ECE5242 Project 4: Reinforcement Learning

Code due date: 5/7/2024 at 11:25am (15 mins before class starts)
Report due date: 5/9/2024 at 11:59pm on Canvas

In this project, you will implement Q-learning and REINFORCE algorithms and learn optimal policies for some MDP environments. You will also implement Value Iteration or Policy Iteration algorithms for a known and simple MDP and see how fast your Q-learning algorithm converges to the optimal Q values.

Simulation Environment:

1. Download code from https://cornell.box.com/v/ECE5242Proj4
2. OpenAI gym : Read its document and install `https://www.gymlibrary.dev/`. Installation instructions are also available here: `https://github.com/Farama-Foundation/Gymnasium`

For Linux and Mac users, this is easily available on pip.
For Windows 10 users, this is available on pip after you install the ubuntu bash.

Upload to Canvas:

1. Code (due 5/7/2024 at 11:25am)
2. Write-up (due 5/9/2024 at 11:59pm)

This assignment consists of several parts. Do each part making sure to answer the questions below as part of your standard writeup (include all the graphs!).

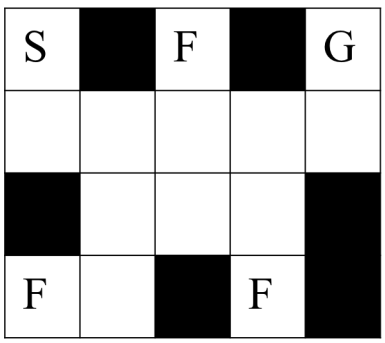
1. Policy Iteration / Value Iteration

In this problem, you will work on a domain, Maze, shown in the below figure. "S" is the starting position and "G" is the goal position. There are three flags located at "F" and the goal of a learning agent is to collect the flags and escape the maze through the goal state as soon as possible. It receives a reward equivalent to the number of flags it has collected at the goal state (i.e. at the current state s, it performs an action a and observes a reward r and the next state s'. If s'=goal state, r=the number of flags it has collected. Otherwise, r=0). The total number of states is 112 and there are 4 cardinal actions available - 0:UP, 1:DOWN, 2:LEFT, 3:RIGHT. A state is defined by a position and a status of the flags (see below for the details). The black blocks represent obstacles and the agent stays at the current state if it performs an action toward an obstacle or off the map. The agent slips with a probability 0.1 and reaches the next clockwise destination (e.g. if it performed UP and slips it will move to its RIGHT). For the maze environment, you must use a discount factor of $\gamma = 0.9$. In the helper functions provided, we use the following conventions:

position state number = 0, . . . , 13 from top to bottom and left to right (free cells only)

flag state number = 0, . . . , 7 identifying which flags have been collected. See the function `num2flag()` in `maze.py` for details.

state number, s = position state number * 8 + flag state number



Implement either Policy Iteration or Value Iteration algorithm and find an optimal policy and the optimal Q values (action-values) for all state and action pairs in Maze. Use the provided code, maze.py, for simulation. Include your optimal Q values (112 by 4 numpy array) as a .npy file in your submission (use numpy.save()) and describe your optimal policy in your writeup by saying what action the robot will take at each timestep in a run following the optimal policy (for this description you can assume a "lucky" run where the robot happens not to slip at all. No need to explain all the alternative "but if the robot slips here, then the robot would..." in this description).

```
# Example for how to use maze.py:
from maze import *
import numpy as np
env = Maze()
initial_state = env.reset()
state = initial_state
action = np.random.choice(4,)
reward, next_state, done = env.step(state, action)
env.plot(state, action)
```

2. Q-learning

Implement Q-learning. Apply it to Maze and learn its Q values for 5000 steps.

- Learning Rate: Experiment with different learning rates.
- Action Policy: Try ϵ-greedy for the action selection rule. Experiment with different hyperparameter values.
- Optimal Q values: In the previous problem, we found the optimal Q values, Q^*, for this domain. Compute RMSE of Q_t at each step t, $Q_{\text{err},t} = RMSE(Q_t, Q^*)$. Which hyperparameter choice converges to Q^* faster? Plot $Q_{\text{err},t=1,\ldots,T}$ of some of methods you tried and discuss your results.
- Performance Evaluation: In order to see its learning progress, pause your learning at every 50 epochs and evaluate your current policy using the function evaluation() in evaluation.py code (please read the description in the code and note that you need to implement the epsilon-greedy policy in get_action_egreedy). Feel free to modify the code further as well. **Plot learning curves with different hyperparameters and discuss your results.**

```
# Example template code for using evaluation.py:
from evaluation import *
Import matplotlib.pyplot as plt
# Some initialization #
eval_steps, eval_reward = [], []
learning = True
while learning:
  # your Q-learning part goes here #
  avg_step, avg_reward = evaluation(Maze(), current_Q_table)
  eval_steps.append(avg_step)
  eval_reward.append(avg_reward)

# Plot example #
f1, ax1 = plt.subplots()
ax1.plot(np.arange(0,5000,50),eval_steps)#repeat for different algs.
f2, ax2 = plt.subplots()
ax2.plot(np.arange(0,5000,50),eval_reward)#repeat for different algs.
```

3. Continuous State Space Problems

In this problem, you will work with Acrobot-v1 and MountainCar-v0 environments in OpenAI Gym (`https://gymnasium.farama.org/environments/classic_control/`). Their state spaces are continuous and their action spaces are discrete. The state vector for Acrobot-v1 is [cos(theta1), sin(theta1), cos(theta2), sin(theta2), theta_dot1, theta_dot2]. The state vector for MountainCar-v0 is [position, velocity]. You can use $\gamma = 0.9$ or can pick a different discount factor for this problem.

- Choose a parameterized policy and implement REINFORCE. You can use a linear function approximator and select features for it. Experiment with different values for hyperparameters.
- Apply the Q-learning algorithm to these environments by either discretizing the state space or using function approximation. Likewise, experiment with different values for hyperparameters.
- Use the same performance evaluation method in the previous problem for REINFORCE and Q-learning with the best hyperparameter values for each. You may have to change some lines in evaluation.py. Plot their learning curves and discuss your results.

A good resource to learn more about RL is:
"Reinforcement Learning: An Introduction" by Sutton and Barto:
http://incompleteideas.net/book/bookdraft2017nov5.pdf

Tips:

1. For OpenAI Gym Environments, think carefully about the discretization. Run each environment and look at the range of values. For gradient-based methods, consider normalizing your state values (ie: scaling them) since one dimension of the state can be much bigger than the other, which can make learning along the gradient harder.
2. For REINFORCE, use a baseline. It can be as simple as a moving average of total rewards. In addition, REINFORCE can be finicky to converge (this is a well-known problem). You may need to run it several times for it to converge to a good policy.
3. For MountainCar, set 'env._max_episode_steps' to be 1000. Your random policy needs enough time to find a possible solution (may not be good) in order to get some sort of feedback. A longer episode length helps the initial learning process.
4. To make the problems easier, you are allowed to use modified/augmented rewards and discount factors for the Gym problems if you want (not for the Maze problem, though).
5. During action exploration on MountainCar, you may cluster your action sampling in time. You can pick an action and then execute that one action for 5 steps in a row. And then after those 5 steps pick a new action (for 5 steps) etc. If you do this "clustering in time", you'll want to make sure you're careful about your gamma (since the gamma should still correspond to the simulation time-step time, not your larger, clustered time-step time chunks). You can think of this as a modification to the training environment where the robot only takes a new action every 5 seconds instead of every 1 second.
6. For Q-Learning on MountainCar, using eligibility traces(See the RL book) can help converge faster as the reward is sparse.