Dynamic Distributed Decision Making
Project 2
MIE567

Hao Tan 999735728
Xiali Wu 999011322
David Molina 1005615318

University of Toronto — March 25, 2020

1 Simulation

In order to implement model-free reinforcement learning algorithms, we need to first have access to a simulator that can recreate the intended behaviours of the environment (MDP) when interacting with it. To do this, create a file called *Gridworld.py* that implements:

1. a function that returns the initial state of the MDP: in this case, you may assume that the initial state is in the bottom right hand corner of the grid.

Listing 1: Returns a random initial state

```python
def initial_state(self):        # return initial state
    return grid.states[4, 4]
```

2. a function that, given a current state and action, returns the next state and associated reward of arriving at that state.

Listing 2: Returns the next state and reward

```python
def transition_reward(self, current_pos, action): # return the transition probability
    # get next position: state: [0, 0], action: [0, 1], new_state = [0, 1] and reward
    self.new_pos = np.array(current_pos) + np.array(action)
    reward = 0 # normally, reward = 0
    # if new pos results in off the grid, return reward -1
    if -1 in self.new_pos or self.size in self.new_pos:
        reward = -1
    if current_pos == [0, 1]: # if in state A, receive + 10
        reward = 10
    if current_pos == [0, 3]: # if in state B, receive + 5
        reward = 5
    if -1 in self.new_pos or self.size in self.new_pos: # if crossing the border;
        self.new_pos = current_pos # agent's new_pos is the same as the current pos
    if current_pos == [0, 1]: # if in state A, transition to state A'
        self.new_pos = [4, 1]
    if current_pos == [0, 3]: # if in state B, transition to state B'
        self.new_pos = [2, 3]
    return self.new_pos, reward
```

2 Implementation

Now that you have defined the environment needed to perform optimization, you are next asked to implement four of the standard reinforcement learning algorithms you have learned in this course so far. This includes:

1. **First-Visit On-Policy Monte Carlo for Control**
2. **Q-learning**
3. **SARSA**
4. **SARSA(λ) using eligibility traces**

Each algorithm should be implemented in a separate Python file and be named appropriately (MonteCarlo.py, QLearning.py, Sarsa.py and SARSALambda.py). For each algorithm you implement, you should have another Python file that allows to run your experiment.

We initialize our domain, and the algorithm with following parameter settings. We applied the same parameter settings for each of the four algorithms. We fix the episode length to 200 and the total number of episodes to 500. We initialize your Q-values to zero. Please refer to the algorithm files.

$$\gamma = 0.99 \tag{1}$$

$$\lambda = 0.9 \tag{2}$$

$$\alpha = 0.1 \tag{3}$$

$$\epsilon \in \{0.01, 0.1, 0.25\} \tag{4}$$

In order to get a better idea of how fast each algorithm converges, practitioners typically run each algorithm independently using the same initial conditions many times (run each one at least 20 times), and average the results from all the trials.

Plotting the performance of each algorithm and each combination, you may plot the mean over all trials (you don't have to plot each individual trial).

For each experiment, you are responsible for reporting (these can be reported as averages of many trials, as mentioned above):

• at the end of each episode of training, you should test the performance of the greedy policy with respect to your learned Q-values (the policy that chooses actions in argmaxQ(s, a)

• you must include a plot of both training performance and test performance for each experiment: you can measure the performance during an episode by the average, total or discounted return you obtained by running that policy

• you should report the final Q-values you obtained for each experiment and the policy you obtained on a typical run: if you obtained different policies or performance, report several typical results you observed

3 Comparison and Write-up

1.For each algorithm, what was the best and worst values of ϵ (in terms of test performance)? How different is the train and test performance between algorithms for each value of ϵ? How different is the train and test performance between values of ϵ for each algorithm (e.g. how sensitive was each algorithm to ϵ)? Why do you think this occurred? Does this coincide with what you learned in lectures about each algorithm?

xxx

2.What was the final policy and Q-values that you typically obtain (typically, as in the majority of the trials)? Are they similar or different across algorithms for each value of ? Does this correspond to an optimal policy (you may refer to your answers for project 1)? Please comment on the variability of each algorithm in terms of what you learned about the algorithms (e.g. bias/variance trade-off).

xxx

3.Which algorithm was most difficult, and which was most easy, to implement and why? Which took the least time (and samples) to train, and which the most? Which algorithm(s) do you think would scale better to larger problems and why?

xxx

4 Appendix