MIE567H1: Dynamic Distributed Decision Making

Project #3: Multi-Agent Reinforcement Learning

Deadline: Wednesday April 8, 2020, 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.

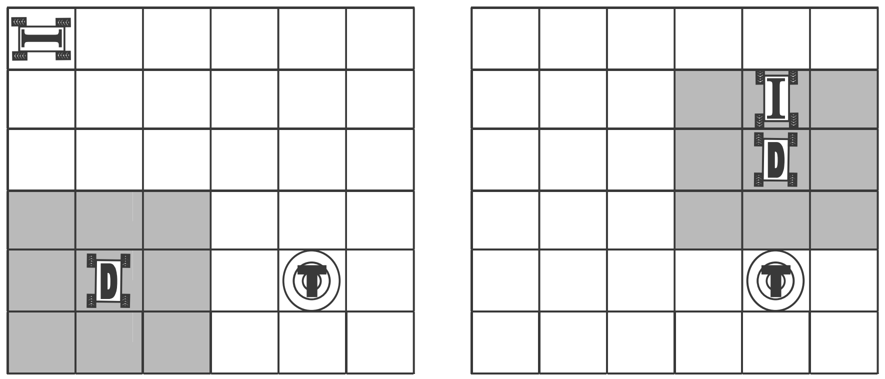
You should provide a **ZIP folder** containing your report in **PDF format** and your Python code. The report should include detailed answers to all questions, including all derivations, analysis and relevant figures. You should include all the code necessary to run your experiments.

In this project, you will be asked to implement three different algorithms to solve a navigation problem involving two agents, and compare your results.

Problem Description: The task you will be implementing for this assignment is defined as follows:

- Time is broken up into discrete epochs.
- The world is a square area that is broken up into a 6-by-6 grid of cells.
- There are two players or agents; both of them can move up, down, left, or right. At each time step, both players take their action simultaneously. Each cannot see the others' action at that time step, but the state of the system and previous actions are fully observable to each. If the chosen action takes an agent outside of the grid, then the agent remains in the current position.
- The invader, marked I, starts in the upper-left corner of the world. The defender, marked D, starts in the bottom-left corner of the world.
- The 9 cells centered around the defender's current position (see Figure 1), is the region where the invader will be captured. The game ends whenever the invader is captured by the defender, or the invader enters the fixed territory cell marked T, whichever comes first.

- The discount factor for future rewards is $\gamma = 0.95$.
- *The goal of the defender is to capture the invader as far away as possible from the territory T. The goal of the invader is to move as close as possible to the territory T before being captured.*



(a) One possible initial positions of the players when the game starts (b) One possible terminal positions of the players when the game ends

Figure 1: The Invader/Defender Problem

Modelling [20 points]: First, you are tasked with modelling the problem as a stochastic game. Then, you are asked to provide a programming description of the problem that will be used to solve it computationally.

1. [10 points] Explain how you would model this problem as a stochastic game:
   - (a) [2 points] What is the state space, and what are the action spaces for each agent?
   - (b) [2 points] Describe the transition probability matrix for this stochastic game.
   - (c) [2 points] Which states, if any, are terminal states?
   - (d) [2 points] Derive one suitable reward function that allows the defender to achieve the goal for this task. Derive one suitable reward function that allows the invader to achieve the goal for this task. Are these the only possible

suitable reward functions? Why or why not? What is the relationship between the defender's and invader's reward functions? Choosing a good reward function is critical to get good results in RL, so please choose carefully.

(e) [2 points] What kind of stochastic game is this (e.g. what special properties does it possess based on what you have learned about stochastic games)?

2. [10 points] Now, you are asked to implement the behavior of the invader/defender domain in a Python file. Create a python file called `InvaderDefender.py`. Here, you should create functions similar to what you have done in Projects 1 and 2, but for the stochastic game. In particular, you would want to consider functions to: return a list of possible states in the game; given a state, check if it is terminal; given a transition, return its probability of occurring and the immediate reward (these should be consistent with your answers to part 1 of the question); and given a state and actions of the agents, return the next state of the game (to simulate the task behavior for Q-learning).

Pre-Implementation [10 points]: The key component of any multi-agent reinforcement learning algorithm is how to decide which actions the agents should take next given the current state of the game. In some cases, this *coordination problem* can be solved using the LP formulation of the minimax equilibrium. Answering the following questions could help you in your implementation:

(a) [1 points] Do you need to maintain separate Q-values or value function for the invader and the defender, or do you only need to maintain one set of Q-values or value function (say, from the defender's perspective)? Why?

(b) [4 points] Given an arbitrary $|\mathcal{A}|$-by-$|\mathcal{A}|$ payoff matrix Q, where rows index the action of agent 1, and columns index the action of agent 2, you have learned how to find the minimax equilibrium by formulating and solving a linear program (LP). Formulate an LP for the defender and an LP for the invader. Define clearly all decision variables, objective function, and constraints.

(c) [1 points] If you were to solve these two linear programs optimally, would you expect the final objective function values to be equal or different? Why?

(d) [2 points] The LPs you defined above may be difficult to solve using standard off-the-shelf LP solvers. The following standard form is useful for this purpose:

$$\begin{aligned} &\min_x c^T x \\ \text{s.t.} &A_{ub} x \leq b_{ub} \\ &A_{eq} x = b_{eq} \\ &x \in [x_{lb}, x_{ub}], \end{aligned}$$

where $c, b_{ub}, b_{eq}, x_{lb}, x_{ub}$ are vectors and A_{ub} and A_{eq} are matrices. Write both LPs you defined previously into this form (explaining what c is, what b_{ub} is, etc).

(e) [2 points] If you had to implement Nash Q-learning for this task, can you use the LP solution above for performing a Q-value update, or do you have to use an algorithm for solving the Nash equilibrium only? Why or why not?

Implementation [40 points]: Now that you have defined the environment needed to perform optimization, you are next asked to implement two of the standard multi-agent reinforcement learning algorithms you have learned in this course.

1. [20 points] Shapley value iteration (model-based)
2. [20 points] MiniMax Q-learning (model-free)

Each algorithm should be implemented in a separate Python file and be named appropriately. 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 three algorithms:

- set the value function or Q-values initially to zeros
- for solving an LP, you should use the scipy package (in this case, make sure you are correctly inputting the bounds)
- for Q-learning, fix the episode length for training and testing to $T = 200$
- the number of episodes can be adjusted to obtain convergence as you see necessary; for value iteration, you can stop updates when the maximum difference in value functions between two iterations drops below 10^{-6}
- in this project, we are giving you full control to experiment with hyper-parameters that you think work well overall (e.g. learning rate, ε), but please be consistent in your choice across algorithms
- for all algorithms, testing should be done using the greedy policy on separate roll-outs at the end of each training episode (as explained for Project 2)

For each algorithm, your report should include:

- a table containing hyper-parameter values you used in your experiments
- plots of the training and testing performance over time as a function of the number of training episodes already performed (for value iteration, your training performance can be the error in V between iterations as in Project 1)

- 2D heat-maps showing: (1) the final value function from the invader's perspective as a function of the invader's starting position, when the defender starting position is fixed at $x = 0, y = 5$, and (2) the final value function from the defender's perspective as a function of the defender's starting position when the invader's starting position is fixed at $x = 0, y = 0$
- a pictorial representation (you may include a drawing based on the output of your code) of the agents' motion in the grid during a typical testing episode once the algorithm has converged.

Write-Up [30 points]: In your report, please provide details about the results of your experiment. Include all relevant plots to support your answers:

- Did your algorithms converge? How many iterations did they take?
- What were the final policies? Do the final policies agree between value iteration and Q-learning? What do you think could explain the differences between them, if any?
- When solving single agent MDPs, the optimal policy is deterministic. However, this is not necessarily the case in stochastic games. Looking at the final policy you obtained, were there any states in which the optimal mini-max equilibrium was non-deterministic (that is, either defender or invader or both agents take action randomly)? Please include a list of all such states in your report. Do you think this result is expected? Why or why not?
- Based on your heat-map plots, for which initial states does the invader win and defender lose (e.g. the invader makes it to the territory and evades capture)? Similarly, for which initial states does the defender win and invader lose (e.g. the invader is captured before reaching the territory)? Are there any starting states in which it is not clear whether invader and defender will win the game? Explain.
- Which algorithm did you feel was more reliable for solving this task? Which algorithm would be more efficient if the state space was considerably larger?