COMS W4701: Artificial Intelligence, Spring 2025

Homework 3

Instructions: Compile all written solutions for this assignment in a single, typed PDF file. Coding solutions may be directly implemented in the provided Python file(s). **Do not modify any filenames or code outside of the indicated sections.** Submit all files on Gradescope in the appropriate assignment bins, and make sure to **tag all pages for written problems**. Please be mindful of the deadline and late policy, as well as our policies on citations and academic honesty.

Problem 1: Mini-Blackjack MDP (12 points)

We will model a mini-blackjack game as a MDP. The goal is to draw cards from a deck containing 2s, 3s, and 4s to obtain a card sum as close to 6 as possible without going over. The possible card sums form the states: 0, 2, 3, 4, 5, 6, "done". The "done" state is terminal. From all other states, one action is to **draw** a card and advance to a new state according to the new card sum, with "done" representing card sums of 7 and 8. Each card value of 2, 3, and 4 can be drawn with probability $\frac{1}{3}$. Alternatively, one may **stop** and receive reward equal to the current card sum, also advancing to "done" afterward.

(a) Draw a state transition diagram of this MDP. The diagram should be a graph with seven nodes, one for each state. Draw edges that represent transitions between states due to the **draw** action only; you may omit transitions due to the **stop** action. Write the transition probabilities adjacent to each edge.

(b) Based on the given information and without solving any equations, what are the optimal actions and values of states 5 and 6? You may assume that $V^*(\text{done}) = 0$. Then using $\gamma = 1$, solve for the optimal actions and values of states 4, 3, 2, and 0 (you should do so in that order). Briefly explain why dynamic programming is not required for this particular problem.

(c) Find the largest possible value of γ that would possibly lead to different optimal actions in both states 2 and 3 (compared to those above). Compute the values of states 3, 2, and 0 for the discount factor that you found. Briefly explain why a lower value of γ decreases the values of these states but not those of the other states (4, 5, 6).

Problem 2: Value Iteration (16 points)

Let's now consider a variation of the mini-blackjack game. If a **draw** action results in a card sum larger than 6, we simply discard our hand and go back to the 0 state. In other words, only **stop** actions will bring us to the "done" state. All other rules remain the same.

(a) Without performing any computation, identify the optimal policy π^* and value function V^* for this game version assuming $\gamma = 1$. Briefly explain how you deduced this solution.

(b) Now suppose we have $\gamma = 0.9$; we will now have to do value iteration to correctly solve for V^* and π^*. Write down the six Bellman updates computed in each iteration. You should have an equation for each $V_{i+1}(s)$, each in terms of $V_i(s)$.

(c) Starting with initial time-limited values $V_0(s) = 0$ for all s, compute two rounds of value iteration, and show the resultant values V_1 and V_2.

(d) Suppose we continue running value iteration until we see the values converge at a certain threshold, and we see that it takes 26 iterations to do so. Suppose we also try running value iteration using $\gamma = 0.5$, and we see convergence to the same threshold in just 3 iterations. Briefly explain the reason for the difference in convergence speeds. Which optimal policy do you expect to have more draw actions vs stop actions and why?

Problem 3: Reinforcement Learning (12 points)

Let's return to the simpler mini-blackjack game from Problem 1 and use reinforcement learning to learn an optimal policy. Again, assume $\gamma = 1$. We initialize all values and Q-values to 0 and observe the following episodes of state-action sequences:

- 0, draw, 2, draw, 4, draw, done (reward = 0)
- 0, draw, 3, draw, done (reward = 0)
- 0, draw, 2, draw, 5, stop, done (reward = 5)
- 0, draw, 3, draw, 5, stop, done (reward = 5)
- 0, draw, 4, draw, 6, stop, done (reward = 6)

(a) Suppose that the above episodes were generated by following a fixed policy. According to Monte Carlo prediction, what are the values of the six states other than the "done" state? Explain whether the *order* in which we see these episodes affects the estimated state values.

(b) Suppose we use temporal-difference learning with $\alpha = 0.5$ instead. Write out **each** of the updates that changes a current state value. Again explain whether the *order* in which we see these episodes affects the estimated state values.

(c) Suppose we are interested in learning a better policy and start allowing for exploration. The values computed using TD learning above are assigned to the relevant Q values, while Q values that were not updated are 0. Consider using Q learning versus SARSA to update the Q values. For which state-action pairs will the converged Q values differ between the two methods and why (assuming exploration is persistent)?

Problem 4: Bernoulli Bandits (12 points)

You will run simulations of a K-armed *Bernoulli* bandit in `bandits.ipynb`. Each arm gives either a reward of 1 with probability p or 0 with probability $1 - p$. `means` is a list of the p probabilities for each arm. `strat` and `param` describe the strategy (ε-greedy or UCB) along with the parameter value (ε or c), and the bandit is simulated M times for T timesteps each. The returned quantities are the average regret and average frequency that the best arm is chosen at each timestep.

`execute()` simulates the bandit given the same arguments above. `params` is expected to be a list, so that we can compare results for different parameter values. The returned values from `bernoulli_bandit()` are plotted vs time on separate plots (regret is plotted cumulatively). The x-axis is logarithmic, so that the trends are better visualized over time. **Thus, logarithmic growth will appear linear on these plots, while linear growth will appear exponential.**

For each part of this problem, you will only need to set the `means` and `strat` arguments as instructed and then call `execute()`. The remaining arguments will be left with their default values: `params=[0.1,0.2,0.3], M=100, T=10000`. You will then copy the resulting plots into your submission and provide responses.

(a) Let's first try an "easier" problem, a 2-armed bandit with means 0.3 and 0.7. Show the two sets of plots and briefly answer the following questions.

- For ε-greedy, which parameter value does better on the two metrics over the long term? Why might your answer change if we look at a shorter time period (e.g., $t < 100$)?
- How does the value of c affect whether UCB does better or worse than ε-greedy? How does c affect the order of growth of regret?

(b) Now let's consider a "harder" 2-armed bandit with means 0.5 and 0.55. Show the two sets of plots and briefly answer the following questions.

- Compare ε-greedy's performance on this problem vs the previous one. Do we need more or fewer trials to maximize the frequency of playing the best arm?
- Compare the performance of ε-greedy vs UCB on this problem on the two metrics. Which generally seems to be less sensitive to varying parameter values and why?

(c) For the last experiment, simulate a 4-armed bandit with means $0.2, 0.4, 0.6, 0.8$. Show the two sets of plots and briefly answer the following questions.

- Among the three bandit problems, explain why ε-greedy performs the worst on this one in terms of regret, particularly for larger values of ε.
- How does UCB compare to ε-greedy here when c is properly chosen (i.e., look at the best performance of UCB)? Explain any differences that you observe.

Problem 5: Crawler Robot (48 points)

You will be training a simple crawler robot modeled as a MDP. When you download the accompanying code files and run `python crawler.py` in your terminal, you should see a GUI pop with a robot on the left side of the screen. It consists of a rigid rectangular body and two joints (an "arm" and a "hand") that can be moved in discrete increments. The state space consists of discrete joint angle combinations, and the action set in each state allows the robot to move one of the joints either up or down. You should also see some buttons on the top third of the GUI that will allow you to adjust various parameters. The robot is initially stuck, but it will eventually learn how to move forward by moving its joints and pushing off the ground below it.

Part 1: Dynamic Programming (20 points)

One method for learning a good policy is to do so entirely offline using dynamic programming. In `DP_Agent.py`, we define a `DP_Agent` class. A `DP_Agent` stores the set of states, a set of values, and a policy. It is also initialized with a discount factor `gamma`. To perform dynamic programming, you will implement policy iteration, a variant of value iteration.

First write `policy_evaluation()`. This will iteratively compute the values for the current policy stored in `self.policy` and store them in `self.values`. The `transition` argument is a `Callable` that returns the successor state and reward given a state and action (all transitions are deterministic). If `None` is returned as a successor state, you can use 0 as its "value". Convergence occurs when the maximum change in any state value is no greater than 10^{-6}.

Next, write `policy_extraction()`, which will store the corresponding actions for all states in the `self.policy` dictionary, computed using the current `state.values`. In addition to `transition`, it also takes in a `valid_actions` argument, a `Callable` that returns a list of all actions for a given state. Note that this is the same procedure that would be run after completion of a value iteration routine, although here we will be using this even when the values are not yet optimal.

Finally, write `policy_iteration()`. This will repeatedly call the two subroutines that you wrote above: `policy_evaluation()`, then `policy_extraction()`. This stops when the "old" policy in `self.policy` *does not change* after running both subroutines. This signals convergence to an optimal policy, equivalent to value convergence in value iteration.

Once you finish all methods, you can test your implementation by running `python crawler.py`. If all goes well, your robot should be able to start moving across the screen with its newfound policy.

Part 2: Reinforcement Learning (16 points)

A second approach for learning a policy is to do so online using reinforcement learning. In `RL_Agent.py`, we define a `DP_Agent` class. A `DP_Agent` stores the set of states, a set of Q-values, and the parameters `alpha`, `epsilon`, and `gamma`.

First write the `choose_action()` method, which performs ε-greedy action selection given the `state` and `valid_actions` list. It should make reference to `Qvalues` if deciding to behave greedily. Next, write the `update()` method, which makes a Q-learning update to the appropriate Q-value given all components of a single transition and the `valid_actions` of the `successor` state. As in Part 1 above, if the successor is `None`, you may set its "Q-value" to 0.

You can test your implementation by running `python crawler.py -q`. The robot will appear to struggle on its own for a while, but after enough time passes it should start moving more regularly. You can decrease the "Time per action" setting at the top left to make the simulation run faster.

Part 3: Analysis (12 points)

1. Let the `DP_agent` run for at least 200 steps. What is the robot's 100-step average velocity? How does this change when you a) increase `gamma` to at least 0.9, and b) decrease it below 0.7 (give it time to settle after each change)? Describe how the discount factor affects the robot's performance.

2. Let the `RL_agent` train until it crosses the screen at least once so that it has learned an optimal or near-optimal policy. Describe how its performance changes when you a) increase and b) decrease `epsilon` by at least 0.25 in each direction.

3. Let the `RL_agent` train until it crosses the screen at least once, and note approximately how many steps it took to do so. Then start a new run and decrease `alpha` to about 0.1 at the very beginning. Describe the effect of the learning rate on the robot's training time.

Coding Submissions

You can submit just the completed `DP_Agent.py` and `RL_Agent.py` files together under the HW3 Coding bin on Gradescope.