RLSS09: Robot Learning Summer School Reinforcement Learning Lab Session

Prepared by:
Francisco S. Melo

http://vislab.isr.ist.utl.pt/RLSS09/

Objective

The purpose of this lab session is to allow you to gain some practical knowledge on some of the fundamental concepts in Reinforcement Learning (RL). We propose a total of 13 different assignments, grouped into 4 distinct sections: "Models", "Planning", "Learning" and "Other". You are not expected to complete all 13 proposed assignments. Ideally, you should complete the assignments in the subsections marked with ($\diamond$). However, if you feel that you are already familiar with some of the topics, or that one of the topics is particularly relevant for your work, you can make a different selection, for example completing the potentially more challenging assignments in the sections marked with ($\star$). The assignments in the subsections marked with ($\star$) typically build on the material covered in the ones immediately before. Our suggestion is that you start by browsing through all the proposed topics and choose after you have read them all.

The different assignments are to be completed in Matlab. Familiarity with this platform can facilitate the completion of the tasks but is not required.

1 Models

Reinforcement Learning problems are typically modeled using Markov Decision Problems (MDP) or a variation thereof. In this section, we explore the properties of this fundamental model, allowing you to gain a better understanding of the different elements that constitute MDPs.

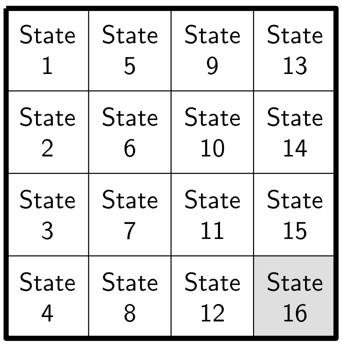
1.1 Markov Decision Problems ($\diamond$)

For our purposes, a Markov Decision Problem (MDP) is described by a tuple $(\mathcal{X}, \mathcal{A}, \mathsf{P}, r, \gamma)$, where $\mathcal{X}$ is the state-space, $\mathcal{A}$ is the action-space, P represents the transition probabilities, r represents the reward function and γ is a discount factor, that represents the particular optimality criterion used.

In this laboratory, we focus on situations in which $\mathcal{X}$ and $\mathcal{A}$ are discrete sets. In robotic terms, we interpret each element x in the set $\mathcal{X}$ as a possible *state* of the robot, and each element a in the set $\mathcal{A}$ as an available *primitive action* of the robot. The transition probabilities describe the *dynamics* of the robot and the reward function encodes the *goal* of the robot – the task that the robot must complete.

Given the state at time t, denoted $X(t)$, and the action at that same time, $A(t)$, the state at time $t+1$, $X(t+1)$, depends only on $X(t)$ and $A(t)$ according to the probabilities:

$$\mathbb{P}\left[X(t+1) = y \mid X(t) = x, A(t) = a\right] = \mathsf{P}(x, a, y).$$



State 1	State 5	State 9	State 13
State 2	State 6	State 10	State 14
State 3	State 7	State 11	State 15
State 4	State 8	State 12	State 16

Figure 1: Simple 16-state environment for robot navigation. The shaded cell corresponds to the goal location.

The robot also receives a (possibly random) reward, $R(t)$, that depends on $X(t)$ and $A(t)$ and whose expectation is given by

$$\mathbb{E}\left[R(t) \mid X(t) = x, A(t) = a\right] = r(x, a).$$

The goal of the robot is to choose the actions $A(t)$ so as to maximize (in expectation) the total discounted reward received:

$$\mathbb{E}\left[\sum_t \gamma^t R(t)\right].$$

A policy is a mapping $\pi : \mathcal{X} \to \mathcal{A}$ that assigns, to each state $x \in \mathcal{X}$, an action $a = \pi(x)$ that the robot should execute whenever in state x. The value of a policy π is the total discounted reward that the robot expects to receive when following policy π. This value generally depends on the initial state of the robot, and can be represented as a function

$$V^\pi(x) = \mathbb{E}\left[\sum_t \gamma^t R(t) \mid X(0) = x, A(t) = \pi(X(t))\right].$$

The optimal policy is the policy π^* that has the largest value at every state, *i.e.*,

$$V^*(x) \geq V^\pi(x)$$

for all $x \in \mathcal{X}$ and all policies π (we have denoted by V^* the value of policy π^*).

Simple Grid World

Consider the environment depicted in Fig. 1. This represents a very simple grid-world that a mobile robot must navigate to reach a goal location, marked as the shaded cell. The robot has available 4 actions each corresponding to movement in one of the 4 possible directions – North (N), South (S), East (E) and West (W) – and a fifth "NoOp" action ($\emptyset$).

Assignment 1.1. In this assignment, you will encode the above problem as an MDP. For the time-being, we assume that each of the actions N, S, E and W has a deterministic outcome.

(a) Launch Matlab and navigate to the folder `rlss_ass11`. In the Matlab main window, execute the command "`init`".

(b) Open the file `GridLoad.m` in the Matlab editor. This is the skeleton of an M-file that generates a structure containing the MDP model for the grid world.

You will notice that, throughout the file, there are several places where you are supposed to replace the entries "`???`" by an adequate value. Write down the transition probabilities for each triplet (x, a, y) for the example above. Recall that, for now, we assume that all actions have a deterministic outcome: with probability 1, the actions N, S, E and W move the robot one step in the corresponding direction. For illustrative purposes, the transition probabilities for action N have been filled up for you. Also, write down the reward function that encodes the goal "Go to state 16 and stop."

(c) Once you have filled all "`???`" entries in the file `GridLoad.m`, save it. In the Matlab window, type "`M = GridLoad`". This instruction calls the function defined in `GridLoad.m` and saves its output to the variable `M`. If the function runs successfully, you should read in the Matlab window something like

```
M =

        nS: 16
        nA: 5
         P: [16x16x5 double]
         r: [16x5 double]
     Gamma: 0.9500
     Model: 1
```

(d) The component `M.P` is a *stochastic matrix*, which means that each line must add to one. Verify this by executing, in the Matlab window, the command "`sum(M.P(`x`, :, `a`))`", where x can be any number between 1 and 16 and a can be any of `N`, `S`, `E`, `W` or `Noop`.

(e) In the Matlab window, execute "`plotPolicy(M)`". This function computes the optimal policy for the MDP `M` and displays it in a figure window. Verify that the policy is as expected.

(f) Try different reward functions by modifying the file `GridLoad.m`. After every change to the file `GridLoad.m`, save the file, reload the MDP using "`M = GridLoad`" and observe the corresponding policy using "`plotPolicy(M)`".

So far, you have modeled a very simple MDP and observed the effect of the reward on the optimal policy. We now move to variations of the same task.

Other Grid Worlds

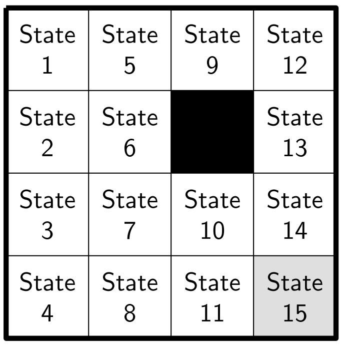
We now consider that the outcome of the four actions N, S, E and W is no longer deterministic. We also consider variations of the same basic task, as represented in Fig. 2.

Assignment 1.2. You will model several versions of the grid-world problem as an MDP. This time, we assume that each of the actions N, S, E and W succeeds with probability 0.65; with probability 0.15 the action completely fails and the robot does not move at all; and with probability 0.2 the action has an unforeseen outcome, and the agent is randomly placed in one of the neighboring cells.

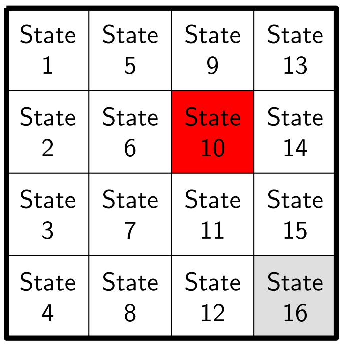
(a) Launch Matlab and navigate to the folder `rlss_ass12`. Clean up the memory by running the "`clear all`" command in the Matlab window. Execute the command "`init`".

(b) Open the file `MDPLoad.m` in the Matlab editor. This is a more complete skeleton of an M-file that generates a structure containing the MDP model for the different grid worlds.

You will find that the values for the deterministic case are already filled. You will also note that the lines corresponding to the models that are yet undefined have been commented, to prevent Matlab from generating errors. Uncomment the lines corresponding to "`case 2`". Complete



(a) Environment with an obstacle.



(b) Environment with a "puddle".

Figure 2: Two variations of the grid-world environment for robot navigation. The shaded cell corresponds to the goal location. The black cell corresponds to a solid obstacle, while the red cell corresponds to a soft obstacle (a "puddle").

the corresponding transition matrices, each marked with a single entry "`???`". Recall that we assume that each of the actions N, S, E and W succeeds with probability 0.65; with probability 0.15 the action completely fails and the robot does not move at all; and with probability 0.2 the action has an unforeseen outcome, and the agent is randomly placed in one of the neighboring cells. Also write down the reward function that encodes the goal "Go to state 16 and stop."

(c) Save the file `MDPLoad` and, in the Matlab window, execute "`M = MDPLoad('stochastic')`". This instruction calls the function defined in `MDPLoad.m`, loads the non-deterministic model and saves its to the variable `M`. If the function runs successfully, you should once again read in the Matlab window something like

```
M =

       nS: 16
       nA: 5
        P: [16x16x5 double]
        r: [16x5 double]
    Gamma: 0.9500
    Model: 2
```

(d) Once again verify that your transition matrix is a stochastic matrix by checking some of the rows of the matrix `M.P`.

(e) In the Matlab window, execute "`plotPolicy(M)`". Verify that the policy is as expected.

(f) Try different transition probabilities by modifying the file `MDPLoad.m` and observe the effect in terms of the policy. Recall that, after every change to the file `MDPLoad.m`, you should save the file, reload the MDP using "`M = MDPLoad('stochastic')`" and observe the corresponding policy "`plotPolicy(M)`".

(g) Open the file `MDPLoad.m` in the Matlab editor. Uncomment the lines corresponding to "`case 3`" and complete the corresponding transition matrices, corresponding to the scenario in Fig. 2(a). Use a similar transition model as in case 2, in which each of the actions N, S, E and W succeeds with probability 0.65; with probability 0.15 the action completely fails and the robot does not move at all; and with probability 0.2 the action has an unforeseen outcome, and the agent is randomly placed in one of the neighboring cells. Also write down the reward function that encodes the goal "Go to state 16 and stop."

(h) Save the file `MDPLoad` and, in the Matlab window, execute the command "`M = MDPLoad('hard_obstacle')`". Verify that your transition matrix is a stochastic matrix by checking some of the rows of the matrix `M.P`.

(i) In the Matlab window, execute "`plotPolicy(M)`". Verify that the policy is as expected.

(j) Finally, again open the file `MDPLoad.m` in the Matlab editor and uncomment the lines corresponding to "`case 4`". Complete the corresponding transition matrices, now corresponding to the scenario in Fig. 2(b). Use a similar transition model as in cases 2 and 3. Write down the reward function that encodes the goal "Go to state 16 while avoiding state 10 and stop."

(k) Save the file `MDPLoad` and, in the Matlab window, execute the command "`M = MDPLoad('soft_obstacle')`". Verify that your transition matrix is a stochastic matrix by checking some of the rows of the matrix `M.P`.

(l) In the Matlab window, execute "`plotPolicy(M)`". Verify that the policy is as expected.

1.2 Extending MDPs ($\star$)

As seen in the previous section, an MDP is a tuple $(\mathcal{X}, \mathcal{A}, \mathsf{P}, r, \gamma)$, where $\mathcal{X}$ denotes the state-space, $\mathcal{A}$ denotes the action-space, P represents the transition probabilities, r is the reward function and γ is the discount factor. One fundamental assumption in MDPs is that the state at time $t+1$, $X(t+1)$, depends only on the state/action at time t. This is known as the *Markov property*, and is summarized by the relation

$$\mathbb{P}\left[X(t+1) = y \mid X(t) = x, A(t) = a\right] = \mathsf{P}(x, a, y).$$

The reward received by the robot at time t, $R(t)$, also exhibits a similar dependence. Its expectation, $r(x, a)$, is a function of the state x and the action a, and therefore, at each time-step does not exhibit any long-term dependency on the past. This Markovian assumption, however useful it is, poses several limitations on the classes of tasks that can be modeled by MDPs. In this second assignment, we study two archetypal situations that require the MDP model to be augmented in specific ways.

Tasks Requiring a Sequence of Steps

Consider once again the simple grid environment depicted in Fig. 1. Suppose now that the task for the robot is to visit state 10 before going to state 16 and stop. If we model this task using an MDP similar to the one implemented in the previous subsection, in which the state-space corresponds to the possible positions for the robot in the environment, there is no reward function that can represent this task (can you see why?) In order to model such task as an MDP, we have to *augment* the state-space to also include the information of whether state 10 has already been visited or not. This augmented state-space has $2 \times 16 = 32$ states.

In this second assignment, you will model a couple of such tasks using the MDP framework, and analyze how this impacts the dimension of the MDP. We will once again use the environment in Fig. 1.

Assignment 1.3. You will model several variations of the simple grid-world problem as MDPs. Therefore, as in Assignment 1.1, we assume that the actions N, S, E and W deterministically move the robot one step in the corresponding direction.

(a) Launch Matlab and navigate to the folder `rlss_ass13`. Clean up the memory by running the "`clear all`" command in the Matlab window. Execute the command "`init`".

(b) Copy the file `GridLoad.m` from the folder `rlss_ass11` to `rlss_ass13`. You will need this file created in Assignment 1.1 to complete this assignment. If you have not completed that assignment, please request a complete `GridLoad.m` from the lab assistant.

(c) Open the file `MDPLoad.m` in the Matlab editor. This is the skeleton of an M-file that generates a structure containing the augmented MDP model for the grid world.

Your goal is to implement a generic MDP loader that, given as a parameter a sequence of states (corresponding to successive goals for the robot), generates the MDP structure that represents that particular task in the simple grid world. To help you in this task you will find instructions and suggestions in the comments throughout the file.

You will also notice that the file begins by loading the simple grid-world MDP, in the line "`M = GridLoad`". This is the line that requires `GridLoad.m` from the Assignment 1.1. Use the simple world MDP structure to complete the transition probabilities for the augmented MDP. As you can easily realize, you can access the variables `nSaux`, `nAaux` and `Paux` that contain, respectively, the number of states and actions and the transition probabilities from the simple grid-world MDP.

(d) Once you have completed implementing the function, save the file. In the Matlab window, you can now try to generate the MDP for the case introduced in the beginning of the section, by executing "`M = MDPLoader([10 16])`". Once the MDP is loaded, you should now observe something like

```
M =

       nS: 32
       nA: 5
        P: [32x32x5 double]
        r: [32x5 double]
    Gamma: 0.9500
    Model: 5
```

(e) You can now visualize the policy for this MDP by executing "`plotPolicy(M)`". This will graphically represent the optimal policy for each of the "stages" of the problem. In the case above (where the robot must visit states 10 and 16), you will observe two grid worlds, side by side. The policy depicted in the first grid-world (on the left) corresponds to the optimal policy *before* visiting state 10, and the one on the right corresponds to the optimal policy *after* visiting state 10 (in the first case, you should ignore the action at state 10).

(f) You probably noticed that, whenever you execute the function `plotPolicy`, a message appears in the Matlab window similar to

```
Running value iteration... Done.
Total running time: 0.06 seconds.
Total iterations:   325.
Average iteration time: 0.000 seconds.
```

Try running this function (`plotPolicy`) for different MDPs obtained using as the argument of `MDPLoad` vectors of different lengths, and record the different running times. You can store the length of the arguments of `MDPLoad` in a vector `x` and the corresponding running time in a vector `y`, and plot the relation between the two by executing `plot(x, y)`. Notice that, as the task completion requires the robot to "memorize" further and further back into the past (*i.e.*, as the task involves a larger sequence of steps), this translates into a growth in the size of the MDP, and consequent growth in the time required to compute the corresponding policy.

Robots with Sensor Noise

Intuitively, we can think of the state $X(t)$ of an MDP as a variable that represents the most up-to-date record of all information relevant for the robot to complete the task. And, in the discussion of MDPs so far, we have assumed that this information was readily available for the robot to use.

However, in real-world tasks, a robot will generally have to rely on its sensors to perceive this information. Alas, sensors are not perfect, and it may happen that the robot may not be able to unambiguously perceive that information. Such situations, in which the robot must rely on noisy sensor measurements to *infer* the state of the world, are generally referred to as having *partial observability*. In the second part of this assignment, you will implement a very simple way to handle this uncertainty. You will also have the opportunity to bring to play some of the notions on Bayesian estimation that have been presented in the lectures.

We depart from the MDP model introduced before. Consider some MDP model that describes a certain task that a robot must complete. Suppose that, at time t, the robot is fully aware of the state of the MDP, $X(t)$. For concreteness, let us suppose that $X(t) = x$. It then executes action $A(t) = \pi(x)$, according to some policy π. The state of the MDP changes and the robot receives a reading from its sensors that we denote as a random variable $Z(t+1)$. For simplicity, we assume that this reading can take values in a finite set $\mathcal{Z}$ of possible readings. In order for the robot to estimate its current state, $X(t+1)$, it must use the information provided by $Z(t+1)$.

In order to infer, from the observation $Z(t)$, what is the underlying state $X(t)$ of the robot, it is necessary some model of observations, *i.e.*, a relation between the state $X(t)$ and the corresponding observation it generates, $Z(t)$. We generally refer to this model as the *observation model* and it is represented by *observation probabilities*,

$$\mathbb{P}\left[Z(t+1) = z \mid X(t+1) = x, A(t) = a\right] = \mathsf{O}(x, a, z).$$

Notice that we allow the observation probabilities to depend on the previous action. This allows us to model situations in which the robot can actively control its sensors. Given this observation model it is now straightforward to apply Bayes rule to get

$$\mathbb{P}\left[X(t+1) = y \mid X(t) = x, A(t) = a, Z(t+1) = z\right] = \frac{\mathsf{P}(x, a, y)\mathsf{O}(y, a, z)}{\sum_{y'} \mathsf{P}(x, a, y')\mathsf{O}(y', a, z)}.$$

This can easily be extended to the case in which the robot, at time t, only knows that it is in a given state x with probability $b_t(x)$. The vector b_t is known as the *belief vector* or simply as the *belief*, and represents the belief of the agent regarding its current state. Then, given the belief at time t, b_t, we can update it according to

$$\begin{aligned} b_{t+1}(y) &\triangleq \mathbb{P}\left[X(t+1) = y \mid X(t) \sim b_t, A(t) = a, Z(t+1) = z\right] \\ &= \frac{\sum_x b_t(x)\mathsf{P}(x, a, y)\mathsf{O}(y, a, z)}{\sum_{x,y'} b_t(x)\mathsf{P}(x, a, y')\mathsf{O}(y', a, z)}. \end{aligned} \tag{1}$$

In this second part of the assignment, you will model the grid-world navigation task but including partial observability. This extended model is known as a *partially observable MDP* (POMDP). A POMDP is, therefore, a tuple $(\mathcal{X}, \mathcal{A}, \mathcal{Z}, \mathsf{P}, \mathsf{O}, r, \gamma)$, where the elements $(\mathcal{X}, \mathcal{A}, \mathsf{P}, r, \gamma)$ correspond to the underlying MDP, and $\mathcal{Z}$ and O represent the observation model: $\mathcal{Z}$ is the space of possible observations and O represents the observation probabilities.

Assignment 1.4. You will model the grid-world as a POMDP. You will also implement a simple routine to track the state of the robot in the grid-world and a simple strategy that uses the optimal policy in the underlying MDP to choose the actions in its partially observable counterpart.

(a) Launch Matlab and navigate to the folder `rlss_ass14`. Clean up the memory by running the "`clear all`" command in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass14`. You will need this file created in Assignment 1.2 to complete this assignment. If you have not completed that assignment, please request a complete `MDPLoad.m` from the lab assistant.

(c) Open the file `POMDPLoad.m` in the Matlab editor. This is the skeleton of an M-file that generates a structure containing the POMDP model for the grid world. Your goal is to implement a POMDP loader that generates the POMDP structure that represents the grid-world problem in the presence of partial observability.

You will notice that the file begins by loading the grid-world MDP, in the line "`M = MDPLoad('stochastic')`". This is the line that requires `MDPLoad.m` from the Assignment 1.2.

To complete the file, you need only to replace the entries "`???`" by adequate values. Write down the observation probabilities for each triplet (x, a, z) for the example above. We assume that the robot is only able to observe the goal state, *i.e.*, the set of possible observations consists of only two elements: "None" and "Goal". Once you have completed implementing the function, save the file.

(d) You will now implement the belief update rule in (1). Open the file `beliefUpd.m` in the Matlab editor. Follow the commented instructions and implement the belief update rule.

You have now a function (`POMDPLoad`) that returns the POMDP model for the grid world, and a function (`beliefUpd`) that, given the POMDP model, updates a belief given a new action and observation. It remains only to define how to choose the actions in the POMDP model.

(e) The action-selection rule you will implement relies on an heuristic known as the *most likely state* (MLS). The idea is to choose the state with highest probability given the current belief, and choose the corresponding optimal MDP action. Open the file `runPolicy.m` in the Matlab editor. Follow the commented instructions to implement the MLS action-selection heuristic.

(f) You are now in position to evaluate the implemented action-selection rule. To do this, in the main Matlab window start by executing "`M = POMDPLoad`", followed by "`evalPolicy(M)`". You will observe the performance of the MLS policy compared to that of the optimal MDP policy. This will look like the following message in the Matlab main window:

```
POMDP MLS EVALUATION:
=====================

Average total discounted reward per episode: 5.075 +- 1.514.
Percentage of successful trials: 0.974.

MDP OPTIMAL POLICY EVALUATION:
==============================

Average total discounted reward per episode: 5.793 +- 1.170.
Percentage of successful trials: 0.997.
```

You will notice how, in this particular environment, the performance of the MLS policy is quite reasonable if compared with that of the optimal MDP policy. However, it is important to refer that this is due to the very simple structure of the problem and it is seldom the case that the MLS heuristic leads to such a good performance.

2 Planning

In the previous section, we focused on the MDP model and some variations thereof. We studied how the different elements of the model impact the optimal policy, and treated the latter as a black box. In this section, we delve into the intricacies of such black box, and implement two standard solution methods for MDPs.

2.1 Value Iteration (◇)

Recall that associated with a policy π is a corresponding *value function*, V^π, that represents how much the robot expects to receive by choosing its actions according to π, depending on the initial state. In other words,

$$V^\pi(x) = \mathbb{E}\left[\sum_t \gamma^t R(t) \mid X(0) = x\right]. \tag{2}$$

By expanding the summation, it is possible to rewrite the expression above recursively as

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_y \mathsf{P}(x, \pi(x), y) V^\pi(y). \tag{3}$$

Similarly, for the optimal policy, π^*, we have

$$V^*(x) = \max_a \left[r(x, a) + \gamma \sum_y \mathsf{P}(x, a, y) V^\pi(y)\right].$$

The two above expressions suggest an iterative process to compute both V^π and V^*. Using (2), we obtain the recursion

$$V^{(k+1)}(x) = r(x, \pi(x)) + \gamma \sum_y \mathsf{P}(x, \pi(x), y) V^{(k)}(y),$$

where $V^{(k)}$ denotes the estimate for V^π at the kth iteration of the algorithm. Similarly, using (3), we get

$$V^{(k+1)}(x) = \max_a \left[r(x, a) + \gamma \sum_y \mathsf{P}(x, a, y) V^{(k)}(y)\right].$$

The first of the above iterations is used to *evaluate* a given policy π. The second is used to compute the value function associated with the optimal policy, V^*. From this function, the optimal policy can be recovered as

$$\pi(x) = \arg\max_a \left[r(x, a) + \gamma \sum_y \mathsf{P}(x, a, y) V^*(y)\right].$$

Notice that, in order to compute π^* from V^* it is still necessary to know r and P. To avoid this, we define the function

$$Q^\pi(x, a) = r(x, a) + \gamma \sum_y \mathsf{P}(x, a, y) V^\pi(y) \tag{4}$$

which, for the optimal policy π^*, becomes

$$Q^*(x, a) = r(x, a) + \gamma \sum_y \mathsf{P}(x, a, y) V^*(y).$$

Clearly, the following relations hold

$$\begin{aligned} V^*(x) &= \max_a Q^*(x, a) \\ \pi^*(x) &= \arg\max_a Q^*(x, a). \end{aligned}$$

Replacing the relation between Q^* and V^* into the definition of Q^*, we also obtain a recursive relation,

$$Q^*(x, a) = r(x, a) + \gamma \sum_y \mathsf{P}(x, a, y) \max_b Q^*(y, b),$$

from where we immediately get the iterative update

$$Q^{(k+1)}(x,a) = r(x,a) + \gamma \sum_y \mathsf{P}(x,a,y) \max_b Q^{(k)}(y,b). \tag{5}$$

In this section, you will implement the iterations above.

Evaluating a Policy

We start by implementing the iteration used to evaluate a policy.

Assignment 2.1. To test your implementation of the value iterations above, we will use the grid-world models implemented in Assignment 1.2.

(a) Launch Matlab and navigate to the folder `rlss_ass21`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass21`. You will need this file created in Assignment 1.2 to complete this assignment. If you have not completed that assignment, please request a complete `MDPLoad.m` from the lab assistant.

(c) Open the file `VI.m` in the Matlab editor. This is the skeleton of an M-file implementing value iteration for policy evaluation. You are supposed to fill-in an adequate expression to implement the value iteration. Define a policy as a column vector `Pol` whose entries `i` correspond to the desired action at state `i`.

(d) Once you have completed implementing value iteration for policy evaluation, save the file.

(e) In Matlab main window, define an arbitrary policy `Pol` using the above format. For the different environments in Assignment 1.2, evaluate your policy (notice that one of the environments has one less state). Recall that you must load the MDP model by executing "`M = MDPLoad(<string>)`", where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`. You can then test your function by executing "`V = VI(M, Pol)`", where `Pol` is the policy you want to evaluate.

(f) For each of the environments in Assignment 1.2, evaluate the corresponding optimal policy using your function (you must first define the optimal policy using the format defined above). Compare the results with those obtained in the previous step.

(g) For each of the value-functions computed in the previous step, compute the corresponding Q-function using (4). Verify that, at each state, the actions that attain the maximum Q-value are, indeed, optimal actions.

Computing the Optimal Policy

You will now implement value iteration to compute the optimal policy.

Assignment 2.2. This assignment is very similar to Assignment 2.1. If you completed the latter, you should be able to quickly finish this one.

(a) Launch Matlab and navigate to the folder `rlss_ass22`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass22`. You will need this file created in Assignment 1.2 to complete this assignment. If you have not completed that assignment, please request a complete `MDPLoad.m` from the lab assistant.

(c) Open the file `VI.m` in the Matlab editor. This is the skeleton of an M-file implementing value iteration to compute Q^*. You are supposed to fill-in an adequate expression to implement the value iteration.

(d) Once you have completed implementing value iteration, save the file.

(e) Repeat all the tests conducted on Assignment 1.2. In particular, load the different models by executing "`M = MDPLoad(<string>)`", where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`. For each of the models, execute "`plotPolicy(M)`". Verify that the policy is as expected.

2.2 Policy Iteration ($\star$)

The methods described in the previous subsection are globally known as *value iteration*, since they evaluate a policy or compute the optimal policy by estimating iteratively the corresponding value/Q-function. These methods are guaranteed to converge to the desired functions, as $k \to \infty$.

Notice, however, that all models considered in here admit a finite set of actions, $\mathcal{A}$. This means that there is only a finite number of possible policies. This observation leads to a different class of methods to compute the optimal policy. In this process, we will also get to a more efficient process of evaluating a policy. This class of methods is generally known as *policy iteration*, and proceeds as follows.

The algorithm starts with some arbitrary policy $\pi^{(0)}$. It evaluates this policy, computing the corresponding value/Q-function, and uses this evaluation to obtain a new, improved policy $\pi^{(1)}$. By iterating through this process, we are guaranteed to attain the optimal policy after a finite number of steps. However, each iteration of this algorithm performs a policy evaluation which is, in turn, an iterative method. To obtain a more efficient method for policy evaluation, we rewrite the recursion for V^π:

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_y \mathsf{P}(x, \pi(x), y) V^\pi(y).$$

Because the state-space $\mathcal{X}$ is finite, V^π can be represented as a vector. he same can be said about $r(x, \pi(x))$. On the other hand, $\mathsf{P}(x, \pi(x), y)$ can be represented as a square matrix. Let $\mathbf{v}_\pi$, $\mathbf{r}_\pi$ and $\mathbf{P}_\pi$ denote this vector-matrix interpretation of $V^\pi(\cdot)$, $r(\cdot, \pi)$ and $\mathsf{P}(\cdot, \pi, \cdot)$, respectively. Then, the above equation becomes

$$\mathbf{v}_\pi = \mathbf{r}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}.$$

We can rewrite this equation as

$$\mathbf{v}_\pi = (\mathbf{I} - \gamma \mathbf{P}_\pi)^{-1} \mathbf{r}_\pi.$$

The expression above states that V^π is the solution to a simple linear system of equations, which can be solved efficiently.

Assignment 2.3. In this assignment you will implement the policy iteration algorithm.

(a) Launch Matlab and navigate to the folder `rlss_ass23`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass23`. You will need this file created in Assignment 1.2 to complete this assignment. If you have not completed that assignment, please request a complete `MDPLoad.m` from the lab assistant.

(c) Open the file `PI.m` in the Matlab editor. This is the skeleton of an M-file implementing policy iteration to compute the optimal policy. You are supposed to fill-in the missing pieces to implement policy iteration.

(d) Once you have completed implementing policy iteration, save the file.

(e) Repeat all the tests conducted on Assignment 1.2. In particular, load the different models by executing “`M = MDPLoad(<string>)`”, where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`. For each of the models, execute “`plotPolicy(M)`”. Verify that the policy is as expected. Compare the running times and number of iterations with those obtained in the previous assignment using value iteration.

3 Learning

So far, we introduced the fundamental MDP model as well as dynamic programming algorithms to solve this class of problems. In this section, we move to the actual *learning*. The main difference between the learning and the planning approaches is that, while in the latter the MDP model is assumed known, in the former it is not. Instead, the robot must *learn* the optimal policy by *interacting* with the environment. This interaction will provide the robot with enough information to figure out the optimal policy.

In this section we cover *temporal difference learning* methods, which can be seen as stochastic versions of the dynamic programming algorithms from the previous section. We also cover other classes of methods, namely *model-based methods* and *policy-gradient methods*.

3.1 Temporal Difference Learning ($\diamond$)

To introduce temporal difference methods, let us first consider the following problem. Suppose that we want to compute the mean μ of an unknown distribution, and we have available a set of N samples $\{w_1, \dots, w_N\}$ from that distribution. One possible estimate for the mean is the *sample mean*, given by

$$\hat{\mu}_N = \frac{1}{N}\sum_{i=1}^{N} w_i.$$

If the samples $\{w_i\}$ arrive sequentially, this very same mean can be computed incrementally as

$$\hat{\mu}_{N+1} = \hat{\mu}_N + \frac{1}{N+1}\big(w_{N+1} - \hat{\mu}_N\big). \tag{6}$$

Going back to the recursive relations for V^π and Q^* introduced in the previous section, we have

$$\begin{aligned} V^\pi(x) &= r(x,\pi(x)) + \gamma \sum_y \mathsf{P}(x,\pi(x),y)V^\pi(y) \\ &= \mathbb{E}_{Y\sim \mathsf{P}_\pi}\left[r(x,\pi(x)) + \gamma V^\pi(Y)\right] \\ Q^*(x,a) &= r(x,a) + \gamma \sum_y \mathsf{P}(x,a,y)\max_b Q^*(y,b) \\ &= \mathbb{E}_{Y\sim \mathsf{P}_a}\left[r(x,a) + \gamma \max_b Q^*(Y,b)\right], \end{aligned}$$

where P_π corresponds to the distribution $\mathsf{P}(x,\pi(x),\cdot)$ and P_a corresponds to the distribution $\mathsf{P}(x,a,\cdot)$. From these expressions, it is clear that the dynamic programming iterations in the previous section merely update the estimates for V^π and Q^* by computing the expectations outlined above. Therefore, by using a similar recursion to (6), we get the following alternative iterations to compute V^π and Q^*

$$V^{(t+1)}(x_t) = V^{(t)}(x_t) + \alpha_t\big(r_t + \gamma V^{(t)}(x_{t+1}) - V^{(t)}(x_t)\big) \tag{7}$$

$$Q^{(t+1)}(x_t,a_t) = Q^{(t)}(x_t,a_t) + \alpha_t\big(r_t + \gamma \max_b Q^{(t)}(x_{t+1},b) - Q^{(t)}(x_t,a_t)\big), \tag{8}$$

where $\{\alpha_t\}$ is a step-size sequence, $\{x_t\}$ is a sample sequence of states observed when following policy π and $\{a_t\}$ and $\{r_t\}$ are the corresponding action and reward sequences.

It is worth explaining with a bit more detail what the above expressions imply. The update rule (7) is used for policy evaluation. Given a policy π to be evaluated, the robot moves around the environment following this policy. At every time step t, it observes the current state, $X(t)$, and chooses the action according to π, *i.e.*, $A(t) = \pi(X(t))$. It then observes the corresponding reward $R(t)$ and next state $X(t+1)$, and the process continues. The transition triplet $(X(t), R(t), X(t+1))$ is used to update the current estimate of V^π, denoted $V^{(t)}$ according to (7). Note that only the component $V^{(t)}(X(t))$ is updated.

As for (8), the robot moves around the environment in an exploratory manner, following some exploratory policy π. As in the previous case, at every time step t it observes the current state, $X(t)$, and chooses the action according to π, *i.e.*, $A(t) = \pi(X(t))$. It then observes the corresponding reward $R(t)$ and next state $X(t+1)$, and the process continues. It then uses the transition tuple $(X(t), A(t), R(t), X(t+1))$ to update the current estimate of Q^*, denoted $Q^{(t)}$ according to (8). Again note that only the component $Q^{(t)}(X(t), A(t))$ is updated.

The two above update equations correspond to two of the best known reinforcement learning algorithms in the literature, to know, TD(0) and Q-learning. In this section you will implement these two algorithms and test them on the environments from Assignment 1.2.

TD(0)

We start by implementing the TD(0) algorithm introduced above.

Assignment 3.1. To test your implementation of TD(0), we will use the grid-world models implemented in Assignment 1.2.

(a) Launch Matlab and navigate to the folder `rlss_ass31`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass31`. You will need this file created in Assignment 1.2 to complete this assignment. If you have not completed that assignment, please request a complete `MDPLoad.m` from the lab assistant.

(c) Open the file `TDLearning.m` in the Matlab editor. This is the skeleton of an M-file implementing the TD-learning algorithm outlined above. You are supposed to fill-in the missing lines and implement TD-learning. A step-size sequence has already been defined for you, which you can access as the variable `SS`.

Adopt the format described in Assignment 2.1 to represent a policy and implement value-iteration accordingly.

Notice also that you will need to generate transitions of the MDP. To this purpose, you can include the line "`[Xnew, Rnew] = MDPStep(M, X, A)`", which generates a transition from state `X` to some state `Xnew` by executing action `A`. This, in turn, yields a reward `Rnew`. For your reference, `X` and `A` can be column-vectors containing multiple states and actions. Note that the actions A used in TD-learning must be generated using the provided policy, `Pol`.

(d) It is important that, when sampling the states to be used in your TD-learning algorithm, you ensure that *all states* are sufficiently sampled, or the algorithm may not be able to properly estimate V^π in all states.

(e) Once you have completed implementing TD-learning, save the file.

(f) In Matlab main window, define an arbitrary policy `Pol` using the above format. For the different environments in Assignment 1.2, evaluate your policy (notice that one of the environments has one less state). Recall that you must load the MDP model by executing "`M = MDPLoad(<string>)`", where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`. You can then test your function by executing "`V = TDLearning(M, Pol)`", where `Pol` is the policy you want to evaluate.

(g) For each of the environments in Assignment 1.2, evaluate the corresponding optimal policy using your function (you must first define the optimal policy using the adequate format). Compare the results with those obtained, for example, in Assignment 1.2.

(h) For each of the value-functions computed in the previous step, compute the corresponding Q-function using (4). Verify that, at each state, the actions that attain the maximum Q-value are, indeed, optimal actions.

Q-learning

In this assignment, you will implement the Q-learning algorithm defined by the iteration (8).

Assignment 3.2. This assignment is very similar to Assignment 3.1. If you completed the latter, you should be able to quickly finish this one.

(a) Launch Matlab and navigate to the folder `rlss_ass32`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass32`. You will need this file created in Assignment 1.2 to complete this assignment. If you have not completed that assignment, please request a complete `MDPLoad.m` from the lab assistant.

(c) Open the file `QLearning.m` in the Matlab editor. This is the skeleton of an M-file implementing Q-learning to compute Q^*. You are supposed to fill-in the missing lines and implement TD-learning. A step-size sequence has already been defined for you, which you can access as the variable `SS`.

Notice that, once again, you need to generate transitions of the MDP. To this purpose, you can include the line "`[Xnew, Rnew] = MDPStep(M, X, A)`", which generates a transition from state `X` to some state `Xnew` by executing action `A`. This, in turn, yields a reward `Rnew`. For your reference, `X` and `A` can be column-vectors containing multiple states and actions. Note that, unlike what happened with TD-learning, with Q-learning you can use any policy to generate the actions.

(d) As in TD-learning, remember that you must ensure that *all states and actions* are sufficiently sampled, or the algorithm may not be able to properly estimate Q^*.

(e) Once you have completed implementing Q-learning, save the file.

(f) Repeat all the tests conducted on Assignment 1.2. In particular, load the different models by executing "`M = MDPLoad(<string>)`", where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`. For each of the models, execute "`plotPolicy(M)`". Verify that the policy is as expected.

(g) Repeat the last step a couple of times, and note that the optimal policy computed varies (although it is always optimal). This is due to the fact that the policy computed depends on the particular sampled trajectory, which makes some of the Q-values be better estimated than others.

3.2 More on Reinforcement Learning ($\star$)

We now delve a little deeper to variations and alternatives to the RL algorithms presented above.

TD(λ)

In the TD(0) algorithm, at every time-step t the robot updates its estimate of V^π based on the latest information obtained from the environment. This information is summarized by the transition triplet $(X(t), R(t), X(t+1))$. However, the information from each such transition is used only in one iteration and is then discarded. It seems reasonable that, in order to potentially improve the performance of the algorithm, these transitions can be used in multiple updates. This is the idea behind TD(λ), a generalization of TD(0).

To understand where TD(λ) comes from, let us again rewrite the recursion for V^π.

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_y \mathsf{P}(x, \pi(x), y) V^\pi(y).$$

This recursion was obtained by expanding the summation in the definition of V^π to separate the immediate reward, $r(x, a)$, from the remaining residual reward, $V^\pi(y)$. In terms of expectation, we have

$$\begin{aligned} V^\pi(x) &= \mathbb{E}\left[R(0) + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} R(t) \mid X(0) = x\right] \\ &= \mathbb{E}\left[R(0) + \gamma V^\pi(X(1)) \mid X(0) = x\right], \end{aligned}$$

Clearly, it is possible to further expand the summation, to yield

$$\begin{aligned} V^\pi(x) &= \mathbb{E}\left[R(0) + \gamma V^\pi(X(1)) \mid X(0) = x\right] \\ &= \mathbb{E}\left[R(0) + \gamma R(1) + \gamma^2 V^\pi(X(2)) \mid X(0) = x\right] \\ &\ \ \vdots \\ &= \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R(t) + \gamma^{T+1} V^\pi(X(T+1))\right]. \end{aligned}$$

Let now λ be a real-valued scalar taking values in $[0, 1]$. Clearly, we can combine all the above expressions to yield

$$V^\pi(x) = (1 - \lambda) \sum_T \lambda^T \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R(t) + \gamma^{T+1} V^\pi(X(T+1))\right].$$

After some shuffling, the above expression finally becomes

$$V^\pi(x) = \mathbb{E}\left[\sum_{t=0}^{\infty} (\gamma\lambda)^t \left[R(t) + \gamma V^\pi(X(t+1)) - V^\pi(X(t))\right]\right] + V^\pi(x).$$

Using the same underlying principle used to derive TD(0), this new recursive expression for V^π can now be turned into the following iterative update

$$V^{(t+1)}(x_t) = V^{(t)}(x_t) + \alpha_t \sum_{\tau=t}^{\infty} \left(r_\tau + \gamma V^{(t)}(x_{\tau+1}) - V^{(t)}(x_\tau)\right). \tag{9}$$

This update uses all samples from time t *into the future* and is, therefore, not very practical to implement. Fortunately, the above algorithm can be equivalently implemented by the following update

$$\begin{aligned} Z^{(t+1)}(x) &= \lambda\gamma Z^{(t)}(x) + \delta_x(X(t)) \\ V^{(t+1)}(x) &= V^{(t)}(x) + \alpha_t Z^{(t)}(x)\left(r_t + \gamma V^{(t)}(x_{t+1}) - V^{(t)}(x_t)\right), \end{aligned} \tag{10}$$

where each $Z^{(t)}$ is an *eligibility vector* and $\delta_x(y)$ represents the Kronecker delta function. Note that, in the above update, *all components of* $V^{(t)}$ are updated at every iteration, as intended.

The parameter λ represents a *forgetting factor*, that defines how past transitions affect the current update. Notice that, for $\lambda = 0$, we recover TD(0).

In the next assignment, you will implement this more general version of TD-learning. This assignment is, in all aspects, similar to Assignment 3.1, except that a more general version of TD-learning is implemented.

Assignment 3.3. To test your implementation of TD(λ), we will use the grid-world models implemented in Assignment 1.2.

(a) Launch Matlab and navigate to the folder `rlss_ass33`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass33`. You will need this file created in Assignment 1.2 to complete this assignment. If you have not completed that assignment, please request a complete `MDPLoad.m` from the lab assistant.

(c) Open the file `TDLearning.m` in the Matlab editor. This is the skeleton of an M-file implementing the TD-learning algorithm outlined above. You are supposed to fill-in the missing lines and implement TD-learning. As before, a step-size sequence has already been defined for you, which you can access as the variable `SS`.

Remember that you will need to generate transitions of the MDP. To this purpose, you can include the line "`[Xnew, Rnew] = MDPStep(M, X, A)`", which generates a transition from state `X` to some state `Xnew` by executing action `A`. This, in turn, yields a reward `Rnew`. For your reference, `X` and `A` can be column-vectors containing multiple states and actions. Note that the actions A used in TD-learning must be generated using the provided policy, `Pol`.

(d) As with TD(0), it is important that you ensure that *all states* are sufficiently sampled, or the algorithm may not be able to properly estimate V^π in all states.

(e) Once you have completed implementing TD-learning, save the file.

(f) In Matlab main window, define an arbitrary policy `Pol` using the format described in Assignment 2.1. For the different environments in Assignment 1.2, evaluate your policy (notice that one of the environments has one less state). Recall that you must load the MDP model by executing "`M = MDPLoad(<string>)`", where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`. You can then test your function by executing "`V = TDLearning(M, Pol)`", where `Pol` is the policy you want to evaluate.

(g) For each of the environments in Assignment 1.2, evaluate the corresponding optimal policy using your function (you must first define the optimal policy using the appropriate format). Compare the results with those obtained, for example, in Assignment 1.2.

(h) For each of the value-functions computed in the previous step, compute the corresponding Q-function using (4). Verify that, at each state, the actions that attain the maximum Q-value are, indeed, optimal actions.

Model-based Learning

The challenge of RL, which sets it apart from the planning approaches such as dynamic programming, is to compute the optimal policy for an MDP *without requiring prior knowledge* on the MDP parameters, namely on P and r. And, as you can easily check from the implementations of TD(λ) and Q-learning, these algorithms do not require explicit knowledge of the MDP model but only whatever information can be obtained by sampling.

The methods relying on temporal differences (TD(λ) and Q-learning) can be seen as stochastic approximations to the dynamic programming updates studied in Section 2. In a sense, they implement sample-based versions of the dynamic programming iterations.

However, a more straightforward approach can be devised to deal with the lack of knowledge on the MDP parameters. Given the experience obtained by the robot when interacting with the environment, it is possible to *estimate the MDP parameters* and then use these to perform the dynamic programming iterations explicitly. Such methods, that build an explicit model of the MDP from the sample data, are known as *model-based*, in contrast with model-free methods like TD(λ) and Q-learning. In fact, as you may have realized, TD(λ) and Q-learning do not explicitly estimate the MDP model but directly estimate the target function (either V^π or Q^*).

We now implement one such model-based method that can be used to compute the optimal Q-function. Let $\mathsf{P}^{(t)}$ denote the estimate of the transition probabilities at time t (*i.e.*, , after the robot has experienced t transitions and has conducted as many updates). Suppose that, at time t, $X(t) = x_t$, $A(t) = a_t$ and $X(t+1) = x_{t+1}$. The estimate $\mathsf{P}^{(t)}$ can then be updated as

$$\mathsf{P}^{(t+1)}(x_t, a_t, y) = \left(1 - \frac{1}{N_t(x_t, a_t)}\right) \mathsf{P}^{(t)}(x_t, a_t, y) + \frac{1}{N_t(x_t, a_t)} \delta_y(x_{t+1}), \tag{11}$$

where, as before, $\delta_x(y)$ denotes the Kronecker delta function, and $N_t(x, a)$ denotes the number of visits to the pair (x, a) up to and including time t. Note that, at time t, only the transition probabilities $\mathsf{P}^{(t)}(x_t, a_t, \cdot)$ are updated, to reflect the new transition just observed. As for the reward function, we have

$$r^{(t+1)}(x_t, a_t) = r^{(t)}(x_t, a_t) + \frac{1}{N_t(x_t, a_t)} \big(r_t - r^{(t)}(x_t, a_t)\big).$$

One practical note: we have implemented the reward function as being deterministic, which means that the above iteration can be replaced by a simple assignment:

$$r(x_t, a_t) = r_t.$$

Once the parameters P and r are properly estimated, we can now use them to implement VI. An interesting aspect of the model-based approaches, however, is that these iterations can be performed as the estimates for P and r are updated. So after updating P and r according to (11) and (3.2), the following update could immediately follow:

$$Q^{(t+1)}(x_t, a_t) = r^{(t+1)}(x_t, a_t) + \gamma \sum_y \mathsf{P}^{(t+1)}(x_t, a_t, y) \max_b Q^{(t)}(y, b). \tag{12}$$

In this assignment, we implement the above algorithm, which is a very simplified implementation of a general class of methods known as *real-time dynamic programming.*

In this assignment, you will implement RTDP as defined by the update equations (11) through (12).

Assignment 3.4. To test your implementation of TD(0), we will use the grid-world models implemented in Assignment 1.2.

(a) Launch Matlab and navigate to the folder `rlss_ass34`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass34`. You will need this file created in Assignment 1.2 to complete this assignment. If you have not completed that assignment, please request a complete `MDPLoad.m` from the lab assistant.

(c) Open the file `RTDP.m` in the Matlab editor. This is the skeleton of an M-file implementing RTDP. It estimates the model parameters P and r and uses these to estimate Q^*. You are supposed to fill-in the missing lines so as to implement RTDP. Notice that, unlike the previous incremental algorithms, the step-size sequence in this case is naturally defined from the number of visits to each state-action pair. Note that, since P is a stochastic matrix, it should be initialized as a stochastic matrix and, at each step, ensure that always remains so.

As in the previous methods, you need to generate transitions of the MDP. To this purpose, you can include the line "`[Xnew, Rnew] = MDPStep(M, X, A)`", which generates a transition from state `X` to some state `Xnew` by executing action `A`. This, in turn, yields a reward `Rnew`. For your reference, `X` and `A` can be column-vectors containing multiple states and actions. Note that, unlike what happened with TD-learning, with Q-learning you can use any policy to generate the actions.

(d) As in TD-learning and Q-learning, remember that you must ensure that *all states and actions* are sufficiently sampled. In this particular case, insufficient visits cause the estimates of P to be wrongly estimated and this may lead to wrong Q^* estimates.

(e) Once you have completed implementing RTDP, save the file.

(f) Repeat all the tests conducted on Assignment 1.2. In particular, load the different models by executing "`M = MDPLoad(<string>)`", where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`. For each of the models, execute "`plotPolicy(M)`". Verify that the policy is as expected.

4 Other

In the previous sections, you had the opportunity to interact with MDPs as well as with dynamic programming and reinforcement learning approaches to solve MDPs. In these sections, we always assumed the reward function r to be defined in advance (even if unknown, in the RL case) so as to encode some desired task. However, it is seldom easy for a "human user" to encode a desired task in the form of a reward function. Instead, it is often easier to *demonstrate* the task to the robot, from which the latter can retrieve the desired policy and/or reward function. In particular, there are several attractive features in recovering the reward function instead of the target policy directly. For example, the reward function can accommodate for differences between the demonstrator and the learner.

If the paradigm of RL is "learning by trial-and-error", we now move to a setting of *learning from demonstration*. Within the MDP formalism, one approach to this problem is *inverse reinforcement learning* (IRL). In this section we implement two simple solution methods for the IRL problem.

4.1 Inverse Reinforcement Learning

Inverse reinforcement learning deals with the problem of recovering a reward function given a policy. In this sense, it deals with the *inverse problem* addressed in the dynamic programming and reinforcement learning approaches discussed in previous sections. This difference is illustrated in Fig. 3.

IRL from Complete Policies

Let π be some given policy, and $(\mathcal{X}, \mathcal{A}, \mathsf{P}, r, \gamma)$ a MDP for which r is unknown. It is known, however, that π is one optimal policy for the MDP $(\mathcal{X}, \mathcal{A}, \mathsf{P}, r, \gamma)$. This means that, for all $a \in \mathcal{A}$ and all $x \in \mathcal{X}$,

$$Q^*(x, \pi(x)) \geq Q^*(x, a).$$

For simplicity, let us assume that r depends only on x, *i.e.*, it is constant for all $a \in \mathcal{A}$. Then, the above expression becomes

$$r(x) + \gamma \sum_y \mathsf{P}(x, \pi(x), y) V^*(y) \geq r(x) + \gamma \sum_y \mathsf{P}(x, a, y) V^*(y)$$

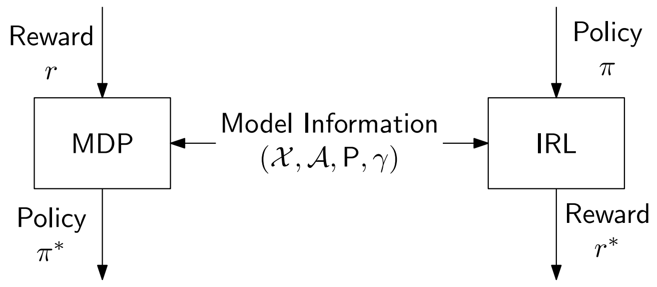


Figure 3: Comparison between RL and IRL.

or, equivalently,

$$\sum_y \big(\mathsf{P}(x, \pi(x), y) - \mathsf{P}(x, a, y)\big) V^*(y) \geq 0.$$

In vector form, this becomes

$$(\mathbf{P}_\pi - \mathbf{P}_a)(\mathbf{I} - \gamma \mathbf{P}_\pi)^{-1}\mathbf{r} \succeq \mathbf{0},$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{P}_a$ denotes the matrix with (x, y) component $\mathsf{P}(x, a, y)$, $\mathbf{P}_\pi$ denotes the matrix with (x, y) component $\mathsf{P}(x, \pi(x), y)$ and $\mathbf{r}$ denotes the column vector with xth component $r(x)$. The above expression provides a constraint that all solutions verify. We now choose the particular solution that maximizes the distance between the optimal action $\pi(x)$ and the second best action, *i.e.*, that maximizes the functional

$$F(\mathbf{r}) = \mathbf{1}^\top \sum_a (\mathbf{P}_\pi - \mathbf{P}_a)(\mathbf{I} - \gamma \mathbf{P}_\pi)^{-1}\mathbf{r},$$

where $\mathbf{1}^\top$ denotes the all-ones row-vector.

In this assignment, you will solve the IRL problem by solving the following linear problem

$$\begin{aligned} \max_{\mathbf{r}} \quad & F(\mathbf{r}) - \lambda \left\|\mathbf{r}\right\|_1 \\ \text{s.t.} \quad & (\mathbf{P}_\pi - \mathbf{P}_a)(\mathbf{I} - \gamma \mathbf{P}_\pi)^{-1}\mathbf{r} \succeq \mathbf{0}, \qquad \text{for all } a \\ & \mathbf{r} \succeq \mathbf{0} \\ & \mathbf{r} \preceq \mathbf{1}, \end{aligned} \tag{13}$$

where we added the 1-norm penalty to privilege sparse solutions.

Assignment 4.1. You will implement the IRL algorithm outlined above and analyze some of its properties.

(a) Launch Matlab and navigate to the folder `rlss_ass41`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass41`. Copy also the file `VI.m` from the folder `rlss_ass22` to `rlss_ass41`. You will need these files to complete this assignment. If you have not completed the corresponding assignments, please request the complete `MDPLoad.m` and `VI.m` from the lab assistant.

(c) Open the file `IRL.m` in the Matlab editor. This is the skeleton of an M-file that implements the above IRL algorithm via linear programming. You will note that some of the steps necessary to build the linear program have already been implemented for you.

To solve the IRL problem, you will use the Matlab function "`linprog`", whose syntax is outlined in the comments in the file. You can also type "help linprog" in the main Matlab window for help on usage and functionality. To solve the IRL problem, you need to define the goal function and constraints in (13), according to the syntax of "`linprog`".

(d) Once you have completed the definition of the IRL progrs, save the file.

(e) In the Matlab window, load any of the different models studied in Assignment 1.3, by executing "`M = MDPLoad(<string>)`", where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`.

(f) Once you loaded the file, execute the command "`[Q, Pol] = VI(M)`". This command will compute both the optimal Q-function for M and the corresponding optimal policy, `Pol`.

(g) To test your IRL implementation, type "`R = IRL(M, Pol)`". Compare the reward function obtained with the one in the model (you can access the reward in the model by typing "`M.r`").

(h) Set the reward of your MDP to the one computed using IRL, by executing "`M.r = R`". This replaces the previous reward function with the one computed by IRL.

(i) You will now verify that the policy for the new reward is as expected. To this purpose, execute "`plotPolicy(M)`" and verify that the policy is as expected. Note that, at the goal state, the selected action is not $\emptyset$ but one other action that does not change the state of the robot. Can you explain this?

A Maximum Likelihood Approach to IRL

In this last assignment, we propose a slightly different view of IRL. In the formulation of the IRL problem described in so far, the robot was provided with a complete and correct description of the policy whose optimal reward function is sought. However, in practical problems, it may occur that the teacher is unable to provide a correct description of the desired policy. How can the same problem be tackled, if we assume that the teacher provides a demonstration of the desired policy but in which we allow some "mistakes"?

To address this situation, we now assume that the robot is given a demonstration of the desired policy, consisting in a set of pairs $\mathcal{D} = \{(x_i, a_i), i = 1, \ldots, N\}$, with $x_i \in \mathcal{X}$ and $a_i \in \mathcal{A}$. We assume that the teacher provides a comprehensive demonstration and that the pairs $(x_i, a_i), i = 1, \ldots, N$, are generated by a demonstrator "described" by an MDP $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathsf{P}, r^*, \gamma)$. The purpose of the robot is to recover the reward function, r^*.

In our model of the demonstrator, we assume that the demonstrator sometimes makes mistakes, but that, when a mistake occurs, the demonstrator is more likely to choose actions closer to the optimal action in terms of value than actions that are further from the optimal action in terms of value. In particular, we assume that the demonstrator chooses its actions according to the distribution

$$\mathbb{P}\left[A = a \mid X = x, r^*\right] = \frac{\exp(\eta Q^*(x, a)}{\sum_b \exp(\eta Q^*(x, b)},$$

where η is a *confidence parameter* that translates the degree of certainty on the demonstrator's choices, and Q^* is the optimal Q-function for the MDP $\mathcal{M}$. This means that the likelihood of the demonstrator for a given reward function r is given by

$$\mathbb{P}\left[\mathcal{D} \mid r\right] = \prod_{i=1}^{N} \frac{\exp(\eta Q_r(x_i, a_i)}{\sum_b \exp(\eta Q_r(x_i, b)},$$

where Q_r denotes the optimal Q-function for the general MDP $(\mathcal{X}, \mathcal{A}, \mathsf{P}, r, \gamma)$.

The maximum likelihood estimate for the policy followed by the demonstrator, henceforth denote as π_{ML}, is easily given by the empirical frequencies,

$$\pi_{\mathrm{ML}}(x, a) = \frac{N(x, a)}{\sum_b N(x, b)},$$

where $N(x, a)$ denotes the number of times that the pair (x, a) occurred in $\mathcal{D}$. It is straightforward to compute one Q-function that yields the above policy (note that this selection is not unique).

One example is

$$Q_{\mathrm{ML}}(x, a) = \frac{1}{\eta} \log(N(x, a)).$$

Finally, we can now estimate a reward function that maximizes the likelihood of the demonstration, by inverting the recursive equation (4), yielding

$$r_{\mathrm{ML}}(x, a) = Q_{\mathrm{ML}}(x, a) - \gamma \sum_{y} \mathsf{P}(x, a, y) \max_{b} Q_{\mathrm{ML}}(y, b).$$

One particularly appealing feature of this method is that is allows an analytical solution, thus making it particularly efficient to solve.

Assignment 4.2. In this last assignment you will implement the maximum likelihood IRL algorithm outlined above.

(a) Launch Matlab and navigate to the folder `rlss_ass42`. Clean up the memory by executing "`clear all`" in the Matlab window. Execute the command "`init`".

(b) Copy the file `MDPLoad.m` from the folder `rlss_ass12` to `rlss_ass42`. Copy also the file `VI.m` from the folder `rlss_ass22` to `rlss_ass42`. You will need these files to complete this assignment. If you have not completed the corresponding assignments, please request the complete `MDPLoad.m` and `VI.m` from the lab assistant.

(c) Open the file `MLIRL.m` in the Matlab editor. This is the skeleton of an M-file that implements the above IRL algorithm. Due to its analytical nature, this algorithm is very simple to implement.

(d) Once you have completed implemented the MLIRL program, save the file.

(e) In the Matlab window, load any of the different models studied in Assignment 1.3, by executing "`M = MDPLoad(<string>)`", where `<string>` can be either `'deterministic'`, `'stochastic'`, `'soft_obstacle'` or `'hard_obstacle'`.

(f) Once you loaded the file, execute the command "`D = getDemo(M)`". This command will generate a demonstration according to the above model.

(g) To test your IRL implementation, type "`R = MLIRL(M, D)`". Compare the reward function obtained with the one in the model (you can access the reward in the model by typing "`M.r`").

(h) Set the reward of your MDP to the one computed using IRL, by executing "`M.r = R`". This replaces the previous reward function with the one computed by IRL.

(i) You will now verify that the policy for the new reward is as expected. To this purpose, execute "`plotPolicy(M)`" and verify that, unlike the previous IRL algorithm, MLIRL is able to selected the action $\emptyset$ in the goal state.

References

[1] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. Technical Report UM-CS-1993-002, Dept. Computer Science, University of Massachusetts at Amherst, 1993.

[2] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

[3] A. Cassandra. Optimal policies for partially observable Markov decision processes. Technical Report CS-94-14, Dept. Computer Sciences, Brown University, 1994.

[4] A. Ng and S. Russel. Algorithms for inverse reinforcement learning. In *Proc. 17th Int. Conf. Machine Learning*, pages 663–670, 2000.

[5] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., 1994.

[6] D. Ramachandran and E. Amir. Bayesian inverse reinforcement learning. In *Proc. 20th Int. Joint Conf. Artificial Intelligence*, pages 2586–2591, 2007.