CS 747: Programming Assignment 3

Total marks: 12

(Prepared by Debabrata and Ashish)

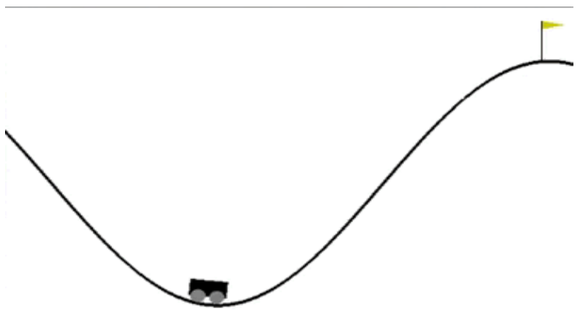
As a part of this assignment, you will write code to learn the *Mountain Car* task given as Example 9.2 by Sutton and Barto (2018). We will use the OpenAI Gym environment for this assignment. You will program Sarsa with linear function approximation, record your results, and present your interpretations.

Unlike the previous assignments, here we give you a code structure (version 1). Please find in-line comments in the files to implement the tasks. To run the code on docker, first install the `gym` and `pyglet` modules on the docker by running `pip install gym` & `pip install pyglet`. Make sure your code runs correctly on the docker container, otherwise you will not be awarded any marks for this assignment.

You can go to the documentation of OpenAI gym to get a fair amount of idea about how an agent must interact with the `gym` environment.

Mountain Car

1. The Mountain Car task is briefly described here.
2. You should be able to implement an agent which can learn to escape the valley successfully and reach the flag. The `gym` environment will terminate the episode after 200 timesteps even if you are not able to reach the goal: that is, 200 is the maximum episode length.



3. As you might have noticed, here the state space is continuous.More formally the state of the car or agent can be denoted by a pair of real numbers (x,v), where x is the the position of the car in the one dimensional track and v is the curent velocity of the car.As per the implementation of the environment in OpenAI Gym $x \in [-1.2,0.6]$ and $v \in [-0.07,0.07]$.
4. And at each timestep the agent can choose to perform one of the three actions encoded as three integers 0,1 and 2.

Num	Action
0	Accelerate to the left.
1	Don't accelerate.
2	Accelerate to the right.

5. For each time step the agent gets a reward of -1 unless it has reached the flag on the top of the mountain, in which case a reward of 0 is awarded and the episode ends. Our objective is to minimise the number of timesteps required to escape the valley, whic is equivalent to maximising the long-term reward.
6. The agent starts at a position selected uniformly at random from [-0.6,-0.4]. The velocity of the agent at the start is 0.
7. Given all this, your task is to implement a Sarsa(0) agent that learns to escape from the valley in minimum number of timesteps, or to maximize the amount of reward you get in each episode. More specifically, for this assignment, you have two tasks at hand.

Task 1: "Tabular" Sarsa

Here your task is to discretise the state space in some way (the way you see fit) so from the agent's point of view, the "state" it sees is one of a finite number (although in reality, the environment/simulator will still deal with real-valued state vector). With a discrete state space, the agent can perform normal Sarsa(0). However, think about it: tabular Sarsa(0) is a

special case of function approximation in which each state-action pair activates a single Boolean feature. So, you will write your tabular case in a form that can also be used in tandem with linear function approximation (as in Task 2).

In the code, you must fill out the function `get_table_features` to return a feature vector corresponding to a tabular representation. Given the state (x,v) this function is expected to return the discretised state that gets activated. It is up to you to decide what granularity of discretisation works best. The other functions, related to Sarsa(0), that you need to fill out are `sarsa_update` and `choose_action`: note that you are dealing with a linear representation: both updating and action selection must work on the weight vector. In this task as well as the next, you are expected to use constant learning and exploration rates. Tune them so your agent is able to sucessfully able to escape from the valley with less than 160 timesteps on average. You will be awarded full marks for this task if your agent is able to get reward (averaged over 100 episodes) of greater than -160.

Task 2: Sarsa with Linear Function Approximation

In this task, you are expected to come up with a feature vector that can perform better than a simple tabular representation as in Task 1. You can implement any reasonable approach: either standard ones such as tile coding or RBFs, or your own hand-designed ones. With such reasonable choices, you should be able to meet the requirement of this task, which is to complete the episode in fewer than 130 steps.

For this task you need to fill the function `get_better_features` which takes state (x,v) as input and returns the representation of the state or the features. The function you use to approximate Q will be linear in the feature vector. With this you should be able to get an average reward of (over 100 episodes) more than -130.

Code Layout

The Code template given has following structure.

- Most of the functions are filled.
- The class `sarsaAgent` holds all the implementation of the agent.The constructor of this class instantiates all the required hyperparamters and weights for the training.
- The `get_table_features` and `get_better_features` functions are described above.
- The `choose_action` function is expected to take state, current weights and epsilon as arguments and return a valid action to take.
- `sarsa_update` function as the name suggests updates the Sarsa weights for Task 1 as well as Task 2. It takes appropriate arguments (more details in the code).
- The `save_weights` function saves the final weights for the tasks.
- The `load_weights` function loads and returns the saved weights.
- The `train` function has to train the agent for a certain number of episodes and obtain the data for plotting graphs for the two tasks. Look at the comments in the code for more details.
- The `test` function is expected to load the saved weights by using the earlier `load_weights` function and runs 100 episodes and returns the average reward.
- You can look at the code template given and the comments for more details. It should be noted for the functions argument structure must not be changed.

Output

You can run the code using command `python mountain_car.py --task [T1/T2] --train [0/1]` . If the `--train` is 0 the code is not expected to output anything. It should just train the agent and save the final weights and plots as `T1.npy` and `T1.jpg` or `T2.npy` and `T2.jpg` based on the `--task` parameter. Additionally we have provided a script (verifyOutput.sh) to verify the output behaviour of the code.

Submission

Create a directory called `submission` and place the following material in it.

- Your `mountain_car.py` file (with the functions filled out).
- The final weights for Task 1 as `T1.npy` and the final weights for Task 2 as `T2.npy`.
- Training plots as T1.jpg and T2.jpg for Task 1 and Task 2, respectively.
- A README file describing how to run your code and obtain the plots and training data.
- A report presenting your observations from these experiments (as a pdf file named `report.pdf`).Also Place the plots in the report and provide accompanying commentary.

Compress the directory into `submission.tar.gz` and upload on Moodle under Programming Assignment 3.

Convince yourself that the results obtained match your expectations. Feel free to be creative and use the simulation environment to test related hypotheses you might find interesting. Report any particular issues you encountered while experimenting with this task. Don't hesitate to include additional numbers or graphs.

Evaluation

Each task is for 6 marks, with the marks being determined by the training plot, test runs, and the accompanying observations.

The TAs and instructor may look at your source code and notes to corroborate the results obtained by your program, and may also call you to a face-to-face session to explain your code.

Deadline and Rules

Your submission is due by 11.55 p.m., Tuesday, November 2nd. Finish working on your submission well in advance, keeping enough time to validate your code and to upload your submission to Moodle.

Your submission will not be evaluated (and will be given a score of zero) if it is not uploaded to Moodle by the deadline. Do not send your code or report to the instructor or TAs through any other channel. Requests to evaluate late submissions will not be entertained.

Your submission will receive a score of zero if your code does not execute on the cs747 docker container. To make sure you have uploaded the right version, download it and check after submitting (but before the deadline, so you can handle any contingencies before the deadline lapses).

You are expected to comply with the rules laid out in the "Academic Honesty" section on the course web page, failing which you are liable to be reported for academic malpractice.