Reinforcement Learning

Alejandro Ribeiro and Sergio Rozada

November 11, 2024

1 Markov Decision Processes

A Markov Decision Process (MDP) is a dynamical system f with state trajectory s_t which we control through the choice of actions a_t. The evolution of the dynamical system is given by

$$s_{t+1} = f\big(s_t, a_t\big) \,. \tag{1}$$

We further postulate that whenever we visit the state s and take action a we collect a reward $r(s,a)$. The state evolution equation in (1) along with the rewards $r(s,a)$ define the MDP. The use of the word "Markov" indicates that the evolution of the process depends on the state s_t and the action a_t but does not depend on past state-action pairs (s_u, a_u) for $u < t$.

An MDP that models walking on a line is shown in Figure 1. The states $s = 0, 1, \dots, N$ are scalar numbers that indicate the walker's current position. When we are in a state other than $s = 0$ or $s = N$ the possible actions are $a = +1$ and $a = -1$. They signify taking a step forward or backward, respectively. When in state $s = 0$ the possible actions are $a = +1$ and $a = 0$, representing a step backward or staying in place. When in state $s = N$ the possible actions are $a = 0$ and $a = -1$, staying in place or walking backwards. It follows from this description that the dynamical system is defined by the state evolution equation

$$s_{t+1} = s_t + a_t. \tag{2}$$

To complete the description of the MDP we assign rewards to each state-action pair. We choose to make $r(s,a) = 0$ for all states s other than

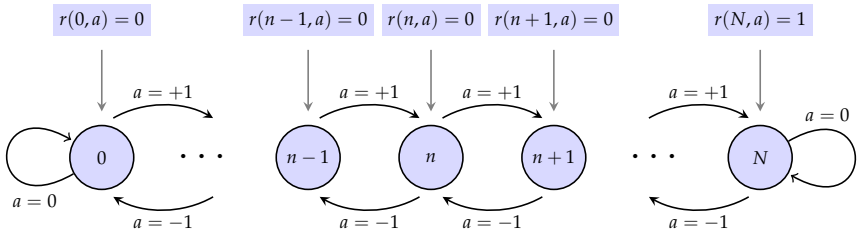

Figure 1. Markov Decision Process (MDP). A walker moves forward when choosing action $a = +1$ and backwards when choosing action $a = -1$. Action $a = 0$ at the border states keeps the walker in place. The rewards $r(N, a) = 1$ for $s = N$ and $r(s, a) = 0$ for other states s incentivize the walker to move forward.

$s = N$ and $r(N, a) = 1$ for $s = N$ irrespectively of the value of the action a. This reward structure is signifying that $s = N$ is a desirable state. It incentivizes the walker to move forward.

1.1 Policies

A policy π is a mapping from states s to actions $a = \pi(s)$ so that whenever we visit state s we take action $a = \pi(s)$. Thus, the system's state s_t when executing policy π evolves according to

$$s_{t+1} = f\big(s_t,\, a_t\big) = f\big(s_t,\, \pi(s_t)\big)\,. \tag{3}$$

In Figure 2 we illustrate two policies for the walker in Figure 1. The forward policy selects $\pi_{\mathrm{F}}(N) = 0$ at the terminal state $s = N$ and $\pi_{\mathrm{F}}(s) = 1$ for other states $s \neq N$. The backward policy selects actions $\pi_{\mathrm{B}}(0) = 0$ and $\pi_{\mathrm{B}}(s) = -1$ for $s \neq 0$.

Notice that different policies accumulate different amounts of rewards. Our goal when working with MDPs is to find the policy π that results in trajectories that accumulate as much reward as possible. Defining this goal mathematically requires some effort (see Section 2.2) but it is clear that for the walker in Figure 1 the optimal policy is to walk forward to reach state $s = N$ which is the only state with nonzero rewards.

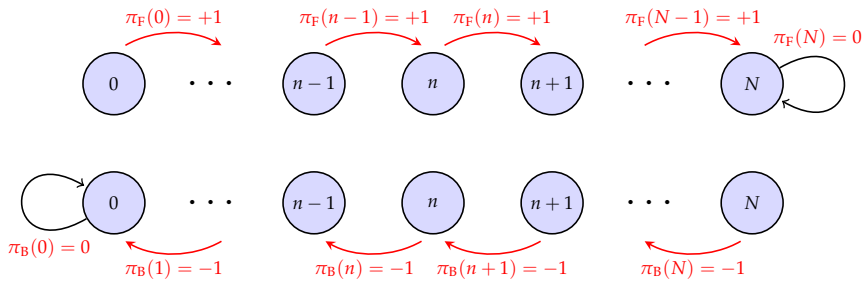

Figure 2. Policies. Policies map states to actions. For the walker in Figure 1 define the forward policy as $\pi_F(N) = 0$ and $\pi_F(s) = 1$ for $s \neq N$ and the backward policy as $\pi_B(0) = 0$ and $\pi_B(s) = -1$ for $s \neq 0$. Different policies accumulate different total rewards. Optimal policies accumulate as much reward as possible.

1.2 Stochastic Policies and MDPs

The MDP in (2) is deterministic. The state-action pair (s_t, a_t) completely determines the next state $s_{t+1} = f(s_t, a_t)$. The policy in (3) is also deterministic. Given the current state s_t completely determines the choice of action $a_t = \pi(s_t)$. Both of these can be stochastic.

In a stochastic MDP the state action pair (s_t, a_t) controls the probability distribution of the next state. I.e., the state s_{t+1} is drawn from the conditional probability distribution

$$s_{t+1} \sim p\big(\, s_{t+1} \,|\, s_t,\, a_t \,\big), \tag{4}$$

Instead of dictating the next state s_{t+1}, the choice of action a_t dictates the likelihood that the next state is s_{t+1}.

Figure 3 shows a stochastic variation of the walker in Figure 1. This walker moves up with probability $p + a$ and it moves down with probability $1 - p - a$. Thus, the MDP is characterized by the stochastic evolution,

$$\begin{aligned} p\big(\, s_{t+1} = s_t + 1 \,|\, s_t,\, a_t \,\big) &= p + a, \\ p\big(\, s_{t+1} = s_t - 1 \,|\, s_t,\, a_t \,\big) &= 1 - p - a. \end{aligned} \tag{5}$$

In (5) actions a must be chose so that $p + a$ and $1 - p - a$ are valid probabilities. For instance, we may have that $p = 1/2$ and $a = +1/4, -1/4$. With these values the model in (5) represents a random walker that can

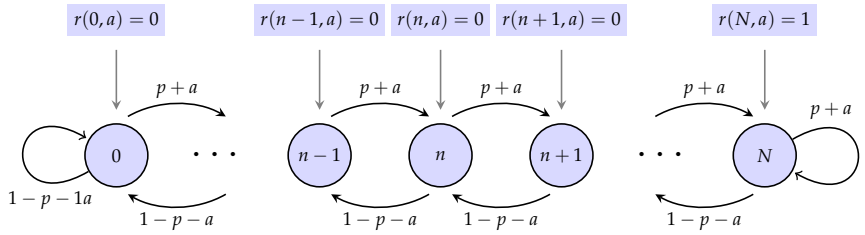

Figure 3. Stochastic MDP.

bias its walk up or down. It can make the probability of walking forward as high as $3/4$ by choosing $a = +1/4$ and the probability of walking backwards as high as $3/4$ by choosing $a = +1/4$.

In addition to working with stochastic MDPs we can also *choose* to work with stochastic policies. A stochastic policy π does not determine the action a_t exactly but the probability of choosing a particular action,

$$a \sim \pi\big(a \,|\, s\big). \tag{6}$$

For example, the random walker in Figure 3 may decide to bias their walk up with probability q and to bias their walk down with probability $1 - q$,

$$\begin{aligned} \pi\big(a = +1/4 \,|\, s\big) &= q, \\ \pi\big(a = -1/4 \,|\, s\big) &= 1 - q. \end{aligned} \tag{7}$$

We will work in this chapter with deterministic MDPs and deterministic policies. They are easier to explain and train. However, we need to be aware of stochastic policies and MDPs because most of the literature on reinforcement learning works with stochastic policies and MDPs. The actor-critic method that we will introduce later works verbatim for stochastic MDP.

2 Q-Functions and Value Functions

We are given a discount factor $\gamma < 1$, we fix a policy π, and we initialize the system with state-action pair $(s, a) = (s_0, a_0)$. We define the Q-function $Q(s, a; \pi)$ as the reward accumulated by executing policy π

when the system is initialized at state-action pair (s, a),

$$\begin{aligned} Q(s, a; \pi) &= \sum_{t=0}^{\infty} \gamma^t \, r(s_t, a_t) \\ &= r(s_0, a_0) + \gamma r(s_1, a_1) + \gamma^2 r(s_2, a_2) + \dots \end{aligned} \tag{8}$$

In this reward accumulation the purpose of the discount factor γ is to give more weight to earlier collection of rewards. Notice that the MDP f and the policy π enter the right hand side of (8) trough the state evolution. This is because actions $a_t = \pi(s_t)$ in (8) follow the policy π and state transitions $s_{t+1} = f(s_t, a_t) = f(s_t, \pi(s_t))$ follow the MDP.

Indeed, the system starts at the state-action pair $(s, a) = (s_0, a_0)$ and collects the reward $r(s, a) = r(s_0, a_0)$. As per (1), the system moves to state $s_1 = f(s_0, a_0) = f(s, a)$. At this point in time we choose action $a_1 = \pi(s_1)$ to collect the reward $\gamma r(s_1, a_1)$ and effect the system to transition into state $s_2 = f(s_1, a_1) = f(s_1, \pi(s_1))$. We then execute action $a_2 = \pi(s_2)$ to collect reward $\gamma^2 r(s_2, a_2)$ and transition into state $s_3 = f(s_2, a_2) = f(s_2, \pi(s_2))$. We keep executing actions as dictated by the policy π so that at arbitrary time t we execute action $a_t = \pi(s_t)$ to collect reward $\gamma^t r(s_t, a_t)$ and transition into state $s_{t+1} = f(s_t, a_t)$.

Notice that in the definition of the Q-function $Q(s, a; \pi)$ in (8), the action a may or may not be the policy action $\pi(s)$.

The Q-function in (8) describes the reward that we collect when the system is initialized at state-action pair $(s, a) = (s_0, a_0)$. In general, we care about the system when it is initialized at a number of different states s. To capture this interest we consider a distribxution $p(s)$ of initial states $s = s_0$ and define the value function of policy π as

$$V(\pi) = \mathbb{E}_{p(s)} \Big[\, Q\Big(\, s, \, \pi(s); \, \pi \, \Big) \, \Big]. \tag{9}$$

Observe that in the definition of the value function we particularize a to $\pi(s)$ in the Q-function. The value function considers the effect of running policy π from $t = 0$ onward averaged over an exogenous choice of the distribution of initial states $s_0 = s$.

2.1 Q-Function Recursion

The form of the Q-function is such that it can be computed recursively. This result is simple to derive but important enough to be highlighted as a proposition.

Proposition 1 *Consider an MDP f as defined in* (1) *along with a policy π as defined in* (3). *The Q-function $Q(s,a;\pi)$ in* (8) *satisfies the recursion*

$$Q\Big(s,\pi(s);\pi\Big) \;=\; r(s,a) + \gamma Q\Big(s^+,\pi(s^+);\pi\Big), \tag{10}$$

for any initial state s and next state $s^+ = f(s,\pi(s))$.

Proof: To prove (10) we write $Q(s,a;\pi) = Q(s_0,a_0;\pi)$ for clarity and we recall that in the definition of the Q-function in (8) actions $a_t = \pi(s_t)$ follow the policy π and state transitions $s_{t+1} = f(s_t,a_t) = f(s_t,\pi(s_t))$ follow the MDP. In particular $s_1 = f(s_0,a_0)$ and $a_1 = \pi(s_1)$.

Begin by separating the term corresponding to $t=0$ in (8) from the other summands. This yields the relationship,

$$Q(s_0,a_0;\pi) \;=\; r(s_0,a_0) + \gamma \sum_{t=1}^{\infty} \gamma^{t-1}\, r(s_t,a_t). \tag{11}$$

Observe now that the summation in the right hand side is the Q-function $Q(s_1,a_1;\pi)$. The sum starts at $t=1$, but this is just a summation index that we can shift backwards. For a reader that needs convincing we write the sum in expanded form,

$$\sum_{t=1}^{\infty} \gamma^{t-1}\, r(s_t,a_t) = r(s_1,a_1) + \gamma r(s_2,a_2) + \gamma^2 r(s_3,a_3) + \dots, \tag{12}$$

which is the same as (8) but with initial condition $(s,a) = (s_1,a_1)$. We have therefore concluded that

$$Q(s_0,a_0;\pi) \;=\; r(s_0,a_0) + \gamma Q(s_1,a_1;\pi) \tag{13}$$

This is the same as (10) because as we have already remarked $s_1 = f(s_0,a_0)$ and $a_1 = \pi(s_1)$. We can then make the substitutions $(s_0,a_0) = (s,\pi(s))$ and $(s_1,a_1) = (s^+,\pi(s^+))$. ■

The claim in Proposition 1 is a rather obvious statement. It says that the reward we accumulate *from* $t = 0$ onwards is the sum of the reward *at* $t = 0$ and the reward we accumulate from $t = 1$ onwards. We just need to discount the reward accumulated from $t = 1$ onwards.

2.2 Optimal Policy

A first approach to define an optimal policy is to maximize the Q-function for a certain initial condition $(s, \pi(s))$,

$$\pi^* \;=\; \underset{\pi}{\operatorname{argmax}}\, Q(s, \pi(s); \pi) \tag{14}$$

Although it seems that the optimal policies π^* in (14) depend on the initial condition $(s, a) = (s_0, a_0)$ this is not quite the case. There exists an optimal policy π^* that is the same for all initial conditions. This is true because having more than one optimal policy contradicts Proposition 1. To see that this is true suppose that policy π^* is optimal for the Q-function $Q(s, \pi(s); \pi)$ but not for the Q-function $Q(s^+, \pi(s^+); \pi)$ that is maximized by policy π^{**}. If this is true,

$$\begin{aligned} Q\Big(s, \pi^*(s); \pi^*\Big) \;&=\; r(s, \pi^*(s)) + \gamma Q\Big(s^+, \pi^*(s^+); \pi^*\Big) \\ &<\; r(s, \pi^*(s)) + \gamma Q\Big(s^+, \pi^{**}(s^+); \pi^{**}\Big), \end{aligned} \tag{15}$$

where the equality follows from (10) and the inequality is true because π^* is not optimal for $Q(s^+, \pi(s^+); \pi)$. This inequality contradicts the statement that π^* is optimal for $Q(s, \pi(s); \pi)$ because we can change the policy to π^{**} from time $t = 1$ and increase the accumulated reward.

A second approach to defining an optimal policy π^* is to maximize the value function $V(\pi)$.

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, V(\pi) \tag{16}$$

Although it may seem that (16) yields a different optimal policy this is not quite the case. Since a policy that is optimal for all initial conditions exists, this policy is also optimal for the average in (16). However equivalent, working with value functions is preferable in practice because finding policies that minimize value functions is easier than finding policies that minimize Q-functions.

Although we can attempt to solve (16) directly over the policy π we chose to introduce a learning parameterization $\pi(s, \alpha)$. This leads to the optimization problem

$$\alpha^* \;=\; \underset{\alpha}{\operatorname{argmax}}\, V(\pi(\alpha)) \;=\; \underset{\alpha}{\operatorname{argmax}}\, \mathbb{E}_{p(s)}\Big[\, Q\Big(\, s,\, \pi(s,\alpha);\, \pi(\alpha)\,\Big)\,\Big]. \quad (17)$$

The advantage of introducing a learning parameterization is that after solving (17) we can execute optimal actions $\pi(s_t; \alpha^*)$ with (simple) evaluations of the learning parameterization. This is our customary reason for using machine learning.

2.3 Circuit Navigation

The trajectory control problem we addressed in Chapter 8 can be written as a particular case of (17). Consider a reference trajectory specified here as set $\mathcal{S}$ of vectors $s_{\mathrm{R}} = [p_{\mathrm{R}}; v_{\mathrm{R}}]$ where p_{R} and v_{R} are coordinates and velocities that we want to track. The pair $[p_{\mathrm{R}}; v_{\mathrm{R}}]$ represents an optimal pair of positions and velocities as determined, e.g., by an expert driver. The state of the system is a vector $s = [p, v]$ that contains the position and velocity of the car. We control the car by choosing accelerations a_t. A first approach to capture the goal of following the expert is the reward function,

$$r_1(s_t, a_t) \;= -\min_{[p_{\mathrm{R}}; v_{\mathrm{R}}] \in \mathcal{S}} \left[\, \frac{a}{2}\|p_t - p_{\mathrm{R}}\|^2 \;+\; \frac{b}{2}\|v_t - v_{\mathrm{R}}\|^2 \right]. \quad (18)$$

The term $-(a/2)\|p_t - p_{\mathrm{R}}\|^2$ keeps the position of the car p_t close to the reference trajectory p_{R}. The term $-(b/2)\|v_t - v_{\mathrm{R}}\|^2$ keeps the car's velocity v_t close to the velocity of the expert v_{R}. The minimum in the reward $r(s_t, a_t)$ is taken over all vectors $s_{\mathrm{R}} = [p_{\mathrm{R}}; v_{\mathrm{R}}]$ in the expert trajectory $\mathcal{S}$. No matter the car's position and velocity, we want to incentivize it to move closest to the nearest point in the expert trajectory.

If the car is initialized at state s with action a and we execute policy π, the Q-function reduces to

$$Q(s, a; \pi) = -\sum_{t=0}^{\infty} \gamma^t \min_{[p_{\mathrm{R}}; v_{\mathrm{R}}] \in \mathcal{S}} \left[\, \frac{a}{2}\|p_t - p_{\mathrm{R}}\|^2 \;+\; \frac{b}{2}\|v_t - v_{\mathrm{R}}\|^2 \right], \quad (19)$$

which we obtain by substituting (18) into (8). Recall that the policy π enters Equation (19) in the generation of the trajectory. States $s_t = [p_t, v_t]$

are the consequence of executing actions $a_u = \pi(s_u)$ and the evolution $s_{u+1} = f(s_u, a_u)$ of the dynamical system for all times $u < t$. Except for the discount factor γ this is the same loss we used to design MPC controllers in Lab 8.

Further consider further a set if *initial* states s_i drawn from a set of N possible initial states. Using this set of states as the initial state distribution $p(s)$ and substituting (19) for the Q-function in (9) yields the value function

$$V(\pi) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{\infty}\gamma^t \min_{[p_R; v_R]\in\mathcal{S}}\left[\frac{a}{2}\|p_{it} - p_R\|^2 + \frac{b}{2}\|v_{it} - v_R\|^2\right], \qquad (20)$$

where p_{it} and v_{it} are the trajectories generated on the dynamical system f when we execute policy π and start the system at initial condition s_i. We see here that minimizing (19) and (20) are indeed equivalent problems as is true in general for MDPs. We also see why solving (20) is preferable in practice. Having a *set* of initial conditions s_i instead of a single initial condition s yields a richer set of states to *explore* as we learn an optimal policy.

A more refined reward combines the reward in (18) with the norm of the acceleration

$$r(s_t, a_t) \ = r_1(s_t, a_t) \ - \frac{c}{2}\|a_t\|^2. \qquad (21)$$

The term $-(c/2)\|a_t\|^2$ penalizes large accelerations. This is to prevent imitation of the expert with swerving trajectories. Penalizing large accelerations moves the car cloer the expert trajectory with smooth movements. The use of different scaling constants a, b, and c in (18) and (21) gives different importance to positions, velocities, and accelerations. E.g., it is clear that keeping positions p_t and p_R close is more important than keeping velocities close and that we should therefore make $a > b$.

Henceforth, we use the reward $r(s_t, a_t)$ in (21).

Task 1 Code a car class whose attributes are an expert trajectory $\mathcal{S}$, the state of the car $s = [p, v]$, and its acceleration a. Make the car simulator a parent of this class. We take this simulator as the system f so that calling `simulator.step(s,a)` computes the state $s' = f(s, a)$. Although not mathematically necessary, it may help with interpretations to know that a step of the car simulator corresponds to $T_s = 50$ms.

Add a reward method that computes the reward $r(s, a)$ collected by the car given the state of its attributes. This method computes the reward in (21).

Instantiate the car and initialize its expert trajectory. To solve this task you need to load the car simulator and the expert trajectory. The simulator can be found here, and the expert trajectory here.

2.4 Observations

Agents in an MDP are assumed to know their states s_t. This is how they can execute the action $a_t = \pi(s_t)$ dictated by their policy. In addition to states, agents may be also assumed to have access to observations o_t. When these observations are available, the policy π is a function of the state and the observation,

$$a_t = \pi(s_t, o_t). \tag{22}$$

In circuit navigation the expert trajectory $\mathcal{S}$ is a possible example of an observation that the agent leverages to make acceleration decisions.

Observations o_t do not affect the evolution of the dynamical system except for their effect in the choice of policy. When choosing actions as in (22), the state of the system still evolves according to the same MDP,

$$s_{t+1} = f\Big(\, s_t,\, a_t \,\Big) = f\Big(\, s_t,\, \pi(s_t, o_t) \,\Big). \tag{23}$$

Observations $o_t = h(s_t)$ are also completely determined by the state s_t through the observation function h. The observation function g, same as the dynamical system f, is unknown.

Since observations o_t are determined by states s_t and state evolution is independent of observations [cf. (23)] – save for their effect in the action choice [cf. (22)] – we think of observations o_t as *side* information that is attached to the state s_t. This information is available to make better decisions $a_t = \pi(s_t, o_t)$ but does not otherwise impact the evolution of the MDP. This is different from states s_t that affect both, the choice of action a_t and the next state s_{t+1}.

In subsequent sections we write policies as $a_t = \pi(s_t)$ even though we most often mean $a_t = \pi(s_t, o_t)$. We do so to avoid complicating notation.

Task 2 Modify the class in Task 1 to add an observation attribute o. Add an observation method that given the state of the car $s = [p, v]$ and the reference trajectory $\mathcal{S}$ computes the observation o consisting of the n_o reference states s_R whose positions p_R are closest to the position of the agent p. For subsequent experiments we recommend that you set $n_o = 10$.

Modify the parent method simulator.step(s,a) to include an update of the observation attribute using the observation method. Modify the `init` method to initialize the observation attribute using the observation method.

3 Reinforcement Learning

In Section 2.2 we define optimal policies of Markov decision processes (MDP). If the MDP transition function f and the reward functions r are known, the optimization problems formulated in Section 2.2 are regular optimization problems. The formulation in (17), in particular, is a standard machine learning problem. However, the function f and the reward r are not known. A possible approach to solve (17) is to design or learn models of f and r and use them to find the policy α^*. This is model predictive control (MPC).

Alternatively, we can learn the policy α^* by probing the MDP with different policies π and observing the collected rewards $V(\pi)$. This is reinforcement learning (RL). We can then think of RL as the process of learning and minimizing the value function. This is in contrast to MPC where we build or learn models of the transition function f and the reward function r as an intermediate step to finding the optimal policy α^*.

4 Policy Optimization

To solve (17) without learning a model of the MDP, we run gradient *ascent* on the policy parameter α. Introduce then an iteration index k, an step size ϵ, an initial value α_0 and proceed through iterations that update α by following gradients $g(\alpha) := \partial V(\pi(\alpha))/\partial\alpha$,

$$\alpha_{k+1} \;=\; \alpha_k \;+\; \epsilon\, g(\alpha_k) \;=\; \alpha_k \;+\; \epsilon\, \frac{\partial}{\partial \alpha} V(\pi(\alpha_k)). \tag{24}$$

In contrast to gradient *descent* we move in the direction of the gradient, not its opposite. This modification is because we are searching for the maximum of $V(\alpha)$, not its minimum.

To implement (24) we need to compute gradients of the value function. This computation is more difficult than it seems but it can be carried out. To give a gradient expression it is convenient to write the value function as

$$V(\pi(\alpha)) = \mathbb{E}_{p(s)}\Big[\, Q\big(\, s,\, \pi(s,\alpha);\, \pi(\alpha)\,\big)\,\Big] = \mathbb{E}_{p(s)}\Big[\, Q\big(\, s,\, a;\, \pi(\alpha)\,\big)\,\Big], \quad (25)$$

where in the second equality it is implicit that $a = \pi(s,\alpha)$. This is just a notation simplification.

We now consider a trajectory s_t generated by policy π. This is a trajectory in which we choose $a_t = \pi(s_t)$ time $t = 0$ onwards. The gradient of the value function can then be written as

$$g(\alpha) = \mathbb{E}_{p(s)}\left[\, \sum_{t=0}^{\infty} \gamma^t \times \frac{\partial}{\partial a_t} Q\Big(\, s_t,\, a_t;\, \pi(\alpha)\,\Big) \times \frac{\partial}{\partial \alpha}\pi(s_t,\alpha)\,\right]. \quad (26)$$

In this expression the gradients of the Q-function are taken with respect to the choice of action a_t and are evaluated at $a_t = \pi(s_t,\alpha)$. I.e., it is implicit that $a_t = \pi(s_t,\alpha)$ as it is implicit in (25) that $a = \pi(s,\alpha)$.

4.1 Critics

In (26), the gradients $\partial\pi(s_t,\alpha)/\partial\alpha$ are with respect to the learning parameterization and can be computed with automated differentiation. The gradients $\partial Q(s_t,a_t;\pi(\alpha))/\partial a_t$ are more difficult to evaluate because we do not know the Q-function. We solve the latter problem with the introduction of a *critic*.

A critic of a given policy π is a function $\tilde{Q}(s,a;\beta)$ that maps the state-action pair $(s,a) = (s,\pi(s))$ and the parameter β to an estimate of the Q-function's value $Q(s,a;\pi)$. Having a critic available we can replace the gradient of the Q-function in (26) for the gradient of the critic to compute gradient estimates,

$$\tilde{g}(\alpha) = \mathbb{E}_{p(s)}\left[\, \sum_{t=0}^{\infty} \gamma^t \times \frac{\partial}{\partial a_t} \tilde{Q}\Big(\, s_t,\, a_t;\, \beta\,\Big) \times \frac{\partial}{\partial \alpha}\pi(s_t,\alpha)\,\right]. \quad (27)$$

The vector $\tilde{g}(\alpha)$ is an estimate of the value function gradient based on derivatives of the critic function. We call it a *critic gradient* for short.

The advantage of replacing the critic $\tilde{Q}(s, a; \beta)$ in the gradient computation is that the critic is a parametric function. As such, the gradients $\partial \tilde{Q}(s, a; \beta)/\partial a$ can be computed with automated differentiation. We can therefore introduce an iteration index k, a step size ϵ and an initial value α_0 to update α_k by following the critic gradients $\tilde{g}(\alpha_k)$,

$$\alpha_{k+1} \;=\; \alpha_k \;+\; \epsilon\, \tilde{g}(\alpha_k). \tag{28}$$

In this recursion the vectors $\tilde{g}(\alpha_k)$ are the critic gradients defined in (27). This is in contrast to the true gradients $g(\alpha_k)$ of the value function $V(\pi)$ that we use in (24) and express in (26).

The downside of using a critic is that it is an approximation, not the true, Q-function. For this to be workable we have to learn a good critic with close function values $\tilde{Q}(s, a; \beta) \approx Q(s, a; \pi(\alpha))$ and close derivatives $\partial \tilde{Q}(s, a; \beta)/\partial a \;\approx\; \partial Q(s, a; \pi(\alpha))/\partial a$. In a sense, (27) does not solve the problem of not knowing the Q-function in (26). It punts it. This is literal. We assume in the following that a critic is available and return to its computation in Section 6.

4.2 Actors

An *actor* is an agent that given critic $\tilde{Q}(s, a; \beta)$, solves the optimization problem

$$\begin{aligned} \alpha^*(\beta) \;&=\; \underset{\alpha}{\text{argmax}}\; \mathbb{E}_{p(s)}\left[\sum_{t=0}^{\infty} \gamma^t \times \tilde{Q}\Big(s_t,\, \pi(s_t; \alpha);\, \beta \Big) \right] \\ &=\; \underset{\alpha}{\text{argmax}}\; \ell_{\text{A}}(\alpha, \beta), \end{aligned} \tag{29}$$

where in the second equality we define the actor loss $\ell_{\text{A}}(\alpha, \beta)$, which we call a loss despite the fact that it is a reward we want to maximize. In this loss we draw initial samples $s_0 = s$ from the distribution $p(s)$. We then generate trajectories s_t from these initial conditions by executing the policy $\pi(\alpha)$. The critic $\tilde{Q}(s, a; \beta)$ is evaluated at states and actions that correspond to this trajectory.

The optimization in (29) is justified by the fact that the gradients of the loss $\ell_{\rm A}(\alpha, \beta)$ have the same form of the critic gradients $\tilde{g}(\alpha)$ in (27),

$$\frac{\partial}{\partial \alpha} \ell_{\rm A}(\alpha, \beta) \;=\; \mathbb{E}_{p(s)} \left[\sum_{t=0}^{\infty} \gamma^t \times \frac{\partial}{\partial a_t} \tilde{Q}\Big(s_t, \, a_t; \, \beta \Big) \times \frac{\partial}{\partial \alpha} \pi(s_t, \alpha) \right]. \tag{30}$$

There is, however, a subtle difference between (27) and (30). In (27) the critic $\tilde{Q}(s, a; \beta)$ approximates the Q-function of policy $\pi(\alpha)$. The same policy whose gradients we are evaluating. In (30) the critic $\tilde{Q}(s, a; \beta)$ is fixed and we evaluate gradients at an arbitrary policy parameter α. This is the reason why in (29) the optimal actor parameter is written as a function of β.

That the critic parameter (29) is fixed is not an insignificant issue. Let us, however, set it aside for the time being and work on its solution. We revisit this challenge in Section 7.

To solve (29) we consider stochastic approximations of the loss $\ell_{\rm A}(\alpha, \beta)$. Formally, consider an initial state $s \sim p(s)$ sampled from the initial state distribution and compute

$$\hat{\ell}_{\rm A}(\alpha, \beta) \;=\; \sum_{t=0}^{\infty} \gamma^t \times \tilde{Q}\Big(s_t, \, \pi(s_t; \alpha); \, \beta \Big). \tag{31}$$

As in (29), the trajectory s_t is generated from the initial condition s by executing the policy $\pi(\alpha)$. We refer to (31) as a rollout. We generate a trajectory by executing (or rolling out) the policy $\pi(\alpha_k)$ from initial condition s. Although we keep a sum running the $t = \infty$ for simplicity, we limit the rollout to a finite limte $t = T$ in practical implementations.

Instead of a single initial state sample s we may work with a batch of B samples $s_i \sim p(s)$ drawn from the initial state distribution. In this case the stochastic approximation of the loss $\ell_{\rm A}(\alpha, \beta)$ becomes

$$\hat{\ell}_{\rm A}(\alpha, \beta) \;=\; \frac{1}{B} \sum_{i=1}^{B} \sum_{t=0}^{\infty} \gamma^t \times \tilde{Q}\Big(s_{it}, \, \pi(s_{it}; \alpha); \, \beta \Big). \tag{32}$$

As in (29) and (31) the trajectories s_{it} are generated from the initial conditions s_i by executing the policy $\pi(\alpha)$.

Task 3 The code in this link provides a `critic` class with a trained critic for the circuit navigation problem desribed in Section 2.3. Run the

method `critic.load` to load the critic and the method `critic.evaluate(s, a)` to evaluate the critic at state-action pair (s, a). The `critic` class is a child of `nn.Module` to allow for computation of gradients with automated differentiation. This critic has been trained on the optimal policy. That is, is approximates the Q-function of the policy $\pi(\alpha^*)$.

Use this critic to find the optimal policy $\pi(\alpha^*)$. Do so by implementing the recursion in (28) with the stochastic approximate gradients in (32). Use $\gamma = 1.0$ as a discount factor. Notice that in (32) policy rollouts extend to $t = \infty$. Use $T = 250$ as a large enough number to approximate infinity.

To find this optimal policy you need a parametric policy. We suggest that you use a fully connected neural network (FCNN) in which the inputs are state (a) and observation (o) attributes of a car object instantiated from the class of Task 2 and the outputs are the acceleration actions (a),

$$a = \pi(s, o; \alpha) = \text{FCNN}(s, o; \alpha). \tag{33}$$

An FCNN with 4 hidden layers containing $n_\ell = 128$ hidden neurons each has worked well in our experiments.

To find the optimal policy you also need to choose batches of initial states to evaluate (32). We suggest that you sample states from the expert trajectory. In our experiments, using $B = 1$ worked good in practice.

Task 4 Implement the optimal policy $\pi(\alpha^*)$ computed in Task 3. Record the sequence of states s_t and actions a_t as the trajectory unfolds. Evaluate the performance by comparing the generated trajectory with an expert trajectory. Plot both trajectories.

5 Exploratory Policies

RL problems are unique among the problems we have studied in that the training data is chosen. In both, the original formulation of the optimal policy in (17) and the actor formulation in (29) we optimize over the initial state distribution $p(s)$. In Task 3 we choose $p(s)$ as the reference trajectory $\mathcal{S}$, but this is an arbitrary choice. We can also, as we did in Lab 8, use dithering of initial states around the reference trajectory.

In RL it is standard to use random exploration around policy trajectories. We describe this formally by introducing an exploratory policy in which actions are chosen at random with probability δ and according to $\pi(\alpha)$ with probability $1-\delta$,

$$\begin{aligned} \pi_{\mathrm{E}}(s,\alpha) &= \mathcal{N}(0,\sigma^2\mathbf{I}), && \text{with probability } \delta, \\ \pi_{\mathrm{E}}(s,\alpha) &= \pi(s,\alpha), && \text{with probability } 1-\delta. \end{aligned} \tag{34}$$

Policy rollouts in (31), or (32) if using batches of initial states, proceed according to the exploratory policy $\pi_{\mathrm{E}}(s,\alpha)$

The idea of exploratory policies is to let the actor visit states around the trajectory generated by the current policy. The rationale for these random visits is to explore during training a set of states that is more representative of the set of states that will be seen during execution. We use a combination of states that we expect to see a priori – drawn from $p(s)$ – and states that we expect to see a posteriori – drawn from the neighborhood of a policy rollout.

Task 5 Repeat Task 3 with an exploratory policy. Choose $\delta = 0.01$ for the probability of choosing a random acceleration. When choosing random accelerations draw them from a white normal distirbution with mean 0 and variance $\sigma^2 = 0.1$.

Implement the optimal policy $\pi(\alpha^*)$ obtained with the exploratory policy. Observe that we use exploration during training but not during execution. Record the sequence of states s_t and actions a_t as the trajectory unfolds. Evaluate the performance by comparing the generated trajectory with an expert trajectory. Plot both trajectories.

6 Critic Training

As stated in Section 4.1, a critic $\tilde{Q}(s,a;\beta)$ is a parametric approximation of the true Q-function $Q(s,a;\pi)$ of a policy π. To train this approximation we use policy rollouts to evaluate the Q-function. Thus, from initial condition $(s,a) = (s,\pi(s))$ we execute policy π and follow the MDP's

dynamics to estimate the Q-function as

$$Q(s,a;\pi) \;=\; Q(s,\pi(s);\pi) \;=\; \sum_{t=0}^{\infty} \gamma^t \, r(s_t,a_t) \tag{35}$$

This is the same as (8), except that we make $a = \pi(s)$ as this choice of initial action is required in the definition of the critic.

To train the critic we evaluate the squared loss of the critic and Q-function difference averaged over an initial distribution of states $p(s)$,

$$\beta^*(\pi) = \underset{\beta}{\operatorname{argmin}} \, \mathbb{E}_{p(s)} \left[\frac{1}{2} \Big\| Q(s,a,\pi) - \tilde{Q}(s,a;\beta) \Big\|^2 \right], \tag{36}$$

where it is implicit that $a = \pi(s)$ in both, the Q-function and its critic.

The loss in (36) can be difficult to train because we need to acquire several estimates of the Q-function. Each of these estimates requires a rollout of the policy π to evaluate the discounted sum of rewards in (35).

A simpler training approach can be derived from the recursion in Proposition 1. This recursion claims that for any pair of states s and $s^+ = f(s,a)$ and their corresponding actions $a = \pi(s)$ and $a^+ = \pi(s^+)$,

$$Q(s,a;\pi) \;=\; r(s,a) + \gamma Q(s^+,a^+;\pi), \tag{37}$$

If we have a critic $\tilde{Q}(s,a;\beta)$ that approximates the Q-function well, this critic must satisfy this same recursion. That is, we must have that

$$\tilde{Q}(s,a;\beta) \;=\; r(s,a) + \gamma \tilde{Q}(s^+,a^+;\beta) + \Delta(s,a,\beta), \tag{38}$$

for some function $\Delta(s,a,\beta)$ that is *small* for all state-action pairs $(s,a) = (s,\pi(s))$ and their subsequent pairs $(s^+,a^+) = (s^+,\pi(s^+))$.

The smaller that the function $\Delta(s,a,\beta)$ is in (37), the closer that $\tilde{Q}(s,a;\beta)$ is to satisfying the Q-function recursion in (37) and, by extension, the closer that $\tilde{Q}(s,a;\beta)$ is to the true Q-function $Q(s,a;\pi)$. We can therefore train the critic $\tilde{Q}(s,a;\beta)$ by minimizing the loss,

$$\beta^*(\pi) \;=\; \underset{\beta}{\operatorname{argmin}} \;\; \mathbb{E}_{p(s)} \left[\frac{1}{2} \Big\| \Delta(s,a,\beta) \Big\|^2 \right], \tag{39}$$

Solving for $\Delta(s, a, \beta)$ in (38) and substituting the result in (39) we obtain the training loss

$$\beta^*(\pi) = \underset{\beta}{\operatorname{argmin}}\, \mathbb{E}_{p(s)} \left[\frac{1}{2} \left\| \left(\tilde{Q}(s, a; \beta) - \gamma \tilde{Q}(s^+, a^+; \beta) \right) - r(s, a) \right\|^2 \right]. \quad (40)$$

This loss has an intuitive interpretation. It says that the difference between the Q-function value $\tilde{Q}(s, a; \beta)$ at state s and the (discounted) Q-function value $\tilde{Q}(s^+, a^+; \beta)$ at state s^+ is the reward $r(s, a)$ accrued in the transition. We penalize deviations from this condition.

In the particular case in which policies π are parameterized by α the critic in $\beta^*(\pi)$ can be written as a function of the parameter α. In this case it is convenient to define the loss $\ell_{\mathrm{C}}(\alpha, \beta)$ to represent the objective of the minimization problem in (40),

$$\begin{aligned} \beta^*(\alpha) &= \underset{\beta}{\operatorname{argmin}}\, \mathbb{E}_{p(s)} \left[\frac{1}{2} \left\| \left(\tilde{Q}(s, a; \beta) - \gamma \tilde{Q}(s^+, a^+; \beta) \right) - r(s, a) \right\|^2 \right] \\ &= \underset{\beta}{\operatorname{argmin}}\, \ell_{\mathrm{C}}(\alpha, \beta). \end{aligned} \quad (41)$$

Actions in (41) are chosen from the policy $\pi(\alpha)$ so that $a = \pi(s, \alpha)$ and $a^+ = \pi(s^+, \alpha)$. The state $s^+ = f(s, a)$ follows from the dynamics of the MDP. It is interesting that the critic loss $\ell_{\mathrm{C}}(\alpha, \beta)$ is minimized with respect to β with α fixed and that the actor loss $\ell_{\mathrm{A}}(\alpha, \beta)$ is maximized with respect to α with β fixed [cf. (41) and (29)].

6.1 Critic Rollouts

To solve the statistical risk minimization (SRM) problem in (41) we consider samples $s_i \sim p(s)$ to formulate and solve the empirical risk minimization (SRM) problem defined by the loss

$$\hat{\ell}_{\mathrm{C}}(\alpha, \beta) = \frac{1}{B} \sum_{i=1}^{B} \frac{1}{2} \left\| \left(\tilde{Q}(s_i, a_i; \beta) - \gamma \tilde{Q}(s_i^+, a_i^+; \beta) \right) - r(s_i, a_i) \right\|^2. \quad (42)$$

As it should go without saying by now, actions in (42) – as they are in (41) – are chosen as $a_i = \pi(s_i, \alpha)$ and as $a_i^+ = \pi(s_i^+, \alpha)$. The next states follow – as they follow in (41) – from execution of the MDP dynamics, $s_i^+ = f(s_i, a_i)$.

An alternative approach to training the critic is to consider policy rollouts. Let s be an initial state and s_t be the trajectory generated by rolling out the policy π. We train the critic $\tilde{Q}(s, a; \beta)$ by minimizing the empirical risk,

$$\hat{\ell}_{\mathrm{C}}(\alpha, \beta) = \sum_{t=1}^{\infty} \frac{1}{2} \left\| \left(\tilde{Q}(s_i, a_i; \beta) - \gamma \tilde{Q}(s_i^+, a_i^+; \beta) \right) - r(s_i, a_i) \right\|^2. \tag{43}$$

The ERM problems defined by the losses in (42) and (43) are *not* equivalent. They are evaluating the critic fit over different datasets. In (42) we measure fit over the initial state distribution $p(s)$. In (42) we measure fit over the distribution of states that are visited when we execute the policy π.

A third alternative is to combine (42) and (43). We consider initial state samples $s_i \sim p(s)$ along with trajectories s_{it} obtained by rolling out the policy π starting at initial condition s_i. We then train the critic to minimize the loss,

$$\hat{\ell}_{\mathrm{C}}(\alpha, \beta) = \frac{1}{B} \sum_{i=1}^{B} \sum_{t=0}^{\infty} \frac{1}{2} \left\| \left(\tilde{Q}(s_{it}, a_{it}; \beta) - \gamma \tilde{Q}(s_{it}^+, a_{it}^+; \beta) \right) - r(s_{it}, a_{it}) \right\|^2. \tag{44}$$

It is worth comparing (43) and (44) with the critic stochastic gradients in (31) and (32). Equations (31) and (43) estimate gradients and risks using policy rollouts. Equations (32) and (44) estimate gradients and risks combining policy rollouts with an exogenous distribution of initial states. In both cases the latter is the preferred training method.

In both cases we are encountering the exploration challenge. Since training states in RL are chosen, we must choose states during training that are representative of states we expect to see during execution. This is the reason why we use a combination of states that we expect to see a priori – drawn from $p(s)$ – and states that we expect to see a posteriori – drawn from a policy rollout. As discussed in Section 5 the exploration challenge is unique among problems we have studied.

Task 6 For a fixed policy π train a critic to estimate the Q-function corresponding to this policy. Train this policy by minimizing (44). Use $\gamma = 0.99$ as a discount factor, $B = 256$ for the batch size, and $T = 250$ as a large enough number to approximate infinity. Sample initial states from the reference trajectory $\mathcal{S}$.

The critic requires a learning parameterization. We suggest that you use an FCNN in which the inputs are state (s), observation (o), and acceleration attributes of a car object instantiated from the class of Task 2. The output of this FCNN is the critic value,

$$\tilde{Q}(s, a, \beta) = \text{FCNN}(s, o, a). \tag{45}$$

An FCNN with 4 hidden layers containing $n_\ell = 128$ hidden neurons each has worked well in our experiments.

Train a critic to estimate the Q-function associated with the optimal policy $\pi(\alpha^*)$ learned in Task 3. Compute the loss $\Delta(s, a, \beta)$ over a trajectory generated by α^* to evaluate the quality of critic estimates of the Q-function.

7 Actor-Critic Reinforcement Learning

In Task 3 we assume that a critic is available and use it to train a policy. In Task 6 we assume a policy is available and use it to train a critic. There seems to be some circular reasoning here that we have not explained well.

To explain it well, recall that the goal of an actor is to maximize the actor loss defined in (29) when a critic is given,

$$\alpha^*(\beta) = \underset{\alpha}{\text{argmax}}\, \ell_{\text{A}}(\alpha, \beta) \tag{46}$$

Similarly, the goal of a critic is to maximize the actor loss defined in (41) when a critic is given,

$$\beta^*(\alpha) = \underset{\beta}{\text{argmin}}\, \ell_{\text{C}}(\alpha, \beta) \tag{47}$$

This is a chicken and egg situation. The problem we would like to solve is finding the actor $\alpha^*(\beta^*(\alpha^*))$. That is, knowing the optimal policy α^* we find the corresponding optimal critic $\beta^*(\alpha^*)$. This is egg in Task 6. Then, knowing this optimal critic we find the optimal policy $\alpha^*(\beta^*(\alpha^*))$. This is chicken in Task 3. We need the chicken α^* to lay the egg $\beta^*(\alpha^*)$ to birth the chicken $\alpha^*(\beta^*(\alpha^*))$.

As is the case of chickens and eggs, the answer is that eggs and chickens evolve together. Thus, for an iteration index k and a policy initialization

α_0 we proceed through the recursive updates,

$$\begin{aligned} \beta_k &= \operatorname*{argmin}_{\beta} \ \ell_{\mathrm{C}}(\alpha_k, \beta), \\ \alpha_{k+1} &= \operatorname*{argmax}_{\alpha} \ \ell_{\mathrm{A}}(\alpha, \beta_k). \end{aligned} \tag{48}$$

For a given policy α_k we compute the critic β_k (the chicken α_k lays the egg β_k). With this critic available we evolve the actor policy to α_{k+1} (the egg β_k births a better chicken α_{k+1}). We then repeat this process with the computation of updated critics and policies.

As we did in Sections 4.2 and 6.1 we use empirical approximations of the statistical losses in (48). Thus, in lieu of the iterations in (48) we use the iterations

$$\begin{aligned} \beta_k &= \operatorname*{argmin}_{\beta} \ \hat{\ell}_{\mathrm{C}}(\alpha_k, \beta), \\ \alpha_{k+1} &= \operatorname*{argmax}_{\alpha} \ \hat{\ell}_{\mathrm{A}}(\alpha, \beta_k). \end{aligned} \tag{49}$$

The empirical actor loss $\hat{\ell}_{\mathrm{A}}(\alpha, \beta_k)$ is given in (32). The empirical critic loss $\hat{\ell}_{\mathrm{C}}(\alpha_k, \beta)$ is given in (44). The iteration described by (49) is the actor-critic method.

Task 7 Train the actor and the critic using the iterative actor-critic method described by (49). Use the same FCNN parameterization of Task 3 for the actor and the same FCNN parameterization of Task 6 for the critic. As we did in Tasks 3 and 6, sample initial conditions from the expert trajectory $\mathcal{S}$. Also, use the same hyper-parameters γ, B and T as in Tasks 3 and 6.

Although (49) calls for iterative critic minimization and actor maximization, in practice we run K_{C} stochastic gradient descent steps on the critic loss and K_{A} stochastic gradient ascent steps on the actor loss. Set $K_{\mathrm{C}} = K_{\mathrm{A}} = 20$.

Actor-critic training is more expensive than separate actor and critic training. To reduce computation cost we use experience replay buffers. This means that we store all the states s_{it}, actions a_{it}, rewards $r(s_{it}, a_{it})$ and the next state s_{it}^{+} and next action a_{it}^{+} that are created by policy rollouts. When implementing gradient steps in the actor and critic losses we draw batches from these histories – as opposed to drawing new batches and generating new trajectories.

Evaluate the accumulated reward per episode and the critic's loss during training. Comment.

Task 8 Implement the policy $\pi(\alpha^*)$ computed in Task 7. Record the sequence of states s_t and actions a_t as the trajectory unfolds. Evaluate the performance by comparing the generated trajectory with an expert trajectory. Plot both trajectories.

8 Report

Do not take much time to prepare a lab report. We do not want you to report your code and we don't want you to report your work. Just give us answers to the specific questions we ask. Specifically, provide the following:

Question	**Report deliverable**
Task 1	Do not report.
Task 2	Do not report.
Task 3	Actor loss as a function of training iteration index.
Task 4	Plot of expert trajectory and reference trajectory.
Task **??**	Actor loss as a function of training iteration index. Plot of expert trajectory and reference trajectory.
Task 6	Critic loss as a function of training iteration index.
Task 7	Actor loss as a function of training iteration index. Critic loss as a function of training iteration index.
Task 8	Plot of expert trajectory and reference trajectory.