Robot Learning

Coursework 2

Thursday 9th February 2023

(Updated on Tuesday 21st February 2023)

Edward Johns

Introduction

In this coursework, you will design and implement your own robot learning method, which you will write in *robot.py*. Then, we will evaluate your method, by testing your *robot.py* on random environments using our own machine, and scoring it based on how well the robot performs. There are no more lab exercises being released for this module, so you are free to work on this coursework during the remaining lab sessions.

Coursework submission. You should submit two files for this coursework, both to Scientia. First, is your *robot.py* file. Second, is a single page PDF titled *robot-learning-coursework-2-description.pdf*, where you will briefly describe your implementation. You may wish to use the template provided on Scientia, titled *robot-learning-coursework-2-description-template.docx*. Both *robot.py* and *robot-learning-coursework-2-description.pdf* will count towards your coursework grade. Your grade for *robot.py* will contribute to 80% of your grade for this coursework, and your grade for *robot-learning-coursework-2-description.pdf* will contribute to 20% of your grade for this coursework. And this coursework contributes to 15% of the entire module.

Submission deadline: Wednesday 1st March at 7pm.

Competition. Your robot will also be entered into a live competition during our final lecture, racing against the robots from other students! This competition is voluntary, unassessed, and just for fun, but you will not need to do any further work for this. I will provide more information on the competition in due course.

The code

There are the following five Python files provided: *robot-learning.py*, *robot.py*, *environment.py*, *graphics.py*, and *model.py*. And you will need to install the following packages, with the version numbers in brackets: *pyglet* (version 2.0.4), *numpy* (version 1.24.2), *scikit-image* (version 0.19.3), *scipy* (version 1.10.0), and *perlin-noise* (version 1.12). Installing these version numbers will hopefully enable the code to run on our machine without any issues, and these are the latest versions, so should be installed by default when

using *pip install*. We will be using Python 3.10 on our machine, but other versions of Python 3 should also be fine.

The code provided is very similar to the code that was provided for Lab Exercise 4. All your implementation should go in the file *robot.py*, and you should not modify any of the other files. Or if you do, for debugging purposes, then make sure you revert these files back to their original content when you do the final testing of your *robot.py*. You may wish to directly copy over some of your code from *robot.py* from Lab Exercise 4, if you made some good progress. However, the code for Lab Exercise 4 was a simplified version of the code for this coursework, to help you gain familiarity with this new environment. There are three main differences between the coursework code, and the code for the previous lab exercises.

The first difference (which was also present in Lab Exercise 4), is that *Robot.model* emulates learning a model, but bypasses the complexities of actually doing the learning. The model has uncertainty, and as the robot physically moves around the environment, this uncertainty decreases for those states and actions nearby the states and actions that the robot has physically visited. As such, physically exploring a certain region of the environment will result in the uncertainty decreasing in that region, which emulates the fact that more data from that region would be available when training the model. *Robot.model.predict()* then allows the robot to make a prediction using the learned model, and also returns the uncertainty of that prediction. Note that the model is updated with each call to *Robot.process_transition()*, and that call is made in *robot-learning.py*, but you should not call *Robot.process_transition()* at any point in *robot.py*. For example, you should not call *Robot.process_transition()* when you are sampling actions during planning, and you should not just generate lots of "fake" transitions and send them to *Robot.process_transition()* to quickly update the model. The model should only be updated with physical actions the robot actually executes. Make sure that you fully understand the difference between these physical actions (which are returned from *Robot.next_action_learning()* and *Robot.next_action_testing()*), and simulated/planned actions (which are calculated during planning by predicting the next state with *Robot.model.predict()*).

The second difference is that the environment dynamics is more complex now, and an action of [0.05, 0.05] no longer necessarily moves the robot upwards-right. Instead, in *environment.py,* as well as the map of the terrain, there is a map of the dynamics, which determines the direction that the robot moves in given its action. You can visualise this by setting *uncertainty* = 0 in *model.py,* then setting draw_*model = True* in *robot-learning.py*, and then running *robot-learning.py*. You will see that an action of [0.05, 0.05] results in the robot moving in a different direction depending on the robot's state. This reflects real-world dynamics, which are unknown to begin with, meaning that we cannot just manually calculate the optimal path between the robot and the goal using an analytical method; the dynamics first needs to be learned. Remember to revert *model.py* to its original content after the above modification.

The third difference is that the state of the robot is now stored in *environment.py*, not in *robot.py*. Therefore, given that the only modifications you can make are to *robot.py,* you cannot simply set the robot to be in any arbitrary state in the environment. In this new setup, there are two ways to change the robot's state. The first is through *environment.step()*, which is called at every timestep in the main loop in *robot-learning.py*. The second is by resetting the environment, such that that the robot returns to its initial state and a new episode begins. However, if you choose to do this, a penalty of 20 seconds is added. This is to emulate the

fact that, in reality, resetting a robot back to its initial state can be very laborious and should only be done sparingly.

Main program flow. The main program flow in *robot-learning.py* is as follows. There are two phases: training and testing. The program begins in the training phase, which continues until *TRAINING_TIME_LIMIT = 600* seconds have elapsed. During the training phase, the robot should explore the environment and attempt to reduce the uncertainty of its model, so that there is a good model to use during the testing phase. During the testing phase, the robot should then attempt to reach the goal, within the time limit of *TESTING_TIME_LIMIT = 200* seconds. As discussed in Lab Exercise 4, time elapses due to both "cpu time" (e.g. planning) and "action time" (e.g. physical steps).

What to implement in robot.py

To begin, for training, try a simple strategy which starts with a period of random exploration, followed by a period of planning towards the goal, using the model You should observe that the model's uncertainty reduces and enables better planning as time goes on. Then for testing, begin by implementing a planning algorithm using the model. You are free to determine your own design, such as the exploration and planning strategies, as well as parameters such as the episode length, planning horizon, and number of samples during planning.

Ideas you may wish to explore:

- Trading off the amount of physical movement with the amount of planning. Is it better to have a small amount of physical movement (which takes lots of time but enables the robot to learn the model better), or is it better to have lots of planning computation (which takes much less time but is only as good as the quality of the model)?
- Trading off the amount of physical exploration with the amount of physical exploitation. During the learning phase, is it better to take random actions to explore the environment, or to take the best actions according to the planning?
- What is the best reward function? Should it be dense or sparse?
- Should the robot ever be reset to its initial state, and if so, when should this be done?
- Parameters such as the episode length and the planning horizon could be dynamic, and could change throughout training or testing.
- You could bias the exploration if you like, such as manually choosing specific actions for the initial exploration, using knowledge of the goal state. You can consider this to be the equivalent of providing a demonstration in imitation learning. However, because the model is unknown, then even if you know where the goal state is, it is not obvious how to provide a good demonstration action, and you will probably find that using the learned model is a better strategy.
- You could use the model's uncertainty, which is returned when using *Robot.model.predict()*, to determine how much to "trust" a particular planned path.
- To solve the task, the robot needs to move within 0.03 of the goal, so it would be wise to ensure that the model is accurate around the goal region.

Things that you are *<u>are allowed</u>* to do:

- Use knowledge of where the goal state is, such as to bias exploration or planning, and of course, in your reward function.
- Use knowledge that there is an "obstacle". In other words, if you wish to bias your exploration or planning based by attempting to estimate where the obstacle is, then this is fine.
- You may have noticed that the bottom-left of the obstacle is always at (0.2, 0.2). You may use this knowledge during exploration or planning, if you like.
- Initialise the planning for testing by using the planning data from training, rather than planning from scratch during testing.
- Change various parameters throughout training or testing, e.g. the planning horizon, the episode length, the reward function, etc.
- Design a structured reward function, such as one which includes waypoints.

Things that you are *<u>not allowed</u>* to do: (violating this could result in a reduced grade)

- Use the GPU, or use multi-threading.
- Import any packages or modules into *robot.py* other than the following: *numpy, scipy*, *time*, and *sys.*
- Take code from other sources (e.g. other students, or online sources) and paste them directly into your code. (However, you may re-use code from the previous lab exercises).
- Call *Robot.process_transition()* at any point in *robot.py*. *Robot.process_transition()* is used to update the model with physical actions the robot takes, and should only be called from *robot-learning.py*.
- Access the true dynamics of the environment (e.g. by importing *environment.py* into *robot.py*, or by calling *Robot.model.environment.dynamics()*). To get access to the (learned) dynamics, you should only use *Robot.model.predict()*.
- Place the robot at a specific state in the environment, e.g. during resets. Besides, this is not actually possible with this code.
- Use "privileged" knowledge about the way that the dynamics is modelled. For example, by inspecting the code, you can see that each cell has a number which rotates the action by a certain angle. You are not allowed to use this knowledge, e.g. by storing that angle for each location and then applying it to all other actions at that location. (Remember that in reality, there would not be any code where you can "inspect" the form of the dynamics function; the dynamics has to be learned). If you want to know where the robot will end up after taking an action, then you must use *Robot.model.predict()*.

Evaluation and grading of *robot.py*

Evaluation. When we evaluate your *robot.py* file that you submit to Scientia, we will run the exact same code that is in the provided *robot-learning.py*. We will set *draw_terrain = True*,

draw_paths = False, and *draw_model = False.* We will evaluate on a machine with an Intel Core i7-7820X CPU and 16 GB of RAM available. We will evaluate your *robot.py* on five random environments, each of which will be just a random instance of the *Environment* class in the code provided. All students will be evaluated on the same five environments.

For each environment, we will run the training phase for TRAINING_TIME_LIMIT = 600 seconds. Then, we will run the testing phase for TESTING_TIME_LIMIT = 200 seconds. As can be seen in *robot-learning.py*, these times include the "cpu time" (the time taken for planning), plus the "action time" (the time taken for physical actions and resets). During the testing phase, there are two potential outcomes. Either the robot will reach the goal within the time limit, in which case the time taken to reach the goal will be recorded. Or, the robot will not reach the goal within the time limit, in which case the closest distance to the goal during the testing phase will be recorded.

Grading. Grading will be done relative to the results of three baselines that we have created. As such, your grade will be normalised based on how difficult the particular environment is. *Baseline 1* simply executes random actions for every step during testing. Baseline *2* is a basic cross entropy method, which can sometimes (but not always) enable the robot to reach the goal during testing, but the path is often very indirect and not very smooth. *Baseline 3* is a method which effectively finds the globally optimal path to the goal, and is calculated using brute force search, which takes a very long time to compute (well over the time limit you have!).

To determine whether your method has "beaten" a baseline, we will do the following. First, we will run the baseline, and see if the robot reached the goal. If the baseline did reach the goal, then your robot needs to reach the goal more quickly than the baseline did, to "beat" the baseline. And your score is based on how much more quickly it reached the goal (using the *time_elapsed* variable). If the baseline did not reach the goal, then either your robot needs to reach the goal, or it needs to get closer to the goal than the baseline did, to "beat" the baseline. And your score is based on how much closer it got to the goal (using the *testing_best_distance* variable). So first, you should design an algorithm that enables the robot to reach the goal, or get close to the goal, in as many environments as possible. Then, once your robot is able to reach the goal reliably, you should continue to improve that algorithm so that the robot reaches the goal as quickly as possible.

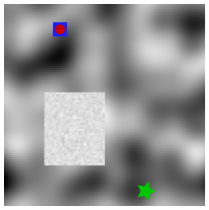
If your *robot.py* performs worse than *Baseline* 1, then you will receive 0%, because you have not implemented anything significant. If your *robot.py* outperforms *Baseline 1*, then you will receive at least 40%. If your *robot.py* outperforms *Baseline 2*, then you will receive at least 60%. And if your *robot.py* reaches the performance of *Baseline 3*, then you will receive 100% (in practice, virtually impossible). To determine the specific grade, we will interpolate between these boundaries based on your score compared to the baselines, as discussed in the previous paragraph. For example, if your *robot.py* performs slightly better than *Baseline 2* but is still very far from the performance of *Baseline 3*, you might score around 70%. The total score will then be the average across these five environments. And your grade for *robot.py* will contribute to 80% of the total grade for this coursework.

Important. When you do your final test of your *robot.*py before submission, make sure that you run your code using exactly the same files provided on Scientia, with only modifications made to *robot.py*. You may wish to re-download the code from Scientia to make sure that any edits you made to the other files are removed. If there are any bugs and your program does not run, you may receive a reduced grade, even if these are tiny bugs, such as typos. We will

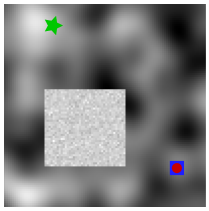
not correct your own bugs or typos. We will run your code exactly as it is in your submission on Scientia, so you should also remove anything that will slow it down, e.g. print statements or unnecessary graphics.

Example scores for Baselines 2 and 3. Below are the scores for Baselines 2 and 3, for four different environments. You may wish to use these to get a sense of how well your own implementation is performing. The random seed for each environment is provided (you can use this in *robot-learning.py*), as well as an image of the environment. These results were averaged over three runs each. Remember that you are not expected to be able to outperform Baseline 3; a solution which performs somewhere between Baseline 2 and Baseline 3 would be a very good outcome.

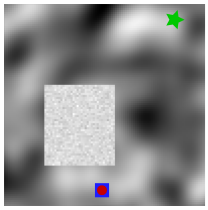
Random seed = 1.
Baseline 2: Reached goal in 187 seconds.
Baseline 3: Reached goal in 43 seconds.



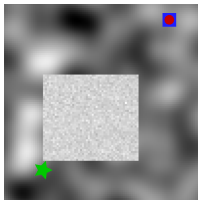
Random seed = 2.
Baseline 2: Reached goal in 169 seconds.
Baseline 3: Reached goal in 35 seconds.



Random seed = 3.
Baseline 2: Reached goal in 155 seconds.
Baseline 3: Reached goal in 44 seconds.



Random seed = 4.
Baseline 2: Did not reach goal in time. Closest distance = 0.23.
Baseline 3: Reached goal in 55 seconds.



What to write in *robot-learning-coursework-2-description.pdf*

In this document, you should write a brief description of your implementation, of no more than one page in length. This description should include two parts. Firstly, a high-level summary of your overall algorithm, relating your implementation back to topics introduced during the lectures, such as planning, model predictive control ,and model-based reinforcement learning. And secondly, a summary of some interesting findings you had when designing your algorithm, such as which strategies seemed to work better than others, and any ideas you introduced into your implementation which you found particularly interesting.

Evaluation and grading of *robot-learning-coursework-2-description.pdf*

This document does not need to be highly detailed and elaborate, and it will be marked generously. If you can show that you have implemented something reasonable, and that you have tried out a few different ideas, and then written at least half a page, you can easily receive full marks. However, we are looking for implementations which generally follow the methods introduced during the lectures, and so implementations which do not use any of the methods from the lectures may not receive full marks. I anticipate that the majority of students will receive full marks for this document. Your score for *robot-learning-coursework-2-description.pdf* will contribute to 20% of the total grade for this coursework.

Any further questions?

For any other questions, please ask me on EdStem, and I promise I will respond! And I will keep everybody updated on any common issues.

Final reminder. Any changes you make to anything other than *robot.py* will not be present later when we evaluate your *robot.py*. Before you submit, make sure that you test your *robot.py* with exactly the same files provided with this coursework.

I really hope that you enjoy this coursework, and that it helps you to understand how robot learning algorithms can be deployed in practice. I look forward to reading about some of your interesting solutions, and to discussing some of your ideas with you in the lab sessions, and I also very much look forward to the live competition in the final lecture!

End of document