Robot Learning

Coursework 1

Released on Wednesday 21st January 2026

Submission deadline: **Friday 6th February at 7pm**

Edward Johns

Coursework 1 covers Lectures 1, 2, 3, and 4. Part 1 should be done after completing Tutorials 1 and 2. Parts 2, 3, and 4 should be done after completing Tutorial 3. Part 5 should be done after completing Tutorial 4. The coursework should be done, and submitted, individually. On Scientia, click on "Data File" and you will download a folder containing the code for this coursework, as well as Word and Latex templates for the document you will submit.

Each of the five parts requires you to create a figure, and to answer two questions. You should submit a document containing these, named *coursework_1.pdf*. In the template provided, the figure placeholders show roughly what size the figures should be, and the "Lore ipsum" shows roughly how much text is required. You should not modify the margins or font size in the template. Each part should be on a new page and should occupy a maximum of one page. Please add your CID number to the cover page.

For each part of this coursework, the figure is worth 2 marks, and each of the 2 answers is also worth 2 marks. This coursework has 5 parts in total, and so the total number of marks you can achieve for this coursework is 30.

When marking the figures, we will be looking for evidence that your implementation is generally correct. Where appropriate, you should add basic axis titles (e.g. if the figure is a graph) and a legend (e.g. to show what the different coloured lines represent), but the specific layout, colours, fonts etc., are not important. Figures will vary across different students, due to different implementations, different parameters, and random numbers.

When marking the answers to the questions, we will be looking for evidence that you understand what the correct answer is, and that you are able to write this clearly and concisely. Even if you believe you know the correct answer, you may still lose a mark if you are not able to explain your answer clearly. You may also lose a mark if you have written incorrect statements, even if you have also written correct statements.

Part 1: Cross-Entropy Method

To begin, similar to Tutorials 1 and 2, create a virtual environment for the code in the folder "Coursework-1-Part-1-Starter-Code", and run *robot-learning.py*. Instead of seeing a robot "circle" moving freely around as in Tutorials 1 and 2, now you should see a robot "arm" moving. The task is for the robot's "hand", visualised by the green circle, to reach the goal, visualised by the red circle, whilst avoiding the obstacle. The state is defined by the two angles of the robot's two joints, and the actions are able to rotate those joints. You should inspect the code, which has a similar format to the code in Tutorials 1 and 2, but with several changes due to the new environment and new type of robot.

Now complete *robot.py* so that the robot performs planning with the cross-entropy method. You may wish to use some of the code you wrote for Tutorials 1 and 2, including the provided solution code. For the reward function, you should use the negative Euclidean distance between the path's final state and the goal state. The algorithm should be developed such that the robot regularly reaches within a distance of 0.01 of the goal state, when following the planned path. To achieve this, there are four key parameters which you should choose in order to achieve good performance: the number of iterations, the number of paths per iteration, the length of the paths, and the number of elite paths.

Figure 1: Similar to Figure 1 in Tutorial 2, create a figure which shows one path per iteration. Each path should show the position of the robot's hand when the mean action sequence is executed at the end of the iteration. Each path should be coloured differently, to indicate the iteration number.

Question 1a: Let use define d as the negative Euclidean distance between the path's final state and the goal, which is used as your reward function. An alternative would be to use $2 \times d$ as the reward function. What effect, if any, would this have on the behaviour of your planning algorithm?

Question 1b: Consider a gradient-based planning algorithm used to optimise several waypoints for the robot to pass through, in order to avoid a collision with the obstacle. Each waypoint is defined as the state at some point along the path the robot will follow. In this algorithm, does the robot's dynamics model need to be differentiable in order to optimise these waypoints? Explain your answer.

Part 2: Model Learning

For Parts 2, 3, and 4, you should now use the code in the folder "Coursework-1-Parts-2,3,4-Starter-Code". There are some differences to the code from Part 1. The main difference is that there are now three panels in the window, with the planning visualisation and the model visualisation split up into two panels. This is because, in the environment with the robot arm, the dynamics cannot be visualised as easily, because the state is now the joint angles, rather than the position of the robot. The right panel is currently blank, and you are welcome to draw whatever you like in there to help you debug your code, if necessary. The *Robot* class now has separate lists of visualisation lines for these two panels.

Another difference since Part 1, is that the robot now has access to the *forward_kinematics()* function. This is the robot's own internal dynamics model, and computes the 2D position of the end of the robot's arm, when given the 2 joint angles. So, although the robot needs to learn the dynamics of the environment, it already has access to its own internal dynamics that maps joint angles to 'hand' positions.

If you run *robot-learning.py*, the robot will randomly move around over a sequence of episodes, with a bias towards moving 'right'. You should modify the code in *robot.py* so that, after 20 episodes, the robot then stops and trains a dynamics model on all the data it has collected, until the loss approximately converges. All data can be used for 'training data', there is no need for 'validation data' in this coursework.

Figure 2: Plot a graph showing the training loss. Here, the x-axis should show the number of minibatches the model has been trained on, and the y-axis should show the loss.

Question 2a: If the training loss were to converge to exactly 0.0, would this mean that the robot would have learned a perfect model of the full environment? Explain your answer.

Question 2b: For the dynamics model to learn the data the robot has collected in Part 2, does the model need to be a non-linear function, or can it be a linear function?

Part 3: Model-Based Reinforcement Learning

Next, implement the algorithm for typical model-based reinforcement learning with exploration, described on slide 39 of Lecture 3, using open-loop planning with the cross-entropy method (CEM). Tune your hyper-parameters to get good performance, e.g. the episode length, the amount of exploration noise, the length of the paths in CEM, the number of sampled paths in each iteration of CEM, the number of minibatches trained on per iteration, the size of the neural network, etc. Run the algorithm long enough that the robot can learn the dynamics well enough to reach near the goal. Note that precisely reaching the goal is not required, but the robot should be able to avoid the obstacle and move to an area around the goal.

Figure 3: Plot the loss curve for the training of the dynamics model during one run of your full algorithm, until the robot is able to reach near the goal. Here, the x-axis should show the number of minibatches the model has been trained on, and the y-axis should show the loss.

Question 3a: Is the average loss lower or higher than in Figure 2? Explain why this is.

Question 3b: In model-based reinforcement learning, the robot trains its model on the (state, action, next state) data it collects when moving through the environment. Would there be any advantage or disadvantage of training on the (state, action, next state) data from the paths sampled during the cross-entropy method planning?

Part 4: Closed-Loop Planning

Next, modify your implementation of model-based reinforcement learning so that the robot uses closed-loop planning instead of open-loop planning. In the standard implementation of this, after taking each step in the environment, the robot replans its entire path rather than continuing with the previous plan. However, this will be very computationally expensive for you. Therefore, you should implement a closed-loop planning algorithm where the robot computes a full plan only three times in each episode: once right at the start of the episode, and twice at intermediate points along the episode, evenly spaced out (e.g. just before, and just after, the robot "avoids" the obstacle).

Figure 4: Create some visualisations in the "Planning" window to show the path for both the open-loop and closed-loop methods, after running each algorithm from scratch, and once the performance is good enough for the robot to reach near the goal by the end of an episode. You should annotate the closed-loop path to indicate where the intermediate points are that the robot performs its replanning (if it is easier, you can just annotate this "manually" using an image editor). Figure 4 should then show two windows: one for the open-loop planning, and one for the closed-loop planning, side-by-side.

Question 4a: Does the state representation for the environment in this coursework (the "robot arm") have the same dimensionality, lower dimensionality, or higher dimensionality, than in the environment used in the tutorials (the "point robot")? Explain your answer.

Question 4b: Explain why open-loop planning may be more suitable than closed-loop planning for tasks that involve a robot running at very high speed.

Part 5: Imitation Learning

For Part 5, you should now use the code in the folder "Coursework-1-Part-5-Starter-Code". This is similar to the code for Parts 2, 3, and 4. However, as with Tutorial 4, there is now a 'Demonstrator' class, which enables the robot to obtain demonstrations of the task. There are also two panels for drawing visualisations, if you wish to use them.

Following Tutorial 4, implement a basic behavioural cloning algorithm, and develop it until the robot is able to avoid the obstacle and reach near to the goal when executing the policy. Note that the robot does not need to reach the goal perfectly, nor does it need to stop at the goal.

Figure 5: Train the robot on 5 demonstrations. Then create a figure which shows paths for the 5 demonstrations (one path each, all in the same colour), as well as the path that the robot follows when executing its trained policy (in a different colour to the demonstrations). You should draw this on the "Demonstrations" window, so that the obstacle and goal state are in the background. The pose of the robot in this figure is not important.

Question 5a: By considering the way in which demonstrations are generated in this code, explain why this robot may have some uncertainty in the optimal action, when trained with behavioural cloning using these demonstrations.

Question 5b: In a basic implementation of this exercise, is likely that the robot will overshoot the goal when executing its trained policy. Explain why this is, and suggest how the demonstrations could be modified so that the robot will stop once it reaches the goal.

End of Coursework 1