21
Reinforcement Learning

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], which encompasses a set of ideas that can generally be viewed as extensions or approximations of the dynamic programming approach, which make it more practical for many robotic 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 decision making, including several more general and challenging problems than have been considered in previous chapters. In particular, the standard reinforcement learning problem is to determine closed-loop control policies that drive an agent to maximize an accumulated reward[3]. However, in the most general case it is not required that a *system model* be known! This paradigm can be represented by Figure 21.1, where it can be seen that given a control input the environment specifies the state and reward, and the environment can be considered to be a black box (it is not necessarily known how the state is generated).

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

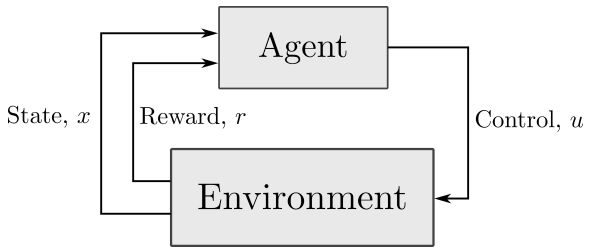

Figure 21.1: In reinforcement learning problems, the robot (agent) learns how to make decisions by interacting with the environment.

Reinforcement learning 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 learn-

ing are introduced, including the formulation of the RL problem, RL algorithms that leverage system models (value iteration/dynamic programming and policy iteration), and a few RL algorithms that do not require system models (Q-learning, policy gradient, actor-critic).

21.1 Problem Formulation

The reinforcement learning problem is effectively the same as the stochastic decision making problem from the previous chapter, but a slightly different notation will be used[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 as:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), \tag{21.1}$$

which is the conditional probability distribution over $\boldsymbol{x}_t$, given the previous state and control.

As was the case for the stochastic decision making problem from previous chapter, RL algorithms seek to compute *policies* $\pi = \{\pi_0, \pi_1, \ldots, \pi_{T-1}\}$ that define the control:

$$\boldsymbol{u}_t = \pi_t(\boldsymbol{x}_t), \tag{21.2}$$

which results in a closed-loop control law. However, instead of a "cost" function (that was used in the previous chapter), reinforcement learning problems are typically formulated with a *reward* function:

$$r_t = R(\boldsymbol{x}_t, \boldsymbol{u}_t), \tag{21.3}$$

where r_t is the reward for being at state $\boldsymbol{x}_t$ and applying a control $\boldsymbol{u}_t$. The goal of the reinforcement learning problem is to maximize the accumulated reward over a number of time steps (in contrast to minimizing cost in the previous chapter). Specifically, the total expected (discounted) reward over T time steps is defined for a particular policy π as:

$$V_T^\pi(\boldsymbol{x}) = E\Big[\sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \pi_t(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\Big], \tag{21.4}$$

where $\gamma \in (0, 1]$ is referred to as a *discount factor*, and the function $V_T^\pi(\boldsymbol{x})$ is referred to as the *value function*[5] for the policy π (starting from $\boldsymbol{x}$ and with T steps to go). The discount factor γ was not used in the decision making problems from the previous chapter, and can be chosen as $\gamma = 1$ for finite horizon problems. However, it is common in RL problems to consider infinite horizons $T = \infty$, and in this case the discount factor must be chosen such that $\gamma < 1$ to ensure the expected reward is finite[6].

The reinforcement learning problem is therefore to find a optimal policy π^* that maximizes the expected reward (i.e. maximizes the value function):

[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 $\boldsymbol{a}$.

[5] Notice the similarity of the value function to the expected cost in the stochastic decision making problem from the previous chapter.

[6] The term "discount factor" is used because it "discounts" future rewards. The intuition behind this can be explained by the phrase "a dollar today is better than a dollar tomorrow".

Definition 21.1.1 (Reinforcement Learning Problem)**.** *The reinforcement learning problem for the system model* (21.1)*, admissible states* $\boldsymbol{x} \in \mathcal{X}$ *and controls* $\boldsymbol{u} \in \mathcal{U}$*, and reward function* (21.3) *is to find an optimal policy* π^* *that satisfies:*

$$\pi^*(\boldsymbol{x}) = \arg\max_{\pi} \; V_T^{\pi}(\boldsymbol{x}), \tag{21.5}$$

for all states $\boldsymbol{x} \in \mathcal{X}$*.*

21.2 Model-based Reinforcement Learning

Model-based reinforcement learning methods rely on the use of an explicit parameterization of the transition model (21.1). While this model could come from physics-based modeling techniques, in many RL settings the model might also be learned from data! For example, for discrete state/action 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.

21.2.1 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 reinforcement learning 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:

$$V_{k+1}^*(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} \; R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^*(\boldsymbol{x}'), \tag{21.6}$$

which is commonly referred to as the *Bellman equation* (note that a shorthand $V_k^*(\boldsymbol{x}) = V_k^{\pi^*}(\boldsymbol{x})$ is used for the optimal value function). In words, this expression says that 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 (21.5) is given in Algorithm 1.

Algorithm 1: Dynamic Programming (Reinforcement Learning)

$V_0^*(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ **to** $T - 1$ **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 $\boldsymbol{x} \in \mathcal{X}$
  $\pi_{T-1-k}^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u} \in \mathcal{U}} \; R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^*(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
return $V_0^*(\cdot), \ldots, V_T^*(\cdot), \pi_0^*(\cdot), \ldots, \pi_{T-1}^*(\cdot)$

In the context of reinforcement learning, this procedure is commonly referred to as *value iteration* and in many cases it is assumed that the horizon T is infinite. For infinite-horizon problems the "value iteration" in Algorithm 1 is performed either over a finite-horizon (which yields an approximate solution), or until convergence to a stationary (i.e. time-invariant) optimal value function/policy[7].

[7] 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 21.2.1 (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 (21.1) 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:

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

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

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

In the next step:

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

with optimal stage policies:

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

Finally, in the last step:

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

with optimal stage policies:

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

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.

21.2.2 *Policy Iteration*

Another common algorithm that can be used to solve the reinforcement learning problem (21.5) is *policy iteration*. The main idea of policy iteration is that if the value function can be computed for any arbitrary policy $\pi = \{\pi_0, \pi_1, \ldots, \pi_{T-1}\}$, then the policy can be incrementally improved to yield a better policy $\pi' = \{\pi_0', \pi_1', \ldots, \pi_{T-1}'\}$.

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:

$$V_{k+1}^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V_k^*(\boldsymbol{x}'), \tag{21.7}$$

which is very similar to the Bellman equation (21.6) 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: π
Result: $V_0^\pi(\cdot), \ldots, V_T^\pi(\cdot)$
$V_0^\pi(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ **to** $T - 1$ **do**
  $V_{k+1}^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi_{T-1-k}(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi_{T-1-k}(\boldsymbol{x})) V_k^\pi(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
return $V_0^\pi(\cdot), \ldots, V_T^\pi(\cdot)$

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 (21.7). 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.

Q-function: The second key component in the policy iteration algorithm is the state-action value function, which is also referred to as the *Q-function*. Unlike the value function, the Q-function is a function of both state and control:

$$Q_{k+1}^\pi(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^\pi(\boldsymbol{x}'), \tag{21.8}$$

and can be interpreted as the "value" of starting in state $\boldsymbol{x}$, applying control $\boldsymbol{u}$, and then following the policy π for the remaining time. In other words, it specifies the value associated with deviating from the given policy π for a single time-step.

Similarly, the *optimal* Q-function is:

$$Q_{k+1}^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^*(\boldsymbol{x}'),$$

where the shorthand notation $Q_k^*(\boldsymbol{x}, \boldsymbol{u}) = Q_k^{\pi^*}(\boldsymbol{x}, \boldsymbol{u})$ is used. Note that from the Bellman equation (21.6) the optimal value function can be written as an optimization over the optimal Q-function:

$$V_{k+1}^*(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} Q_{k+1}^*(\boldsymbol{x}, \boldsymbol{u}).$$

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

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: π
Result: π'
$V_0^\pi(\cdot), \ldots, V_T^\pi(\cdot) \leftarrow$ PolicyEvaluation(π)
for $k = 0$ **to** $T-1$ **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 $\boldsymbol{x} \in \mathcal{X}$
  $\pi'_{T-1-k}(\boldsymbol{x}) = \arg\max_{\boldsymbol{u} \in \mathcal{U}} Q_{k+1}^\pi(\boldsymbol{x}, \boldsymbol{u})$, for all $\boldsymbol{x} \in \mathcal{X}$
return $\pi' = \{\pi'_0, \ldots, \pi'_{T-1}\}$

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.

21.3 Model-free Reinforcement Learning

The value and policy iteration algorithms are applicable only to problems where the probabilistic system model $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ is *known*. In many real-world robotics applications this may not be the case!

In general, many reinforcement learning algorithms address problems where it is assumed that the agent only has access to the rewards r_t and states $\boldsymbol{x}_t$, but does not know how they are generated. Therefore the agent must "learn" policies from experience, or in other words from trajectory data:

$$\{\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T\}, \quad \{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{T-1}\}, \quad \{r_0, r_1, \ldots, r_{T-1}\}.$$

21.3.1 Q-Learning

One algorithm for solving the more challenging model-free problem is *Q-learning*. The core idea behind Q-learning is that it is possible to collect data samples $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$, and over time learn what the optimal Q-function $Q^*(\boldsymbol{x}, \boldsymbol{u})$ is. For simplicity an infinite-horizon ($T = \infty$) problem will be considered, such that the optimal value and Q-functions will be stationary, and in particular the optimal Q-function will satisfy:

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

Importantly, even if the model and reward function are unknown, for a given sample $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ that is collected by interacting with the environment, the *expectation*:

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

will hold for the optimal Q-function. Therefore, 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.

Algorithm 4: Q-learning

Data: Set $\mathcal{S}$ of trajectory samples $\{\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}\}$, learning rate α
Result: $Q(\boldsymbol{x}, \boldsymbol{u})$
Initialize $Q(\boldsymbol{x}, \boldsymbol{u})$ for all $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{u} \in \mathcal{U}$
for $\{\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}\} \in \mathcal{S}$ **do**
  $Q(\boldsymbol{x}_t, \boldsymbol{u}_t) \leftarrow Q(\boldsymbol{x}_t, \boldsymbol{u}_t) + \alpha\Big(r_t + \gamma \max_{\boldsymbol{u} \in \mathcal{U}} Q(\boldsymbol{x}_{t+1}, \boldsymbol{u}) - Q(\boldsymbol{x}_t, \boldsymbol{u}_t)\Big)$
return $Q(\boldsymbol{x}, \boldsymbol{u})$

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 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-value pair. One variation of the Q-learning algorithm to handle large or continuous state and control spaces is to parameterize the Q-function as $Q_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{u})$ and to simply update the parameters θ. While this method often works well in practice, convergence is not guaranteed. Another disadvantage of Q-learning is that sometimes the Q-function can be complex and therefore potentially hard to learn.

21.3.2 *Policy Gradient*

The *policy gradient* method for reinforcement learning problems can also be applied to situations where the model is unknown. 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*[8] 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

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

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.

Considering the original problem (21.5), the objective function can be written as:

$$J(\boldsymbol{\theta}) = E\big[\sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t))\big],$$

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:

$$r(\tau) = \sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t), \tag{21.9}$$

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:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx r(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau). \tag{21.10}$$

In particular the sampled quantities $r(\tau)$ can be directly computed from (21.9), and it turns out that the term $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ can be evaluated quite easily as[9]:

$$\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \sum_{t=0}^{T-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t). \tag{21.11}$$

[9] Using Bayes' rule: $p_{\boldsymbol{\theta}}(\tau) = p(\boldsymbol{x}_0) \Pi_{t=1}^{T-1} p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}) \pi_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{u}_{t-1})$. 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 (21.9).
2. Compute $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ for the sampled trajectory using (21.11), which only requires computing gradients related to the current policy $\pi_{\boldsymbol{\theta}}$.
3. Approximate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ using (21.10).

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[10].

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.

[10] 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.

21.3.3 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[11].

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:

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

[11] 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!

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 $\boldsymbol{\phi}$.

21.4 Deep Reinforcement Learning

Neural networks are a powerful function approximator that can be utilized in reinforcement learning algorithms. For example, in Q-learning the Q-function can be approximated by a neural network to extend the approach to nonlinear, continuous state space domains. In policy gradient methods, the policy $\pi_{\boldsymbol{\theta}}$ can

be parameterized as a neural network, enabling the policy to operate on high-dimensional states including images (i.e. visual feedback)!

21.5 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[12]. This leads to the *exploration vs exploitation* trade-off: a fundamental challenge in reinforcement learning.

[12] 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.