Homework 3

Please write the following on your homework:

- Name
- Collaborators (write none if no collaborators)
- Source, if you obtained the solution through research, e.g. through the web.

While you may collaborate, you **must write up the solution yourself**. While it is okay for the solution ideas to come from discussion, it is considered as plagiarism if the solution write-up is highly similar to your collaborator's write-up or to other sources. For the programming assignment, we will be running an **automated plagiarism detector**. Highly similar programs will be investigated.
Your solution should be submitted by **Sunday 1 November 11.59pm**. Scanned handwritten solutions are acceptable but must be legible.
Late Policy: A late penalty of 20% per day will be imposed for the written assignment (no submission accepted after 5 late days) unless prior permission is obtained. Late submissions will not be accepted for the programming assignment.

---

1. **Q-Learning with Continuous State**
   Consider a system with a single continuous state variable x and actions a_1 and a_2. An agent can observe the value of the state variable as well as the reward in the observed state. Assume a discount factor $\gamma = 0.9$.

   (a) Assume that function approximation is used with $Q(x, a_1) = w_{0,1} + w_{1,1}x + w_{2,1}x^2$ and $Q(x, a_2) = w_{0,2} + w_{1,2}x + w_{2,2}x^2$. Give the Q-learning update equations.

   (b) Assume that $w_{i,j} = 1$ for all i, j. The following transition is observed: $x = 0.5$, observed reward $r = 10$, action a_1, next state $x' = 1$. What are the updated values of the parameters assuming a learning rate of 0.5?

2. **Policy Gradient with Continuous State**
   Assume that Q-function with function approximation is used together with the softmax function to form a policy $\pi_\theta(s, a) = e^{Q_\theta(s,a)} / \sum_{a'} e^{Q_\theta(s,a')}$. Assume that there are two actions with $Q(x, a_1) = w_{0,1} + w_{1,1}x + w_{2,1}x^2$ and $Q(x, a_2) = w_{0,2} + w_{1,2}x + w_{2,2}x^2$ for a real valued variable x.

   (a) Give the update equations for the *REINFORCE* algorithm. Assume that the the return at the current step is G and the action taken is a_1.

   (b) Assume that $w_{i,j} = 1$ for all i, j and return $G = 5$ is received. What are the updated values of the parameters assuming $x = 0.5$ and a learning rate of 0.5?

3. **Programming Assignment**

   In the second homework, we have learned to solve non-deterministic planning problems with Value Iteration and Monte Carlo Tree Search. However, both methods have a problem with scaling on a larger task In this task, we will learn to solve slightly larger planning problems in deterministic environments with *Deep Q Learning*. Despite the fact that we are solving a deterministic problem, the method that we are about to learn can also solve non-deterministic problems. We chose to limit the task to be deterministic just for simplicity and better predictability during training. Before proceeding further, please follow the instructions given below to complete the setup required for this programming assignment.

Installation Instructions[1]

- Remove `cs4246/base` image from the docker container with command : `docker rmi cs4246/base`
- Pull the updated docker image with the required dependencies with command : `docker pull cs4246/base`. Other required docker commands can be found here.

We will be using the same gym_grid_environment (`https://github.com/cs4246/gym-grid-driving`) to simulate the solutions. All dependencies have been installed in the **updated** docker image "cs4246/base" which you have downloaded.

By now, you should be familiar with the gym_grid_environment. Go through the IPython Notebook file on Google Colab here if you have not already done so in the previous programming assignment. You are now ready to begin solving the task.

Task: Deep Q Learning (6 Points)

Problem Definition

In the previous two homework, we assume that we somehow have the model of the environment. PDDL requires the determinized version of the MDP, VI requires the MDP, and MCTS requires a simulator of the environment. Here, we want to tackle a problem in which we don't have the underlying details of the environment. What we have instead is just the observation (image) of the top view of the street. Fortunately, DQN can be trained without any of those details.

In this task, we will implement a DQN algorithm with the following features.

- *Replay buffer*: During the training, the DQN agent will interact with the environment and produce a sequence of transition (experiences) that can be used for training data. However, training directly with such experiences may lead to instability due to the data

[1]The files to be installed are very large. Do your installation in while in SoC if it is too slow to do at home.

being temporally correlated. To remove correlation in the data, experiences are instead stored in a memory buffer. At every training step, the experiences are then sampled to train the agent. The memory is often called replay buffer since it contains the "replay" of the agent's interaction.

- *Epsilon greedy*: Training a DQN agent (and RL agent in general) is highly dependent on the ability to find good regions (i.e., regions at which it yields high value). If good regions fail to be discovered during training, the agent will be unlikely to perform well. However, most of the time, we won't know of a good heuristic to determine which region is useful for training, hence random exploration strategy is employed. One such strategy is called the epsilon greedy exploration strategy, in which the agent chooses to perform a random action with some probability at every step during experience gathering.
- *Target Network*: DQN loss function computes the distance between the Q values induced by the model with its expected Q values (given by the next state Q values and current state reward). However, such an objective is hard to optimize because it optimizes for a moving target distribution (i.e., the expected Q values). To minimize such a problem, instead of computing the expected Q values using the model that we are currently training, we use a target model (target network) instead, a replica of the model that is synchronized every few episodes.

We provide an agent template that has a code snippet that allows you to train the DQN agent and test the agent in the server. The use of the agent template is required. Your agent definition is located in the `agent/dqn.py`. What you have to do is to fill all functions that are marked with "FILL ME" listed below.

- Replay Buffer functions:
  - `ReplayBuffer.__init__`: initialize the replay buffer.
  - `ReplayBuffer.push`: add transition to replay buffer.
  - `ReplayBuffer.sample`: sample from the replay buffer.
- `BaseAgent.act`: computes an action of a given state following the ϵ-greedy formulation: with $(1 - \epsilon)$ probability output the actual action, otherwise, with ϵ probability output random action.
- `compute_loss`: compute the temporal difference loss δ of the DQN model Q, incorporating the target network $\hat{Q}$: $\delta = Q(s, a) - (r + \gamma \max_a \hat{Q}(s', a)$; with discounting factor γ and reward r of state s after taking action a and giving next state s'. To minimize this distance, we can use Mean Squared Error (MSE) loss or Huber loss. Huber loss is known to be less sensitive to outliers and in some cases prevents exploding gradients (e.g. see Fast R-CNN paper by Ross Girshick).

You can see more details in the `agent/dqn.py` code comments.

Training & Testing the Agent

You can train the agent by executing `python dqn.py --train`. It will train the agent neural network model for 2,000 episodes, with 10 training steps for each episode. The training will take about 6-10 minutes in Tembusu (See 3) using GPU. If you are training on CPU, it might take around 30 minutes to an hour depending on the spec of your machine.

Training Deep Q Learning model (Deep Reinforcement Learning model in general) is a difficult task, and highly sensitive to changes in the hyperparameters, initializations, and changes in the environment. To avoid that, we have included a function that automatically checks whether you happen to be unlucky with your initialization, and stops the program immediately to avoid wasting time. It will print "Bad initialization. Please restart the training.", indicating that you should re-execute the training command.

As a rule of thumb for debugging, if your rewards are not increasing consistently after a few episodes, it means that there is something wrong with your implementation. You should try to figure out what's wrong and fix it first before continuing.

After the training has completed, your agent will be tested automatically. The model of your agent will also be saved automatically at `agent/model.pt`. To manually test the saved agent with the example test cases, you can execute: `python dqn.py`. Your agent will be evaluated on 600 randomly generated test cases of the same size and specification. If you have implemented all functions correctly, you should get an average reward of ≥ 8.5.

Grading

We follow the following grading scheme for this task:

- If avg reward ≥ 8.0: Point = 6 (maximum point),
- Else: Point = 0.

*Point computed in the submission server.

Compute Resources

Since we are working with a problem with a larger scale, you might benefit from larger (and highly available compute resources). We recommend you to use Google Colaboratory and Tembusu cluster (if available).

Google Colaboratory is just a modified Jupyter Notebook instance running on Google Cloud Server with decent GPUs. It's free, but your instance will get reset every 12 hours. So please do make a backup of your files if you decided to use it. If you are not familiar with it, please check out the introduction to Google Colab first.

To run the agent code in Google Colaboratory (or Jupyter Notebook in general), you have to follow the following instruction:

(a) At the beginning of the notebook, put a code block to install the dependencies, e.g., torch, numpy, gym-grid-driving.

(b) Split the agent/dqn.py into two code blocks: one before the "if __name__ == '__main__'" and the other after it. Next we will refer those two blocks as block 1 and block 2.

(c) To run your agent, you just have to execute the code blocks sequentially from top to bottom.

Tembusu: There is no change needed to the code apart from the fact that you have to install the dependencies yourself. Please refer to the manual setup guide for more details.

Non-SoC students can go to the following URL to create/re-enable their SoC account: `https://mysoc.nus.edu.sg/~newacct`. For information on using the cluster, see `https://dochub.comp.nus.edu.sg/cf/services/compute-cluster`.

Submission Instructions

Please follow the instructions listed here. You have to make submission as a zipped agent with the correct submission format. Please make sure to read the frequently asked questions to avoid problems.

- Please make sure that you have completed all functions that are marked with "FILL ME".
- The maximum size of the zipped agent is 5MB.

Extra Notes:

(a) Please make sure that everything that you need for the agent to run resides in the agent/ folder.

(b) Please do not print anything to the console, as it might interfere with the grading script.

(c) There will be 3 daily submission limit for each task. Please use it wisely.

(d) The time limit of each task will be 600 seconds. If your program fails to complete on time, the system will show a "TimeoutException" error.

(e) Please make sure that you submit only what's necessary for the agent to run.

(f) Please upload everything inside the template including MANIFEST.in