3 Model-based and Model-free RL for Robot Control

The previous chapter introduced the deterministic and stochastic sequential decision making problems, and demonstrated how these problems can be solved by dynamic programming. While dynamic programming is a powerful algorithm, it also suffers from several practical challenges. This chapter briefly introduces some of the key ideas in *reinforcement learning*[1,2], a set of ideas which aims to solve a more general problem of behaving in an optimal way within a given *unknown* environment. That is, the reinforcement learning setting assumes only the ability to (1) interact with an unknown environment and (2) receive a reward signal from it. How the actions affect the future state evolution or the future reward is not known *a priori*. Reinforcement learning includes a class of approximation algorithms which can be much more practical than dynamic programming in real world applications.

[1] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019

[2] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018

Reinforcement Learning

Reinforcement learning (RL) is a broad field that studies autonomous sequential decision making, but extends to more general and challenging problems than have been considered in previous chapters. The standard RL problem is to determine closed-loop control policies that drive an agent to maximize an accumulated reward[3]. However, in the general case it is not required that a *system model* be known! This paradigm can be represented by Figure 3.1, where it can be seen that given a control input the environment specifies the next state and reward, and the environment can be considered to be a black box (it is not necessarily known how the state is generated, nor the reward computed).

[3] Note that the maximization of "reward" in the context of reinforcement learning is essentially equivalent to minimization of "cost" in optimal control formulations.

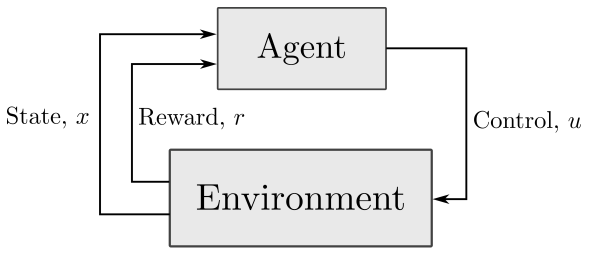

Figure 3.1: In reinforcement learning problems, the robot (agent) learns how to make decisions by interacting with the environment.

To account for this model uncertainty (which is notably distinct from the state transition uncertainty inherent in a stochastic but *known* system model, as considered in the previous chapter), an agent must instead learn from its experience to produce good policies. Concisely, RL deals with the problem of how to learn to act optimally in the long term, from interactions with an unknown environment which provides only a momentary reward signal.

RL is a highly active field of research, and has seen successes in several applications including acrobatic control of helicopters, games, finance, and robotics. In this chapter the fundamentals of reinforcement learning are introduced, including the formulation of the RL problem, RL algorithms that leverage system models ("model-based" methods; value iteration/dynamic programming and policy iteration), and a few RL algorithms that do not require system models ("model-free" methods; Q-learning, policy gradient, actor-critic).

3.1 Problem Formulation

The problem setting of reinforcement learning is similar to that of stochastic sequential decision making from the previous chapter, but here we will adopt slightly different notation more consistent with how Markov Decision Processes (MDPs) are typically framed in this community.[4] The state and control input for the system is denoted as $\boldsymbol{x}$ and $\boldsymbol{u}$, and the set of admissible states and controls are denoted as $\mathcal{X}$ and $\mathcal{U}$. However, the stochastic state transition model will now be written explicitly as a probability distribution (where before this was implicit in the influence of the stochastic variables $\boldsymbol{w}$ on the system dynamics f):

[4] The fields of optimal control and reinforcement learning have significant overlap, but each community has developed its own standard notation. Most often, the state in the optimal control community is represented by $\boldsymbol{x}$ and in the RL community as $\boldsymbol{s}$. Similarly, in control theory the control input is $\boldsymbol{u}$ while in the RL community it is referred to as an action $\boldsymbol{a}$.

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), \tag{3.1}$$

which is the conditional probability distribution over $\boldsymbol{x}_t$, given the previous state and control. The environment also has a reward function which defines the reward associated with every state and control

$$r_t = R(\boldsymbol{x}_t, \boldsymbol{u}_t). \tag{3.2}$$

The goal of the RL problem is to interact with the environment over a (possibly infinite) time horizon and *accumulate the highest possible reward in expectation*. To accommodate infinite horizon problems and to account for the fact that an agent is typically more confident about the ramifications of its actions in the short term than the long term, the accumulated reward is typically defined as the *discounted total expected reward* over time

$$E\left[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \boldsymbol{u}_t)\right], \tag{3.3}$$

where $\gamma \in (0, 1]$ is referred to as a *discount factor*. The tuple

$$\mathcal{M} = \big(\mathcal{X}, \mathcal{U}, p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), R(\boldsymbol{x}_t, \boldsymbol{u}_t), \gamma\big)$$

defines the Markov Decision Process (MDP), the environment in which the reinforcement learning problem is set.

In this chapter we will consider infinite horizon MDPs for which the notion of a stationary policy π applied at all time steps, i.e.,

$$\boldsymbol{u}_t = \pi(\boldsymbol{x}_t), \tag{3.4}$$

is appropriate. The goal of the RL problem is to choose a policy that maximizes the cumulative discounted reward

$$\pi^* = \arg\max_{\pi} \; E\left[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)))\right] \tag{3.5}$$

where the expectation is notionally computed with respect to the stochastic dynamics p, but in practice is estimated empirically by drawing samples from the environment encoding $\mathcal{M}$ (i.e., in constructing π we may not assume exact knowledge of $\mathcal{M}$).

3.1.1 *Value function*

A policy π defines a value function which corresponds to the expected reward accumulated starting from a state $\boldsymbol{x}$

$$V^{\pi}(\boldsymbol{x}) = E\big[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_t(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\big], \tag{3.6}$$

which can also be expressed in the *tail* formulation

$$V^{\pi}(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V^{\pi}(\boldsymbol{x}'). \tag{3.7}$$

The optimal policy π^* satisfies *Bellman's equation*

$$\begin{aligned} V^{\pi^*}(\boldsymbol{x}) = V^*(\boldsymbol{x}) &= \max_{\boldsymbol{u} \in \mathcal{U}} \left(R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}') \right) \\ \pi^*(\boldsymbol{x}) &= \arg\max_{\boldsymbol{u} \in \mathcal{U}} \left(R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}') \right) \end{aligned} \tag{3.8}$$

and also satisfies $V^*(\boldsymbol{x}) = V^{\pi^*}(\boldsymbol{x}) \geq V^{\pi}(\boldsymbol{x})$ for all $\boldsymbol{x} \in \mathcal{X}$ for any alternative policy π. That is, the optimal policy induces the maximum value function and solves the RL problem of maximizing the accumulated discounted reward.

3.1.2 *Q-function*

Motivated by Bellman's equation above, in addition to the (state) value function $V^{\pi}(\boldsymbol{x})$ it makes sense to define the state-action value function $Q^{\pi}(\boldsymbol{x}, \boldsymbol{u})$ which corresponds to the expected reward accumulated starting from a state $\boldsymbol{x}$ and taking a first action $\boldsymbol{u}$ before following the policy π for all subsequent time steps. That is,

$$Q^{\pi}(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^{\pi}(\boldsymbol{x}'). \tag{3.9}$$

Similarly, the *optimal* Q-function is:

$$Q^*(\boldsymbol{x},\boldsymbol{u}) = R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) V^*(\boldsymbol{x}'),$$

where the shorthand notation $Q^*(\boldsymbol{x},\boldsymbol{u}) = Q^{\pi^*}(\boldsymbol{x},\boldsymbol{u})$ is used. Note that from the Bellman equation (3.8) the optimal value function can be written as an optimization over the optimal Q-function:

$$V^*(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} Q^*(\boldsymbol{x},\boldsymbol{u}),$$

so,

$$Q^*(\boldsymbol{x},\boldsymbol{u}) = R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) \max_{\boldsymbol{u}' \in \mathcal{U}} \Big(Q^*(\boldsymbol{x}',\boldsymbol{u}') \Big).$$

Therefore, instead of computing the optimal value function using value iteration it is possible to deal directly with the Q-function!

3.2 *Model-based Reinforcement Learning*

Model-based reinforcement learning methods rely on the use of an explicit parameterization of the transition model (3.1), which is either fit to observed transition data (i.e., learned) or, in special cases, known *a priori*. For example, for discrete state/control spaces it is possible to empirically approximate the transition probabilities $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ for every pair $(\boldsymbol{x}_t, \boldsymbol{u}_t)$ by counting the number of times each transition occurs in the dataset! More sophisticated models include linear models generated through least squares, or Gaussian process or neural network models trained through an appropriate loss function. Given a learned model, the problem of optimal policy synthesis reduces to the sequential decision making problem of the previous chapter.

3.2.1 *Value Iteration (Dynamic Programming)*

While the dynamic programming algorithm was covered in the previous chapter, it will also be included here in the context of the RL problem formulation. In this case, the "principle of optimality" again says that the optimal *tail* policy is optimal for *tail* subproblems, which leads to the recursion:

$$V^*_{k+1}(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} \Big(R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) V^*_k(\boldsymbol{x}') \Big), \tag{3.10}$$

which is commonly referred to as the *Bellman recursion*. In words, the optimal reward associated with starting at the state $\boldsymbol{x}$ and having $k+1$ steps to go can be found as an optimization over the immediate control by accounting for the (expected) optimal tail rewards. The full dynamic programming algorithm for solving the RL problem (3.5) is given in Algorithm 1.

In the context of RL, this procedure is commonly referred to as *value iteration* and in many cases it is assumed that the horizon N is infinite. For infinite-horizon problems the "value iteration" in Algorithm 1 is performed either over

Algorithm 1: Dynamic Programming/Value Iteration (RL)

```
$V_0^*(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ to $N-1$ do
    $V_{k+1}^*(\boldsymbol{x}) = \max_{\boldsymbol{u}\in\mathcal{U}} R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) V_k^*(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
    $\pi_{N-1-k}^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}\in\mathcal{U}} R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) V_k^*(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
return $V_0^*(\cdot), \dots, V_N^*(\cdot), \pi_0^*(\cdot), \dots, \pi_{N-1}^*(\cdot)$
```

a finite-horizon (which yields an approximate solution), or until convergence to a stationary (i.e. time-invariant) optimal value function/policy[5].

[5] In the infinite horizon case, the optimal value function is unique and the optimal policy is stationary and deterministic, but not necessarily unique.

To solidify the relationship between value iteration in the context of RL and dynamic programming in the context of stochastic decision making from the previous chapter, the inventory control example from the previous chapter is revisited:

Example 3.2.1 (Inventory Control)**.** Consider again the inventory control problem from the previous chapter, where the available stock of a particular item is the state $x_t \in \mathbb{N}$, the control $u_t \in \mathbb{N}$ adds items to the inventory, the demand w_t is uncertain, and the dynamics and constraints are:

$$\begin{aligned} x_t &= \max\{0, x_{t-1} + u_{t-1} - w_{t-1}\}, \\ p(w=0) &= 0.1, \quad p(w=1) = 0.7, \quad p(w=2) = 0.2. \end{aligned}$$

and

$$x_t + u_t \leq 2.$$

Based on the dynamics, the probabilistic model (3.1) is given by:

$$\begin{aligned} p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 0) &= \{1, 0, 0\}, \\ p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 1) &= \{0.9, 0.1, 0\}, \\ p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 2) &= \{0.2, 0.7, 0.1\}, \\ p(x_t = \{0,1,2\} \mid x_{t-1} = 1, u_{t-1} = 0) &= \{0.9, 0.1, 0\}, \\ p(x_t = \{0,1,2\} \mid x_{t-1} = 1, u_{t-1} = 1) &= \{0.2, 0.7, 0.1\}, \\ p(x_t = \{0,1,2\} \mid x_{t-1} = 2, u_{t-1} = 0) &= \{0.2, 0.7, 0.1\}, \end{aligned}$$

where some transition values are not explicitly written due to the control constraints. Next, the reward function is defined as:

$$\begin{aligned} R(x_t, u_t) &= -E\big[u_t + (x_t + u_t - w_t)^2\big], \\ &= -\big(u_t + (x_t + u_t - E[w_t])^2 + Var(w_t)\big), \end{aligned}$$

and a discount factor of $\gamma = 1$ is used. As in the previous chapter, this reward penalizes (a negative reward is a penalty) ordering new stock and having available stock at the next time step (i.e. having to store stock).

Algorithm 1 can now be applied, starting with the value function with no steps to go:

$$V_0^*(x) = 0,$$

and then recursively computing:

$$\begin{aligned}
V_1^*(0) &= \max_{u \in \{0,1,2\}} -\big(u + (u-1.1)^2 + 0.29\big) = -1.3, \\
V_1^*(1) &= \max_{u_2 \in \{0,1\}} -\big(u + (1+u-1.1)^2 + 0.29\big) = -0.3, \\
V_1^*(2) &= -\big((2-1.1)^2 + 0.29\big) = -1.1,
\end{aligned}$$

where $E[w] = 1.1$ and $Var(w) = 0.29$. The optimal stage policies associated with this step are:

$$\pi_{N-1}^*(0) = 1, \quad \pi_{N-1}^*(1) = 0, \quad \pi_{N-1}^*(2) = 0.$$

In the next step:

$$\begin{aligned}
V_2^*(0) &= \max_{u \in \{0,1,2\}} -\big(u + (u-1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x=0,u) V_1^*(x') = -2.5, \\
V_2^*(1) &= \max_{u \in \{0,1,\}} -\big(u + (1+u-1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x=1,u) V_1^*(x') = -1.5, \\
V_2^*(2) &= -\big((2-1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x=2,u=0) V_1^*(x') = -1.68,
\end{aligned}$$

with optimal stage policies:

$$\pi_{N-2}^*(0) = 1, \quad \pi_{N-2}^*(1) = 0, \quad \pi_{N-2}^*(2) = 0.$$

Finally, in the last step:

$$\begin{aligned}
V_3^*(0) &= \max_{u \in \{0,1,2\}} -\big(u + (u-1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x=0,u) V_2^*(x') = -3.7, \\
V_3^*(1) &= \max_{u \in \{0,1,\}} -\big(u + (1+u-1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x=1,u) V_2^*(x') = -2.7, \\
V_3^*(2) &= -\big((2-1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x=2,u=0) V_2^*(x') = -2.818,
\end{aligned}$$

with optimal stage policies:

$$\pi_{N-3}^*(0) = 1, \quad \pi_{N-3}^*(1) = 0, \quad \pi_{N-3}^*(2) = 0.$$

These results are, in fact, identical to the results from the example in the previous chapter! The only difference is the formulation of the problem in the RL framework instead of the stochastic decision making problem framework.

3.2.2 *Policy Iteration*

Another common algorithm that can be used to solve the reinforcement learning problem (3.5) is *policy iteration*. The main idea of policy iteration is that if the value function can be computed for any arbitrary finite horizon policy $\pi = \{\pi_0, \pi_1, \dots, \pi_{N-1}\}$, then the policy can be incrementally improved to yield a better policy $\pi' = \{\pi'_0, \pi'_1, \dots, \pi'_{N-1}\}$.

Policy Evaluation: The first key element of the policy iteration algorithm is *policy evaluation*, which is used to compute the value function $V_k^\pi(\boldsymbol{x})$ for a given policy π. Policy evaluation is based on the recursion:

$$V_{k+1}^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V_k^\pi(\boldsymbol{x}'), \tag{3.11}$$

which is very similar to the Bellman equation (3.8) except that there is no optimization over the control (since it is fixed). The policy evaluation algorithm is given in Algorithm 2.

```
Algorithm 2: Policy Evaluation
  Data: $\pi$
  Result: $V_0^\pi(\cdot), \dots, V_N^\pi(\cdot)$
  $V_0^\pi(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
  for $k = 0$ to $N - 1$ do
      $V_{k+1}^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi_{N-1-k}(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi_{N-1-k}(\boldsymbol{x})) V_k^\pi(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
  return $V_0^\pi(\cdot), \dots, V_N^\pi(\cdot)$
```

In the infinite-horizon case where a stationary policy is used, the iteration in Algorithm 2 stops when the value function has converged to its stationary value. Indeed, since the infinite horizon value function is the stationary point of this recursion, it is possible to directly solve for it by setting both $V_{k+1}^\pi = V_k^\pi = V_\infty^\pi$ in (3.11). In the case of a discrete state space with N possible states, this creates a linear system of N equations which can be used to solve for V_∞^π directly.

Policy Iteration Algorithm: The policy iteration algorithm incrementally updates the policy by performing local optimizations of the Q-function. In particular, a single iteration of the policy update is shown in Algorithm 3. It can be proven

```
Algorithm 3: Policy Iteration Step
  Data: $\pi$
  Result: $\pi'$
  $V_0^\pi(\cdot), \dots, V_N^\pi(\cdot) \leftarrow$ PolicyEvaluation($\pi$)
  for $k = 0$ to $N - 1$ do
      $Q_{k+1}^\pi(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^\pi(\boldsymbol{x}')$ for all $\boldsymbol{x} \in \mathcal{X}$
      $\pi'_{N-1-k}(\boldsymbol{x}) = \arg\max_{\boldsymbol{u} \in \mathcal{U}} Q_{k+1}^\pi(\boldsymbol{x}, \boldsymbol{u})$, for all $\boldsymbol{x} \in \mathcal{X}$
  return $\pi' = \{\pi'_0, \dots, \pi'_{N-1}\}$
```

theoretically that under the policy iteration algorithm the value function is monotonically increasing with each new policy, and the procedure is run until convergence. While policy iteration and value iteration are quite similar, policy iteration can end up converging faster in some cases.

3.3 *Model-free Reinforcement Learning*

The value and policy iteration algorithms are applicable only to problems where the model $\mathcal{M}$ is *known*, i.e., they rely on direct access to the probabilistic system dynamics $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ and reward function $R(\boldsymbol{x}_t, \boldsymbol{u}_t)$, or at least learned approximations of these functions fit to observed data. Model-free RL algorithms, on the other hand, sidestep the explicit consideration of p and R entirely.

3.3.1 *Q-Learning*

The canonical model-free reinforcement learning algorithm is *Q-learning*. The core idea behind Q-learning is that it is possible to collect data samples $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ from interaction with the environment, and over time learn the long-term value of taking certain actions in certain states, i.e., directly learning the optimal Q-function $Q^*(\boldsymbol{x}, \boldsymbol{u})$. For simplicity an infinite-horizon ($N = \infty$) problem will be considered, such that the optimal value and Q-functions will be stationary, and in particular the optimal Q-function will satisfy:

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}', \boldsymbol{u}').$$

In a model-free context, the dynamics p above are notional (i.e., the problem is described by some MDP $\mathcal{M}$, we just don't know exactly what it is).[6] We may instead rewrite the above equation in terms of an expectation over trajectory samples drawn from p (i.e., drawn from the environment as a "black box") while implementing the policy $\boldsymbol{u}_t = \pi^*(\boldsymbol{x}_t)$:

[6] Or even if we have a environment simulator, in which case it could be argued that the dynamics are exactly described by the simulation code, the dynamics are too complex/opaque to be considered in this form.

$$Q^*(\boldsymbol{x}_t, \boldsymbol{u}_t) = E\big[r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')\big],$$

or equivalently,

$$E\left[\left(r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')\right) - Q^*(\boldsymbol{x}, \boldsymbol{u})\right] = 0,$$

where $(r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')) - Q^*(\boldsymbol{x}, \boldsymbol{u})$ is known as the *temporal difference error*. The idea of Q-learning is that an approximation of the optimal Q-function can be improved over time by collecting data and trying to ensure that the above conditions holds. This leads to the Q-learning algorithm described in Algorithm 4. The iterations of Q-learning, each a local deterministic correction to the Q-function, in aggregate aim to ensure that the expected temporal difference error is 0.

Q-learning is referred to as a *model-free* method because it forgoes explicitly estimating the true (unknown) system dynamics, and directly estimates the Q-function. It is also called a *value-based* model-free method since it does not directly build the policy, but rather estimates the optimal Q-function to implicitly define the policy. Q-learning is also called an *off-policy* algorithm because the Q-function can be learned from stored experiences and does not require interacting with the environment directly.

```
Algorithm 4: Q-learning
    Data: Set $\mathcal{S}$ of trajectory samples $\{x_t, u_t, r_t, x_{t+1}\}$, learning rate $\alpha$
    Result: $Q(x, u)$
    Initialize $Q(x, u)$ for all $x \in \mathcal{X}$ and $u \in \mathcal{U}$
    for $\{x_t, u_t, r_t, x_{t+1}\} \in \mathcal{S}$ do
        $Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \alpha\Big(r_t + \gamma \max_{u \in \mathcal{U}} Q(x_{t+1}, u) - Q(x_t, u_t)\Big)$
    return $Q(x, u)$
```

Q-learning can be guaranteed to converge to the optimal Q-function under certain conditions, but has some practical disadvantages. In particular, unless the number of possible states and controls are finite and relatively small, it can be intractable to store the Q-value associated with each state-control pair. Another disadvantage of Q-learning is that sometimes the Q-function can be complex and therefore potentially hard to learn.

Fitted Q-learning: One variation of the Q-learning algorithm to handle large or continuous state and control spaces is to parameterize the Q-function as $Q_\theta(x, u)$ and to simply update the parameters $\boldsymbol{\theta}$. This approach is also known as *fitted Q-learning*. While this method often works well in practice, convergence is not guaranteed.

A principled way of performing fitted Q-learning involves minimizing the expected squared temporal difference error for the Q-function

$$E\left[\Big(\big(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(x_{t+1}, u')\big) - Q_\theta(x_t, u_t)\Big)^2\right].$$

For a given parameterization θ fitted Q-learning minimizes the total temporal difference error over all collected transition samples

$$\theta^* = \arg\min_\theta \frac{1}{|S_{\text{exp}}|} \sum_{(x_t, u_t, x_{t+1}, r_t) \in S_{\text{exp}}} \big(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(x_{t+1}, u') - Q_\theta(x_t, u_t)\big)^2$$

where S_{exp} denotes the experience set of all transition tuples with a reward signal. This minimization is typically performed using stochastic gradient descent, yielding the parameter update

$$\theta \leftarrow \theta + \alpha \left(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(x_{t+1}, u') - Q_\theta(x_t, u_t)\right) \nabla_\theta Q_\theta(x_t, u_t)$$

applied iteratively for each $(x_t, u_t, x_{t+1}, r_t) \in S_{\text{exp}}$.

3.3.2 *Policy Gradient*

The *policy gradient* method is another algorithm for model-free reinforcement learning. This approach, which directly optimizes the policy, can be particularly useful for scenarios where the optimal policy may be relatively simple compared to the Q-function, in which case Q-learning may be challenging.

In the policy gradient approach, a class of *stochastic*[7] candidate policies $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$ is defined based on a set of parameters $\boldsymbol{\theta}$, and the goal is to *directly* modify the parameters $\boldsymbol{\theta}$ to improve performance. This is accomplished by using trajectory data to estimate a gradient of the performance with respect to the policy parameters $\boldsymbol{\theta}$, and then using the gradient to update $\boldsymbol{\theta}$. Because this method works directly on a policy (and does not learn a model or value function), it is referred to as a *model-free policy-based* approach.

[7] A stochastic policy defines a distribution over actions at a given state, is useful for exploration, and sometimes is even required for optimality.

Considering the original problem (3.5), the objective function can be written as:

$$J(\boldsymbol{\theta}) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t))\Big],$$

where the $J(\boldsymbol{\theta})$ notation is used to explicitly show the dependence on the parameters. Implementing a policy gradient approach therefore requires the computation of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. One of the most common approaches is to *estimate* this quantity using data, using what is known as a *likelihood ratio method*.

Let τ represent a *trajectory* of the system (consisting of sequential states and actions) under the current policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$. As a shorthand notation, consider the total discounted reward over a trajectory τ to be defined written as:

$$r(\tau) = \sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t), \qquad (3.12)$$

such that $J(\boldsymbol{\theta})$ can be expressed equivalently as $J(\boldsymbol{\theta}) = E\big[r(\tau)\big]$. Additionally, let the probability that the trajectory τ occurs be expressed by the distribution $p_{\boldsymbol{\theta}}(\tau)$. Then the expectation from the objective function can be expanded as:

$$J(\boldsymbol{\theta}) = \int_{\tau} r(\tau) p_{\boldsymbol{\theta}}(\tau) d\tau,$$

and its gradient given by:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int_{\tau} r(\tau) p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) d\tau.$$

From standard calculus $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \frac{1}{p_{\boldsymbol{\theta}}(\tau)} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)$, which replaces the use of the gradient $\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)$ with $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$. This is a very useful "trick" when it comes to approximately computing the integral, as will be seen shortly.

Rather than explicitly computing this integral it is much easier to approximate using sampled data (i.e. sampled trajectories). This is possible since the integral can be written as the expectation $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = E\big[r(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)\big]$, which can be estimated using a Monte Carlo method. While in general a number of sampled trajectories could be used to estimate the gradient, for data efficiency it is also possible to just use a single sampled trajectory τ and approximate:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx r(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau). \qquad (3.13)$$

In particular the sampled quantities $r(\tau)$ can be directly computed from (3.12), and it turns out that the term $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ can be evaluated quite easily as[8]:

$$\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \sum_{t=0}^{N-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t). \qquad (3.14)$$

[8] Using Bayes' rule: $p_{\boldsymbol{\theta}}(\tau) = p(\boldsymbol{x}_0)\Pi_{t=1}^{N-1} p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}) \pi_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{u}_{t-1})$. Then the log converts the product into a sum.

Importantly, notice that only the gradient of the policy is needed, and knowledge of the transition model $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ is not! This occurs because only the policy is dependent on the parameters $\boldsymbol{\theta}$.

In summary, the gradient of $J(\theta)$ can be approximated given a trajectories τ under the current policy $\pi_{\boldsymbol{\theta}}$ by:

1. Compute $r(\tau)$ for the sampled trajectory using (3.12).

2. Compute $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ for the sampled trajectory using (3.14), which only requires computing gradients related to the current policy $\pi_{\boldsymbol{\theta}}$.

3. Approximate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ using (3.13).

The process of sampling trajectories from the current policy, approximating the gradient, and performing a gradient descent step on the parameters $\boldsymbol{\theta}$ is referred to as the *REINFORCE* algorithm[9].

[9] There are some other modified versions of this algorithm, for example some contain a baseline term $b(\boldsymbol{x}_0)$ in the gradient by replacing $r(\tau)$ with $r(\tau) - b(\boldsymbol{x}_0)$ to reduce the variance of the Monte Carlo estimate.

In general, policy-based RL methods such as policy gradient can converge more easily than value-based methods, can be effective in high-dimensional or continuous action spaces, and can learn stochastic policies. However, one challenge with directly learning policies is that they can get trapped in undesirable local optima. Policy gradient methods can also be data inefficient since they require data from the *current* policy for each gradient step and cannot easily reuse old data. This is in contrast to Q-learning, which is agnostic to the policy used and therefore doesn't waste data collected from past interactions.

3.3.3 *Actor-Critic*

Another popular reinforcement learning algorithm is the *actor-critic* algorithm, which blends the concepts of value-based and policy-based model-free RL. In particular, a parameterized policy $\pi_{\boldsymbol{\theta}}$ (actor) is learned through a policy gradient method along side an estimated value function for the policy (critic). The addition of the critic helps to reduce the variance in the gradient estimates for the actor policy, which makes the overall learning process more data-efficient[10].

[10] This is a similar variance reduction approach to adding a baseline $b(\boldsymbol{x}_\tau)$ to the REINFORCE. In fact the baseline is chosen as the value function!

In particular, the policy $\pi_{\boldsymbol{\theta}}$ is again learned through policy gradient like in the REINFORCE algorithm, but with the addition of a learned approximation of the value function $V_{\boldsymbol{\phi}}(\boldsymbol{x})$ as a baseline:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{t=0}^{N-1} \big(r(\tau) - V_{\boldsymbol{\phi}}(\boldsymbol{x}_0)\big) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t).$$

Recall that the value function $V(\boldsymbol{x})$ quantifies the expected total return starting from state $\boldsymbol{x}$ (i.e. the average performance). Therefore the quantity $r(\tau) - V_{\boldsymbol{\phi}}(\boldsymbol{x}_0)$ now represents a performance increase over average. Of course in this method the learned value function approximation $V_{\boldsymbol{\phi}}(\boldsymbol{x})$ is also updated along with the policy by performing a similar gradient descent on the parameters $\boldsymbol{\phi}$.

3.4 Deep Reinforcement Learning

Neural networks are a powerful function approximator that can be utilized in reinforcement learning algorithms.

Q-learning: In Q-learning the Q-function can be approximated by a neural network to extend the approach to nonlinear, continuous state space domains.

Policy Gradient: In policy gradient methods, the policy π_θ can be parameterized as a neural network, enabling the policy to operate on high-dimensional states including images (i.e. visual feedback)!

Actor-Critic: In actor-critic methods, both the policy π_θ and the value function V_ϕ can be parameterized as a neural network which often leads to a space efficient nonlinear representations of the policy and the value function.

3.5 Exploration vs Exploitation

When learning from experience (e.g. using Q-learning, policy gradient, actor-critic, deep RL, etc.) it is important to ensure that the experienced trajectories (i.e. the collected data points) are meaningful! For example, an abundance of data related to a particular set of actions/states will not necessarily be sufficient to learn good policies for *all* possible situations. Therefore an important part of reinforcement learning is *exploring* different combinations of states and actions. One simple approach to exploration is referred to as ϵ-greedy exploration, where a random control is applied instead of the current (best) policy with probability ϵ.

However, exploration can lead to suboptimal performance since any knowledge accumulated about the optimal policy is ignored[11]. This leads to the *exploration vs exploitation* trade-off: a fundamental challenge in reinforcement learning.

[11] In other words, actions with known rewards may be foregone in the hope that exploring leads to an even better reward.

Bibliography

[1] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019.

[2] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018.