COMS W4701: Artificial Intelligence, Summer 2024

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: Bernoulli Bandits (18 points)

You 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 used to play this bandit (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 (highest mean) 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 arguments as instructed and then call `execute()`. You will then copy these plots into your submission and provide responses.

1. (6 pts) Let's first try an "easier" problem: a 2-armed bandit with means 0.3 and 0.7, simulated for $T = 10000$ timesteps and averaged over $M = 100$ simulations. Use ε-greedy with ε values $0.1, 0.2, 0.3$, and then use UCB with c values $0.1, 0.2, 0.3$. 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?

2. (6 pts) Now let's consider a "harder" 2-armed bandit with means 0.5 and 0.55. Perform the same simulations as in the previous part with all other values unchanged. Show the two sets of plots and briefly answer the following questions.

    - Compare ε-greedy's performance on this problem vs the previous one on the two metrics. For this problem, do we need greater or fewer (or about the same) number of 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?

3. (6 pts) For the last experiment, simulate a 4-armed bandit with means $0.2, 0.4, 0.6, 0.8$, keeping all other values the same. Do so using both ε-greedy and UCB, and show both sets of plots. 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 2: Reinforcement Learning (24 points)

The performance of the robot from HW2 has degraded, and the original model no longer appears to be valid. In particular, the *off* state is now a terminal state. Suppose we observe the following two episodes of state and reward sequences following the policy π of going *slow* in both states.

- Episode 1: $high, 2, high, 1, low, 1, low, 1, low, 0, off$
- Episode 2: $high, 2, high, 2, high, 1, low, 0, off$

1. (6 pts) We use first-visit Monte Carlo to perform prediction for the policy π. Again using $\gamma = 0.5$, show the calculations for finding the individual state return values G in each episode. Then compute the estimated state values $V^\pi(high)$ and $V^\pi(low)$.

2. (6 pts) Suppose that we apply different weights to the returns in each episode when computing the average values $V^\pi(high)$ and $V^\pi(low)$. Compute the values obtained by applying a weight $\alpha = 0.8$ to the returns in episode 2 (and correspondingly, $1 - \alpha = 0.2$ in episode 1). Briefly describe a scenario in which this weighting scheme may give more accurate value estimates.

3. (6 pts) We send the robot off to do some active reinforcement learning. It has the following Q-values so far: $Q(high, fast) = 2, Q(high, slow) = 0, Q(low, fast) = -1, Q(low, slow) = 1$. Starting off in the *high* state, it takes the greedy action, receives a reward of $+2$, and lands in the *low* state. It then takes the exploratory action, receives a reward of $+1$, and stays in the *low* state. Show the resultant Q-value updates performed by the Q-learning algorithm, using $\gamma = 0.5$ and $\alpha = 0.8$.

4. (6 pts) Recompute the first Q-value update using SARSA instead of Q-learning. Briefly explain how the SARSA update results in a different Q-value, making reference to how the robot "interprets" its second transition.

Problem 3: Recommendation Model (24 points)

A certain app is deciding whether to show you a personalized **ad**, represented by a binary random variable A. It can possibly consider two factors: whether you browse the **virtual** marketplace and whether you make a trip to the **physical** store location (it can track your location!). These decisions are represented by binary random variables V and P. The joint distribution over all three variables is as follows:

V	P	A	$\Pr(V, P, A)$
$+v$	$+p$	$+a$	0.12
$+v$	$+p$	$-a$	0.08
$+v$	$-p$	$+a$	0.18
$+v$	$-p$	$-a$	0.12
$-v$	$+p$	$+a$	0.06
$-v$	$+p$	$-a$	0.14
$-v$	$-p$	$+a$	0.09
$-v$	$-p$	$-a$	0.21

1. (3 pts) Find the marginal distributions of each of the three random variables.
2. (6 pts) Find the joint distributions of each of the three different *pairs* of the random variables.
3. (6 pts) Using the distributions you found above, show whether each pair of random variables is independent.
4. (6 pts) Find the following conditional distributions: $\Pr(V|+a)$, $\Pr(P|+a)$, $\Pr(V, P|+a)$.
5. (3 pts) Can we conclude that V and P are conditionally independent given A without further calculations? If not, what additional calculations do we need to verify?

Problem 4: Crawler Robot (34 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 (14 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`.

First write the `value_iteration()` method. This should run value iteration to find the optimal values V^* for all states and store them in the `values` dictionary. Two `Callable`s (function handles) are given as arguments: `valid_actions` returns a list of actions given a state, and `transition` 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 may occur when the maximum change in any state value is no greater than 10^{-6}.

After we run value iteration, we need to derive a policy π^*. Write `policy_extraction()`, which will store the optimal actions for all states in the `policy` dictionary. Once you finish this, 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 (14 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 (6 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.