Assignment 2: Reinforcement Learning

October 2, 2018

1 INTRODUCTION

In the previous exercise you have programmed the ability to classify products based on features of those products. This is important for a robot that needs to detect products in the supermarket. Another important ability is to learn to optimally navigate the supermarket, for example to a product or to cash register. In this exercise you will program a robot to learn such a route based on rewards. The supermarket will be modeled as a labyrinth (further referred to as 'maze'). The target location is a rewarded location in that maze, and the robot will learn to navigate from a fixed start to this rewarded target by exploration and repetition. The algorithm you will use is Reinforcement Learning (RL). You will implement a model-free off-policy version of RL called Q-learning, and you will use a tabular representation for $Q(s,a)$, i.e., a matrix with two dimensions S and A; S being all possible states (the locations in the maze), and A being all actions possible in each state (up, down, left, right). A value in the matrix at location s,a represents the $Q(s,a)$ value, i.e., the value of an action a in state s.

Essentially, the problem you need to solve boils down to learning a SxA matrix of values, each value representing the utility of an s,a pair. Every step your robot takes updates *one* such value, namely the last visited s,a pair. As we use Q-learning, and Q-learning is off-policy (updates are not based on actual actions chosen), we will update this value using the value of the best possible next action $Q(s',a_{max})$. The full update rule is:

$Q(s,a)_{new} = Q(s,a)_{old} + \alpha(r + \gamma Q_{max}(s',a_{max}) - Q(s,a)_{old})$ where $Q_{max}(s',a_{max})$ is the Q value of the best action so far in state s'.

The robots learns by taking random actions, and after each action updating $Q(s,a)$. Action selection (the policy) is something you will have to experiment with. You will implement the method called ϵ-Greedy. This method will take either a random action with a probability of ϵ

(exploration) or a greedy action with a probability of $1-\epsilon$ (exploitation). A random action is selected out of the 4 possible actions. A greedy action is an action with the highest predicted value so far, so it chooses the action with the highest $Q(s,a)$ for the state s we are now in. By varying the parameter ϵ you can thus control how much the robot explores. By varying α and γ you can control the speed of learning and the discount factor respectively.

1.1 ASSIGNMENT

You and your team will have to write a fully functioning Q-learning algorithm and ϵ-Greedy action selection algorithm. You can make use of the Python 3/Java template we have developed for you to focus on the actual algorithms of action selection and learning. The Python template is new this year and contains the same functionality as the Java framework. You will also have to write a report explaining what you did. Please include answers to all the questions from Section 2 in a structured but natural fashion. Where possible, include plots and graphics to support your answers.

You should use the toy_maze.txt provided with this exercise to start with. Your robot's starting location is fixed at the top left of the maze. Your target location is at the bottom right, and has a reward of 10. Start and target locations in the easy_maze.txt are also at the top left and bottom right respectively.

The following paragraph describes the Java framework but as the Python framework is transcribed from this template, the explanation is also applicable to the Python template. The project contains 6 classes in the package: State, Action, Maze, Agent, QLearning and EGreedy. You can load a maze from a file by using the constructor Maze(File file). This will parse the file text and construct a 2D maze initing all necessary data. The Maze class has helper functions to retreive valid actions: ActionList[] getValidActions(Agent r). Further, the Maze class has a method to set the reward for a particular state: void setR(State s), and a method to retrieve the reward for a state: double getR(State s). The QLearning class has a method: double getQ(State s, Action a) to retreive a Q value for a state/action pair and a method: double[] getActionValues(State s, ArrayList<Action> actions) that returns the associated action values for all actions in <actions> in that order. It further has a method: void setQ(State s, Action a, double q) to set the Q value for a state/action pair to the value q (please note this is not updating the value but setting, no learning is done here). The Agent class is simply a class that maintains the current coordinates of the robot (x and y coordinates). To get the current state of the agent, use: State getState(Maze m). It also has a method: void reset(), setting the robot back to its starting location, which is fixed at creation of the agent by yourself, and printing the number of steps since the previous reset to the console. Further it has a method to execute an action you select in a maze: State doAction(Action a, Maze m), returning the new state.

You will find three classes in the package mysolution that you need to work on: the MyQLearning class the MyEGreedy class, and the RunMe class. MyEGreedy inherits all the functionality of EGreedy. MyQLearning inherits all the functionality of QLearning. You should start looking

at RunMe first. You only need to program in these three classes for the assignment. There is no need to make changes to the other classes. Further, we have added TODO's in the classes to help you figure out what to do.

In the RunMe class we have helped you setting up the learning agent and the maze. Have a look at that. Further, you should implement the main agent cycle, i.e., doing actions updating the Q values etc... in the right order.

In the MyQLearning class you should implement the following method:

void updateQ(State s, Action a, double r, State s_next, ArrayList<Action> possibleActions, double alfa, double gamma), to update Q values according to the Q-Learning algorithm, where possibleActions is the list of actions you want to consider in state s_next (for $maxQ_a$ calculation).

In the MyEGreedy class you should implement the following methods: Action getRandomAction(Agent r, Maze m), to select an action at random in State s. Action getBestAction(Agent r, Maze m, QLearning q), to select the best possible action currently known in State s. Action getEGreedyAction(Agent r, Maze m, QLearning q, double epsilon), to select between random or best action selection based on ϵ.

1.2 DELIVERABLES

You will have to create a report to answer the questions and a file containing the classes of the unknown samples. Besides this, you have to clean up your code and deliver it as a compressed archive. The following files should be delivered by replacing XX with your group number(eg. 12,31,03):

- **Group_XX.zip**: Compressed file for submission. Should contain the following:
  - ***Group_XX_report.pdf***: The report as **PDF Document (.pdf)** . The number of pages should not exceed 5.
  - ***Group_XX_code***: The package containing the java code for the three classes you need to program (see below) in a directory equal to the package tudelft.rl.mysolution should be inside this **File Folder**. Easiest is to simply add the folder tudelft.rl.mysolution from the java tempplate project provided to the zipfile.

To speed up the grading process, we want you to deliver your work exactly as outlined above. **If your files do not contain this format, your work will not be graded!**

The deadline for the assignment is **Friday, October 5th at 23:59**.

2 QUESTIONS

The following sections will help you to develop the algorithms. The questions must be answered in the report.

2.1 DEVELOPMENT

This sections helps you to implement the algorithm step by step.

1. First you need to make sure your agent can do something at all, so you implement getRandomAction, getBestAction, and getEGreedyAction in MyEGreedy. How do you ensure that the agent does not have a bias for selecting the same action over and over in getBestAction() if it did not learn something yet?
2. Second, you need to implement the agent's cycle in RunMe (selecting actions, executing them, calling the updateQ etc...). Note that the agent will not be able to learn yet, as you have not implemented updateQ in MyQLearning yet. Remember that you need to reset the agent when it is at it's goal location. Explain your cycle in your report.
3. Implement the stopping criterion to 30000 steps. The agent will stop the run after 30000 steps.
4. Run your agent, it should print to the console a list of numbers, representing the steps taken between resets to its starting position. The period between two resets is called a trial. You should be able to observe that the number of steps the agent takes per trial does not decrease. Make a plot of the average of 10 runs (one run is one launch of RunMe) to show that the agent indeed does not learn. The x-axis in your plot refers to the trial in your runs, the y-axis refers to the average number of steps needed in that trial. Note that you might not have the same number of trials per run if your stopping criterion is not based on number of trials. Make this plot for both mazes given. The toy_maze plot is the easier maze of the two. Explain your plots.
5. Now implement updateQ() in MyQLearning. Set α, γ and ϵ to 0.7, 0.9 and 0.1 respectively. This should be a straightforward implementation of Q-Learning. Explain your method in your report.
6. Now run your agent again on the mazes. You should be able to see that the numbers decrease over time during a run. If you don't first make sure that you allow the robot to try long enough. If it still does not learn, make sure you have set α, γ and ϵ to the default values. Again, make a plot of the average of 10 runs to show that the agent indeed does learn. Make this plot for both mazes given. Explain your plots.

2.2 TRAINING

Here you experiment with action selection and training

1. Play around with the ϵ parameter of your action selection algorithm. Make a couple of plots of 10 run averages (same plots as above) to study the effect of varying epsilon between 0 and 1. Explain your plots.
2. Play around with the learning rate α of your QLearning algorithm. Again make a couple of plots to study the effect of varying α between 0 and 1. Explain your plots.
3. What are the trade-offs between a high and a low ϵ?

2.3 OPTIMIZATION

Here you play around with optimization of γ and ϵ. You do all of these exercises on the toy_maze.

1. Add a second reward, sized 5, at the location (9,0) (so at the top right). Remember to add this location to the condition to reset the agent (it is a second goal). Now run the agent again for a couple of times. What do you observe, and how can you explain this?
2. Invent a way to overcome the previous problem using an adaptive ϵ. Explain your method and show with a plot that the agent now indeed converges to the optimal solution.
3. Can you, using your previous solution, experimentally find a γ for which the optimal solution is in fact to go up and get the smaller reward, rather than going down for the larger? Explain and show plots for the different values of γ you try.