UNIVERSITAT DE VALÈNCIA

NOTES

Reinforcement Learning

Author:
Pablo CARRERA

"People worry that computers will get too smart and take over the world, but the real problem is that they're too stupid and they've already taken over the world."

Pedro Domingos

Contents

Chapter 1

Introduction

1.1 What is Reinforcement Learning?

The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. We explore designs for machines that are effective in solving learning problems of scientific or economic interest, evaluating the designs through mathematical analysis or computational experiments. The approach we explore, usually known as reinforcement learning, is much more focused on goal-directed learning from interaction than are other approaches to machine learning.

Reinforcement learning is learning what to do, how to map situations to actions, so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics, trial-and-error search and delayed reward, are the two most important distinguishing features of reinforcement learning, so we consider reinforcement learning to be a third machine learning paradigm, alongside supervised learning and unsupervised learning:

- Reinforcement learning is different from supervised learning. In supervised learning the system learn from a training set of labeled examples provided by a knowledgeable external supervisor. Each example is a description of a situation together with a specification, the label, of the correct action the system should take to that situation, which is often to identify a category to which the situation belongs. The object of this kind of learning is to make the system able to generalize its responses so that it acts correctly in situations not present in the training set. This is an important kind of learning, but alone it is not adequate for learning from interaction. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. In uncharted territory, where one would expect learning to be most beneficial, an agent must be able to learn from its own experience.

- Reinforcement learning is also different from unsupervised learning. In unsupervised learning the system find the structure hidden in collections of unlabeled data. Although one might be tempted to think of reinforcement learning as a kind of unsupervised learning because it does not rely on examples of correct behavior, reinforcement learning is trying to maximize a reward signal instead of trying to find hidden structure. Uncovering structure in an agent's experience can certainly be useful in reinforcement learning, but by itself does not address the reinforcement learning problem of maximizing a reward signal.

We formalize the problem of reinforcement learning using ideas from dynamical systems theory, specifically, as the optimal control of incompletely-known Markov decision processes. The idea is to capture the most important aspects of the real problem, facing a learning agent interacting over time with its environment to achieve a goal. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. Markov decision processes are intended to include just these three aspects (sensation, action, and goal) in their simplest possible forms without trivializing any of them. Any method that is well suited to solving such problems we consider to be a reinforcement learning method.

One of the key challenges that arise in reinforcement learning is the trade-of between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best.

1.2 Elements of Reinforcement Learning

As we said, there are two elements in every Reinforcement Learning system:

- Agent:
- Environment:

and beyond them, one can identify four main subelements:

- Policy: it defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. In general, policies may be stochastic, specifying probabilities for each action.
- Reward signal: it defines the goal of a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the reward. The agent's sole objective is to maximize the total reward it receives over the long run. The reward signal thus defines what are the good and bad events for the agent. The reward signal is the primary basis for altering the policy, if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. In general, reward signals may be stochastic functions of the state of the environment and the actions taken.
- Value function: the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards

determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states. Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no values, and the only purpose of estimating values is to achieve more reward. Nevertheless, it is values with which we are most concerned when making and evaluating decisions. Action choices are made based on value judgments. We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run. Unfortunately, it is much harder to determine values than it is to determine rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime.

- Model of the environment: it is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. Models are used for planning, that is, deciding on a course of action by considering possible future situations before they are actually experienced.

1.3 Structure of these notes

In the first part of the notes we describe almost all the core ideas of reinforcement learning algorithms in their simplest forms: that in which the state and action spaces are small enough for the approximate value functions to be represented as arrays, or tables. In this case, the methods can often find exact solutions, that is, they can often find exactly the optimal value function and the optimal policy. This part comprises chapters 2,3,4,5.

In the second part of the notes we extend the tabular methods presented in the first part to apply to problems with arbitrarily large state spaces. In many of the tasks to which we would like to apply reinforcement learning the state space is combinatorial and enormous. In such cases we cannot expect to find an optimal policy or the optimal value function even in the limit of infinite time and data, our goal instead is to find a good approximate solution using limited computational resources.

Chapter 2

Finite Markov Decision Processes

Finite Markov Decision Processes, or MDPs, are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward.

2.1 The agent-environment interface

MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions.

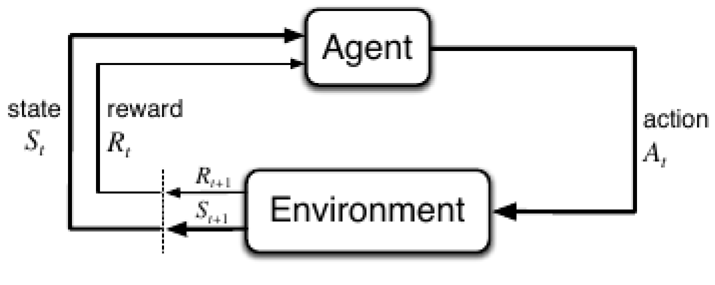

FIGURE 2.1: Schema of the agent–environment interaction in a Markov decision process.

More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives some representation of the environment's state, $S_t \in \mathcal{S}$, and on that basis selects an action, $A_t \in \mathcal{A}(S_t)$. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in \mathcal{R}$, and finds itself in a new state S_{t+1}. Then, the MDP and agent give rise to a sequence or trajectory that begins like this $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$

In a finite MDP, the sets of states $\mathcal{S}$, actions $\mathcal{A}$, and rewards $\mathcal{R}$ all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action:

$$p(s', r|s, a) \doteq \mathrm{P}(S_t = s', R_t = t|S_{t-1} = s, A_{t-1} = a)$$

so the function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \longrightarrow [0, 1]$, that defines the dynamics of the MDP, is an ordinary deterministic function of four arguments. And since it involves a conditional probability distribution, it satisfies that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1$$

which implies that the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, S_{t-1} and A_{t-1}, and not at all on earlier states and actions. This is best viewed a restriction not on the decision process, but on the state, which must include information about all aspects of the past agent-environment interaction that make a difference for the future. If it does, then the state is said to have the Markov property.

2.2 Rewards and returns

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent. At each time step, the reward is a simple number $R_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of reward it receives, that is, maximizing not immediate reward, but cumulative reward in the long run. If we want it to do something for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It is thus critical that the rewards we set up truly indicate what we want to accomplish. In particular, the reward signal is not the place to impart to the agent prior knowledge about how to achieve what we want it to do.

In general, we seek to maximize the expected return, where the return is defined as some specific function of the reward sequence $G_t \doteq f(R_{t+1}, R_{t+2}, ..., R_T)$, where T is a final time step. In the simplest case the return is the sum of the rewards:

$$G_t \doteq R_{t+1} + ... + R_T = \sum_{k=t+1}^{T} R_k$$

This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent-environment interaction breaks naturally into subsequences, which we call episodes. Each episode ends in a special state called the terminal state, followed by a reset to a standard starting state. Then the next episode begin independently of how the previous one ended. Thus the episodes can all be considered to end in the same terminal state, with different rewards for the different outcomes. Tasks with episodes of this kind are called episodic tasks. If we consider episodic tasks as a series of episodes, each of which consists of a finite sequence of time steps, we number the time steps of each episode starting anew from zero, so then, we have to refer not just to S_t but to $S_{t,i}$, the state representation at time t of episode i. However, it turns out that when we discuss episodic tasks we almost never have to distinguish between different episodes, so usually the last index will be dropped.

However, in many cases the agent-environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. Then the above return formulation is problematic for continuing tasks because the final time step would be $T = \infty$ and then also the return. conceptually but much simpler mathematically. The additional concept that we need is that of discounting, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses A_t to maximize the expected discounted return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=t+1}^{\infty} \gamma^{k-(t+1)} R_k$$

where $0 \leq \gamma \leq 1$ is called the discount rate. If $\gamma < 1$, the infinite sum has a finite value as long as the reward sequence is bounded. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards, but in general acting to maximize immediate reward can reduce access to future rewards so that the return is reduced. As γ approaches 1, the return objective takes future rewards into account more strongly, that is, the agent becomes more farsighted. It is important to note the following recursive rule

$$G_t = R_{t+1} + \gamma G_{t+1}$$

It is useful to establish one notation that enables us to talk precisely about both cases simultaneously. This can be achieved by considering episode termination to be the entering of a special absorbing state that transitions only to itself and that generates only rewards of zero. Then we get the same return whether we sum over the first T rewards or over the full infinite sequence.Thus, we can define the return, in general, using the convention of omitting episode numbers when they are not needed, as

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-(t+1)} R_k$$

where we include the possibility of $T = \infty$ and $\gamma = 1$, but not both at the same time.

2.3 Policies and value functions

Reinforcement learning algorithms usually involve estimating value functions that estimate how good it is for the agent to be in a given state. This notion is defined in terms of expected return. The rewards the agent can expect to receive in the future depend on what actions it will take, so value functions are defined with respect to particular ways of acting, called policies.

Formally, a policy is a mapping from states to probabilities of selecting each possible action, if the agent is following policy π at time t, then $\pi_t(a|s)$ is the probability that $A_t = a$ if we are in $S_t = s$. Like p, π is an ordinary function which defines a probability distribution over a $A(s)$ for $s \in S$. Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter

$$v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t | S_t = s\right]$$

for $s \in \mathcal{S}$. Note that the value of the terminal state, if any, is always zero. We call the function $v_\pi(s)$ the state-value function for policy π.

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s,a)$, as the expected return of taking the action a in the state s following policy π

$$q_\pi(s,a) \doteq \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right]$$

and we call q_π the action-value function for policy π. By definition the following relation is satisfied

$$\begin{aligned}
&v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t | S_t = s\right] \\
&\implies v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right] \\
&\implies v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a)
\end{aligned}$$

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return

$$\begin{aligned}
&v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t | S_t = s\right] \\
&\implies v_\pi(s) = \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\
&\implies v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} | S_t = s'\right]\right] \\
&\implies v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \left[r + \gamma v_\pi(s')\right]
\end{aligned}$$

which is known as the Bellman equation and represent the relation between the value of an state and the subsequent states. Basically the value function $v_\pi(s)$ is the unique solution to the Bellman equation. For the action-value function

$$\begin{aligned}
&q_\pi(s,a) \doteq \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right] \\
&\implies q_\pi(s,a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\
&\implies q_\pi(s,a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \left[r + \gamma v_\pi(s')\right] \\
&\implies q_\pi(s,a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \left[r + \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')\right] \\
&\implies q_\pi(s,a) = \sum_{a' \in \mathcal{A}} \pi(a'|s') \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \left[r + q_\pi(s', a')\right]
\end{aligned}$$

we obtain a similar result, which is denoted as the Bellman equation for the action-value function. Again, basically the action-value function $q_\pi(s,a)$ is the unique solution to the Bellman equation.

2.4 Optimality

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs value functions define a partial ordering over policies, a policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π for all states. In other words, $\pi > \pi'$ if and only if $v_\pi(s) > v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies, the optimal policy. Although there may be more than one, we denote all the optimal policies by π_*. They share the same state-value function, called the optimal state-value function,

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

for all $s \in \mathcal{S}$. Optimal policies also share the same optimal action-value function

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

Because $v_*(s)$ is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values, but because it is the optimal one its consistency condition can be written in a special form without reference to any specific policy. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state

$$\begin{aligned}
&v_*(s) = \max_a q_*(s,a) \\
&\implies v_*(s) = \max_a \mathbf{E}\left[G_t | S_t = s, A_t = a\right] \\
&\implies v_*(s) = \max_a \mathbf{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\
&\implies v_*(s) = \max_a \mathbf{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \\
&\implies v_*(s) = \max_a \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a)\left[r + \gamma v_*(s')\right]
\end{aligned}$$

For finite MDPs, the Bellman optimality equation for v_* has a unique solution. If the dynamics p of the environment are known, then in principle one can solve this system of equations using any one of a variety of methods for solving systems of nonlinear equations. One can also solve the related set of equations for $q_*(s,a)$.

Once one has v_*, it is relatively easy to determine an optimal policy. For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy. We can also think of this as a one-step search, if we have the optimal value function, v_* , then the actions that appear best after a one-step search will be optimal actions. So, by means of v_* , the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state. Hence, a one-step-ahead search yields the long-term optimal actions.

Having q_* makes choosing optimal actions even easier, the agent does not even have to do a one-step-ahead search, for any state s, it can simply find any action that maximizes $q_*(s,a)$. The action-value function effectively caches the results of all one-step-ahead searches. It provides the optimal expected long-term return as a value that is locally and immediately available for each state–action pair.

Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the reinforcement learning problem. However, this solution is rarely directly useful, it leads to an exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. This solution relies on at least three assumptions that are rarely true in practice: we accurately know the dynamics of the environment, we have enough computational resources to complete the computation of the solution and the system satisfies the Markov property. But for the kinds of tasks in which we are interested, one is generally not able to implement this solution exactly because various combinations of these assumptions are violated. Then many reinforcement learning methods can be understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions.

We use the term generalized policy iteration (GPI) to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes.

Chapter 3

Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can bused to compute optimal policies given a perfect model of the environment as a MDP. Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically.

We usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets, $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$, are finite, and that its dynamics are given by a set of probabilities $p(s', r|s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, $r \in \mathcal{R}$, and $s' \in \mathcal{S}^+$. DP algorithms are then obtained by turning Bellman equations into assignments, that is, into update rules for improving approximations of the desired value functions.

3.1 DP prediction

3.1.1 Policy evaluation

First we consider how to compute the state-value function v_π for an arbitrary policy π, this is called policy evaluation. If the environment's dynamics are completely known, then the Bellman equation is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns (the $v_\pi(s)$, $s \in \mathcal{S}$). So in principle, its solution is a straightforward computation. For our purposes, iterative solution methods are most suitable.

Consider a sequence of approximate value functions $v_0, v_1, v_2, ...$, each mapping $\mathcal{S}^+$ to $\mathbb{R}$. The initial approximation v_0 is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation as an update rule:

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \left[r + \gamma v_k(s') \right]$$

for all $s \in \mathcal{S}$. Each iteration updates the value of every state once to produce the new approximate value function. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called iterative policy evaluation and is usually applied until the difference between two consecutive iterations of the value function estimate go beyond a given threshold θ.

Algorithm 1 Iterative policy evaluation, for estimating $V \approx v_\pi$

```
1: $V(s) \longleftarrow 0$
2: **loop** until $\Delta < \theta$
3:     $\Delta \longleftarrow 0$
4:     **loop** for $s \in \mathcal{S}$
5:         $v \longleftarrow V(s)$
6:         $V(s) \longleftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} p(s', r|s, a) \left[r + \gamma V(s')\right]$
7:         $\Delta \longleftarrow \max(\Delta, |v - V(s)|)$
8:     **end loop**
9: **end loop**
```

3.1.2 Policy improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function v_π for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$.

Consider selecting a in s and thereafter following the existing policy, π. The value of this way of behaving is, from chapter 2,

$$q_\pi(s, a) = \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$$

and then, the key criterion is whether this is greater than or less than $v_\pi(s)$. If it is greater, that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time, then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall. That this is true is a special case of a general result called the policy improvement theorem. Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$, $q_\pi(s, \pi'(s)) \geq v_\pi(s)$. Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states, $v_{\pi'}(s) \geq v_\pi(s)$.

A natural extension is to consider changes at all states and to all possible actions, selecting at each state the action that appears best according to $q_\pi(s, a)$. In other words, to consider the new greedy policy π' given by

$$\begin{aligned} \pi'(s) &= \operatorname{argmax}_a \left[q_\pi(s, a)\right] \\ \implies \pi'(s) &= \operatorname{argmax}_a \left[\sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} p(s', r|s, a) \left[r + q_\pi(s', a')\right]\right] \end{aligned}$$

where argmax_a denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). The greedy policy takes the action that looks best in the short term, after one step of lookahead, according to v_π. By construction, the greedy policy meets the conditions of the policy improvement theorem, so we know that it is as good as, or better than, the original policy. The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called policy improvement.

Suppose the new greedy policy π' is as good as, but not better than, the old policy π, then $v_\pi = v_{\pi'}$, and then for all $s \in \mathcal{S}$:

$$\begin{aligned} v_{\pi'}(s) &= \max_a \left[\mathbb{E}\left[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) | S_t = s, A_t = a\right]\right] \\ \implies v_{\pi'}(s)' &= \max_a \left[\sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s', r|s, a) \left[r + q_\pi(s', a')\right]\right] \end{aligned}$$

but this is the same as the Bellman optimality equation, and therefore, $v_{\pi'}$ must be v_* , and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

3.2 DP control

3.2.1 Policy iteration

Once a policy π has been improved using v_π to yield a better policy π' we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} ... \xrightarrow{I} \pi_* \xrightarrow{E} v_{\pi_*}$$

where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy improvement. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations. This way of finding an optimal policy is called policy iteration.

Algorithm 2 Policy iteration, for estimating $\pi \approx \pi_*$

```
Initialization
 1: $V(s) \longleftarrow 0$
 2: $\pi(s) \longleftarrow$ random
Policy evaluation
 1: loop until $\Delta < \theta$
 2:     $\Delta \longleftarrow 0$
 3:     loop for $s \in \mathcal{S}$
 4:         $v \longleftarrow V(s)$
 5:         $V(s) \longleftarrow \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s', r|s, \pi(s)) \, [r + \gamma V(s')]$
 6:         $\Delta \longleftarrow \max(\Delta, |v - V(s)|)$
 7:     end loop
 8: end loop
Policy improvement
 1: $\textit{policy stable} \longleftarrow \textit{True}$
 2: loop for $s \in \mathcal{S}$
 3:     $\textit{old action} \longleftarrow \pi(s)$
 4:     $\pi(s) \longleftarrow \text{argmax}_a \left[\sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s', r|s, a) \, [r + \gamma V(s')]\right]$
 5:     if $\textit{old action} \neq \pi(s)$ then
 6:         $\textit{policy stable} \longleftarrow \textit{False}$
 7:     end if
 8: end loop
 9: if $\textit{policy stable} = \textit{True}$ then
10:     return $V$ and $\pi$
11: else
12:     go back to policy evaluation
13: end if
```

3.2.2 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation, this results in a high computational cost. Policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state), this algorithm is called value iteration and it can be written as a particularly simple update operation that combines policy improvement and truncated policy evaluation steps, for all $sin\mathcal{S}$:

$$v_{k+1}(s) = \max_a \left[\mathbb{E}\left[R_{t+1} + \gamma v_{\pi'}(S_{t+1})|S_t = s, A_t = a\right]\right]$$
$$\Longrightarrow v_{k+1}(s)' = \max_a \left[\sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s', r|s, a) \left[r + v_k(s')\right]\right]$$

where note that the value iteration is obtained simply by turning the Bellman optimality equation into an update rule.

Algorithm 3 Value iteration, for estimating $\pi \approx \pi_*$

```
1: $V(s) \longleftarrow 0$
2: loop until $\Delta < \theta$
3:     $\Delta \longleftarrow 0$
4:     loop for $s \in \mathcal{S}$
5:         $v \longleftarrow V(s)$
6:         $V(s) \longleftarrow \max_a \left(\sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s', r|s, a) \left[r + \gamma V(s')\right]\right)$
7:         $\Delta \longleftarrow \max(\Delta, |v - V(s)|)$
8:     end loop
9: end loop
10: $\pi(s) = \operatorname{argmax}_a \left(\sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s', r|s, a) \left[r + \gamma V(s')\right]\right)$
```

3.3 Asynchronous Dynamic Programming

A major drawback to the DP methods is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set, so if this set is very large, then even a single sweep can be prohibitively expensive. Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms update the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be updated several times before the values of others are updated once. To converge correctly, however, an asynchronous algorithm must continue to update the values of all states, it cannot ignores any state after some point of computation.

Asynchronous algorithms also make it easier to intermix computation with real-time interaction. To solve a given MDP, we can run an iterative DP algorithm at the same time that an agent is actually experiencing the MDP. The agent's experience can be used to determine the states to which the DP algorithm applies its updates. At the same time, the latest value and policy information from the DP algorithm can guide the agent's decision making. For example, we can apply updates to states as the agent visits them. This makes it possible to focus the DP algorithm's updates onto parts of the state set that are most relevant to the agent.

3.4 Generalized Policy Iteration

Policy iteration consists of two simultaneous, interaction processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes alternate, each completing before the other begins, but this is not really necessary. In value iteration, for example, only a single iteration of policy evaluation is performed between each policy improvement. In asynchronous DP methods, the evaluation and improvement processes are interleaved at an even finer grain. In some cases a single state is updated in one process before returning to the other. As long as both processes continue to update all states, the ultimate result is typically the same, the convergence to the optimal value function and an optimal policy.

Chapter 4

Monte Carlo Methods

Although we have complete knowledge of the environment, it would not be easy to apply DP methods to compute the value function. DP methods require the distribution of next events, all of the probabilities must be computed before DP can be applied, and such computations are often complex and error-prone. In contrast, generating the sample episodes required by Monte Carlo methods is easy. This is the case surprisingly often, the ability of MC methods to work with sample episodes alone can be a significant advantage even when one has complete knowledge of the environment dynamics.

4.1 MC Prediction

Formally, suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in a episode is called a visit to s. An obvious way to learning the state-value function is from experience, then we can simply compute it averaging the returns observed after visits to that state:

- First-visit MC method estimates $v_\pi(s)$ as the average of the returns following first visit to s in each episode.
- Every-visit MC method estimates $v_\pi(s)$ as the average of the returns following all visit to s in each episode.

Algorithm 4 First-visit MC prediction, for estimating $V \approx v_\pi$

```
1: $V(s) \longleftarrow 0$
2: $returns(s) \longleftarrow$ empty list
3: loop forever
4:     Generate a episode following $\pi$: $S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T$
5:     $G \longleftarrow 0$
6:     loop for each step in reverse ordering: $t = T-1, T-2, ..., 0$
7:         $G \longleftarrow \gamma G + R_{t+1}$
8:         if $S_t$ not in $S_0, S_1, ..., S_{t-1}$ then
9:             Append $G$ to $returns(S_t)$
10:            $V(S_t) \longleftarrow$ mean($returns(S_t)$)
11:        end if
12:    end loop
13: end loop
```

Algorithm 5 Every-visit MC prediction, for estimating $V \approx v_\pi$

```
1: $V(s) \longleftarrow 0$
2: $returns(s) \longleftarrow$ empty list
3: **loop** forever
4:     Generate a episode following $\pi$: $S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T$
5:     $G \longleftarrow 0$
6:     **loop** for each step in reverse ordering: $t = T-1, T-2, ..., 0$
7:         $G \longleftarrow \gamma G + R_{t+1}$
8:         Append $G$ to $returns(S_t)$
9:         $V(S_t) \longleftarrow \text{mean}(returns(S_t))$
10:    **end loop**
11: **end loop**
```

An important fact about the MC methods is that the estimates for each state are independent. The estimate for one state does not build upon the estimate of any other state, as is the case in DP. In particular, note that the computational expense of estimating the value of a single state is independent of the number of states.

4.1.1 Monte Carlo Estimation of Action Values

The policy evaluation problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π. The MC methods for this are essentially the same as just presented for state values, except now we talk about visits to a state-action pair rather tan to a state. A state action pair (s, a) is said to be visited in an episode if ever the state s is visited and the action a is taken in it.

The only complication is that many state-action pairs may never be visited. For policy evaluation to work for action-value pairs, we must assure continual exploration. One way to do this is by specifying that episodes start in a state-action pair, and that every pair has a nonzero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. We call this the assumption of exploring starts.

The assumption of exploring starts is sometimes useful, but it cannot relied in general, particularly when learning directly from actual interacting with an environment. The common alternative approach to assuring that all state-action pairs are encountered is to consider only policies that are stochastic with a nonzero probability of selecting all action in each state.

4.2 MC Control

Now we are going to use Monte Carlo estimation to approximate optimal policy. Te overall idea is to proceed according to the same pattern as in DP, according to the idea of GPI. To begin, let us consider a MC version of classical policy iteration. In this method we perform alternating complete steps of policy evaluation and policy improvement

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} ... \xrightarrow{I} \pi_* \xrightarrow{E} q_{\pi_*}$$

where $\xrightarrow{E}$ denotes a complete policy evaluation and $\xrightarrow{I}$ denotes a complete policy improvement. Policy evaluation is done as described in the preceding section, and policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an action-value function, so no model is needed to construct the greedy policy, just

$$\pi(s) \doteq \text{argmax}_a q(s,a)$$

We made two unlikely assumptions above in order to easily obtain this guarantee of convergence for MC method:

- The episodes have exploring starts.
- The policy evaluation could be done with an infinite number of episodes.

so, in order to obtain a practical algorithm we will have to remove both assumptions.

The second assumption is easy to remove, in fact, is the same issue arises in classical DP methods such as iterative policy evaluation. In both DP and MC cases, there are two ways to solve the problem. One is to hold firm the idea of approximating q_{π_k} in each policy evaluation, measurements and assumptions are made to obtain bounds on the magnitude and probability of error in the estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficiently small. A second approach is to give up trying to complete policy evaluation before returning to policy improvement. On each evaluation step we move the value function toward q_{π_k}, but we do not expect actually to get close except over many steps. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. This is known as Monte Carlo ES (with Exploring Starts), and stability is achieved only when both policy and the value function are optimal.

Algorithm 6 Monte Carlo ES, for estimating $\pi \approx \pi_*$

```
1: $\pi(s) \longleftarrow$ arbitrary
2: $Q(s,a) \longleftarrow$ arbitrary
3: $returns(s,a) \longleftarrow$ empty list
4: loop forever
5:     Chose $S_0$, $A_0$ randomly such that all pairs have nonzero probability
6:     Generate a episode following $\pi$: $S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T$
7:     $G \longleftarrow 0$
8:     loop for each step in reverse ordering: $t = T-1, T-2, ..., 0$
9:         $G \longleftarrow \gamma G + R_{t+1}$
10:        if $(S_t, A_t)$ not in $(S_0, A_0), (S_1, A_1), ..., (S_{t-1}, A_{t-1})$ then
11:            Append $G$ to $returns(S_t, A_t)$
12:            $Q(S_t, A_t) \longleftarrow \text{mean}(returns(S_t, A_t))$
13:            $\pi(S_t) \longleftarrow \text{argmax}_a Q(S_t, a)$
14:        end if
15:    end loop
16: end loop
```

4.3 MC Control without Exploring Starts

The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches to ensuring this, resulting in what we call on-policy methods, which attempt to evaluate or improve the policy what is used to make decisions, and off-policy methods, which evaluate or improve a policy different from that used to generate the data.

In on-policy control methods the policy is generally soft, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$, but gradually shifted closer and closer to a deterministic optimal policy. A example of these policies are ε-greedy policies, meaning that most of the time they choose an action that has maximal estimated action values, but with probability ε they instead select an action at random. That is, all nongreedy actions are given the minimal probability of selection $\varepsilon/|\mathcal{A}|$, and the remaining bulk of probability $1-\varepsilon+\varepsilon/|\mathcal{A}|$ is given to the greedy action.

Algorithm 7 On-policy first-visit MC control (ε-soft policies), for estimating $\pi \approx \pi_*$

```
1: $\pi(s) \longleftarrow$ arbitrary $\varepsilon$-soft policy
2: $Q(s,a) \longleftarrow$ arbitrary
3: $returns(s,a) \longleftarrow$ empty list
4: **loop** forever
5:     Chose $S_0$, $A_0$ randomly such that all pairs have nonzero probability
6:     Generate a episode following $\pi$: $S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T$
7:     $G \longleftarrow 0$
8:     **loop** for each step in reverse ordering: $t = T-1, T-2, ..., 0$
9:         $G \longleftarrow \gamma G + R_{t+1}$
10:        **if** $(S_t, A_t)$ not in $(S_0, A_0), (S_1, A_1), ..., (S_{t-1}, A_{t-1})$ **then**
11:            Append $G$ to $returns(S_t, A_t)$
12:            $Q(S_t, A_t) \longleftarrow \text{mean}(returns(S_t, A_t))$
13:            $A^* \longleftarrow \text{argmax}_a Q(S_t, a)$
14:            **loop** for each $a \in \mathcal{A}$
15:                $\pi(a|S_t) \longleftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}| & \text{if } a \neq A^* \end{cases}$
16:            **end loop**
17:        **end if**
18:    **end loop**
19: **end loop**
```

Using the natural notion of greedy policy for ε-soft policies, one is assured of improvement in every step, except when the best policy has been found among the ε-soft policies. This brings us to roughly the same point as in the previous section. Now we only achieve the best policy among the ε-soft policies, but on the other hand, we have eliminated the assumption of exploring starts.

4.4 Off-policy Prediction via Importance Sampling

The on-policy approach is actually a compromise, it learns action values not for the optimal policy, but for a near-optimal policy that still explores. a more straightforward approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is used to generate behavior. The policy being learned about is called the target policy, and the policy used to generate behavior is called the behavior policy. In this case we say that learning is from data "off" the target policy, and the overall process is termed off-policy learning.

We begin by considering the prediction problem, in which both target and behavior policies are fixed. That is, suppose we wish ti estimate v_π or q_π, but all we have are episodes following another policy b, where $b \neq \pi$. in this case π is the target policy and b is the behavior policy, and both policies are considered fixed and given. In order to use episodes from b to estimate π, we require that every action taken under π is also taken, at least occasionally, under b. That is, we require that $\pi(a|s) > 0$ implies $b(a|s) > 0$, this is called the assumption of coverage. It follows from coverage that b must be stochastic in states where it is not identical yo π. The target policy π, on the other hand, may be deterministic, and, in fact, this is a case of particular interest in control applications. In control, the target policy is typically the deterministic greedy policy with respect to the current estimate of the action-value function.

Almost all off-policy methods utilize importance sampling, a general technique for estimating expected values under one distribution given samples from another by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the importance-sampling ratio. Given a starting state S_t, the probability of the subsequent state-action trajectory $A_t, S_{t+1}, A_{t+1}, ..., S_T$ occurring under any policy π

$$P_\pi(A_t, S_{t+1}, ..., S_T|S_t) \doteq \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)...\pi(A_{T-1}|S_{T-1})p(S_T|S_{T+1}, A_{T+1})$$
$$\implies P_\pi(A_t, S_{t+1}, ..., S_T|S_t) = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)$$

thus, the relative probability of the trajectory under the target and behavior policies, the importance-sampling ratio, is

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)}$$
$$\implies \rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

so it only depends on the two policies and the sequence, not on the MDP dynamics.

Recall that we wish to estimate the expected returns (values) under the target policy, but all we have are the returns G_t due to the behavior policy. These returns have the wrong expectation $v_b(s) = \mathbb{E}[G_t|S_t = s]$, but the ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value $v_\pi(s) = \mathbb{E}[\rho_{t:T-1}G_t|S_t = s]$. Then to estimate $v_\pi(s)$ we simply scale the returns by the ratios and average the returns over many episodes, in the so called ordinary importance sampling

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}$$

or in the weighted importance sampling

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}}$$

where we use a notation in which all episodes are concatenated, in such a way that time steps increase continually across episode boundaries, so that $\mathcal{T}(s)$ denote the time steps in which the state s is visited, and $T(t)$ denote the termination following t and G_t denote the return after t up through $T(t)$. Then $\{G_t\}_{t\in\mathcal{T}(s)}$ are the returns that pertains to state s and $\{\rho_{t:T(t)-1}\}_{t\in\mathcal{T}(s)}$ are the importance-sampling ratios.

Formally, the difference between first-visit methods of the two kinds of importance sampling is expressed in their biases and variances. Ordinary importance sampling is unbiased whereas weighted importance sampling is biased (though the bias converges asymptotically to zero). On the other hand, the variance of ordinary importance sampling is in general unbounded because of the variance of the ratios can be unbounded, whereas in the weighted estimator the largest weight on any single return is one. In fact, assuming bounded returns, the variance of the weighted importance sampling estimators converges to zero even is the variance of the ratios themselves is infinite. in practice, the weighted estimator has dramatically lower variance and is strongly preferred.

The every-visit methods for ordinary and weighted importance sampling are both biased, though,again, he bias falls asymptotically to zero as the number of samples increases. In practice, every-visit methods are often preferred because they remove the need to keep track of which states have been visited and because the are much easier to extend to approximations.

4.5 Incremental Implementation

Monte Carlo prediction methods can be implemented incrementally, on a episode-by-episode basis:

- On-policy methods: we average the returns

$$V_{n+1} \doteq V_n + \frac{1}{n}[G_n - V_n]$$

- Off-policy methods with ordinary importance sampling: we have to form a weighted average of returns. Suppose we have a sequence $G_1, .., G_{n-1}$, all starting at the same state and each with a corresponding random weight W_k, we form the estimate

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{n}$$

and update it as we obtain a single additional G_n as

$$V_n \doteq V_n + \frac{W_n}{n}[G_n - V_n]$$

- Off-policy methods with weighted importance sampling: we have to form a weighted average of returns,but with a slightly different increment. We form the estimate

  $$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}$$

  and update it as we obtain a single additional G_n as

  $$V_n \doteq V_n + \frac{W_n}{C_n}[G_n - V_n]$$

  where C_n is the cumulative sum of the weights given to the firsts n returns, thus $C_{n+1} = C_n + W_{n+1}$.

Algorithm 8 Off-policy every-visit MC prediction, for estimating $Q \approx q_\pi$

```
1: $Q(s,a) \longleftarrow 0$
2: $C(s,a) \longleftarrow 0$
3: **loop** forever
4:     $b \longleftarrow$ any policy with coverage of $\pi$
5:     Generate a episode following $b$: $S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T$
6:     $G \longleftarrow 0$
7:     $W \longleftarrow 0$
8:     **loop** for each step in reverse ordering: $t = T-1, T-2, ..., 0$
9:         $G \longleftarrow \gamma G + R_{t+1}$
10:        $C(S_t, A_t) \longleftarrow C(S_t, A_t) + W$
11:        $Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
12:        $W \longleftarrow W\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
13:        **if** W = 0 **then**
14:            exit loop
15:        **end if**
16:    **end loop**
17: **end loop**
```

4.6 Off-policy MC Control

Off-policy MC control methods follow the behavior policy while learning about and improving the target policy, this require that the behavior policy has a nonzero probability of selecting all actions that might be selected by the target policy (coverage), that is, to explore all possibilities, we require that the behavior policy be soft.

Algorithm 9 Off-policy every-visit MC control, for estimating $\pi \approx \pi_*$

```
1: $Q(s, a) \longleftarrow 0$
2: $C(s, a) \longleftarrow 0$
3: $\pi(s) \longleftarrow \text{argmax}_a Q(s, a)$
4: **loop** forever
5:     $b \longleftarrow$ any soft policy
6:     Generate a episode following $b$: $S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T$
7:     $G \longleftarrow 0$
8:     $W \longleftarrow 0$
9:     **loop** for each step in reverse ordering: $t = T-1, T-2, ..., 0$
10:        $G \longleftarrow \gamma G + R_{t+1}$
11:        $C(S_t, A_t) \longleftarrow C(S_t, A_t) + W$
12:        $Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
13:        $\pi(s) \longleftarrow \text{argmax}_a Q(s, a)$
14:        **if** $A_t! = \pi(S_t)$ **then**
15:            exit loop
16:        **end if**
17:        $W \longleftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
18:    **end loop**
19: **end loop**
```

Chapter 5

Temporal-Difference Learning

5.1 TD Prediction

Both, TD and MC methods use experience to solve the prediction problem, given some experience following a policy π, both type of methods update their estimate V of v_π for nonterminal states S_t occurring in that experience. Roughly speaking MC methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$ as

$$V(S_t) \longleftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

where G_t is the actual return following time t and α is a constant step-size parameter. So, they must wait until the end of the episode to determine the increment to $V(S_t)$. However, TD method need to wait only until next time step, at time $t+1$ they immediately form a target and make a useful update. The simplest TD method makes

$$V(S_t) \longleftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

immediately on transition to S_{t+1} and receiving R_{t+1}. This TD method is called TD(0), or one-step TD, because it is a special case of the TD(λ) method, and it uses as a target just $R_{t+1} + \gamma v_\pi(S_{t+1})$.

Algorithm 10 Tabular TD(0), for estimating $V \approx v_\pi$

```
1: $V(s) \longleftarrow 0$
2: **loop** for each episode
3:     $S \longleftarrow$ initial state
4:     **loop** for each step
5:         $A \longleftarrow \pi(S)$
6:         Take action $A$, observe $R, S'$
7:         $V(S) \longleftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
8:         $S' \longleftarrow S$
9:     **end loop**
10: **end loop**
```

The Monte Carlo target is an estimate, because the expected value $\mathbb{E}[G_t|S_t = s]$ is not known, a sample return is used in place of the expected return. The DP target is also an estimate, not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate $V(S_{t+1})$ is used instead. Then, the TD target is an estimate for both reasons, it samples the expected values in $\mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$ and it uses the current estimate V instead of the true v_π, that is, TD methods combine the sampling of MC methods with the bootstrapping of DP methods.

Note that the quantity in brackets in the $TD(0)$ update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1}$. This quantity, called the TD error, arises in various forms throughout reinforcement learning

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

and note that this is the error in the estimate made at that time. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. Also note that if the array V does not change during the episode, then the MC error can be written as a sum of TD errors

$$\begin{aligned}
&G_t - V(S_t) = R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \\
&\implies G_t - V(S_t) = \delta_t + \gamma[G_{t+1} - \gamma V(S_{t+1})] \\
&\implies G_t - V(S_t) = \delta_t + \gamma\delta_{t+1} + \gamma^2[G_{t+2} - \gamma V(S_{t+2})] \\
&\implies G_t - V(S_t) = \delta_t + \gamma\delta_{t+1} + ... + \gamma^{T-1}[G_T - \gamma V(S_T)] \\
&\implies G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k
\end{aligned}$$

5.2 Sarsa: On-policy TD Control

The first step is to learn an action-value function rater than a state-value function. in the previous section we considered transitions from state to state and learned the values of these states, now we consider transitions from a state-action pair to another state-action pair, so the update rule is

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

and it is done after every transition from a nonterminal state S_t, if S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ that make up a transition from one state-action pair to the next. This quintuple five rise to the name Sarsa for the algorithm.

It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π.

Algorithm 11 Sarsa (on-policy TD control), for estimating $Q \approx q_*$

```
 1: $Q(s, a) \longleftarrow 0$
 2: loop for each episode
 3:     $S \longleftarrow$ initial state
 4:     Choose $A$ from $S$ using policy derived from $Q$ ($\varepsilon$-greedy for example)
 5:     loop for each step
 6:         Take action $A$, observe $R, S'$
 7:         Choose $A'$ from $S'$ using policy derived from $Q$ ($\varepsilon$-greedy for example)
 8:         $Q(S, A) \longleftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$
 9:         $S' \longleftarrow S$
10:         $A' \longleftarrow A$
11:     end loop
12: end loop
```

5.3 Q-learning: Off-policy TD Control

We can develop an off-policy TD control algorithm, known as Q-learning, defined by the update rule

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

in this case the learned action-value function Q directly approximates q_* the optimal action-values function, independent of the policy being followed. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated.

Algorithm 12 Q-learning (off-policy TD control), for estimating $\pi \approx \pi_*$

```
 1: $Q(s, a) \longleftarrow 0$
 2: loop for each episode
 3:     $S \longleftarrow$ initial state
 4:     loop for each step
 5:         Choose $A$ from $S$ using policy derived from $Q$ ($\varepsilon$-greedy for example)
 6:         Take action $A$, observe $R, S'$
 7:         $Q(S, A) \longleftarrow Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$
 8:         $S' \longleftarrow S$
 9:     end loop
10: end loop
```

5.4 Expected Sarsa

Consider now the learning algorithm that is just like Q-learning except hat instead of the maximum over next state-action pairs it uses the expected values, taking into account how likely each action is under the current policy. That is, consider the algorithm with the update rule

$$\begin{aligned} &Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbb{E}_\pi \left[Q(S_{t+1}, A_{t+1})|S_{t+1}\right] - Q(S_t, A_t)] \\ &\implies Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned}$$

so given the next state S_{t+1}, this algorithm moves deterministically in the same direction as Sarsa mover in expectation, and accordingly it is called Expected Sarsa.

Expected Sarsa is more complex computationally than Sarsa, but it eliminates the variance due to the random selection of A_{t+1}. Given the same amount of experience we might expect it to perform slightly better than Sarsa, and indeed it generally does. Note that in the case that π is a greedy policy then Expected Sarsa is exactly Q-learning, so in this sense it subsumes and generalizes Q-learning while reliably improving over Sarsa with a small additional computational cost.

5.5 Maximization Bias and Double Learning

All the control algorithms that we have discussed so far involve a maximization in the construction of their target policies. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias usually called maximization bias. One way to view the problem is that it is due to using the same samples both to determine the maximizing action and to estimate its value.

Suppose we divided the samples in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(s)$, each an estimate of the true value $q(a)$ for all $a \in \mathcal{A}$. We could then use one estimate, say Q_1, to determine the maximizing action $A^* = \text{argmax}_a Q_1(a)$, and other, say Q_2, to provide the estimate of its value $Q_2(A^*) = Q_2(\text{argmax}_a Q_1(a))$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(A^*) = Q_1(\text{argmax}_a Q_2(a))$, so the two approximate value functions are treated completely symmetrically. This is the idea of double learning.

Algorithm 13 Double Q-learning, for estimating $\pi \approx \pi_*$

```
 1: $Q_1(s,a) \longleftarrow 0$
 2: $Q_2(s,a) \longleftarrow 0$
 3: loop for each episode
 4:     $S \longleftarrow$ initial state
 5:     loop for each step
 6:         Choose $A$ from $S$ using policy derived from $(Q_1 + Q_2)/2$
 7:         Take action $A$, observe $R, S'$
 8:         if choice$(1,2) = 1$ then
 9:             $Q_1(S,A) \longleftarrow Q_1(S,A) + \alpha\,(R + \gamma Q_1(S', \text{argmax}_a Q_2(S',a)) - Q_1(S,A))$
10:         else
11:             $Q_2(S,A) \longleftarrow Q_2(S,A) + \alpha\,(R + \gamma Q_2(S', \text{argmax}_a Q_1(S',a)) - Q_2(S,A))$
12:         end if
13:         $S' \longleftarrow S$
14:     end loop
15: end loop
```

Bibliography

[1] Greg Brockman et al. *OpenAI Gym*. 2016.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd. Adaptive Computation and Machine Learning. MIT Press, 2018. ISBN: 9780262039246.