CMPUT 366Assignment 3: Dynamic Programming and Monte Carlo methods

Due: Tuesday Oct 16 by gradescope

There are a total of 100 points available on this assignment, and 5 bonus points.

Question 1. Exercises from SB textbook. [**18 points total**]. This question has two parts corresponding to two exercises from the book.

Exercise 4.3 [9 points, 3 for each equation] (*policy evaluation equations for action-value functions*). Show all the steps in your answer for full marks

Exercise 4.5 [9 points] *(policy iteration for action values)*

Exercise 4.3.

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s]$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s] \quad 4.3$$
$$= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma V_\pi(s')] \quad 4.4$$

$$V_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_k(S_{t+1}) \mid S_t = s]$$
$$= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma V_k(s')] \quad 4.5.$$

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s]$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = A] \quad 4.3'$$
$$= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \sum_{a'} \pi(a' \mid s') \cdot q_\pi(s', a')\right] \quad 4.4'$$

$$q_{k+1}(s,a) = \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \sum_{a'} \pi(a' \mid s') \cdot q_k(s', a')\right]$$

4.5'

Exercise 4.5

1. Init.

$Q(s, a) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$, $a \in A(s)$.

2. Policy Evaluation.

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in S$, $a \in A(s)$:

$q \leftarrow Q(s, a)$

$Q(s,a) \leftarrow \sum_{s', r} p(s', r \mid s, a)[r + \gamma Q(s', \pi(s'))]$

$\Delta \leftarrow \max(\Delta, |q - Q(s, a)|)$

until $\Delta < \theta$ (a small positive number ....).

3. Policy Improvement:

policy-stable $\leftarrow$ true

for each $s \in S$:

old action $\leftarrow \pi(s)$

$\pi(s) \leftarrow \arg\max_a Q(s, a)$

if old-action $\neq \pi(s)$, then policy-stable $\leftarrow$ false.

If policy-stable, then stop and return $Q \approx q_*$ and $\pi \approx \pi_*$

else go to 2.

Question 2. Programing exercise. There are three parts worth **82 points** total**. Submit your plots with the code submission in Gradescope, not with the written part!**

Part 1. Programming exercise. **[32 points]**. **Exercise 4.9** from Sutton and Barto second Ed. You are not expected to use the RL-Glue interface for this exercise. **But you must use python3!**

Please submit all your code and your plot of the value estimates, and the plot of the final policy.

Part 2. Programming exercise. **[40 points]**.

Implement On-policy Monte Carlo Control with Exploring Starts for action values (described in Section 5.3) on the Gambler's problem described in Chapter 4.

We will make two changes from the problem specification in the book. First we will set pr(heads) = 0.55. In addition we will **not allow** the **zero** bet action; the agent must always make a bet of one or greater.

You will use RL-Glue. We will provide you with the experiment program to run the experiment. You can either use the plotting script provided or use the data generated by gambler_exp.py and plot it some other way. Look at gambler_exp.py to see how the data is stored. You have to modify mc_agent.py correctly, to implement the Monte Carlo control method and modify gambler_env.py to implement the gamblers problem. Then run the experiment and generate the graphs you need. Please complete the implementation of (1) the Monte Carlo agent (in mc_agent.py) and the Gambler's problem (in gambler_env.py). It is your job to complete them—they currently will not run.

Please use the code we have provided. **Do not modify** rl_glue.py. You will only need to modify mc_agent.py, gambler_env.py, and possibly gambler_exp.py if you want to change the number of runs or how the data is stored.

You will plot the Monte Carlo agent's estimate of $v_*(s)$ during training. To do this: plot on the y-axis V(s) for each state on the x-axis. The Monte Carlo Exploring Starts algorithm iteratively updates an action-value function Q(s,a). You will compute V(s) using the following relation: $V(s) = \max_a Q(s, a)$ for all $s \in S$. You will plot V(s) after 100 episodes, 1000 episodes, and 8000 episodes. The plotted V should be produced by averaging over 10 independent runs. That is, your plot should contain 3 lines: V after 100 episodes averaged over 10 runs, V after 1000 episodes averaged over 10 runs, and V after 8000 episodes averaged over 10 runs. The experiment program you have been given does the averaging for you. You just have to make sure agent_message in mc_agent.py does the correct thing. You may experiment with more than 8000 episodes if you like. Our implementation takes 3 to 5 minutes to run 8000 episodes and 10 runs. Please perform at least 10 runs, but feel free to test more runs (typically leading to more clear results). Number of runs and averaging is done in the experiment program which is provided for you.

Hint: the initial policy used for Monte Carlo is important for getting good performance in this problem. Therefore we want to initialize the policy in a smart way. One example: initially pi(s) = min(s, 100-s) (i.e. bet all the needed capital to win). Other initializations might work better. Feel free to explore. Note: this is the initial policy. The agent's goal is to learn and change the policy over time.

Please submit your plot and **all** code used to generate your results.

Part 3. [10 points]. Discuss and compare how the plot produced by Monte Carlo Control with Exploring Starts is similar and different from the plot produced by value iteration (in Question 2, part 1). Discuss why Dynamic Programming is suitable for the Gambler's problem, and why the Monte Carlo method with exploring starts might be less suitable for tasks like the Gambler's problem. Experiment with pr(heads) different than 0.55, to see what happens and report those results and submit those plots.

Bonus Question. Programming. **[5 points]** *(implement constant-alpha Monte-Carlo ES agent to find the optimal policy in Blackjack)*

This question requires implementing an RL-agent that performs constant-alpha Monte-Carlo Exploring Starts algorithm (described in Section 5.3). You will then run some experiments with that agent. We will provide the environment program for Blackjack and the experiment program. Feel free to modify the experiment program as needed. All you need to do is write the agent and run some experiments. We will also provide a random agent to get you started.

We will use a blackjack MDP similar to that described in Examples 5.1 and 5.3 in the book. One difference is that we introduce a special additional state corresponding to the beginning of a blackjack game. There is only one action available in that state, and it results in a transition to one of the regular states treated in the book . We include this initial state so that you can learn its value, which will be the overall value for playing the game. The initial state is numbered 0, and the regular states are numbered 1-180. (All actions are the same in state 0.)

The regular states are represented by the tuple (playerSum, dealerShowing, usableAce). The playerSum, which is between 12 and 20, is the sum of the cards held by the player; dealerShowing, which is between 1 and 10, represents the card that the dealer is offering to the player; usableAce is 0 or 1 depending on whether the player has a usable ace. There are 180 valid tuples, so we represent them by integers form 1 to 180. You do not have to worry about converting the state tuples to and from state integers because the your agent will only interact with the states as integers. Note that in this implementation, the playerSum is never 21. Since 21 is the best possible hand in blackjack, we assume the player always chooses the 'stick' action when their playerSum is 21. Similarly, since the player can not exceed 21 by choosing 'hit' when their playerSum is less than 12, we assume the player always chooses the 'hit' action when their playerSum is less than 12. Thus, the sequence of states in an episode begins with 0, is followed by some states between 1 and 180, and finishes in the terminal state represented by the Python value 'False'. The two actions, which are permitted in all non-terminal states, are 0 for 'stick' and 1 for 'hit'.

First test the equiprobable-random policy, run a number episodes and observe the returns from the initial state (assuming γ=1). These returns should all be either -1, 0, or +1, with an average of around -0.27 or so. If they don't, then something is off.

Next copy the random agent to file to produce a new file MCESAgent.py. In it, implement the Monte-Carlo Exploring starts algorithm (constant alpha version from Ch 6). Set alpha=0.01 and run for perhaps 1,000,000 episodes, observing the average return every 10,000 episodes, which should increase over episodes as learning progresses. After learning, print out your learned policy using agent message ``printPolicy''. One way to assess the quality of your policy is how similar it looks to that given in the textbook.

Try 5,000,000 episodes and print the policy again.

Now experiment with the settings of alpha, and the number of episodes to find a setting that reliably produces a better policy than that obtained with alpha=0.01 and 5,000,000 episodes. Report the settings, the final policy (by printPolicy) and the reliable performance level obtained by running the policy.

part 3.

MC agent does not use the model of environment.

DP has used environment and converges faster than MC agent. In this problem, the model has not been used.

So DP is better than MC in this problem.