National University of Singapore
School of Computing

Semester 2, AY2021-22

CS4246/CS5446 AI Planning and Decision Making

Issued: 24 Mar 2022 **Due: 20 April 2022 at 23:59**

Assignment 3

Information

- This assignment is worth 10 marks out of 100. The score you obtain will be scaled to 10.
- This assignment should be submitted individually.

On collaboration & information source

- You are allowed to discuss solution ideas on the forums. However, you *must write up the solutions independently*.
- It is considered as plagiarism if the solution write-up is highly similar to other students write-ups or to other sources.
- If you obtained the solution through research, e.g. through the web, state your source in the answer rationale in the quiz.

Submission

- The written part of the assignment should be submitted via the corresponding LumiNUS quiz.
- Things to note:
  - You will have only one attempt at the quiz on LumiNUS.
  - There is no time limit for the attempt. However, the quiz closes on the designated deadline.
  - Use the *Rationale* field in the LumiNUS quiz to show your working, or paste the well-formatted answer, or both!
- For the programming part, please **upload the zip to the aiVLE evaluation server**.
- Additionally, the zip file containing your solution to the programming assignment should be submitted on LumiNUS
- ***Late submission:***
  - **Hard deadline for written part** is April 20, 2022, no late submissions are allowed. There will not be any penalty for submissions until April 20, 2022
  - For the **programming component**, you will incur a late penalty of 50% of your score for submissions after April 20, 2022.

– Hard deadline for programming component: April 22, 2022 @ 23:59. No submission will be accepted after that.

Written questions (14 marks)

1. **[6 marks] POMDP Modeling**

   Kermit the Frog is stuck at a corner of a lotus pond. There are only two lotus leaves at the corner, one Red and one Green, and staying on Green long enough (to bend the stem to open a path) will lead him out of the corner, which is desirable. But poor Kermit is color-blind and cannot tell exactly which leaf he is on; he has also hurt his hind legs. When Kermit jumps, there is only a 60% chance that he will land on the other leaf. Also, Kermit can estimate his position (on Red or Green) from the texture of the leaf with 80% accuracy.

   Stating any additional assumptions you may need, formulate Kermit's problem as a sequential decision problem. Carefully list out all the components of the model.

2. **[4 marks] POMDP Belief Filtering**

   Consider a problem with two states s_1 and s_2, current belief $b(s_1) = 0.6$ and $b(s_2) = 0.4$. For action a. The transition probabilities are as follows:

   - $P(s_1|s_1, a) = 0.3$ and $P(s_2|s_1, a) = 0.7$
   - $P(s_2|s_2, a) = 0.4$ and $P(s_1|s_2, a) = 0.6$

   The observation probabilities are as follows:

   (a) The observation o_2 reports that the agent is in state s_2 but agent is in state s_1 with 70% probability.

   (b) The observation o_2 reports that the agent is in state s_2 and agent is in state s_2 with 30% probability.

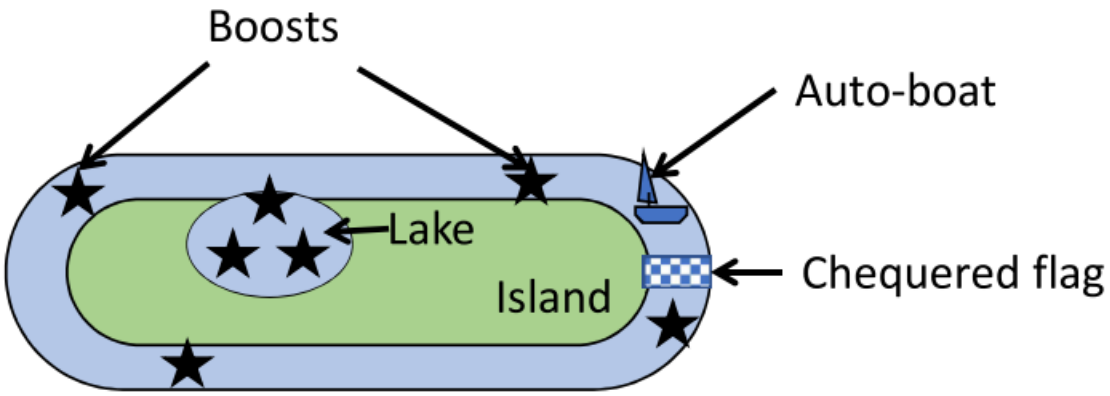
   Assume that the evidence received is $e = o_2$.

   Fill in the blanks: $b'(s_1) =$ ____
   $b'(s_2) =$ ____

   Find the values rounded to 3 decimal places. Show your working in the rationale field. Answers with no rationale will get only half the marks.

3. **[4 marks] Reinforcement learning – Modeling**

   You are building an RL agent to control the racing auto-boat in a simple video game. The boat starts from the chequered flag, and is supposed to circle around the water track (shown in blue). When auto-boat reaches the chequered flag it completes the race. There is a small lake (blue oval) on the island (the green patch) that is connected to the outer track via a small



opening. The reward for reaching the flag is R. There are additional entities that spawn at frequent, regular intervals which boost the auto-boat's power (denoted by the stars; positions are fixed). Auto-boat gets a positive reinforcement $B \approx R/20$ on knocking out the stars. Auto-boat can move forward and turn left or right by 15 degrees. Assume full observability.

Explain what could be a *potential problem* with this setup.

Programming Assignment (aiVLE submission)

In Assignment 2, 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.

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 docker image 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.

Programming problem: Deep Q Learning

[6 marks]

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 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) 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 instruction below:

1. At the beginning of the notebook, put a code cell to install the dependencies, e.g., torch, numpy, gym-grid-driving.
2. Open the agent/__init__.py and copy the content into a new cell in the Colab. Place the code before "if __name__ == '__main__' " in one cell, and the code after on another cell. The **argparse** will not work and should be removed. You also have to specify the **model_path** manually because **script_path** will no longer work and should also be removed.
3. To run your agent, you just have to execute the code blocks sequentially from top to bottom.

An example Colab: HW3 Colab Example.

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.

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:

1. Please make sure that everything that you need for the agent to run resides in the agent/ folder (including the model).
2. Please do not print anything to the console, as it might interfere with the grading script.
3. There will be **2 daily submission** limit for the task. Please use it wisely.
4. The time limit of the task will be 600 seconds. If your program fails to complete on time, the system will show a "TimeoutException" error.
5. Please make sure that you submit only what's necessary for the agent to run.
6. Please upload everything inside the template including MANIFEST.in