Lab : Reinforcement Learning

In this lab, we will solve a control problem in the Markov Decision Problem (MDP) framework, using Dynamic Programming (DP) and Reinforcement Learning (RL) techniques.

1 Markov Decision Problems

Suppose we have a robot in a simple maze and we want it to learn how to reach a goal position. The maze is represented by a grid, each cell is a possible state for the robot. We choose a 4×4 grid and number the states from 1 to 16. Our robot will start in state 1 and the goal position will be state 16. The robot can choose among 5 actions : going north (N), south (S), west (W), east (E), or not moving (NoOp). When the robot has reached the goal position and does not move, we give him an abstract reward.

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. The shaded cell corresponds to the goal location.

More formally, our problem is a Markov Decision Problems (MDP) which is described by a tuple $(\mathcal{X}, \mathcal{U}, \mathcal{P}_0, \mathcal{P}, r, \gamma)$ where $\mathcal{X}$ is the state-space, $\mathcal{U}$ the action-space, $\mathcal{P}_0$ the distribution of the initial state, $\mathcal{P}$ the transition function, r the reward function, and $\gamma \in [0, 1]$ a parameter called the discount factor.

The state in which the robot starts, denoted x_0, is drawn from the initial state distribution $\mathcal{P}_0$. We will choose it such that the robot always starts in state 1. Then, given the state x_t at time t and the action u_t at that same time, the next state x_{t+1} depends only on x_t and u_t according to the transition function $\mathcal{P}$:

$$\mathcal{P}(x_t, u_t, x_{t+1}) = \Pr(x_{t+1}|x_t, u_t)$$

If the robot tries to get out of the grid (i.e. if he uses action N in state 1), he will stay in his current state. The robot also receive a reward r_t according to the reward function $r(x_t, u_t)$, which depends on the state x_t and the action u_t. We choose to give a reward 1 when the robot is in state 16 and does the action NoOp, and 0 for any other state-action pair.

Questions

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, W and NoOp) has a deterministic outcome. Open the file `MDP.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 $\mathcal{P}$(state, action, next state) 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, and the action NoOp make it stay in its current state. For illustrative purposes, the transition probabilities for action N have been filled up for you.

2. Once you have filled all "`???`" entries in the file `MDP.m`, save it. In the `main.m` file, add the instruction "`M = MDP()`" to store the MDP structure in the variable `M`. Then, launch `main.m`. If the script runs successfully, you should read in the Matlab window something like

```
M =

       nX: 16
       nU: 5
       P0: [16x1 double]
        P: [16x5x16 double]
        r: [16x5 double]
    gamma: 0.9500
```

3. The component `M.P` reprensents a probability distribution over the next state, which means that it must sum to one over its third dimension. Verify this by executing, in the Matlab window, the command "`sum(M.P(x, u, :))`", where x can be any number between 1 and 16 and u can be any of `N`, `S`, `E`, `W` or `NoOp`. It should always return 1.

The goal of the robot is to choose the actions u_t so as to maximize (in expectation) its long term reward. We define this long term reward as the total discounted reward received :

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(x_t, u_t) \mid x_0 \sim \mathcal{P}_0(.), x_{t+1} \sim \mathcal{P}(x_t, u_t, .)\right]$$

The discount factor γ defines a trade-off between immediate and long term rewards : for a small γ, a greedy behaviour (i.e. seeking the greatest immediate reward) will be more rewarding.

A policy is a mapping $\pi : \mathcal{X} \to \mathcal{U}$ that assigns, to each state x, an action $u = \pi(x)$ that the robot should execute whenever in state x.

The state value of a policy π is the total discounted reward that the robot expects to receive when starting from a given state x and following policy π :

$$V^\pi(x) = \mathbb{E}\left[\sum_{t} \gamma^t r(x_t, u_t) \mid x_0 = x, u_t = \pi(x_t), x_{t+1} \sim \mathcal{P}(x_t, u_t, .)\right] \tag{1}$$

The state-action value of a policy π is the total discounted reward that the robot expects to receive when starting from a given state x, taking the action u and then following policy π :

$$Q^\pi(x, u) = \mathbb{E}\left[\sum_{t} \gamma^t r(x_t, u_t) \mid x_0 = x, u_0 = u, u_t = \pi(x_t), x_{t+1} \sim \mathcal{P}(x_t, u_t, .)\right] \tag{2}$$

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

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

for all $x \in \mathcal{X}$ and all policies π, where V^* denotes the value of policy π^*.

Note that we can also relate the optimal state-action value Q^* to V^* and the optimal policy π^* :

$$\begin{aligned} V^*(x) &= \max_u Q^*(x,u) \\ \pi^*(x) &= \operatorname*{argmax}_u Q^*(x,u) \end{aligned} \tag{3}$$

2 Dynamic Programming

Dynamic Programming is a family of methods for computing the optimal policy of an MDP. There are two approaches : Value Iteration (VI) computes the optimal state-action value Q^* and deducts the optimal π^* from it, Policy Iteration computes directly the optimal policy π^*. As their name suggest, they are both iterative : we start with in initial guess $Q^{(0)}$ of the optimal state-value function in VI, or with an initial policy $\pi_{(0)}$ in PI.

2.1 Value Iteration

In Equation (1), we have expressed the value function as a sum over time. By expanding the summation, we can rewrite it recursively as

$$V^\pi(x) = r(x,\pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,\pi(x),y) V^\pi(y) \tag{4}$$

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

$$V^*(x) = \max_u \left[r(x,u) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,u,y) V^*(y) \right] \tag{5}$$

The two above expressions suggest an iterative process to compute both V^π and V^*. Using (4), we obtain the recursion

$$V^{(k+1)}(x) = r(x,\pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,\pi(x),y) V^{(k)}(y)$$

where $V^{(k)}$ denotes the estimate for V^π at the kth iteration of the algorithm. Similarly, using (5), we get

$$V^{(k+1)}(x) = \max_u \left[r(x,u) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,u,y) V^{(k)}(y) \right]$$

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

$$\pi^*(x) = \operatorname*{argmax}_u \left[r(x,u) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,u,y) V^*(y) \right]$$

Notice that, in order to compute π^* from V^* it is still necessary to know r and $\mathcal{P}$. To avoid this, we can use the state-action value function (2) in the recursive form

$$Q^\pi(x,u) = r(x,u) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,u,y) V^\pi(y) \tag{6}$$

which, for the optimal policy π^*, becomes

$$Q^*(x,u) = r(x,u) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,u,y) V^*(y) \tag{7}$$

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

$$Q^*(x,u) = r(x,u) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,u,y) \max_v Q^*(y,v)$$

from where we immediately get the iterative update

$$Q^{(k+1)}(x,u) = r(x,u) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,u,y) \max_v Q^{(k)}(y,v) \tag{8}$$

The Value Iteration method works as follows :

- Provide an initial guess $Q^{(0)}$ of the optimal state-action value.
- For $k = 1, 2, 3, \cdots$, apply repeatedly equation (8) until convergence (i.e. $\|Q^{(k)} - Q^{(k-1)}\| = 0$ or small enough)
- Compute the corresponding (optimal) policy using equation (3)

This method is guaranteed to converge to the optimal state-action value function and policy, as $k \to \infty$.

Questions

1. 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. We define a policy as a column vector `pol` whose entries `i` correspond to the desired action at state `i`.
2. Once you have implemented Value Iteration, call it from the main script. To see what the policy looks like, you can use the plotPolicy function (see `plotPolicy.m`).

2.2 Policy Iteration

We must note that the Value Iteration approach admit a finite set of actions $\mathcal{U}$. 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 (PI), and proceeds as follows.

The algorithm starts with some arbitrary policy $\pi^{(0)}$. It evaluates this policy, computing the corresponding state-action value 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 :

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi(x), y) V^\pi(y)$$

Because the state-space $\mathcal{X}$ is finite, V^π can be represented as a vector. The same can be said about $r(x, \pi(x))$. On the other hand, $\mathcal{P}(x, \pi(x), y)$ can be represented as a square matrix. Let

$\mathbf{V}_\pi$, $\mathbf{R}_\pi$ and $\mathbf{P}_\pi$ denote this vector-matrix interpretation of $V^\pi(x)$, $r(x, \pi(x))$ and $\mathcal{P}(x, \pi(x), y)$, respectively. Then, the above equation becomes

$$\mathbf{V}_\pi = \mathbf{R}_\pi + \gamma \mathbf{P}_\pi \mathbf{V}_\pi.$$

We can rewrite this equation as

$$\mathbf{V}_\pi = (\mathbf{I} - \gamma \mathbf{P}_\pi)^{-1} \mathbf{R}_\pi. \tag{9}$$

The expression above states that $\mathbf{V}_\pi$ is the solution to a simple linear system of equations, which can be solved efficiently. We can easily compute the corresponding state-value function using equation (6), and then deduce a new policy from it :

$$\pi^{(k+1)}(x) = \underset{u}{\operatorname{argmax}}\ Q^{\pi^{(k)}}(x, u) \tag{10}$$

The Policy Iteration method works as follows :

- Provide an initial policy $\pi^{(0)}$
- For $k = 0, 1, 2, 3, \cdots$, do ...
- Compute the state value function of $\pi^{(k)}$ using equation (9)
- Compute the state-action value of $\pi^{(k)}$ using equation (6)
- Compute a new policy $\pi^{(k+1)}$ using equation (10)
- ... until the policy converge (i.e. $\pi^{(k+1)}(x) = \pi^{(k)}(x)$ for all state x)

Questions

1. 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.
2. Once you have implemented Policy Iteration, test it like you did for Value Iteration.
3. Compare how many iterations are necessary for VI and PI to converge.
4. Add a second goal : modify the reward function to give 0.9 when the robot is in state 2 and does not move. Why does the robot choose to go to this new goal instead of the first one ? Explain what parameter is responsible of this behaviour.

3 Reinforcement Learning

In Dynamic Programming, we assumed that our robot knew the transition and reward function of the MDP. We now suppose that the robot only knows the state-space and the action-space and must experiment with his environment to find out the optimal policy. In this context we can use Reinforcement Learning techniques (RL), which can be divided in two branches :

- The Model-Based approach consist in building a transition and reward function model based on the sampled obtained by interacting with the environment, and use Dynamic Programming with these models to compute the optimal policy.
- The Model-Free approach does not build an explicit model of the environment.

We will focus here on the Model-Free approach.

3.1 Temporal Difference Learning

Although we don't have a direct access to the transition and reward function of the MDP, we get samples from it when interacting with the environment : when the robot goes into state y after performing action u in state x, we now that y is a sample from the unknown distribution $\mathcal{P}(x, u, .)$. Therefore, we can estimate the value functions by formulating them in expectation :

$$\begin{aligned} V^\pi(x) &= r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi(x), y) V^\pi(y) \\ &= \mathbb{E}\left[r(x, \pi(x)) + \gamma V^\pi(y) \mid y \sim \mathcal{P}(x, \pi(x), .)\right] \end{aligned} \tag{11}$$

$$\begin{aligned} Q^*(x, u) &= r(x, u) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, u, y) \max_v Q^*(y, v) \\ &= \mathbb{E}\left[r(x, u) + \gamma \max_v Q^*(y, v) \mid y \sim \mathcal{P}(x, u, .)\right] \end{aligned} \tag{12}$$

For both (11) and (12), the difference between the left and right hand side is called the temporal difference error (TD error) :

$$\delta_{V^\pi}(x) = \mathbb{E}\left[r(x, \pi(x)) + \gamma V^\pi(y) - V^\pi(x) \mid y \sim \mathcal{P}(x, \pi(x), .)\right] \tag{13}$$

$$\delta_{Q^*}(x, u) = \mathbb{E}\left[r(x, u) + \gamma \max_v Q^*(y, v) - Q^*(x, u) \mid y \sim \mathcal{P}(x, u, .)\right] \tag{14}$$

It can be interpreted as the difference between the reward we received ($r(x, u)$) and the reward we predicted using our value function ($\gamma V^\pi(y) - V^\pi(x)$ or $\gamma \max_v Q^*(y, v) - Q^*(x, u)$). If the TD error is greater (resp. lower) than 0, we got a better (resp. worse) reward than expected and the true value of the state (and action for the state-action value) is greater (resp. lower). Therefore, we must update our value function in the direction of the TD error.

3.2 TD(0)

TD(0) estimate incrementally the state value function of a policy. A time t, we note $V^{(t)}$ our estimation of the state value function and $\delta_t \equiv \delta_{V^{(t)}}(x_t)$ the TD error of the current state x_t. The state value of the corrected can be corrected as follows :

$$\begin{aligned} V^{(t+1)}(x_t) &= V^{(t)}(x_t) + \alpha \delta_t \\ &= V^{(t)}(x_t) + \alpha \left[r(x_t, u_t) + \gamma V^{(t)}(x_{t+1}) - V^{(t)}(x_t)\right] \end{aligned} \tag{15}$$

where α is a small positive constant called the learning rate. If α is small enough, $V^{(t)}$ converge asymptotically (as $t \to \infty$) to V^π. The complete algorithm is :

- Provide an initial state value function estimate $V^{(0)}$
- For $t = 0, 1, 2, 3, \cdots$, do ...
- Draw a sample (x_t,u_t,r_t,x_{t+1}) according to the policy π and the MDP transition and reward functions.
- Update the state value function for state x_t using equation (15)
- ... until a sufficient number of iterations

Questions

1. First, open the file `MDPStep.m` in the Matlab editor. This function returns a next state and a reward given a state and an action. You are supposed to fill-in the missing lines. To draw a number according to a vector of probabilities, you can use the function "`discreteProb`" (see `discreteProb.m` for details).

2. Open the file `TD.m` in the Matlab editor. This is the skeleton of an M-file implementing TD(0). You are supposed to fill-in the missing lines. To obtain samples from MDP, you will use the MDPStep function which return a reward and a next state given a MDP, a state and an action : "`[y, r] = MDPStep(MDP, x, u)`".

3. Create a policy and use TD(0) to evaluate it. You can use the optimal policy we obtained using VI or PI, and compare the TD(0) evaluation with the Q-value computed by VI (using the relation $\forall x \in \mathcal{X}, V^\pi(x) = Q^\pi(x, \pi(x))$).

3.3 Q-Learning

Q-Learning estimate incrementally the optimal state-action value function. As opposed to TD(0), the policy is not fixed but based on the current estimation $Q^{(t)}$ of Q^*. At this point, we must introduce one of the most important aspect of Model-Free Reinforcement Learning : the exploration/exploitation dilemma.

If we wanted to evaluate exactly our policy, i.e. compute the exact optimal state-action value function, we would have to try every possible action in every possible state. But obviously, exploring the whole state-action space is contradictory with the goal of our robot, maximizing the long term reward. However, if we take the risk of trying new actions we might as well find a better policy ! Therefore, we have to make a compromise between trying new actions (exploration) and following our best policy so far (exploitation).

One way to realize this compromise is to choose a random action instead of $\text{argmax}_u Q^{(t)}(x_t, u)$ with a small probability ϵ. A more subtle way is to weight the probability of each action by its Q-value. This is called a soft-max policy :

$$\pi^{(t)}(u|x) = \frac{\exp(Q^{(t)}(x,u)/\tau)}{\sum_{v \in \mathcal{U}} \exp(Q^{(t)}(x,v)/\tau)}$$

where τ is a positive parameter called the temperature. As $\tau \to 0$, the soft-max policy becomes equivalent to the greedy policy $\text{argmax}_u Q^{(t)}(x, u)$.

Now, going back to the evaluation problem, we can use the TD error $\delta_t \equiv \delta_{Q^{(t)}}(x_t, u_t)$ to update the current estimate $Q^{(t)}$ of the optimal state-value function Q^* :

$$\begin{aligned} Q^{(t+1)}(x_t, u_t) &= Q^{(t)}(x_t, u_t) + \alpha\delta_t \\ &= Q^{(t)}(x_t, u_t) + \alpha \left[r(x_t, u_t) + \gamma \max_{u_{t+1} \in \mathcal{U}} Q^{(t)}(x_{t+1}, u_{t+1}) - Q^{(t)}(x_t, u_t) \right] \end{aligned} \tag{16}$$

where α is a small positive constant called the learning rate. If α is small enough, $Q^{(t)}$ converge asymptotically (as $t \to \infty$) to Q^*.

Using the soft-max policy, the complete Q-Learning algorithm is

- Provide an initial state-action value function estimate $Q^{(0)}$
- For $t = 0, 1, 2, 3, \cdots$, do ...
- Draw a sample (x_t,u_t,r_t,x_{t+1}) according to the soft-max policy over $Q^{(t)}$ and the MDP transition and reward functions.

- Update the state-action value function for state x_t and action u_t using equation (16)
- ... until a sufficient number of iterations

Questions

1. First, open the file `softmax.m` in the Matlab editor. This function must return a soft-max distribution over actions, given a state, a policy and a temperature. You are supposed to fill-in the missing lines.
2. Open the file `QLearning.m` in the Matlab editor. This is the skeleton of an M-file implementing Q-Learning. You are supposed to fill-in the missing lines.
3. Run Q-Learning several times. What do you observe ?
4. Compare the Q-function and the policy computed using Q-Learning with the ones you obtained with VI and PI.

4 Model-Based Reinforcement Learning

We are now going to use Model-Based Reinforcement Learning (MBRL) techniques to solve our MDP. In the Model-Based approach, we learn a model of the transition and the reward function (resp. $\mathcal{P}$ and r) and apply Dynamic Programming (DP) techniques using this model instead of the true MDP. If this model is good enough, we will find the optimal policy of the MDP.

Let $\hat{\mathcal{P}}^{(t)}$ 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). The estimate $\hat{\mathcal{P}}^{(t)}$ can then be updated as

$$\hat{\mathcal{P}}^{(t+1)}(x_t, u_t, y) = \left(1 - \frac{1}{N_t(x_t, u_t)}\right) \hat{\mathcal{P}}^{(t)}(x_t, u_t, y) + \frac{1}{N_t(x_t, u_t)} \delta_y(x_{t+1}), \tag{17}$$

where $\delta_y(x) = 1$ if $y = x$ and 0 else, and $N_t(x, u)$ denotes the number of visits to the pair (x, u) up to and including time t. Note that, at time t, only the transition probabilities $\hat{\mathcal{P}}^{(t)}(x_t, u_t, \cdot)$ are updated, to reflect the new transition just observed. For the reward function estimate $\hat{r}$, we simply have

$$\hat{r}^{(t+1)}(x_t, u_t) = r_t \tag{18}$$

Therefore, we don't have to estimate the reward function explicitly, i.e. we can use the reward samples instead.

5 Real-Time Dynamic Programming

Once $\hat{\mathcal{P}}$ and $\hat{r}$ are properly estimated, we can now use them in a Value Iteration scheme. An interesting aspect of the model-based approaches, however, is that these iterations can be performed as the estimates for $\mathcal{P}$ and r are updated. So after updating $\hat{\mathcal{P}}$ and $\hat{r}$ according to (17) and (18), the following update could immediately follow :

$$Q^{(t+1)}(x_t, u_t) = \hat{r}^{(t+1)}(x_t, u_t) + \gamma \sum_{y \in \mathcal{X}} \hat{\mathcal{P}}^{(t+1)}(x_t, u_t, y) \max_{v \in \mathcal{U}} Q^{(t)}(y, v) \tag{19}$$

This technique is called Real-Time Dynamic Programming.

Questions

1. Open the file `RTDP.m` in the Matlab editor. This is the skeleton of an M-file that implements the RTDP algorithm. You are supposed to fill-in the missing lines. Note that for each iteration we reuse the next state from the previous iteration as the current state.
2. Suppose now that the reward is stochastic : modify `MDPStep.m` by adding a Gaussian noise of standard deviation 0.1 to the reward (see Matlab function `randn`). What happens if you set the standard deviation to 1.0 ?
3. Modify RTPD to handle this stochastic reward by computing the model $\hat{r}$ of the mean reward for each state and action.