732A96/TDDE15 ADVANCED MACHINE LEARNING

LAB 3: REINFORCEMENT LEARNING

JOSE M. PEÑA
IDA, LINKÖPING UNIVERSITY, SWEDEN

1. INSTRUCTIONS

- **Deadline for individual and group reports**
  See LISAM.
- **What and how to hand in**
  Each student must send a report to LISAM with his/her solutions to the lab. The file should be named FirstName_LastName.pdf. The report must be concise but complete. It should include (i) the code implemented or the calls made to existing functions, (ii) the results of such code or calls, and (iii) explanations for (i) and (ii).

  In addition, students must discuss their lab solutions in a group. Each group must compile a collaborative report that will be used for presentation at the seminar. The report should clearly state the names of the students that participated in its compilation and a short description of how each student contributed to the report. This report should be submitted via LISAM. The group reports are corrected and graded. The individual reports are also checked, but feedback on them will not be given. A student passes the lab if the group report passes the seminar and the individual report has reasonable quality, otherwise the student must complete his/her individual report by correcting the mistakes in it.

  Attendance to the seminar is obligatory. In the seminar, some groups will be responsible for presenting their group reports. Each student in these groups must be prepared to individually present an arbitrary part of the report. The selection of the speakers is done randomly during the seminar. In the seminar, some groups will act as opponents to the reports provided by the presenters. The opponent group should examine the group report of the presenter group before the seminar, and prepare a minimum of three questions, comments and/or improvements. The opponent group will ask these questions during the seminar. Check LISAM for the list of presenter and opponent groups.

2. QUESTIONS

The purpose of the lab is to put in practice some of the concepts covered in the lectures.

2.1. **Q-Learning.** The file `RL_Lab1.R` in the course website contains a template of the Q-learning algorithm.[1] You are asked to complete the implementation. We will work with a grid-world environment consisting of $H \times W$ tiles laid out in a 2-dimensional grid. An agent acts by moving up, down, left or right in the grid-world. This corresponds to the following Markov decision process:

- State space: $\mathcal{S} = \{(x, y) | x \in \{1, \ldots, H\}, y \in \{1, \ldots, W\}\}$.
- Action space: $\mathcal{A} = \{up, down, left, right\}$.

Additionally, we assume state space to be fully observable. The reward function is a deterministic function of the state and does not depend on the actions taken by the agent. We assume the agent gets the reward as soon as it moves to a state. The transition model is defined by the agent moving in the direction chosen with probability $(1-\beta)$. The agent might also slip and end up moving in the direction to the left or right of its chosen action, each with probability $\beta/2$. The transition model is unknown to the agent, forcing us to resort to model-free solutions. The environment is episodic and all states with a non-zero reward are terminal. Throughout this lab we use integer representations of the different actions: Up=1, right=2, down=3 and left=4.

2.2. **Environment A.** For our first environment, we will use $H = 5$ and $W = 7$. This environment includes a reward of 10 in state (3,6) and a reward of -1 in states (2,3), (3,3) and (4,3). We specify the rewards using a reward map in the form of a matrix with one entry for each state. States with no reward will simply have a matrix entry of 0. The agent starts each episode in the state (3,1). The function `vis_environment` in the file `RL_Lab1.R` is used to visualize the environment and learned action values and policy. You will not have to modify this function, but read the comments in it to familiarize with how it can be used.

When implementing Q-learning, the estimated values of $Q(S, A)$ are commonly stored in a data-structured called Q-table. This is nothing but a tensor with one entry for each state-action pair. Since we have a $H \times W$ environment with four actions, we can use a 3D-tensor of dimensions $H \times W \times 4$ to represent our Q-table. Initialize all Q-values to 0. Run the function `vis_environment` before proceeding further. Note that each non-terminal tile has four values. These represent the action values associated to the tile (state). Note also that each non-terminal tile has an arrow. This indicates the greedy policy for the tile (ties are broken at random).

Your are requested to carry out the following tasks.

- Implement the greedy and ϵ-greedy policies in the functions `GreedyPolicy` and `EpsilonGreedyPolicy` of the file `RL_Lab1.R`. The functions should break ties at random, i.e. they should sample uniformly from the set of actions with maximal Q-value.
- Implement the Q-learning algorithm in the function `q_learning` of the file `RL_Lab1.R`. The function should run one episode of the agent acting in the environment and update the Q-table accordingly. The function should return the episode reward and the sum of the temporal-difference correction terms $R + \gamma \max_a Q(S', a) - Q(S, A)$ for all steps in the episode. Note that a transition model taking β as input is already implemented for you in the function `transition_model`.
- Run 10000 episodes of Q-learning with $\epsilon = 0.5$, $\beta = 0$, $\alpha = 0.1$ and $\gamma = 0.95$. To do so, simply run the code provided in the file `RL_Lab1.R`. The code visualizes the Q-table and a greedy policy derived from it after episodes 10, 100, 1000 and 10000. Answer the following questions:
  - What has the agent learned after the first 10 episodes ?
  - Is the final greedy policy (after 10000 episodes) optimal? Why / Why not ?
  - Does the agent learn that there are multiple paths to get to the positive reward ? If not, what could be done to make the agent learn this ?

[1] If you do not see four arrows in line 16 of the template, then do the following: File/Reopen with Encoding/UTF-8.

2.3. **Environment B.** This is a 7×8 environment where the top and bottom rows have negative rewards. In this environment, the agent starts each episode in the state (4, 1). There are two positive rewards, of 5 and 10. The reward of 5 is easily reachable, but the agent has to navigate around the first reward in order to find the reward worth 10.

Your task is to investigate how the ϵ and γ parameters affect the learned policy by running 30000 episodes of Q-learning with $\epsilon = 0.1, 0.5$, $\gamma = 0.5, 0.75, 0.95$, $\beta = 0$ and $\alpha = 0.1$. To do so, simply run the code provided in the file `RL_Lab1.R` and explain your observations.

2.4. **Environment C.** This is a smaller 3×6 environment. Here the agent starts each episode in the state (1,1). Your task is to investigate how the β parameter affects the learned policy by running 10000 episodes of Q-learning with $\beta = 0, 0.2, 0.4, 0.66$, $\epsilon = 0.5$, $\gamma = 0.6$ and $\alpha = 0.1$. To do so, simply run the code provided in the file `RL_Lab1.R` and explain your observations.

2.5. **REINFORCE.** The file `RL_Lab2.R` in the course website contains an implementation of the REINFORCE algorithm.[2] Your task is to run the code provided and answer some questions. Although you do not have to modify the code, you are advised to check it out to familiarize with it. The code uses the R package `keras` to manipulate neural networks.

We will work with a 4×4 grid. We want the agent to learn to navigate to a random goal position in the grid. The agent will start in a random position and it will be told the goal position. The agent receives a reward of 5 when it reaches the goal. Since the goal position can be any position, we need a way to tell the agent where the goal is. Since our agent does not have any memory mechanism, we provide the goal coordinates as part of the state at every time step, i.e. a state consists now of four coordinates: Two for the position of the agent, and two for the goal position. The actions of the agent can however only impact its own position, i.e. the actions do not modify the goal position. Note that the agent initially does not know that the last two coordinates of a state indicate the position with maximal reward, i.e. the goal position. It has to learn it. It also has to learn a policy to reach the goal position from the initial position. Moreover, the policy has to depend on the goal position, because it is chosen at random in each episode. Since we only have a single non-zero reward, we do not specify a reward map. Instead, the goal coordinates are passed to the functions that need to access the reward function.

2.6. **Environment D.** In this task, we will use eight goal positions for training and, then, validate the learned policy on the remaining eight possible goal positions. The training and validation goal positions are stored in the lists `train_goals` and `val_goals` in the code in the file `RL_Lab2.R`. You are requested to run the code provided, which runs the REINFORCE algorithm for 5000 episodes with $\beta = 0$ and $\gamma = 0.95$.[3] Each training episode uses a random goal position from `train_goals`. The initial position for the episode is also chosen at random. When training is completed, the code validates the learned policy for the goal positions in `val_goals`. This is done by with the help of the function `vis_prob`, which shows the grid, goal position and learned policy. Note that each non-terminal tile has four values. These represent the action probabilities associated to the tile (state). Note also that each non-terminal tile has an arrow. This indicates the action with the largest probability for the tile (ties are broken at random). Finally, answer the following questions:

- Has the agent learned a good policy? Why / Why not ?
- Could you have used the Q-learning algorithm to solve this task ?

2.7. **Environment E.** You are now asked to repeat the previous experiments but this time the goals for training are all from the top row of the grid. The validation goals are three positions from the rows below. To solve this task, simply run the code provided in the file `RL_Lab2.R` and answer the following questions:

[2] If you do not see four arrows in line 19 of the code, then do the following: File/Reopen with Encoding/UTF-8.

[3] Note that we are performing back-propagation in a neural network in each episode. Even though the network is small, this is quite computationally heavy and may take some time. Specifically, it takes around 12 wall-clock minutes in my laptop (Intel(R) Core(TM) i7-6600U CPU at 2.60GHz with 16.0 GB RAM and Windows 10). If someone finds a way to speed up the code, I would love to hear about it. The same code implemented in Python is faster. So, `keras` seems to get along better with Python than with R.

- Has the agent learned a good policy? Why / Why not ?
- If the results obtained for environments D and E differ, explain why.

ACKNOWLEDGMENTS

Thanks to Joel Oskarsson for developing this lab in Python for the course “Machine Learning for Industry”. This is a translation to R of that lab. Any errors are my sole responsibility.