MIE567H1: Dynamic Distributed Decision Making

Project #2: Reinforcement Learning

Deadline: Thursday March 26, 2019, 11:59 pm on Quercus
Maximum Points: 100

Instructions: Please read all instructions carefully before attempting each part of the problem. All programming should be done in Python - please ensure that your code runs on a recent version of Python (at least 3.5). Jupyter notebooks are not allowed. You need to implement all the algorithms yourself, and you are allowed to use only standard packages.

You should provide a **ZIP folder** containing your report in **PDF format** and your Python code. The report should include detailed answers to all questions, including all derivations, analysis and relevant figures. You should include all the code necessary to run your experiments.

In Project 1, you were asked to implement two exact algorithms to solve a Gridworld domain. In this project, you will be asked to implement four different model-free algorithms to solve the same problem, and compare your results.

Problem Description: Figure 1 (below) shows a rectangular grid world representation of the problem. The cells of the grid correspond to the states of the environment. At each cell, four possible actions exist, each corresponding to one compass direction (e.g. north, east, west, south), which deterministically cause the agent to move one cell in the respective direction on the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions have zero reward, except those that move the agent out of the special states A and B. From state A, all four actions yield a reward of +10 and take the agent to A'. From state B, all actions yield a reward of +5 and take the agent to B'.

Simulation [10 points]: 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. The following Figure 2 illustrates the overall work-flow.

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.

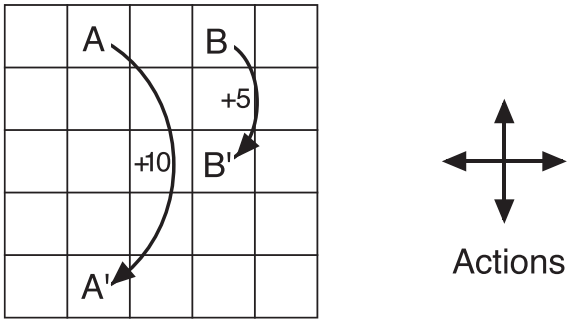

Figure 1: Gridworld domain

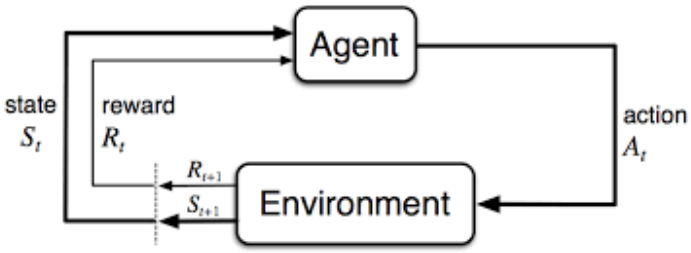

Figure 2: Model-Free RL Workflow

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

Implementation [60 points]: 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 `TdLambda.py`). For each algorithm you implement, you should have another Python file that allows to run your experiment.

Here, you will have to initialize your domain, and the algorithm with your parameter settings, and run your algorithm.

For each of the four algorithms:

- run each algorithm using the discount factor $\gamma = 0.99$
- fix the episode length to 200 and the total number of episodes to 500
- you may initialize your Q-values to zero
- for SARSA(λ), you may set $\lambda = 0.9$
- you may set the learning rate to $\alpha = 0.1$ for all algorithms (if you are experiencing unstable convergence with this choice, you may increase or decrease α, but in this case please use the same value when comparing all algorithms)
- train using the epsilon-greedy policy you learned in class: for each algorithm, you should perform an experiment for different values of $\varepsilon \in \{0.01, 0.1, 0.25\}$ and report the performance of each one (see below for what to report)

Furthermore, the algorithms in model-free RL are inherently random due to the randomness of the exploration policy and possibly the stochasticity of the MDP (not relevant here, since it is deterministic). 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. Do not forget to reset or re-initialize your Q-values between trials. When plotting the performance of each algorithm and each ε 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 $\arg max_a Q(s, a)$
- 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

Comparison of Algorithms and Write-Up [30 points]:
Once you run each experiment, please report your answers to the following questions for each algorithm:

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?

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 ε? 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).

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?