

CS 4122: Reinforcement Learning

Programming Assignment #3 (for Module 3)

Release date: 13th April, 2025, 10:30 pm (IST)

Due date: 27th April, 2025, 11:59 pm (IST)

Maximum Score: 100 marks (this assignment is graded)


Read the following before you move forward:

1. This programming assignment is worth 10% of the total marks of this course.
2. Late policy: You can be late by a maximum of 3 days. Your assignment will not be accepted more than 3 days after the due date. The actual score and received score (received score is actual score minus penalty for late submission) are related as follows:
$$\text{received score} = \begin{cases} \text{actual score} & ; \text{late by 0 days} \\ 0.9 \cdot \text{actual score} & ; \text{late by 1 day} \\ 0.8 \cdot \text{actual score} & ; \text{late by 2 days} \\ 0.7 \cdot \text{actual score} & ; \text{late by 3 days} \\ 0 & ; \text{late by more than 3 days} \end{cases}$$
3. This is a team project. You have to do the project with the team that was finally selected for you. Only one submission per team. To submit, go to your team's Google drive folder. The Google drive link was emailed to you. Then go to the sub-folder titled **Programming Assignment 3** and drop the following files (DON'T zip/compress these files, DON'T submit any other files): (i) report.pdf (ii) GymTraffic.py (iii) training.py (iv) Trained policies as .npy files (v) testing.py Sections 4-6 have deliverables. Read these deliverables carefully to understand what needs to be submitted in the report and in the Python codes.
4. The first page of report.pdf should contain a detailed description of the contribution made by individual team members. NOTE: Documenting the work is NOT a valid contribution.
5. Even though each team member may contribute to a specific section of the assignment, every team member should have the technical understanding of the entirety of the project. I will be conducting random assessment to check student's understanding of the assignment. Assignment marks will depend on the random assessment as well.
6. Plagiarism, if detected, will lead to a score of ZERO for all the team member; no exceptions!
 - a. It is ok to take help from AI tools. It is down right stupid to copy-paste from AI tools and think that is acceptable.
 - b. It is ok to take help from other teams. But, it is a bad idea to keep open another team's work in front of you while you do the assignment. While you may think you are just taking help, your reports and your codes may get heavily influenced by the other team to the point that it will qualify as being plagiarized.

Traffic Light Control using Reinforcement Learning

Coding Custom Environments

One of the tasks of this programming assignment is to code a **custom environment** using OpenAI Gymnasium. More specifically, this environment is going to simulate a traffic intersection controlled by a traffic light. **Coding custom environments is one of the most important concepts that you will learn in this course as far as applied RL is concerned.** This is because you will get pre-built RL algorithms online that you can use. But, the practical applications where you will be applying these RL algorithms will be very specific to your case. And hence, with high probability, you have to make environments custom designed for your practical application.



Section 1: Installation

Installation instructions are same as programming assignment 1. In fact, you don't need any deep learning libraries (like Tensorflow, Keras, and Pytorch) for this programming assignment.

Section 2: Sharing workload

Just like programming assignment 2, sharing workload is a little tricky because the tasks are **not decoupled**. The following are my two cents about sharing workload:

1. Section 4 is about designing custom OpenAI Gymnasium environment. This is where the **coupling** comes into picture; you can't fully attempt the later sections without completing section 4. So, I suggest that every team member should contribute to section 4. This is also important because of the reason mentioned here.
2. In section 5, there are two tasks related to implementing TD-based RL algorithms. These two tasks are decoupled. You can easily divide these tasks among yourself and do it independently. Also, you really don't have to wait for section 4 to complete before you can attempt section 5. You can still write a skeleton of the code and then fill in the blanks after section 4 is completed.

Section 3: Useful resources

1. In section 4, you will be coding a custom environment. To learn how to code a custom environment, refer to [lecture 0, part 4](#) (there are two video lectures in it containing two examples).
2. In sections 5, you will be training RL agents using temporal difference (TD) based learning algorithms. You can find the pseudocode of these learning algorithms in [lecture 26 and 27 slides](#).
3. Here is a [video lecture](#) that demonstrates how to implement **Q-Learning in Python**:

<https://youtu.be/OJXztVLbRUk>

Section 4: Custom OpenAI Gym Environment (35 marks)

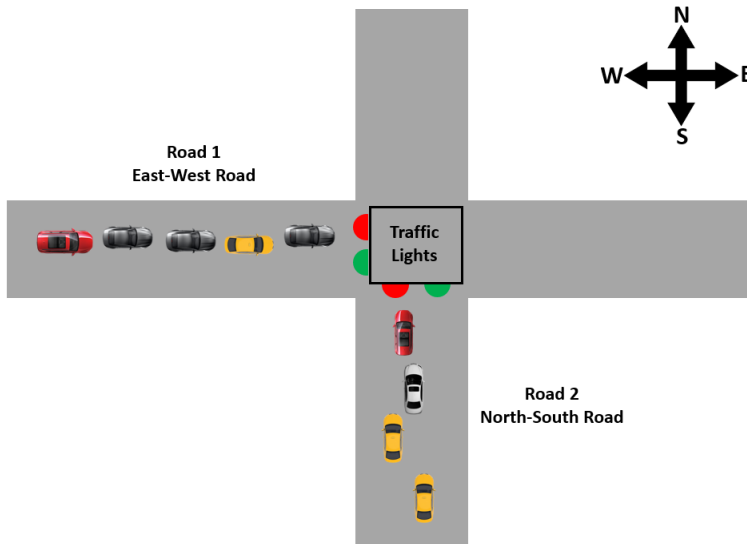


Fig. 1

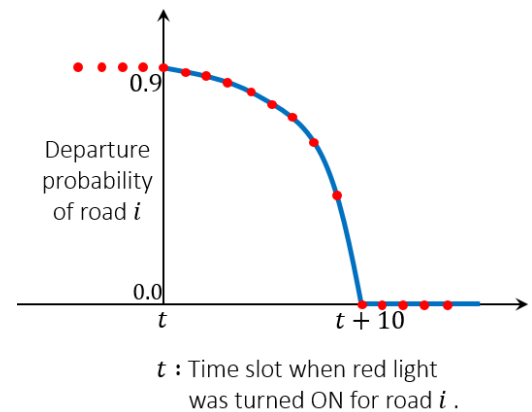


Fig. 2

In this section, your task is to code a custom OpenAI Gymnasium environment for the setup discussed in the following paragraphs.

Consider a traffic intersection as shown in Fig. 1. It consists of an east-west road (road 1) and north-south road (road 2) being controlled by a traffic light. The traffic light can show either green (GO) or red (STOP); there is no yellow light. The objective of the traffic light controller is to minimize the average wait time of the vehicles in the traffic intersection which can be shown to be equivalent¹ to minimizing the average queue length of the total number of vehicles in roads 1 and 2².

We consider that the time is slotted where each time slot is of 1 sec duration. Vehicles come and join the back of the queue corresponding to each of the two roads. For each of the two roads, at most one vehicle joins the queue in any time slot. The probability that a vehicle will join roads 1 and 2 at any time slot is 0.28 and 0.4 respectively. These are called arrival probabilities.

When the traffic light turns green for a given road, the vehicle starts departing from the front of the queue. The departure model of the vehicles is same for both the roads. The remaining paragraph describes this model. At most one vehicle can depart the queue in any time slot. The probability that a vehicle will depart in a time slot is called the departure probability. The departure probability is 0.9. As soon as the traffic light turns red for a given road, the vehicles are unlikely to stop departing from that road immediately. This is captured in our model by using a departure probability that decreases with time as shown in Fig. 2. To elaborate, let's say that at time slot t , the traffic light for road k turns red. The departure probability of road i at time t is 0.9 (same as this). The departure probability decreases from time t to $t + 10$ and at time $t + 10$ it becomes 0. Since each time slot is of 1 sec duration, this model

¹ This equivalence is due to Little's law. You don't have to know about Little's law to do the rest of the assignment.

² Even though the objective is to minimize the average queue length, the RL algorithms that you will code in the subsequent sections will minimize the discounted queue length. This is because of simpler convergence guarantee of discounted rewards and something called Blackwell's optimality that I mentioned in module 2. That said, as far as coding the environment is concerned, there will be no change whether we consider average or discounted queue length.

essentially means that after the traffic light turns red, the vehicles will take **10 sec** to come to a complete halt which seems reasonable. The exact formula of the departure probability at any time slot $t + \delta$ between time slots t and $t + 10$ is,

$$0.9 \left(1 - \frac{\delta^2}{100} \right)$$

Let \tilde{k} denote the “other” road, i.e. if $k = 1$ then $\tilde{k} = 2$ and if $k = 2$ then $\tilde{k} = 1$. If the traffic light of road k turns red at time slot t , then the following point holds:

- The traffic light of road \tilde{k} **MUST** be red between time slots t to $t+9$. This is because from time slot t to $t+9$, the traffic is still flowing from road k . Therefore, if traffic starts flowing from road \tilde{k} as well, it can lead to an accident.
- The traffic light of road \tilde{k} will become green in time slot $t + 10$ provided that traffic light of road k did NOT become green again between time slot $t + 1$ to $t+9$. This implicitly means that even if the traffic light of a road turns red, it can turn green again before the traffic light of the other road turns green.

Needless to say, in every time slot, the traffic light of at most one of the roads can be green.

Deliverables:

1. **(10 marks)** Model this problem as an MDP by documenting the following in **report.pdf**:
 - a. **States and state space.** Keep the number of states as small as possible. Otherwise, it is going to slow down training in the subsequent sections. HINT: Queue lengths of the roads will not exceed 1810. Think why! This observation is critical because the state space can't be infinite.
 - b. **Action and action space.**
 - c. **Reward.**
 - d. **State transition equations.**
2. **(25 marks)** Using the answer to the first question, code a custom OpenAI Gymnasium environment for the setup described in the previous page. The skeleton of this environment is there in **GymTraffic.py** that is provided to you. **GymTraffic.py** contains **GymTrafficEnv** class. Implement the following functions of **GymTrafficEnv** class:
 - a. **__init__()**: This function should initialize all the system parameters like the arrival probabilities, and the parameters related with departure probability.
 - b. **reset()**: The function to reset the environment in the beginning of every episode. Set the queue lengths of the roads uniformly at random between 0 to 10.
 - c. **step()**: This function should take the action as input and return the following:
 - i. Next state.
 - ii. Current reward.
 - iii. *terminated*. This should be *False*.
 - iv. *truncated*. This setup is a continuous task. But, truncate an episode after 1800 time slots, i.e. each episode is of 30 min with the duration of one time slot being 1 sec.
 - v. *info*. This should be an empty dictionary.

Section 5: Training RL algorithms (50 marks)

In this section, you will be coding three temporal difference (TD) based learning algorithms for the environment that you coded in the previous section. You must follow these two points while you implement these algorithms:

1. **Truncating state space:** In TD-based learning approaches we learn Q-values for all state-action pairs. But the state space of the queue length is large for this problem. Because of this you will realize that the size of the Q-matrix is so large that it may not even fit in RAM. To resolve this situation, we will NOT learn Q-values for all state-action pairs. Rather, we will learn Q-values for state-action pairs that we are highly likely to encounter. In this section we will assume that a queue length of more than 20 is highly unlikely. Accordingly, we will learn Q-values for all those state-action pairs whose queue length is less than or equal to 20. This will reduce the dimension of the Q-matrix. But, one question remains unanswered: It is still possible that the queue length can become more than 20, however low be the probability. How to decide what action to take in such situations? ANSWER: If queue length is greater than 20, then we will assume that the queue length is 20 and decide the action accordingly.
2. **Exploration strategy:** For all the three algorithms you MUST use an ϵ –greedy policy derived from the current Q-function as the **behavior policy**. For the exploration part of the ϵ –greedy policy, you MUST NOT choose all the actions with equal probability. Rather, during exploration, the probability of choosing an action a for state x MUST be,

$$\frac{e^{\bar{q}(x,a)}}{\sum_{\tilde{a} \in \mathcal{A}(x)} e^{\bar{q}(x,\tilde{a})}}$$

where,

$$\bar{q}(x,a) = \frac{q(x,a)}{\nu + \sum_{\tilde{a} \in \mathcal{A}(x)} |q(x,\tilde{a})|}$$

In the above two mathematical expressions, $q(x,a)$ is the Q-function, $\bar{q}(x,a)$ is the normalized Q-function, $\mathcal{A}(x)$ is the action space for state x , ν is a very small positive number, and $|\cdot|$ implies the absolute value operator. In the exponential term, we use $\bar{q}(x,a)$ instead of $q(x,a)$ because $q(x,a)$ can be either very big or very small causing numerical issues. ν is used to prevent division by zero (it is a hyperparameter). The above exploration strategy ensures that:

- Actions that has higher Q-value is more likely to be chosen. Actions with higher Q-value are likely to yield higher reward.
- All the actions have a finite probability of being chosen.

Deliverables:

1. **(15 marks)** Implement the original version of SARSA for the custom environment that you coded in the previous section. You must implement this code in the function **SARSA()** of the Python script **training.py** that is provided to you. **You MUST use the exploration strategy mentioned in the previous page otherwise you will get ZERO.** **SARSA()** should return the optimal policy (NOT the Q-value) as a Numpy array. Submit the final policy that you learned as **policy1.npy**.

- (15 marks) Implement expected-SARSA for the custom environment that you coded in the previous section. You must implement this code in the function **ExpectedSARSA()** of the Python script **training.py** that is provided to you. You MUST use the exploration strategy mentioned in the previous page otherwise you will get ZERO. ExpectedSARSA() should return the optimal policy (NOT the Q-value) as a Numpy array. Submit the final policy that you learned as **policy2.npy**.
- (20 marks) In Module 2, we discussed that if the state transition probabilities and reward probabilities are known, then we can compute the Q-function from the value function as follows,

$$q^{\pi}(x, a) = r(x, a) + \beta \sum_{x' \in S} P[x'|x, a] V^{\pi}(x')$$

This essentially means that if the state transition probabilities and reward probabilities are known, we can design a variant of SARSA where rather than updating the Q-function, we can update the value function just like we did in TD-based policy evaluation as follows,

$$V(x) = V(x) + \alpha(r + \beta V(x') - V(x))$$

In this variant of SARSA, when we have to choose an action for a state x , we first compute $q(x, a)$ for all $a \in \mathcal{A}(x)$ from $V(x)$ using this formula. We then use ε -greedy policy derived from $q(x, a)$ to choose the action. Your deliverables for this are the following:

- Document a neatly written/typed pseudocode of this variant of SARSA in **report.pdf**.
- If we have state transition probabilities and reward probabilities, then we can directly use value/policy iteration to compute the optimal policy. What is the advantage (if any) of this variant of SARSA over value/policy iteration? You must document the answer to this in **report.pdf**. *HINT: We discussed this in the very beginning of lecture 23.*
- What is the advantage (if any) of this variant of SARSA over the original version of SARSA? You must document the answer to this in **report.pdf**. *HINT: Think about the number of parameters that is being estimated/updated in original SARSA and this variant of SARSA.*
- Implement this variant of SARSA for the custom environment that you coded in the previous section. You must implement this code in the function **ValueFunctionSARSA()** of the Python script **training.py** that is provided to you. ValueFunctionSARSA() should return the optimal policy (NOT the Q-value) as a Numpy array. Submit the final policy that you learned as **policy3.npy**.

Read these following points before attempting the above deliverables:

- For all the three algorithms, you don't need to learn for more than 2000 episodes. Use discount factor $\beta = 0.997$ and learning rate $\alpha = 0.1$. It is your task to decide the schedule of the exploration probability.
- The Python script **training.py** already contains the skeleton of the code where the number of episodes, discount factor, and the learning rate are already set. The function also saves the policy that you learn as .npy files. You MUST NOT TAMPER with this skeleton. Just implement **SARSA()**, **ExpectedSARSA()**, and **ValueFunctionSARSA()**.
- Any additional functions required to implement these three algorithms must be declared inside **training.py**.

Section 6: Testing RL algorithms (15 marks)

In this section, you will test the RL algorithms that you trained in the previous section.

Deliverables: (15 marks) You are provided a Python script `testing.py`. Implement the function `TestPolicy()` of `testing.py` that simulates the custom environment that you coded in section 4 for [just one episode](#) using the policies that you trained in section 5. After this simulation, the function should plot the queue length of both the roads as a function of time slots. The function should also display the average of the sum of queue lengths of both the roads and the actions taken by the policy over one episode.