AI Planning and Decision Making
National University of Singapore
Prof Lee Wee Sun

2020
CS4246/CS5446

Mini Project

Submission Deadline: **Sunday 15 November 2020 11.59pm**.

The project can either be done individually or in a team of 2 students. You are required to submit one task for evaluation on the server. In addition to that, you need to submit a half page report to the LumiNUS workbin containing

- Team Members' names and student ID (one or two students)
- A one paragraph description (up to half a page) of how you solved the second task.

Only one report is required for each team (whether it has one or two members).

Grading: There is only one problem to do for the project. Grading will be based on the score achieved on the evaluation server (scoring scheme to be decided after the deadline based on the performance of the class).

If done in a team, the best score of both team members (after the deadline) will be used for each task. If the paragraph description describing your method for the second task is missing or unsatisfactory (cannot be understood), one mark will be deducted. If your half page report is not submitted, one mark will be deducted.

Late Policy: Late submission will not be accepted. If your report is not received on time, your scores from the server will be used with a one mark deduction.

Prizes: As part of the project, we will have a competition. There is a leaderboard on the evaluation server. The top scoring team at the end of the project will receive a $100 prize and the runner-up will receive a $50 prize.

Task Description

By now, you should be familiar with the gym_grid_environment. Go through the IPython Notebook file on Google Colab here. All information to build an MDP model is available in the notebook. If present, stochasticity can only be in the speeds of other cars.

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. The environment is stochastic, which means that other cars will have a non-deterministic speed. 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. Unlike Homework 3, the minimum and maximum speed of each lane is not known in advance, i.e. it can change between trials, although within each trial it is fixed. This is one of the main issues: the model is not Markov if the observation is used directly as the state. Thinking about how to make your state representation closer to being Markov could be one way to improve performance. Another main issue, if learning is used, is scaling up to the large problem size. See Appendix 2 to see descriptions of methods of scaling up in practice. We suggest you consider reward shaping as a way to get started but the other methods may also be useful.

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.

Rules

- **You are free to design your agent**, with the agent template that is provided. There is no restriction on the method you use to construct the agent, whether you use search, learning, or a hand-crafted policy. However, the maximum size of the zipped agent is 20MB.

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.

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 algorithm requires the model to be known; in the project the model is not known at the start of each trial and you only have 40 or 50 steps to complete the trial. Also, value iteration is unlikely to scale to problems of this size. 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.

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 can only have access to an observation, not a state, in the form of image (tensor).
- In the testing server, agent will have no access to environment's MDP, or its simulator. During training, you can access everything.
- The environment will change at the start of each trial.
- 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 should use the knowledge that you get from doing the past homeworks, relating many concepts taught in the lectures.

Docker[1]

Please update your **cs4246/base** docker image by following the instruction bellow:

1. Remove `cs4246/base` image from the docker container with command :
   `docker rmi cs4246/base`
2. Pull the updated docker image with the required dependencies with command :
   `docker pull cs4246/base`.

Other required docker commands can be found here. If you are not using Docker, then you must update the **gym-grid-driving** package in your environment.

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 testcases for the task and **models.py** containing function approximations in the form of neural networks. **env.py** describes the MDP of the task.

[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.

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 before the evaluation. 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.
- **reset (optional):** This function will be called once after every `env.reset` (executed once before each trial/episode). This function can be useful if you want to reset the internal state of the agent before every episode. For example, if your agent policy depends on the history of the environment state, then you might want to remove the history of the previous episode inside this function.
- **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 testcases. 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 reasonable 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. You can also see the example implementations of deep Q-learning in the homework 3.

Dependencies

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

- Fast Downward solver
- Numpy (*np.dot* and *np.matmul* is 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.

Appendix 1: Extra Information

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 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.
3. Put everything inside the agent/env.py and agent/models.py into two different cell and place them right after the dependencies cell.
4. To run your agent, you just have to execute the code blocks sequentially from top to bottom.

An example Colab from HW3: 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.

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 submit zipped agents with the correct submission format. Please make sure to read the frequently asked questions to avoid problems.

- Please use the agent template as a skeleton for your agent.
- The maximum size of the zipped agent is 20MB

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

Appendix 2: 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. 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.

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 Homework 3. 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. In the project, 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 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[2]. As an example, in learning to play DOTA 2, a simple reward would be to give a positive score for winning the game and a negative score for losing the game. However, OpenAI uses domain knowledge about DOTA 2 to provide useful intermediate reward during the game [3]. Read more about OpenAI's agent at https://openai.com/blog/openai-five/.

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 supplementary lecture on function approximation and machine learning.

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[4].

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

[2]https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/NgHaradaRussell-shaping-ICML1999.pdf
[3]https://gist.github.com/dfarhi/66ec9d760ae0c49a5c492c9fae93984a
[4]https://www.aaai.org/Papers/ICAPS/2007/ICAPS07-045.pdf
[5]https://www.nature.com/articles/nature16961

Zero, on the other hand, combines MCTS with policy and value function learning to iteratively improve the policy and value function[6].

Other 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).

[6]https://deepmind.com/blog/article/alphago-zero-starting-scratch
[7]http://www.machinelearning.org/archive/icml2009/papers/119.pdf