Robot Learning

Lab Exercise 3

Wednesday 1st February 2023

Edward Johns

Welcome to the third lab exercise for the Robot Learning course! By the end of this exercise, you will have implemented some of the methods introduced in Lecture 3. Before starting this exercise, you should have completed Lab Exercise 2. And by following this exercise, you should then be able to complete the tasks in Parts 3 and 4 of Coursework 1. This exercise consists of 2 parts. After Part 1 you will be able to complete Part 3 of Coursework 1, and after Part 2 you will be able to complete Part 4 of Coursework 1.

As with Lab Exercise 2, there are the following four Python files: *robot-learning.py*, *robot.py*, *environment.py*, and *graphics.py*. However, there is also now a fifth Python file: *torch-example.py*. Whilst the overall code structure is similar to that in Lab Exercise 2, there have been some changes to make the code easier to manage, now that you will be implementing different ideas. Therefore, you should begin this exercise using these new provided files, rather than the files from previous exercises.

Preliminaries (~30 minutes)

To begin, set up a new virtual environment for this exercise, and then install the following five Python packages: *numpy*, *pyglet*, *scikit-image*, *torch*, and *matplotlib*. This exercise will use Torch, a deep learning framework, for learning the model. Therefore, the next thing to do is to try running *torch-example.py*. This should train a simple network with supervised learning on some dummy data, and print out the losses during training whilst also showing a graph of the losses. The file *torch-example.py* is not directly used in this exercise, but exists to help you understand how to use Torch to train a neural network. Therefore, if you are not familiar with Torch, I suggest you spend some time looking at *torch-example.py* to understand how it can be used for supervised learning. Note that in this lab exercise, and both the courseworks, you should use your CPU rather than your GPU for training. This is the default behaviour in *torch-example.py*.

Then, try running *robot-learning.py*. You should see the robot randomly moving around the environment. The first thing you may notice is that the environment now has two obstacles, one in the bottom-left and one in the top-right, and that now, the robot is actually able to move right through these obstacles. The obstacle in the bottom-left has a darker shade, representing the fact that this obstacle has a lower terrain than the top-right obstacle, and therefore, has lower impedance when the robot tries to move through it.

Take a look at the code to try to understand what is going on. You can see that there is now a function *Robot.next_action_random()*, which executes random actions for one episode, and then resets the robot to its initial state. There is also a function *Robot.next_action_open_loop()*, which itself calls *Robot.planning_random_shooting()*. In this exercise, to keep the planning simple, you will be using random shooting instead of the cross entropy method. In *robot-learning.py*, if you replace the line *action =*

robot.next_action_random() with *action = robot.next_action_open_loop()*, the robot will plan an action sequence using random shooting, plot its plan in orange, and then physically execute this sequence in an open loop.

Next, look at *Graphics.draw_visualisations()*. Replace the *if 0:* statement with *if 1:*, such that the function *draw_model()* is called. You will see a gride overlaid on the environment, which visualises the robot's model. In each cell, a line is drawn showing, according to the model, the next state (small white circle), if the current state (small white square) is the centre of that cell, and the action is to move the robot diagonally up-right. This action can be changed, and is the second argument in *draw_model()*. The first argument is the robot's model. Note that currently, in *Robot.model()*, the true environment dynamics is used.

Part 1: Model Learning (~60 minutes)

In this part, you will implement a model learning algorithm, and then visualise this learned model.

To begin, change *robot-learning.py* back to its original content, so that the robot takes a random action at every timestep. Then, every time the robot takes a physical action, store this transition data (state, action, next state) in a dataset. Once the robot has collected 5000 transitions (5000 physical timesteps), stop the data collection, and start training on this dataset.

To train on the dataset, you can modify the code in *torch-example.py*. You will need to modify the dimensionality of the inputs and outputs in the network, and you may wish to experiment with the network size and the learning rate. Also, you should divide the dataset into training and validation subsets, and use early stopping to prevent overfitting. Or, if your network does not seem to be overfitting, you can just stop training when the performance appears to saturate.

Once you have done this, you should be able to answer Part 3 of Coursework 1.

Part 2: Closed-loop Planning (~30 minutes)

In this part, now that the robot is able to learn the model, you can move on to implementing closed-loop planning using this learned model.

To begin, modify *robot-learning.py* so that the robot uses *Robot.next_action_open_loop()*, and hence *Robot.planning_random_shooting()*, to choose its next action, using the learned model. Initially, *Robot.model()* uses the true environment dynamics, but in real-world robot learning, this is not directly available. So, you should now replace this with the model learned in Part 1, such that the robot performs planning by simulating actions with the learned model. Then, observe as the robot physically takes actions in the environment, and follows its plan in an open loop.

Next, you will introduce closed-loop control by implementing the model-predictive control algorithm, which was introduced in Lecture 2. First, make a copy of the function *Robot.next_action_open_loop()*, rename it to *Robot.next_action_closed_loop()*, and call this in *robot-learning.py* when computing the robot's next action. Then, modify the contents of *Robot.next_action_closed_loop()*, such that the robot performs planning at every timestep, before each physical action it takes in the environment, rather than just before the first action as with open-loop planning. You should not modify the values of *Robot.planning_horizon*, *Robot.num_samples*, or *Robot.episode_length*.

Once you have done this, you should be able to answer Part 4 of Coursework 1.

Part 3: Model-based Reinforcement Learning (~120 minutes)

In this part, now that the robot is able to learn the model and perform closed-loop planning, you can move on to implementing a full (but simple) model-based reinforcement learning algorithm. This part is optional, and not required for Coursework 1, but implementing it will help you to understand model-based reinforcement learning, and prepare you for Coursework 2.

To do this, you should modify *Robot.next_action_closed_loop()* to follow the "Typical model-based reinforcement learning algorithm" described in Lecture 3. The main difference between this, and the implementation that you already have, is that rather than randomly exploring the environment to collect the data to train the model, the robot will use its planned actions to collect this data. In this way, the data that the model is trained on will be better aligned with the actual states and actions that the robot expects to physically experience in the environment.

For each call of *Robot.next_action_closed_loop()*, i.e. each time the robot takes a physical action, you should make the following changes. First, you should add some noise to the action calculated by the planning. For example, this could be from a normal distribution centred on the planned action. Then, you should record the transition experienced by the robot after taking this action, and add this to a dataset. Next, you should sample a minibatch from the dataset and update the model by training on this minibatch.

You are free to experiment with various design choices in this algorithm, such as the amount of exploration noise, the number of minibatches trained on per timestep (this may be greater than 1, or less than 1), as well as the design choices explored during Part 1 of this exercise (e.g. the network architecture), and other design choices you may not have explored significantly yet (e.g. the reward function).

End of exercise