ECE5242 **Project 4: Reinforcement Learning**

Code due date: **4/18/2019 at 1:40pm** on CMSX, <NetID>_projec4.zip
Report due date: **4/23/2019 at 11:59pm** on CMSX, <NetID>_project4.pdf

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. For all environments, the discount factor is fixed to $\gamma = 0.9$.

Simulation Environment:

1. Download codes from https://upenn.box.com/v/ECE5242Proj4
2. OpenAI gym : Read its document and install https://gym.openai.com/docs.
   Installation instructions are also available here: https://github.com/openai/gym

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: on CMSX

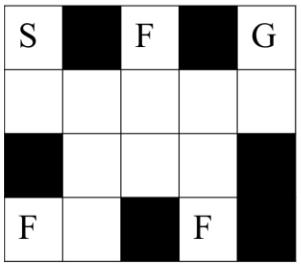
(1) Code (due 4/18/2019 1:40pm, <NetID>_project4.zip)
(2) Write-up (due 4/23/2019 11:59pm, < NetID >_project4.pdf)

This assignment consists of several parts. Do each part and answer the questions in the writeup (include all the graphs and plots!).

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 (i.e. It performed UP, but moved to its RIGHT).

- state number, s = position state number * flag state number
- position state number = 0, … , 13 from top to bottom and left to right.
- flag state number = 0, …, 7. See the function `num2flag()` in `maze.py` for details.



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()`)

<example for using `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.

a) *Learning Rate*: Experiment with different learning rates.

b) *Action Policy*: Try ϵ-greedy for the action selection rule. Experiment with different hyperparameter values.

c) *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, $Q_{err,t} = RMSE(Q_t, Q^*)$. Which hyperparameter converges to Q^* faster? Plot $Q_{err,t=1,\ldots,T}$ of some of methods you tried and discuss your results.

d) *Performance Evaluation*: In order to see its learning progress, pause your learning at every 50 steps and evaluate your current policy using the function `evaluation()` in `evaluation.py` code (please read the description in the code). Feel free to modify the code. Plot learning curves with different hyperparameters and discuss your results.

<example for using `evaluation.py`>

```
from evaluation import *
Import matplotlib.pyplot as plt

# Some initialization #
eval_steps, eval_reward = [], []
while learning:
      # your Q-learning part #
      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.range(0,5000,50),eval_steps)#repeat for different algs.
f2, ax2 = plt.subplots()
ax2.plot(np.range(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://gym.openai.com/envs/#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].

a) 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.

b) 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.

c) 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.
1) 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.
2) 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.
3) For Q-Learning on MountainCar, using eligibility traces(See the RL book) can help converge faster as the reward is sparse.