SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Olivier Sigaud
RL Labs: Instructions
2020-2021

---

1 Final report instructions and evaluation

- Upload you lab report on the Moodle page of the class, following the "Upload your lab report" link.
- You must upload your report included as a pdf file and a copy of your source files, altogether included into an archive whose name must be `RL_NAME1_NAME2` (.zip or .tar.gz).
- For each function required in a question, the corresponding piece of code should be copy-pasted into your report to facilitate evaluation.
- The evaluation will take into account the quantity of work done, the accuracy of the results and the quality of the content (including written english).

2 MDPs and mazes

A reinforcement learning agent interacts with an environment represented as a Markov Decision Process (MDP). It is defined by a tuple (S, A, P, r, γ) where S is the state space, A the action space, $P(state_t, action_t, state_{t+1})$ the transition function, $r(state_t, action_t)$ the reward function and $\gamma \in [0, 1]$ the discount factor.

2.1 Documentation of the provided code

Some code is provided to create mazes, transform them into MDPs and visualize them together with policies or value functions. It is contained into three files: `maze.py`, `mdp.py` and `maze_plotter.py`. The following sections give an overview of this code.

2.1.1 Content of the maze.py file

A maze is represented as an object of the `Maze` class. It is defined as a grid of $width \times height$ cells, and some of these cells contain a wall.

The `build_maze(width,height,walls,hit=False)` function is used to create a Maze, where `walls` is a list of the number of the cells which contain a wall. The `hit` parameter has an impact on the MDP reward function: if `hit` is `true`, the agent is penalized each time it tries to move to a wall cell. Otherwise, the agent is just rewarded when it reaches terminal states. In the provided function, the list of terminal states is a singleton corresponding to the last cell that the agent can visit.

Apart from representing the two reward functions described above, the `Maze` class contains a constructor whose only role is to create the MDP corresponding to the maze and the maze plotter used to display simulations. A key point is that only cells where there is no wall are considered as states of the underlying MDP. To facilitate the correspondence between mazes and MDPs, each free cell (i.e. with no wall) knows the number of its corresponding MDP state.

The maze constructors also builds the action space, the initial state distribution, the transition function and the reward function of the MDP. Once all these data structures have been created, the resulting MDP is built.

2.1.2 Content of the mdp.py file

The `mdp.py` file contains the `SimpleActionSpace` class and the `Mdp` class.

The `SimpleActionSpace` class contains the list of actions and a method to sample from this list. In our maze environment, the possible actions for the agent are going north, south, east or west (resp. [0, 1, 2, 3]).

The `Mdp` class is designed to be compatible with the OpenAI gym interface[1]. The main methods are `reset(self, uniform=False)`, which resets the MDP into an initial state drawn from the initial state distribution, and `step(self, u,deviation=0)` which is used to let the agent perform a step in the environment, sending and action and receiving the next state, the reward, and a signal telling whether a terminal state was reached.

2.1.3 Content of the maze_plotter.py file

The code to display the effect of the algorithms in these environments is in `maze_plotter.py`, in the `MazePlotter` class. In order to visualize the environment, you use the `new_render()` function to initialize the rendering, then `render(V, policy,agent_pos)` to refresh the maze with either the newly calculated state values and the policy, or the state-action values, and eventually the current position of the agent. There is also a `render_pi(policy)` function which only displays the policy (useful for POLICY ITERATION). The function `save_fig(title)` is used to save the last render into a file.

You can see examples of calls to these different visualizations in the functions defined in `dynamic_programming.py` and `reinforcement_learning.py`.

2.2 Playing with differents MDPs (optional)

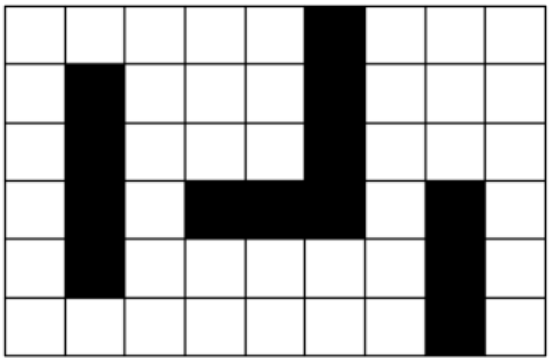

Figure 1: A sample maze

Though it is not crucial for these labs, you may have a look at how MDPs and mazes are defined and try to design different ones.

Code question 1:(**OPTIONAL and easy**) In `dynamic_programming.py`, in the `run()` function, modify the call to the `build_maze` function so as to get a maze corresponding to Figure 1.

Code question 2: (**OPTIONAL and more difficult**) Copy the `mdp.py` file into `my_mdp.py` and modify the MDP so that actions have a stochastic outcome: each time the agent chooses a direction, there is a `proba_success` chance that it performs the expected movement, and the remaining chances are evenly distributed between available directions, i.e. directions where there is no wall.

Later, when running your dynamic programming and reinforcement learning functions, you can evaluate them on the various MDPs you have designed and see if there is a difference in convergence time and properties.

3 Dynamic Programming

The goal of an RL agent is to find the optimal behaviour, defined by a policy π that assigns an action (or distribution over actions) to each state so as to maximize the agent's total expected reward. In order to estimate how good a state is, either

[1] `https://gym.openai.com/`

a state value function $V(x)$ or a state-action value function $Q(x,u)$ is used.

Dynamic programming algorithms are used for planning, they require a full knowledge of the MDP from the agent (in contrast to "true" RL where the agent does not know the transition and reward functions). They find the optimal policy by computing a value function V or an action-value function Q over the state space or state-action space of the given MDP. VALUE ITERATION and POLICY ITERATION are two standard dynamic programming algorithms. You should study both of them using both V and Q, as these algorithms contain the basic building blocks for most RL algorithms.

3.1 Value iteration

3.1.1 Value iteration with the V function

When using the V function, VALUE ITERATION aims at finding the optimal values V^* based on the Bellman Optimality Equation:

$$V^*(s) = \max_a \left[r(s,a) + \gamma \sum_{y \in S} P(s,a,y) V^*(y) \right].$$

In `dynamic_programming.py`, the `value_iteration_v(mdp)` function provides the code of VALUE ITERATION using the V function. It is given as an example from which you can derive other instances of dynamic programming algorithms. Look at it more closely, this will help for later questions:

- you can ignore the `mdp.new_render()` and `mdp.render(...)` functions which are here to provide the visualization of the iterations.
- find in the code the loop over states, the main loop that performs these updates until the values don't change significantly anymore, the main update equation. Found them? OK, you can continue...

3.1.2 Value iteration with the Q function

The state-action value function $Q^\pi(s,a)$ defines the value of being in state s, taking action a then following policy π. The Bellman Optimality Equation for Q^* is

$$Q^*(s,a) = r(s,a) + \gamma \sum_y P(s,a,y) \max_{a'} Q^*(y,a').$$

Code question 3: By taking inspiration from the `value_iteration_v(mdp)` function given in `dynamic_programming.py`, fill the blank (given with '#Q[x,u]=...') in the code of `value_iteration_q(mdp)` and run it. Your final report must contain the missing piece of code.

3.2 Policy Iteration

The POLICY ITERATION is more complicated than VALUE ITERATION. Given a MDP and a policy π, POLICY ITERATION iterates the following steps:

- Evaluate policy π: compute V or Q based on the policy π;
- Improve policy π: compute a better policy based on V or Q.

This process is repeated until convergence, i.e. when the policy cannot be improved anymore.

When using V, $V^\pi(s)$ is the expected return when starting from state s and following policy π. It is processed based on the Bellman Expectation Equation for deterministic policies:

$$V^\pi(x) = r(s,\pi(s)) + \gamma \sum_y P(x,\pi(x),y) V^\pi(y),$$

where:

- π is a deterministic policy, meaning that in a state s, the agent always selects the same action,
- $r(s,\pi(s))$ is the reward obtained from taking action $\pi(s)$ in state s,
- $P(s,\pi(s),y)$ is the probability of reaching state y when taking action $\pi(s)$ in state s,

- $V^\pi(y)$ is the value of the state y under policy π,
- $\gamma \in [0,1]$ is a discount factor defining the relative importance of long term rewards over short term ones (the closer to 0, the more the agent focuses on immediate rewards).

When using Q, the Bellman Expectation Equation with deterministic policy π for Q becomes:

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{y \in S} P(s,a,y) Q^\pi(y, \pi(y)).$$

The policy can then be updated as follows:

$$\pi^{(t+1)}(x) = \arg\max_a Q^{\pi^{(t)}}(s,a).$$

3.2.1 Intermediate functions

In order to facilitate the coding of POLICY ITERATION algorithms, we first define a set of useful functions. When it is not provided, the code for all the functions below should be given in your report.

Code question 4: In `dynamic_programming.py`, code a function `get_policy_from_q(q)`, where `q` is the state-action value function. Give the code in our report.

The more complicated function `get_policy_from_v(v)`, where `v` is the state value function, is given. The function `improve_policy_from_v(mdp,v,policy)` is also given. It is very similar to `get_policy_from_v(v)`, except that it takes a policy as argument and improves this policy when possible.

The functions `evaluate_one_step_v(mdp,v,policy)`, where `mdp` is a given MDP, `v` is some value function in this MDP and `policy` is some policy and the function `evaluate_v(mdp,policy)` are given. These functions give the value function V^π corresponding to policy π.

Code question 5: By drawing inspiration on the given functions, code a function `evaluate_one_step_q(mdp, q,policy)`, where `q` is some action value function, and a function `evaluate_q(mdp,policy)`. Give the code in our report.

3.2.2 Policy Iteration functions

Code question 6: By using the above functions, fill the code of the `policy_iteration_q(mdp)` function, run it and visualize it. Include the missing code and a picture of the final configuration in your report.

Code question 7: By using the above functions, fill the code of the `policy_iteration_v(mdp)` function, run it and visualize it. Include the missing code and a picture of the final configuration in your report.

3.3 Comparisons

We want to compare the efficiency of the various dynamic programming methods using either the V or the Q functions.

Study question 8: In all your dymanic programming functions, add code to count the number of iterations and the number of elementary V or Q updates. Use the provided `Chrono` class to measure the time taken. Use the provided `create_maze(...)` function to create a large maze (around 10×10), run your functions and collect the resulting numbers.
Build a table where you compare the various dymanic programming functions in terms of iterations, elementary operations and time taken. Also run the provided `plot_convergence_vi_pi(...)` function to visualize the convergence of the various algorithms.
Given all these results, discuss the relative computational efficiency of these methods.

4 Reinforcement learning functions

Reinforcement Learning is about finding the optimal policy in an MDP which is initially unknown to the agent. More precisely, the state and action spaces are known, but the agent does not know the transition and reward functions. Generally speaking, the agent has to explore the MDP to figure out which action in which state leads to which other state and reward. The model-free case is about finding this optimal policy just through very local updates, without storing any information about previous interactions with the environment. Principles of these local updates can already be found in the Temporal Difference (TD) algorithm, which iteratively computes optimal values for all state using local updates. The most widely used model-free RL algorithms are Q-LEARNING, SARSA and actor-critic algorithms. In the `reinforcement_learning.py` file, we focus on the first two.

4.1 TD learning

Given a state and an action spaces as well as a policy, TD(0) computes the state value of this policy based on the following equations:

$$\delta_t = r(s_t, a_t) + \gamma V^{(t)}(s_{t+1}) - V^{(t)}(s_t)$$
$$V^{(t+1)}(s_t) = V^{(t)}(s_t) + \alpha\delta_t$$

where δ is the TD error and α is a parameter called "learning rate".

The code is provided in `reinforcement_learning.py`, so that you can take inspiration later on. The important part is the computation of δ, and the update of the values of V.

To run TD learning, a policy is needed as input. Such a policy can be retreived by using the `policy_iteration_q(mdp)` function defined in the `dynamic_programming.py` file.

Code question 9: Fill the code of the `temporal_difference(...)` function and run it. Put into your report the corresponding code and a picture of the final configuration.

4.2 Q-learning

The Q-LEARNING algorithm accounts for an agent exploring an MDP and updating at each step a model of the state action-value function stored into a Q-table. It is updated as follows:

$$\delta_t = r(s_t, a_t) + \gamma \max_{a \in A} Q^{(t)}(s_{t+1}, a) - Q^{(t)}(s_t, a_t)$$

$$Q^{(t+1)}(s_t, a_t) = Q^{(t)}(s_t, a_t) + \alpha\delta_t.$$

The Q-LEARNING function in `reinforcement_learning.py` gives the code of Q-LEARNING.

Code question 10: Fill the code of the `q_learning(...)` function and run it. Put into your report the corresponding code and a picture of the final configuration.

4.2.1 Learning dynamics

By watching carefully the values while the agent is learning, you can see that the agent favors certains paths over others which have a strictly equivalent value. This can be explained easily: as the agent chooses a path for the first time, it updates the values along that path, these values get higher than the surrounding values, and the agent chooses the same path again and again, increasing the phenomenon. Only steps of random exploration can counterbalance this effect, but they do so extremely slowly.

4.2.2 Exploration

In Q-LEARNING function, action selection is based on a *softmax* policy. Instead, it could have relied on ε-*greedy*.

Code question 11: Copy-paste the Q-LEARNING function and replace the call to the *softmax* policy with an ε-*greedy* policy. The `softmax(...)` and `egreedy(...)` functions are available in `toolbox.py`. Rename both functions `q_learning_soft(...)` and `q_learning_eps(...)`. Put the relevant piece of code into your report.

4.3 SARSA

The SARSA algorithm is very similar to Q-learning. At first glance, the only difference is in the update rule. However, to perform the update in SARSA, one needs to know the action the agent will take when it will be at the next state, even if the agent is taking a random action.

This implies that the next state action is determined in advance and stored for being played at the next time step.

Code question 12: By taking inspiration from the above Q-LEARNING function, write the `sarsa(...)` function that implements the corresponding algorithm, then run it. Put the corresponding code into your report. As above, copy-paste the resulting code to get a `sarsa_soft(...)` and a `sarsa_eps(...)` function.

4.4 Comparisons and hyper-parameters

Study question 13: Compare in your report the number of steps needed by Q-LEARNING and SARSA to converge on the given MDP using the *softmax* and ε-*greedy* exploration strategies. To figure out, use the provided `plot_ql_sarsa(m,epsilon,tau,nb_episodes,timeout,alpha,render)` function with various values for `epsilon` (e.g. 0.001, 0.01, 0.1) and `tau` (e.g. 0.1, 5, 10) and comment the obtained curves.

Study question 14: When using *softmax*, the three main hyper-parameters of Q-LEARNING and SARSA are α, τ and γ. By varying the values of these hyper-parameters and watching the learning process and behavior of the agent, explain in your report their impact on the algorithm. Using additional plotting functions will also be welcome.

5 Optional questions for going further

The lab instructions above are taken from a richer set of questions available here:
`https://github.com/osigaud/rl_labs_notebooks` All the questions below are optional and can be addressed independently from each other. Trying to address these further questions will result in a bonus on your final grade.

5.1 Generalized Policy Iteration

The GENERALIZED POLICY ITERATION (GPI) algorithm is described in `http://incompleteideas.net/book/first/ebook/node46.html`. It draws a continuum between VALUE ITERATION and POLICY ITERATION.

Code question 15: Code the GPI algorithm and parametrize it to obtain the VALUE ITERATION regime, the POLICY ITERATION regime, and something intermediate. Provide your code and report the performance of the various regimes.

5.2 Actor-critic architecture

As the name implies, an actor-critic architecture contains both a policy (the actor) and a critic. Note that studying actor-critic algorithms is a must if you want to study deep RL afterwards, because several state-of-the-art deep RL algorithms are actor-critic.

Code question 16: Code a basic actor-critic algorithm and compare it to Q-LEARNING. Provide your code and report the comparison.

5.3 Replay buffer

A replay buffer is used in deep RL algorithms to decorrelate samples resulting from the interaction between the agent and its environment. The simplest form is a long FIFO list, from which samples are drawn randomly to train a critic.

Code question 17: Add a replay buffer to your reinforcement learning algorithm and observe its effect on learning efficiency in the tabular case. Provide your code and report your observations.