided.

Written Questions:

1. Are your policy and value iteration results the same? Do you expect them to be?
2. Compare the runtime of your policy and value iteration implementations. Which runs faster? Why?

Part 3: MountainCar Environment

This assignment uses the Mountain Car environment, a classic benchmark in reinforcement learning. The task is simple to describe but challenging to solve: an underpowered car starts in a valley and must drive up a steep hill to reach the goal flag. Because the engine is not strong enough to climb directly, the car must learn to build momentum by moving back and forth.

Environment Details

State Space (Continuous): Each state is represented as a 2D vector:

$$s = \begin{bmatrix} \text{position} \\ \text{velocity} \end{bmatrix}, \quad \text{with position} \in [-1.2, 0.6], \quad \text{velocity} \in [-0.07, 0.07].$$

Action Space (Discrete): The agent chooses one of three discrete actions at each step:

- 0: Accelerate left,
- 1: No acceleration,
- 2: Accelerate right.

Transition Dynamics:

- Velocity is updated using the chosen action, a small engine force, and the effect of gravity $\cos(3 \cdot \text{position})$.
- Position is updated based on velocity.
- Both position and velocity are clipped to their allowed ranges.
- If the car reaches the left boundary (-1.2), velocity is reset to zero.

Reward Function:

- At every time step: $r = -1$ (penalizing long episodes).
- Reaching the goal position (position ≥ 0.5) ends the episode with success.

Initial State:

- Position is initialized uniformly in $[-0.6, -0.4]$.
- Velocity is initialized to 0.

Termination Conditions:

- Car reaches the goal position (position ≥ 0.5).
- Episode length exceeds 200 steps.

Please consult the file `mountain_car.py` for further details on the mountain car environment for this problem.

3.1 Nearest neighbors interpolation [20 points]

Value Iteration can only work when the state and action spaces are discrete and finite. If these assumptions do not hold, we can approximate the problem domain by coming up with a discretization such that the previous algorithm is still valid. The MountainCar domain, as described in class, has a continuous state space that will prevent us from using value or policy iteration to solve it directly. One of the solutions that we came up with for this was nearest-neighbor interpolation, where we discretize the actual state space S into finitely many states and act as if we are in the nearest to our actual state s. Implement Value Iteration with nearest-neighbor interpolation on the MountainCar domain .Run the script and report the state value heatmap for MountainCar, discretizing each dimension of the state space (position and velocity) into 21, 51, and 101 bins.

3.2 Linear Interpolation [30 points] [GRAD QUESTION]

Nearest-neighbor interpolation is able to approach the optimal solution if you use a fine-grained approximation, but doesn't scale well as the dimensionality of your problem increases. A more powerful discretization scheme that we discussed in class is n-linear interpolation, an n- dimensional analogue of linear interpolation. Just as before, report the state value heatmap for MountainCar with discretization resolutions of 21, 51, and 101 points per dimension.