National University of Singapore

CS4246/CS5446 AI Planning and Decision Making

Homework 3

Submission Deadline: **April 17, 2020 @ 23:59**

Instructions

Instructions differ from earlier homeworks. Please pay attention

- In this homework, the written questions should be submitted individually.
- There are two programming tasks, you need to work on the programming tasks as a team of two people.
- The report for the programming part should be submitted as a team.

On Collaboration (Written questions)

- While you may collaborate, you *must write up the solution yourself*.
- It is okay for the solution ideas to come from discussions. However, it is considered as plagiarism if the solution write-up is highly similar to your collaborator's write-up or to other sources.

Submission

- Your solution for both the written part and programming assignment should be submitted on LumiNUS.
- For the programming assignment, code should be uploaded to aiVLE evaluation server **in addition to LumiNUS.**
- ***Late submission:*** You will incur a late penalty of 20% of your score for the late submissions.
  - Late submissions will be computed for written and programming parts separately.
  - No submission will be accepted after April 20, 2020 @ 23:59.
- Please refer to the submission instructions for both parts in the relevant sections below.

Marks distribution

- Written questions: 7 marks
- Programming assignment, Task 1: 4 marks
- Programming assignment, Task 2: 14 marks
- Report describing Task 2: 5 marks

Written questions

Instructions for written questions

Please write the following at the top of your files (for the written assignment)

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

Submission

- For the written part, please type out your answers and submit a PDF file.
- Name the file `[YourMatricNumber]_Homework3-Written.pdf`

Questions

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?

Programming Assignment

Instructions for programming assignment

- The programming assignment must be done as a team of **two students**.
- You are required to submit the programming assignment to both LumiNUS and the aiVLE server.
- In addition, you need to submit a report to the LumiNUS workbin containing:
  - Team Members' names and student ID (one or two students)
  - Description (up to two pages) on how you solved the task describing your implementation (including the parameters and their values, if any) and detailing your observations (what worked, what did not work, how good or how bad was your trials etc.,).

Submission

- Please follow the instructions in Appendix 1 to prepare your submission.
- Submit the same zip files to both LumiNUS and aiVLE server.

Introduction

In the second homework, we 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.

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 "cs4246/base" which you should have downloaded in the previous homework. 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 1: Deep Q Learning

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.

Template

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[1]).

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 Appendix 2: Compute Resources) using GPU. If you are training on CPU, it might take around 30 minutes to an hour depending on 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: 4 marks,
- Else: 0 marks.

[1] https://arxiv.org/abs/1504.08083

Lane	# Cars	Min Speed	Max Speed
1	7	-1	-2
2	8	-1	-2
3	6	-1	-1
4	6	-1	-3
5	7	-1	-2
6	8	-1	-2
7	6	-2	-3
8	7	-1	-1
9	6	-1	-2
10	8	-2	-2

Table 1: Task 2 MDP.

Task 2: Solving Large Stochastic Environments

Problem Definition

In this task, your agent must be able to solve a large stochastic environment with the dimension 50×10, which is 10 lanes with each having a width of 50. Each lane will have 6-8 cars with speed ranging from 1 to 3 cells per time step. The environment is stochastic, which means that other cars will have a non-deterministic speed. Agent's car will remains deterministic. At every time step, the car will move at a different speed, uniformly sampled within the range of minimum and maximum speed of the lane. The detail of the MDP is given on Table 1.

You can also get the detail of the MDP from **agent/env.py**. To modify or understand the configuration of an environment, you might want to read the documentation of gym-grid-driving. If required, you can go through the IPython Notebook file on Google Colab here. All information to build an MDP model is available in the notebook.

Rules

1. **You are free to design your agent**, you may use the methods introduced in the earlier homeworks, or any other method, as long as you use the agent template that is provided.
2. **Any approach that involves handcrafting policies is not allowed.** We will check and judge your submissions and award **0 mark** if we believe that you violate this rule.

Relationship to the Previous Homework

The task is similar to the tasks in Homework 2. In Homework 2, you are tasked to build an agent that can solve non-deterministic environment. However, it only requires you to solve a very small sized problems. The value iteration does not scale well, whereas Monte Carlo Tree Search, albeit able to scale, is likely too slow to work well under the time constraints. In the Homework 1, the agent can actually solve similar sized problems to this one. However, it can only solve deterministic problems. In this homework, you are introduced to deep Q-learning, which essentially introduces

function approximation in the policy, allowing it to scale better (if trained appropriately). However, training deep reinforcement learning agent is often a difficult task, due to many issues such as convergence, sensitivity, and exploration.

Constraints

Due to its stochastic nature, the agent will be evaluated on 600 different environment configurations (i.e., initial car position) of the same size and details (e.g., number of cars, lane speed, etc). Your agent should be able to run on all test cases within 600 seconds. Half the test cases are restricted to a horizon of 40 (must reach the goal in 40 steps to be successful) while the remaining half have a horizon of 50; hence it is desirable to get to the goal within 40 steps but taking longer will still be rewarded half the time.

Goal

Your goal in this task is to build an agent that meet the following criterion:

- Agent should be be able to handle stochasticity.
- Agent has access to the state in the form of 3D tensor.
- In the testing server, agent will have no access to environment's MDP, or its simulator. During training, you can access everything.
- Agent should be optimized to get to the goal in less than 40 steps.
- Its runtime should be reasonably fast since it's going to be run on 600 different environments within set time limit (600 seconds).

You can use the knowledge that you get from doing the past homeworks, relating many concepts taught in the lectures.

Agent

We provide agent template that has a code snippet that allows you to interact with the environment in the server. The use of agent template is required. Similar to the previous task and homeworks, the main entry point of the agent is located in **`agent/__init__.py`**. The other notable files inside the agent/ folder are **`env.py`** which provides the definition of the example test case and **`models.py`** containing function approximations in the form of neural networks. **`env.py`** describes the MDP.

Implementing an Agent

Your agent definition is located in the main entry point **`agent/__init__.py`**. In there, you'll find a definition of an example random agent (defined as *ExampleAgent* class). In the example, the agent does nothing except outputting random action, disregarding the state that it's currently in. Your job is to improve the performance of the agent by making it doing something more sensible. To do that, you'll need to implement the following functions properly:

- **`__init__` (optional):** This function will be called to prepare your agent, once on each test case. For example, if you have agents that handle different test cases differently, then you want to load the appropriate agent. Another example might be if you use function approximation (e.g., neural networks) as a policy, then you can load the appropriate model here.
- **`initialize` (optional):** This function will be called once every run on each test case. In this function, you'll receive:
  1. The path to the fast_downward solver.
  2. The admissible speed range of the agent.
  3. The discount factor used for the task.

  For example, if you are using PDDL solver or MCTS, you might want to use (1) to define the PDDL problem or as a simulator. (2) and (3) is required if your agent uses PDDL, whereas (4) can be used if your agent solves the MDP of the task.

  If you are using offline solver such as Fast Downward, you'll want to compute the sequence of actions here.
- **`step` (required):** This is the main bulk of your agent. It governs how your agent behaves in a state. It receives state information at every step, and is required to output an action. For an online planner such as MCTS, the search will have to be computed in here to output the best action to take.

  If you are using function approximator, then you should compute the output in here as well. In the Homework 3 we use neural networks as a function approximator for the agent.

  If you are using offline solver, or precompute the sequence of actions in the **`initialize`** function, then you have to output the decision sequentially in here.
- **`update` (optional):** This function will be called once after every environment step with the action given by the **step** function. It'll receive the transition information from the environment, e.g., state, action, reward, next state, and whether the task is done or not. This update function is important if your policy is defined as such it depends on the history. If that's the case, then you wants to save the transition (or subset of it) to be used during in **`step`** function for taking the next action.

For more details, you can read through the commented codes in the file.

Testing an Agent

You can test your agent before submitting by executing: **`python __init__.py`**. By executing the command, your agent will be tested against the predefined set of test cases. You can see how your agent performs as well as how long it takes for your agent to completes execution. Do note that local runtime might be different from the server runtime. The test program also gives remarks on how fast your program is. Please aim to get "fast" or "safe" remarks before submitting. In the case where you still fail after doing so, please aim for faster time, by either making your model to be simpler or through some performance optimization.

Function Approximation

If you decided to use function approximation in the form of neural networks as your agent, you may use the architectures that we have defined in **`agent/models.py`**. We have verified that the model that is given (and its smaller variations) will be able to be run 600 times within the time limit as well as meet the size requirement.

The model is written using PyTorch. If you are not familiar with deep (reinforcement) learning, you might want to take a look at PyTorch 60-minutes deep learning tutorial and DQN tutorial with PyTorch.

Dependencies

You can only use the following programs/libraries as your dependencies.

- Fast Downward solver
- Numpy (`np.dot` and `np.matmul` are not supported)
- OpenAI Gym Environment
- PyTorch

Internet connection will be disabled during program execution. Any attempt made to download dependencies will fail and therefore will crash the execution. Please monitor wiki/Known-Issues for any known issues with some dependencies.

Report

The report is worth 5 marks. Page length limit: 2 A4 pages, single column format. Font size: between 10-12 point font.

In the report, clearly indicate the Full name (as it appears on your Matric card) and Student number (AxxxxxxxZ) of both the team members. Failing which, we may not be able to track and assign marks to you.

In the report you must describe your implementation. You can include the following details. Do note that it is not mandatory to include all these details, and this is not an exhaustive list. Include anything that you think is relevant.

- Use this opportunity to include the details of the parameters used, what specific configuration/values did you use, if any.
- Any particular "trick" that worked well for you. (or that did not work!)
- What else did you try before arriving at this solution?
- Intuition/explanation/observations of why did the other methods fail?
- Any graphs you could plot comparing various methods you tried etc.,

Grade breakdown for programming assignment

As the problem is non-deterministic, 600 runs will be done with your policy. Marks are awarded for the agent performance as follows:

- At least 85% success: 14 marks.

- At least 60% but below 85% success: 10 marks
- At least 40% but below 60% success: 6 marks
- At least 20% but below 40% success: 2 marks
- Below 20% success: 0 mark

Both team members can submit the files independently; we will consider the best score among the submissions (after the deadline) for grading.

Appendix 1: Submission Instructions

Please follow the instructions listed here. You have to make 2 separate submissions for the two tasks, which are zipped agents with the correct submission format. Please make sure to read the frequently asked questions to avoid problems.

Task 1

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

Task 2

- Please use the agent template as a skeleton for your agent.
- The maximum size of the zipped agent is 30MB

Extra Notes:

1. Please make sure that everything that you need for the agent to run resides in the agent/ folder.
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 each task. Please use it wisely.
4. 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.
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`
7. Please use relative import to import module(s) to `__init__.py`. For example, to import `agent/models.py` you can write: "`from .models import *`"

Appendix 2: 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:

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

   !pip install torch numpy git+https://github.com/cs4246/gym-grid-driving.git

2. 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.

3. **Task 2:** Put everything inside the `agent/env.py` and `agent/models.py` into two different code blocks and place them before the block 1 and 2.

4. 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.

Appendix 3: Scaling Up

Getting planning and reinforcement learning methods to scale up often involves a substantial amount of trial and error. Small problems can usually be solved reliably by known methods, e.g. small MDPs can be solved reliably using value iteration if the model is known or using Q-learning for unknown models. When the problem is large, you often have to try various different methods or combinations of methods; in such cases, knowing what has worked for other problems can be helpful in reducing the amount of trial and error that you need to do. For Task 2 of the homework, your policy need to run 600 times in 600 seconds. Online search methods are unlikely to do well without enough time to run, hence you likely need to use some form of function approximation. However, the online search methods may still be useful in some combination with the function approximation. We tell some stories that may or may not be useful for helping you solve Task 2.

One way to use an online planner together with function approximation is to use the online planner as an expert for imitation learning. You can learn about imitation learning in the CS5446 lecture on Week 8 (or the webcast of that). For example, MCTS has been used as an expert to train a function approximator for Atari games[2].

Deterministic problems are usually easier to scale than stochastic problems. When the International Probabilistic Planning Competition was introduced in 2004, most entries were based on MDPs. However, the winner was surprisingly a deterministic planner, with replanning at every time step[3].

AlphaGo used imitation learning from expert games to initialize the function approximator before using reinforcement learning to improve the value function and policy[4]. The learned policy and value function are finally used with MCTS to beat Lee Sedol, one of the best Go players. AlphaGo Zero, on the other hand, combines MCTS with policy and value function learning to iteratively improve the policy and value function[5].

Why not just use function approximation directly with reinforcement learning from scratch? For example, you are asked to use Deep-Q learning to directly solve a smaller version of the problem in Task 1. This is unlikely to work well (at least it did not when we tried it . . .). The exploration problem is often difficult, resulting in very slow learning using just reinforcement learning, compared to imitation learning or combination of different methods. For Task 2, the goal is at least 50 steps away from the agent, and the agent has no indication on which direction is good to explore as no reward is provided until the goal is reached.

If you have some prior knowledge of the goal or value function, reward shaping [6] can be used to modify the reward signal to provide information that encourages the reinforcement learning agent to explore useful parts of the state space. Distance and subgoal based reward shaping are often used, and under appropriate conditions reward shaping can maintain the optimal policy. Other

[2] https://papers.nips.cc/paper/5421-deep-learning-for-real-time-atari-game-play-using-offline-monte-carlo-tree-search-planning.pdf

[3] https://www.aaai.org/Papers/ICAPS/2007/ICAPS07-045.pdf

[4] https://www.nature.com/articles/nature16961

[5] https://deepmind.com/blog/article/alphago-zero-starting-scratch

[6] https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/NgHaradaRussell-shaping-ICML1999.pdf

ideas for using prior knowledge for speeding up learning include the use a curriculum[7] where easier instances are presented first and made more difficult during learning.

Each time you try a method or combination of methods, it may take hours to run reinforcement learning (or to generate training examples). You should think before trying a method and perhaps try small experiments (e.g. to check the quality/runtime of specific methods of generating training examples).

[7] http://www.machinelearning.org/archive/icml2009/papers/119.pdf