CSCI 4160/6963, ECSE 4965/6965
Reinforcement Learning
Homework 6

Overview

In this homework, you will develop the main reinforcement learning algorithms in the case where the Markov decision process (MDP) is known and finite. Specifically, first you will build a finite-horizon policy for the Mountain Car (MC) benchmark implemented in the OpenAI Gymnasium package (using finite-horizon dynamic programming). Additionally, you will build an infinite-horizon policy (using either policy iteration or value iteration) for the inverted pendulum (also implemented in OpenAI Gym).

In MC, you need to learn a controller to drive a car up a hill. Since the car doesn't have enough power, you need to back up the left hill first in order to gain speed. After each step, you get a small negative reward (smaller if you apply larger thrust); when you get to the goal, you get a positive reward of 100. MC is an interesting benchmark because it's quite easy to solve for a human, yet it is hard for RL because it gives sparse rewards (you only get a positive reward when you reach the goal).

In the inverted pendulum, you control the thrust of the rotating motor and your task is to keep the pendulum upright. After each step, you get a small negative reward, which is smaller if you apply higher thrust or if the pendulum is farther from the top (so reward is close to 0 if the pendulum is not moving and is close to the top). The pendulum is a classical control task since it is an unstable non-linear system but is still amenable to analysis.

In this homework, you will approximate the simulator with an MDP. The MC simulator has a continuous state space (position is between [-1.2, 0.6], velocity is between [-0.07, 0.07], and thrust is between [-1, 1], where negative thrust means "left" and positive thrust means "right"). The starting position is chosen randomly between [-0.6, -0.4], where the bottom of the valley is at $-\pi/6$ (starting velocity is 0). Your job will be to discretize the state and action space and approximate the simulator with a finite-state MDP.

Similar to the Mountain Car case, you will also need to build an MDP for the pendulum. The pendulum environment is a bit trickier – although it has only two states (orientation θ, measured in radians, and angular velocity, $\dot{\theta}$), you actually do not observe the states directly but instead observe $cos\theta, sin\theta, \dot{\theta}$. To get around this issue, you are given a small function that will infer θ from your measurements such that $\theta \in [-\pi, \pi]$. Velocity ranges between $[-8,8]$.

After that, you can build the MDP in exactly the same way as before (possibly with a different number of states).

Logistics

All coding assignments in this course will be in Python. If you need help with Python, please talk to me ahead of time, so we can discuss the best way to get familiar with it.

For this assignment, you will be using your own computer, which should be sufficient. The code should not require more than 10 minutes to run on any standard laptop.

You are provided with skeleton starting code. Please use the Python libraries in the `requirements.txt` file that's provided on LMS.

OpenAI environments are nice because they are all run in the same way, with very little changes from one to the next. To create the two environments, you need to run

```
env = gym.make('MountainCarContinuous-v0').env.unwrapped
```

```
env = gym.make('Pendulum-v1').env.unwrapped
```

where the final unwrapped variable just makes sure you can modify the environment state directly (which you will need to do). When you want to set the env state to a new state s, just call

```
env.state = s
```

If you want to render the environment, you also need to set the `render_mode='human'` flag.

Once you have an environment, you will need a control policy, pi, that gives you an action for each state. Given your action, you can call the `step` function of the environment:

```
observation, reward, done, __, ___ = env.step(action)
```

where `observation` is the next state, `reward` is the one-step reward, and the boolean `done` is set to true when you reach the goal. You shouldn't need the other two variables returned by `step`. So, once you call `step`, you can use `observation` to decide your action in the next step. In the case of the pendulum, you will also need to convert the environment observation to the states, using the provided `get_state_from_observation` function:

```
observation = get_state_from_observation(observation)
```

Don't forget to call `env.reset()` every time you start a new episode!

Grading

The first main part of this assignment is to build good enough MDPs. You will need to discretize each environment sufficiently so that the rewards in the MDP are similar enough to those in the simulator.

Once you have an MDP for each environment, the second main part of this assignment is to build good enough policies. Again, this means also having a good enough MDP, since your policies will be learned from the MDPs.

Note that while your policy is trained on the MDP, it must be evaluated on the real simulator. In particular, when you run the simulator, you must 1) collect your current observation, 2) figure out which MDP state that observation corresponds to, 3) compute the optimal action for that MDP state, and 4) send that action to the simulator.

Specifically, your Mountain Car policy needs to achieve an average (undiscounted) reward of at least 90 (**92 for graduate students**), as averaged over 100 simulator runs, each consisting of at most 150 steps. You need to build a finite-horizon dynamic programming policy for a horizon of 150 steps, with no discounting.

Your Pendulum policy needs to achieve an average (undiscounted) reward of at least -300 (**-250 for graduate students**), as averaged over 100 simulator runs, each consisting of 200 steps. You need to build an infinite-horizon policy (using either policy iteration or value iteration; you are not required to implement both). Since you will be using an infinite-horizon method, during training you should use a discounted reward, $\gamma = 0.9$.

You will lose 5 points for each 1 point of reward below the target in the case of Mountain Car and 5 points for each 10 points of reward below the target in the case of Pendulum.

Make sure your code prints out some indication of progress as well as all required information – we should not have to insert any print statements in order to find out whether your code works!

Finally, please provide answers to each of the questions below:

1) Why is the policy in the case of Mountain Car time-dependent? In other words, why does the value of a given state depend on the current time step?

2) What is the trade-off of having more states in your MDP? Think about how closely the MDP approximates the simulator.

Hints and Tips

- I suggest your start by playing with the simulator a bit and maybe building a simple policy first. This will let you get a feel for how the environment works and what the challenges are.
- Feel free to test your dynamic programming implementation on the cliff environment from the previous homework.
- To build your MDP, you need to discretize the position and velocity space. This will effectively give you a 2D grid of states, which are your MDP states. Each cell in that grid has ranges for velocity and position. You will also need to discretize your action space – I suggest you just pick a few actions in the action space of each task. Then, for each state, you need to calculate which action takes you to which other state (and with what probability). So you can have the following iteration: for each MDP state, iterate through all actions and for each action you 1) sample a random position and velocity from the state's ranges; 2) take the action; 3) record which state that action sends you to (by calling the step function in the environment); 4) repeat 1-3 a number of times (say 100 times) and then count the fraction of time that action led you to a particular MDP state.
- Keep in mind that even though the OpenAI simulator is deterministic, your MDP will not be. In other words, it is likely that a given action can send you to different states in your MDP. This is because an MDP state represents a range of positions and velocities, so depending on the actual value, you may end up in a different place.
- It is up to you to decide what data structure you want to use for the MDP. The easiest is to probably use a dictionary, where each key is an MDP state; the corresponding value is also a dictionary where each key is an action – the corresponding value is a tuple: 1) a list of probabilities (effectively a row of your transition matrix) and 2) a list of corresponding rewards. The other option is to use a numpy array, which tends to be faster than dictionaries.
- I suggest writing an extra function that will return a dictionary that contains the bounds for each state in your MDP (recall each state covers some position and velocity range). This will make it easier to iterate through all the states.
- Similarly, you might want to write a little function that maps an individual observation (actual position and velocity) to an MDP state.
- By convention, in the pendulum environment, negative angular velocity (and negative angles) means you are going in a clockwise direction.
- While the two control tasks are similar once you have an MDP, the Pendulum environment is sufficiently different in absolute terms, since velocity ranges from -8 to 8. So, you might need more states in the MDP to discretize sufficiently.

- I have provided you with a way of saving and loading your MDP so that you don't have to rebuild it every time you run your code. This is done using the pickle functionality in Python, which is a JSON-like file format for storing data structures to disk.

Submission

Please use `LMS` to submit a zip file containing 1) your **.py** code (in three files, as provided), along with instructions on how to run it, 2) a **.pdf** file containing your answers to the above questions. The deadline is **11:59pm, Thursday, Oct. 30**.