INF8250AE(Fall 2024) : Reinforcement Learning — Given on: Sep 23

Assignment #2 — Due on : Oct 23, 10 pm

- The primary objective of this assignment is to evaluate your comprehension and application of TD and Monte-Carlo learning algorithms. Through this task, you will demonstrate your ability to understand the core concepts, implement the algorithms, and analyze their performance.
- This is an individual assignment. You are not allowed to discuss the problems with other students.
- Make sure to write your code only in the `.py` files provided. Avoid creating new files. Do not rename the files or functions, as it will interfere with the autograder's ability to evaluate your work correctly. Also, do not change the input or output structure of the functions.
- When Submitting to GradeScope, be sure to submit:
  1. A `'.zip'` file containing all your Python codes (in `.py` format) to the `'Assignment 2 - Code'` section on Gradescope.
  2. A `'pdf'` file that contains your answers to the questions and generated plots to the `'Assignment 2 - Report'` entry.
- Part of this assignment will be auto-graded by Gradescope. You can use it as immediate feedback to improve your answers. You can resubmit as many times as you want. We provide some tests that you can use to check that your code will be executed properly on Gradescope. These are not meant to check the correctness of your answers. We encourage you to write your own tests for this.
- You cannot use ChatGPT or any other code assistance tool for the programming part; however, if you use ChatGPT to edit the grammar in your report, you have to explicitly state it in the report.
- If you have questions regarding the assignment, you can ask for clarifications in Piazza. You should use the corresponding tag for this assignment.

1 TD for the variance of returns

In previous lectures, we defined the state value function as the expected sum of discounted rewards for any policy π:

$$v_\pi(s) := \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \quad \forall s \in \mathcal{S},$$

and showed that it satisfies a Bellman equation:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left(r + \gamma v_\pi(s')\right), \quad \forall s \in \mathcal{S}.$$

Similarly, we define the second moment of the sum of discounted rewards for any policy π:

$$m_\pi(s) := \mathbb{E}_\pi \left[\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \right)^2 | S_t = s \right].$$

1. (8 points)
   (a) (6 points) Show that the second moment satisfies the recursion:

   $$m_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left(r^2 + 2\gamma r v_\pi(s') + \gamma^2 m_\pi(s')\right), \quad \forall s \in \mathcal{S}.$$

   (b) (2 points) How many equations do we need to solve in order to compute $m_\pi(s)$ for all $s \in \mathcal{S}$?

2. (7 points)
   (a) (2 points) We now define the variance of the return:

   $$w_\pi(s) := \text{Var} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right].$$

   Explain how w_π can be calculated.
   (b) (5 points) Outline a variant of TD(0) algorithm that concurrently estimates the first moment (i.e., the value function v_π) and the second moment (i.e., m_π) from experience to recursively estimate the variance w_π. Report the pseudocode in the .pdf file.

2 Racetrack

Consider a racetrack environment based on the OpenAI gym library. The agent must drive a race car around a turn as fast as possible without running on gravels. In our simplified version, the track is a discrete set of grid positions.

At the beginning of each episode, the agent is in one of the randomly sampled states from the starting line with zero velocity on both x and y coordinates. The episode ends when the car crosses the finish line.

The agent controls its acceleration through discrete increments on the velocity components. Each increment may be $+1$ (accelerate), -1 (decelerate) or 0 (keep the same velocity) in one step and per coordinate. This makes 3 actions for x, times 3 actions for y, equals 9 actions in total. The y velocity is non-positive (no backward move), both velocity components are less than 5, and cannot both be zero (except at the starting line).

The agent gets -1 for each step on the track, -1000 if it touches gravels, 50 if it reaches the finishing line and -500 if it runs out of time. In all but the first case, the episode ends and the chronometer restarts. For your convenience, the environment is implemented in `environment.py`.

2.1 Helper Methods

1. (10 points)
   (a) (2 points): Understanding the module `environment.py`
   According to the code, what is the state-space? What is the action space? How many elements does each space have? Report your answer in the .pdf file.
   (b) (4 points): Completing the method `make_eps_greedy_policy` in `utils.py`
   Implement an epsilon-greedy policy over the state-action values of an environment. Besides the ϵ parameter, it should take as input a dictionary where the keys correspond to all state-action pairs, and the values are the corresponding Q-value estimates. The method should return a sampled action.
   (c) (4 points): Completing the method `generate_episode` in `utils.py`
   Create a function `generate_episode` which takes as input a policy π (e.g., the output from `make_eps_greedy_policy` in Question 1.(b)) and the environment. This function should return 3 lists, each containing respectively the episode's states, actions, and rewards generated from π.

2.2 On-Policy Monte-Carlo

2. (6 points): Completing the method `fv_mc_estimation` in `algorithms.py`
   The first-visit Monte Carlo prediction algorithm evaluates any given policy π. It takes as input a policy from which it generates episodes and averages the returns according to the 'first-visit' rule. One of the building blocks to this value prediction algorithm is to compute the return from one sampled episode. Complete the method `fv_mc_estimation` according to the following:
   - it takes as input a triplet of lists `states`, `actions`, `rewards` (for example, an output from `generate_episode` in Question 1.(c))
   - it returns a dictionary where each key is a unique pair of `state`, `action` encountered during the episode, and each value is the corresponding return computed according to the 'first-visit' rule.
3. (6 points)
   (a) (4 points): Training and Plotting results
   We now aim to do control and find an approximate optimal policy. The script `run_fv_mc.py` runs the first-visit Monte-Carlo method `fv_mc_control` in

`algorithms.py` for 5 different runs. It then plots the average undiscounted return obtained across the 5 different runs as a function of episodes (the x-axis corresponds to each episode from 1 to `num_episodes`, and the y-axis is the average return obtained at each episode). It also saves the results from all runs, which will be useful in the next question. Run the script `run_fv_mc.py` and report the plot you obtain in the .pdf file. Please note that this process takes some time to complete.

(b) (2 points) Why do we observe some variance on the return, even after convergence? In other words, why don't we get the same return at the limit even though all runs converge to the same policy?

4. (8 points): Evaluating a trained policy

   (a) (3 points) Take the last learned state-action value dictionary from the previous runs. Now set $\epsilon = 0$ and evaluate the 0-greedy policy on 5 episodes. Report the accumulated returns you get in each episode run.

   (b) (3 points) Take again the last state-action value dictionary from the previous runs but this time, set $\epsilon = 0.5$ to evaluate an ϵ-greedy policy on 5 episodes. Report the accumulated returns you get in each episode run.

   (c) (2 points) Compare the accumulated returns you obtain on average in Questions 4.(a) and 4.(b). What do you remark? Why does this happen?

3 Off-policy Monte-Carlo

5. (3 points): Completing the method `make_eps_greedy_policy_distribution` in `utils.py`

   Before implementing off-policy control, complete the method `make_eps_greedy_policy_distribution`. Similarly as `make_eps_greedy_policy`, it takes a dictionary of state-action values and an ϵ parameter as inputs, but instead of returning a *sampled action*, it returns the full policy *distribution over actions* $a \mapsto \pi(a|s)$. The file *utils.py* also provides the helper function `convert_to_sampling_policy` to transform such a policy back to an action sampling policy.

   In the remainder of this section, we take a uniform policy distribution as the behavior policy to generate episodes.

6. (8 points)

   (a) (5 points): Completing the method `is_mc_estimate_with_ratios` in `algorithms.py`

   Complete `is_mc_estimate_with_ratios` to compute the importance sampling estimates of the state-action pairs for an episode of a given target policy for the uniform behavior policy. Let it also return all the associated importance sampling ratios.

   (b) (3 points): Training and Plotting results

   The script `run_off_policy_mc_eval.py` runs ordinary importance sampling to predict the value of one of the ϵ-greedy policies trained in Question 3.(a). It then

plots the average off-policy Monte Carlo estimates of $v_\pi(s)$ for some specified states against the number of episodes. Execute the script and report the plots in the .pdf file.

7. (11 points): Training and comparing results
   (a) (5 points): Completing the method `ev_mc_off_policy_control` in `algorithms.py`
   Now, we can do off-policy control and find an approximate optimal policy, even though actions may be selected according to a suboptimal policy. In this part, we are interested in implementing an every-visit Monte-Carlo method that computes ordinary importance-sampling ratios. Complete the method `ev_mc_off_policy_control` in `algorithms.py`.
   (b) (3 points) The script `run_ev_mc.py` runs the off-policy Monte-Carlo method `ev_mc_off_policy_control` for 5 different runs and plots the average undiscounted return obtained across the 5 different runs as a function of episodes. Run the script `run_ev_mc.py` and report the plot you obtain in the .pdf file.
   (c) (3 points) By observing the convergence plots obtained from Questions 3.(a) and 7.(b), which of the two algorithms between on-policy and off-policy Monte-Carlo necessitated fewer episodes (or samples) to converge? What can we hypothesize about the sample complexity of on-policy versus off-policy control?

4 TD-Control

Continuing with the same Reacetrack environment as before, we now investigate TD(0)-control methods. First, carefully read and understand the code provided for a base class `Agent` in `td_algos.py`. It will serve as the parent class for all learning agents you will implement in the remaining sections.

8. (4 points): Completing the method `train_episode` in `td_algos.py`
   Implement the method `train_episode` in `td_algos.py`. It is similar to `generate_episode` from Question 1.(c), but it takes an agent as an argument instead of the policy and simultaneously trains the agent while generating an episode. *Hint:* Use the `agent_step` method from the `Agent` class to concurrently get an action and train the agent.

4.1 Sarsa

9. (12 points): In this part, we want to do on-policy control using Sarsa.
   (a) (6 points) Complete the method `agent_step` from the `SarsaAgent` class in `td_algos.py` according to the following:
      - It takes as input a current state s, current action a, immediate reward r and next state s' (sars')

- It updates the Q-value function according to Sarsa rule (make sure you correctly handle terminal states)
- It returns the next action to take

(b) (3 points) We can now do control. The script `run_td_sarsa.py` sets $\epsilon = 0.05$, $\gamma = 0.99$, `num_episodes` $= 2000$ and `step_size` $= 0.1$ and trains a Sarsa policy over 5 different runs. It additionally saves the results from all runs, which will be useful in the next question. Execute the script `run_td_sarsa.py` and display the convergence plot of returns you obtained in the .pdf file.

(c) (3 points) Take all Sarsa agents trained in the 5 previous runs. Now set $\epsilon = 0$ and sample one episode per 0-greedy policy (5 episodes total). Report the accumulated return you get for each of the episode runs.

4.2 Q-learning (off-policy Sarsa)

We now aim to do off-policy control using Q-learning.

10. (2 points) Why is Q-learning called an 'off-policy' algorithm?

11. (12 points)

(a) (6 points) Similarly as in Question 9.(a), complete the method `agent_step` from the `QLearningAgent` class in `td_algos.py`. It takes the same inputs as `agent_step` from `SarsaAgent` and returns an action to take, but this time, the Q-value function must be updated according to Q-learning (make sure you correctly handle terminal states).

(b) (3 points) We can now do control. Modify the script `run_td_sarsa.py` to train Q-learning over 5 different runs, under the same parameters as in Question 9.(b). Execute the script and display the convergence plot of returns you obtained in the .pdf file.

(c) (3 points) Take all Q-learning agents trained in the 5 previous runs. Set $\epsilon = 0$ again and sample one episode per 0-greedy policy (5 episodes total). Report the accumulated return you get for each of the episode runs.

12. (3 points): Comparing trained policies

Take the last learned state-action value dictionary from Sarsa and Q-learning runs. Set $\epsilon = 0.2$ and evaluate the ϵ-greedy policy of both `sarsa` and `q_learning` on 5 episodes. Report the returns you get in each run, for each algorithm. Which algorithm did you find less sensitive to this change? Explain why.