COMS W4701: Artificial Intelligence, Spring 2024

Homework 3a

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: MDPs and Dynamic Programming (18 points)

A mobile robot is moving around on a rechargeable battery. There are three battery level states: *high*, *low*, and *off*. In the first two states, the robot may move *fast* or *slow*, while in *off*, the robot may only *recharge*. Transitions are stochastic; moving *fast* is guaranteed to lower the robot's battery level, while moving *slow* may sometimes do so. A transition and reward function are defined for this robot as follows.

s	a	s'	$T(s,a,s')$	$R(s,a,s')$
high	*fast*	*low*	1.0	+3
high	*slow*	*high*	0.5	+2
high	*slow*	*low*	0.5	+2
low	*fast*	*off*	1.0	+2
low	*slow*	*low*	0.75	+2
low	*slow*	*off*	0.25	+1
off	*recharge*	*high*	1.0	-2

1. (4 pts) Consider the policy π in which the robot goes *fast* in both the *high* and *low* states and *recharges* in the *off* state. Write down the system of linear equations describing the value function V^π in terms of γ, and then solve for the values using $\gamma = 0.5$. (You don't need to do the last step by hand, but please state if you are using any programs to help with it.)

2. (6 pts) Suppose the values that you obtained above are the time-limited values V_i in iteration i of value iteration, which is being used here to find the optimal policy π^*. Show the computations done in the next iteration, and find the next set of time-limited values V_{i+1}.

3. (5 pts) We find that the optimal values are $V^*(high) = 4.42, V^*(low) = 2.84, V^*(off) = 0.21$ at convergence. Show the computations done by policy extraction to find π^*.

4. (3 pts) Now suppose $\gamma = 0$; find the optimal policy for this scenario. (You should be able to do so without resorting to dynamic programming.) Briefly explain how changing γ to 0 also changes any optimal actions from those you found in part 3 above with $\gamma = 0.5$.

Problem 2: Reinforcement Learning (22 points)

After having been in operation for a long time, we find that the robot's performance has degraded, and the original model no longer appears to be valid. In particular, the robot has lost the ability to recharge itself in the *off* state, so we will treat it as a terminal state (we can still manually recharge it ourselves). 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. (4 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. (4 pts) We want the robot to try different actions to learn a better policy. It has the following Q-values so far: $Q(high, fast) = 2, Q(high, slow) = 0, Q(low, fast) = -1, Q(low, slow) = 1$. What is its current greedy policy? If we know that all possible future rewards are nonnegative, is it possible for the robot to learn a different policy if it always acts greedily and never explores? Explain your answer.

4. (4 pts) We send the robot off to do some active reinforcement learning. Starting off in the *high* state, it takes the greedy action (according to its Q-values from part 3 above), 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 $\alpha = 0.8$.

5. (4 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: Crawler Robot (30 points)

You will be training a simple crawler robot to move using dynamic programming and reinforcement learning. When you download the accompanying code files and run `python crawler.py` in your terminal, you should see a GUI pop with a robot toward 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 (12 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 `Callables` (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 (12 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 HW3a Coding bin on Gradescope.