

## Chapter 2

# Multi-arm Bandits


The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that *evaluates the actions taken* rather than *instructs by giving correct actions*. This is what creates the need for *active exploration*, for an explicit *trial-and-error search* for good behavior. Purely *evaluative feedback* indicates how good the action taken is, but not whether it is the best or the worst action possible. Evaluative feedback is the basis of methods for function optimization, including evolutionary methods. Purely *instructive feedback*, on the other hand, indicates the correct action to take, *independently of the action actually taken*. This kind of feedback is the basis of *supervised learning*, which includes large parts of pattern classification, artificial neural networks, and system identification. In their pure forms, these two kinds of feedback are quite distinct: *evaluative feedback depends entirely on the action taken*, whereas *instructive feedback is independent of the action taken*. There are also interesting intermediate cases in which evaluation and instruction blend together.

In this chapter we study the evaluative aspect of reinforcement learning in a simplified setting, one that does not involve learning to act in more than one situation. This *nonassociative* setting is the one in which most prior work involving evaluative feedback has been done, and it avoids much of the complexity of the full reinforcement learning problem. Studying this case will enable us to see most clearly how evaluative feedback differs from, and yet can be combined with, instructive feedback.

The particular nonassociative, evaluative feedback problem that we explore is a simple version of the  $n$ -armed bandit problem. We use this problem to introduce a number of basic learning methods which we extend in later chapters to apply to the full reinforcement learning problem. At the end of this chapter, we take a step closer to the full reinforcement learning problem by discussing


what happens when the bandit problem becomes associative, that is, when actions are taken in more than one situation.

## 2.1 An $n$ -Armed Bandit Problem

Consider the following learning problem. You are faced repeatedly with a choice among  $n$  different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or *time steps*. 

This is the original form of the *n-armed bandit problem*, so named by analogy to a slot machine, or “one-armed bandit,” except that it has  $n$  levers instead of one. Each action selection is like a play of one of the slot machine’s levers, and the rewards are the payoffs for hitting the jackpot. Through repeated action selections you are to maximize your winnings by concentrating your actions on the best levers. Another analogy is that of a doctor choosing between experimental treatments for a series of seriously ill patients. Each action selection is a treatment selection, and each reward is the survival or well-being of the patient. Today the term “ $n$ -armed bandit problem” is sometimes used for a generalization of the problem described above, but in this book we use it to refer just to this simple case.

In our  $n$ -armed bandit problem, each action has an expected or mean reward given that that action is selected; let us call this the *value* of that action. If you knew the value of each action, then it would be trivial to solve the  $n$ -armed bandit problem: you would always select the action with highest value. We assume that you do not know the action values with certainty, although you may have estimates.

If you maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest. We call this a *greedy action*. If you select a greedy action, we say that you are *exploiting* your current knowledge of the values of the actions. If instead you select one of the nongreedy actions, then we say you are *exploring*, because this enables you to improve your estimate of the nongreedy action’s value. Exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce the greater total reward in the long run. For example, suppose the greedy action’s value is known with certainty, while several other actions are estimated to be nearly as good but with substantial uncertainty. The uncertainty is such that at least one of these other actions probably is 

actually better than the greedy action, but you don't know which one. If you have many time steps ahead on which to make action selections, then it may be better to explore the nongreedy actions and discover which of them are better than the greedy action. Reward is lower in the short run, during exploration, but higher in the long run because after you have discovered the better actions, you can exploit *them* many times. Because it is **not possible both to explore and to exploit with any single action selection**, one often refers to the “conflict” between exploration and exploitation.


In any specific case, whether it is **better to explore or exploit** depends in a complex way on the **precise values of the estimates**, **uncertainties**, and the **number of remaining steps**. There are many sophisticated methods for balancing exploration and exploitation for particular mathematical formulations of the  $n$ -armed bandit and related problems. However, most of these methods make strong assumptions about stationarity and prior knowledge that are either violated or impossible to verify in applications and in the full reinforcement learning problem that we consider in subsequent chapters. The guarantees of optimality or bounded loss for these methods are of little comfort when the assumptions of their theory do not apply.

In this book we do not worry about balancing exploration and exploitation in a sophisticated way; we worry only about balancing them at all. In this chapter we present several simple balancing methods for the  $n$ -armed bandit problem and show that they work much better than methods that always exploit. The need to balance exploration and exploitation is a distinctive challenge that arises in reinforcement learning; the simplicity of the  $n$ -armed bandit problem enables us to show this in a particularly clear form.

## 2.2 Action-Value Methods

Action Selection Method (ASM)

We begin by looking more closely at some simple methods for estimating the values of actions and for using the estimates to make action selection decisions. In this chapter, we denote the **true (actual) value of action  $a$  as  $q(a)$** , and the **estimated value on the  $t$ th time step as  $Q_t(a)$** . Recall that the true value of an action is the mean reward received when that action is selected. One natural way to estimate this is by averaging the rewards actually received when the action was selected. In other words, if by the  $t$ th time step **action  $a$  has been chosen  $N_t(a)$  times prior to  $t$ , yielding rewards  $R_1, R_2, \dots, R_{N_t(a)}$** , then its value is estimated to be



$$Q_t(a) = \frac{R_1 + R_2 + \dots + R_{N_t(a)}}{N_t(a)}. \quad (2.1)$$

If  $N_t(a) = 0$ , then we define  $Q_t(a)$  instead as some default value, such as  $Q_1(a) = 0$ . As  $N_t(a) \rightarrow \infty$ , by the law of large numbers,  $Q_t(a)$  converges to  $q(a)$ . We call this the *sample-average method for estimating action values* because each estimate is a simple average of the sample of relevant rewards. Of course this is just one way to estimate action values, and not necessarily the best one. Nevertheless, for now let us stay with this simple estimation method and turn to the question of how the estimates might be used to select actions.

The *simplest action selection rule* is to select the action (or one of the actions) with highest estimated action value, that is, to select at step  $t$  one of the greedy actions,  $A_t^*$ , for which  $Q_t(A_t^*) = \max_a Q_t(a)$ . This *greedy action selection method* can be written as

$$A_t = \operatorname{argmax}_a Q_t(a), \quad (2.2)$$

where  $\operatorname{argmax}_a$  denotes the value of  $a$  at which the expression that follows is maximized (with ties broken arbitrarily). Greedy action selection *always exploits* current knowledge to *maximize immediate reward*; it spends no time at all sampling apparently inferior actions to see if they might really be better. A *simple alternative* is to *behave greedily most of the time*, but every once in a while, say with *small probability  $\varepsilon$* , instead to *select randomly from amongst all the actions with equal probability* independently of the action-value estimates. We call methods using this near-greedy action selection rule  *$\varepsilon$ -greedy methods*. An advantage of these methods is that, in the limit as the number of plays increases, every action will be sampled an infinite number of times, guaranteeing that  $N_t(a) \rightarrow \infty$  for all  $a$ , and thus ensuring that all the  $Q_t(a)$  converge to  $q(a)$ . This of course implies that the probability of selecting the optimal action converges to greater than  $1 - \varepsilon$ , that is, to near certainty. These are just asymptotic guarantees, however, and say little about the practical effectiveness of the methods.

To roughly assess the relative effectiveness of the greedy and  $\varepsilon$ -greedy methods, we compared them numerically on a suite of test problems. This was a set of 2000 randomly generated  $n$ -armed bandit tasks with  $n = 10$ . For each bandit, the action values,  $q(a)$ ,  $a = 1, \dots, 10$ , were selected according to a normal (Gaussian) distribution with mean 0 and variance 1. On  $t$ th time step with a given bandit, the *actual reward  $R_t$*  was the  $q(A_t)$  for the bandit (where  $A_t$  was the action selected) plus a *normally distributed noise term* that was mean 0 and variance 1. Averaging over bandits, we can plot the performance and behavior of various methods as they improve with experience over 1000 steps, as in Figure 2.1. We call this suite of test tasks the *10-armed testbed*.

Figure 2.1 compares a greedy method with two  $\varepsilon$ -greedy methods ( $\varepsilon = 0.01$  and  $\varepsilon = 0.1$ ), as described above, on the 10-armed testbed. Both methods

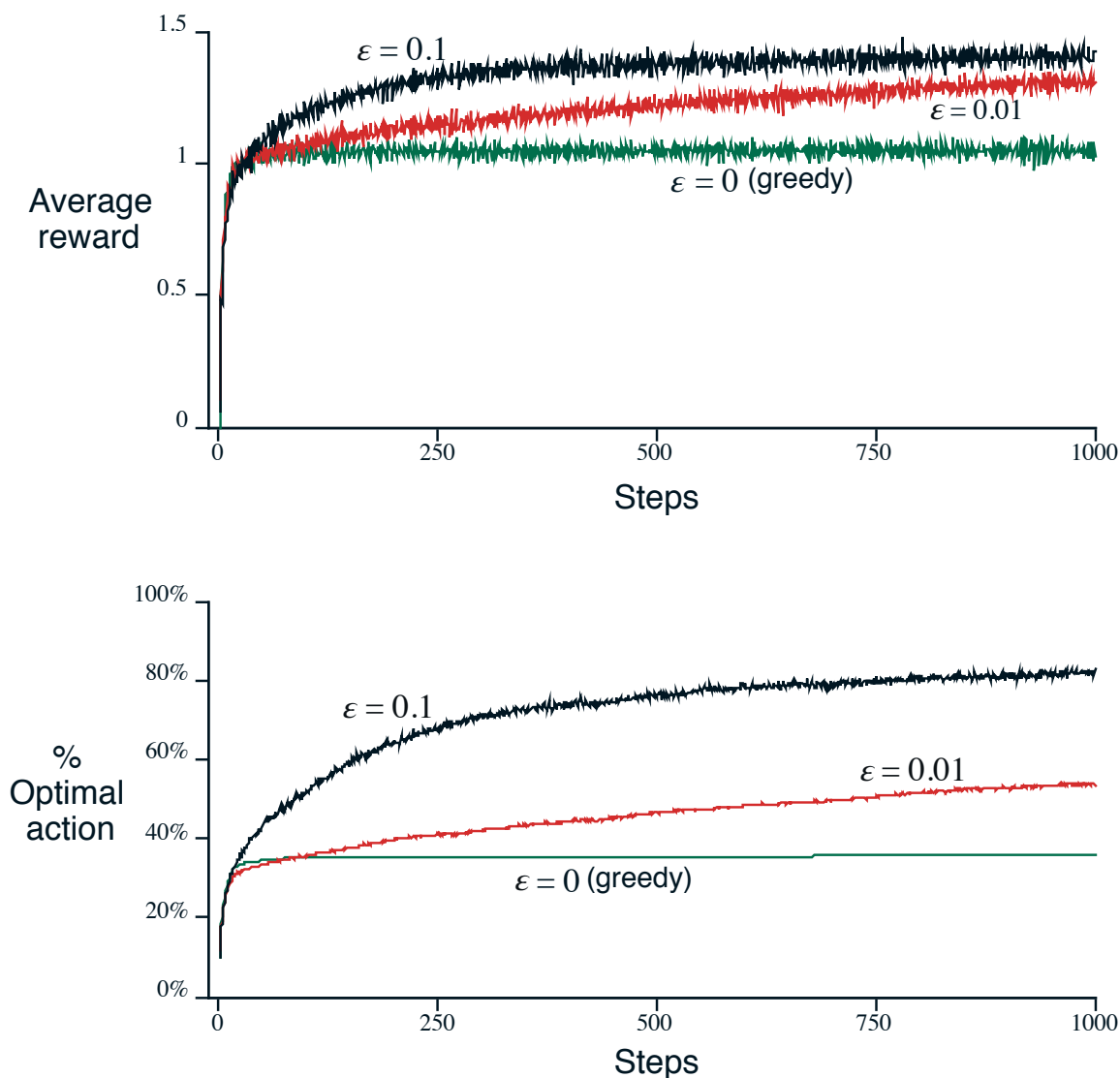


Figure 2.1: Average performance of  $\epsilon$ -greedy action-value methods on the 10-armed testbed. These data are averages over 2000 tasks. All methods used sample averages as their action-value estimates. The detailed structure at the beginning of these curves depends on how actions are selected when multiple actions have the same maximal action value. Here such ties were broken randomly. An alternative that has a similar effect is to add a very small amount of randomness to each of the initial action values, so that ties effectively never happen.

formed their action-value estimates using the sample-average technique. The upper graph shows the increase in expected reward with experience. The greedy method improved slightly faster than the other methods at the very beginning, but then leveled off at a lower level. It achieved a reward per step of only about 1, compared with the best possible of about 1.55 on this testbed. The greedy method performs significantly worse in the long run because it often gets stuck performing suboptimal actions. The lower graph shows that the greedy method found the optimal action in only approximately one-third of the tasks. In the other two-thirds, its initial samples of the optimal action were disappointing, and it never returned to it. The  $\varepsilon$ -greedy methods eventually perform better because they continue to explore, and to improve their chances of recognizing the optimal action. The  $\varepsilon = 0.1$  method explores more, and usually finds the optimal action earlier, but never selects it more than 91% of the time. The  $\varepsilon = 0.01$  method improves more slowly, but eventually performs better than the  $\varepsilon = 0.1$  method on both performance measures. It is also possible to reduce  $\varepsilon$  over time to try to get the best of both high and low values.

The advantage of  $\varepsilon$ -greedy over greedy methods depends on the task. For example, suppose the reward variance had been larger, say 10 instead of 1. With noisier rewards it takes more exploration to find the optimal action, and  $\varepsilon$ -greedy methods should fare even better relative to the greedy method. On the other hand, if the reward variances were zero, then the greedy method would know the true value of each action after trying it once. In this case the greedy method might actually perform best because it would soon find the optimal action and then never explore. But even in the deterministic case, there is a large advantage to exploring if we weaken some of the other assumptions. For example, suppose the bandit task were nonstationary, that is, that the true values of the actions changed over time. In this case exploration is needed even in the deterministic case to make sure one of the nongreedy actions has not changed to become better than the greedy one. As we will see in the next few chapters, effective nonstationarity is the case most commonly encountered in reinforcement learning. Even if the underlying task is stationary and deterministic, the learner faces a set of banditlike decision tasks each of which changes over time due to the learning process itself. Reinforcement learning requires a balance between exploration and exploitation.

## 2.3 Incremental Implementation

The action-value methods we have discussed so far all estimate action values as sample averages of observed rewards. The obvious implementation is to

maintain, for each action  $a$ , a record of all the rewards that have followed the selection of that action. Then, when the estimate of the value of action  $a$  is needed at time  $t$ , it can be computed according to (2.1), which we repeat here:

$$Q_t(a) = \frac{R_1 + R_2 + \cdots + R_{N_t(a)}}{N_t(a)},$$

where here  $R_1, \dots, R_{N_t(a)}$  are all the rewards received following all selections of action  $a$  prior to play  $t$ . A problem with this straightforward implementation is that its memory and computational requirements grow over time without bound. That is, each additional reward following a selection of action  $a$  requires more memory to store it and results in more computation being required to determine  $Q_t(a)$ .

As you might suspect, this is not really necessary. It is easy to devise incremental update formulas for computing averages with small, constant computation required to process each new reward. For some action, let  $Q_k$  denote the estimate for its  $k$ th reward, that is, the average of its first  $k - 1$  rewards. Given this average and a  $k$ th reward for the action,  $R_k$ , then the average of all  $k$  rewards can be computed by

$$\begin{aligned} Q_{k+1} &= \frac{1}{k} \sum_{i=1}^k R_i \\ &= \frac{1}{k} \left( R_k + \sum_{i=1}^{k-1} R_i \right) \\ &= \frac{1}{k} \left( R_k + (k-1)Q_k + Q_k - Q_k \right) \\ &= \frac{1}{k} \left( R_k + kQ_k - Q_k \right) \\ &= Q_k + \frac{1}{k} [R_k - Q_k], \end{aligned} \tag{2.3}$$

which holds even for  $k = 1$ , obtaining  $Q_2 = R_1$  for arbitrary  $Q_1$ . This implementation requires memory only for  $Q_k$  and  $k$ , and only the small computation (2.3) for each new reward.

The update rule (2.3) is of a form that occurs frequently throughout this book. The general form is

$$NewEstimate \leftarrow OldEstimate + StepSize [Target - OldEstimate]. \tag{2.4}$$

The expression  $[Target - OldEstimate]$  is an *error* in the estimate. It is reduced by taking a step toward the “Target.” The target is presumed to

indicate a desirable direction in which to move, though it may be noisy. In the case above, for example, the target is the  $k$ th reward.

Note that the **step-size parameter** (*StepSize*) used in the incremental method described above **changes from time step to time step**. In processing the  $k$ th reward for action  $a$ , that method uses a step-size parameter of  $\frac{1}{k}$ . In this book we denote the step-size parameter by the symbol  $\alpha$  or, more generally, by  $\alpha_t(a)$ . We sometimes use the **informal shorthand**  $\alpha = \frac{1}{k}$  to refer to this case, leaving the **dependence of  $k$  on the action implicit**.

## 2.4 Tracking a Nonstationary Problem

The averaging methods discussed so far are appropriate in a stationary environment, but not if the bandit is changing over time. As noted earlier, we often encounter reinforcement learning problems that are effectively nonstationary. In such cases it makes sense to weight recent rewards more heavily than long-past ones. One of the most popular ways of doing this is to use a constant step-size parameter. For example, the incremental update rule (2.3) for updating an average  $Q_k$  of the  $k - 1$  past rewards is modified to be

$$Q_{k+1} = Q_k + \alpha [R_k - Q_k], \quad (2.5)$$

where the **step-size parameter  $\alpha \in (0, 1]$ <sup>1</sup> is constant**. This results in  $Q_{k+1}$  being a **weighted average of past rewards** and the initial estimate  $Q_1$ :

$$\begin{aligned} Q_{k+1} &= Q_k + \alpha [R_k - Q_k] \\ &= \alpha R_k + (1 - \alpha) Q_k \\ &= \alpha R_k + (1 - \alpha) [\alpha R_{k-1} + (1 - \alpha) Q_{k-1}] \\ &= \alpha R_k + (1 - \alpha) \alpha R_{k-1} + (1 - \alpha)^2 Q_{k-1} \\ &= \alpha R_k + (1 - \alpha) \alpha R_{k-1} + (1 - \alpha)^2 \alpha R_{k-2} + \\ &\quad \dots + (1 - \alpha)^{k-1} \alpha R_1 + (1 - \alpha)^k Q_1 \\ &= (1 - \alpha)^k Q_1 + \sum_{i=1}^k \alpha (1 - \alpha)^{k-i} R_i. \end{aligned} \quad (2.6)$$

We call this a weighted average because the sum of the weights is  $(1 - \alpha)^k + \sum_{i=1}^k \alpha (1 - \alpha)^{k-i} = 1$ , as you can check yourself. Note that the **weight,  $\alpha (1 - \alpha)^{k-i}$ , given to the reward  $R_i$  depends on how many rewards ago,  $k - i$ , it was**

<sup>1</sup>The notation  $(a, b]$  as a set denotes the real interval between  $a$  and  $b$  including  $b$  but not including  $a$ . Thus, here we are saying that  $0 < \alpha \leq 1$ .



observed. The quantity  $1 - \alpha$  is less than 1, and thus the weight given to  $R_i$  decreases as the number of intervening rewards increases. In fact, the weight decays exponentially according to the exponent on  $1 - \alpha$ . (If  $1 - \alpha = 0$ , then all the weight goes on the very last reward,  $R_k$ , because of the convention that  $0^0 = 1$ .) Accordingly, this is sometimes called an *exponential, recency-weighted average*.

Sometimes it is convenient to vary the step-size parameter from step to step. Let  $\alpha_k(a)$  denote the step-size parameter used to process the reward received after the  $k$ th selection of action  $a$ . As we have noted, the choice  $\alpha_k(a) = \frac{1}{k}$  results in the sample-average method, which is guaranteed to converge to the true action values by the law of large numbers. But of course convergence is not guaranteed for all choices of the sequence  $\{\alpha_k(a)\}$ . A well-known result in stochastic approximation theory gives us the conditions required to assure convergence with probability 1:

$$\checkmark \sum_{k=1}^{\infty} \alpha_k(a) = \infty \quad \text{and} \quad \checkmark \sum_{k=1}^{\infty} \alpha_k^2(a) < \infty. \quad (2.7)$$

The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence.

Note that both convergence conditions are met for the sample-average case,  $\alpha_k(a) = \frac{1}{k}$ , but not for the case of constant step-size parameter,  $\alpha_k(a) = \alpha$ . In the latter case, the second condition is not met, indicating that the estimates never completely converge but continue to vary in response to the most recently received rewards. As we mentioned above, this is actually desirable in a nonstationary environment, and problems that are effectively nonstationary are the norm in reinforcement learning. In addition, sequences of step-size parameters that meet the conditions (2.7) often converge very slowly or need considerable tuning in order to obtain a satisfactory convergence rate. Although sequences of step-size parameters that meet these convergence conditions are often used in theoretical work, they are seldom used in applications and empirical research.

## 2.5 Optimistic Initial Values

All the methods we have discussed so far are dependent to some extent on the initial action-value estimates,  $Q_1(a)$ . In the language of statistics, these methods are *biased* by their initial estimates. For the sample-average methods,

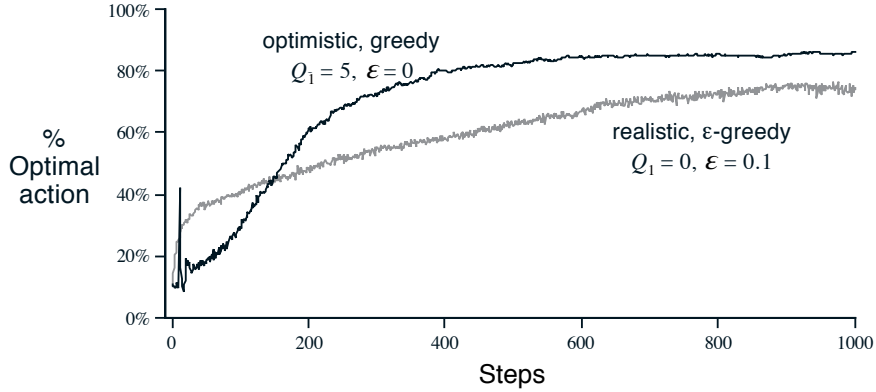


Figure 2.2: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter,  $\alpha = 0.1$ .

the bias disappears once all actions have been selected at least once, but for methods with constant  $\alpha$ , the bias is permanent, though decreasing over time as given by (2.6). In practice, this kind of bias is usually not a problem, and can sometimes be very helpful. The downside is that the initial estimates become, in effect, a set of parameters that must be picked by the user, if only to set them all to zero. The upside is that they provide an easy way to supply some prior knowledge about what level of rewards can be expected.

Initial action values can also be used as a simple way of encouraging exploration. Suppose that instead of setting the initial action values to zero, as we did in the 10-armed testbed, we set them all to  $+5$ . Recall that the  $q(a)$  in this problem are selected from a normal distribution with mean 0 and variance 1. An initial estimate of  $+5$  is thus wildly optimistic. But this optimism encourages action-value methods to explore. Whichever actions are initially selected, the reward is less than the starting estimates; the learner switches to other actions, being “disappointed” with the rewards it is receiving. The result is that all actions are tried several times before the value estimates converge. The system does a fair amount of exploration even if greedy actions are selected all the time.

Figure 2.2 shows the performance on the 10-armed bandit testbed of a greedy method using  $Q_1(a) = +5$ , for all  $a$ . For comparison, also shown is an  $\epsilon$ -greedy method with  $Q_1(a) = 0$ . Initially, the optimistic method performs worse because it explores more, but eventually it performs better because its exploration decreases with time. We call this technique for encouraging exploration *optimistic initial values*. We regard it as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration. For example, it is not well suited to

nonstationary problems because its drive for exploration is inherently temporary. If the task changes, creating a renewed need for exploration, this method cannot help. Indeed, any method that focuses on the initial state in any special way is unlikely to help with the general nonstationary case. The beginning of time occurs only once, and thus we should not focus on it too much. This criticism applies as well to the sample-average methods, which also treat the beginning of time as a special event, averaging all subsequent rewards with equal weights. Nevertheless, all of these methods are very simple, and one of them or some simple combination of them is often adequate in practice. In the rest of this book we make frequent use of several of these simple exploration techniques.

## 2.6 Upper-Confidence-Bound Action Selection

Exploration is needed because the estimates of the action values are uncertain. The greedy actions are those that look best at present, but some of the other actions may actually be better.  $\varepsilon$ -greedy action selection forces the non-greedy actions to be tried, but indiscriminately, with no preference for those that are nearly greedy or particularly uncertain. It would be better to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates. One effective way of doing this is to select actions as

$$A_t = \operatorname{argmax}_a \left[ Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}} \right], \quad (2.8)$$

where  $\ln t$  denotes the natural logarithm of  $t$  (the number that  $e \approx 2.71828$  would have to be raised to in order to equal  $t$ ), and the number  $c > 0$  controls the degree of exploration. If  $N_t(a) = 0$ , then  $a$  is considered to be a maximizing action.

The idea of this *upper confidence bound (UCB)* action selection is that the square-root term is a measure of the uncertainty or variance in the estimate of  $a$ 's value. The quantity being max'ed over is thus a sort of upper bound on the possible true value of action  $a$ , with the  $c$  parameter determining the confidence level. Each time  $a$  is selected the uncertainty is presumably reduced;  $N_t(a)$  is incremented and, as it appears in the denominator of the uncertainty term, the term is decreased. On the other hand, each time an action other  $a$  is selected  $t$  is increased; as it appears in the numerator the uncertainty estimate is increased. The use of the natural logarithm means that the increase gets smaller over time, but is unbounded; all actions will eventually be selected, but

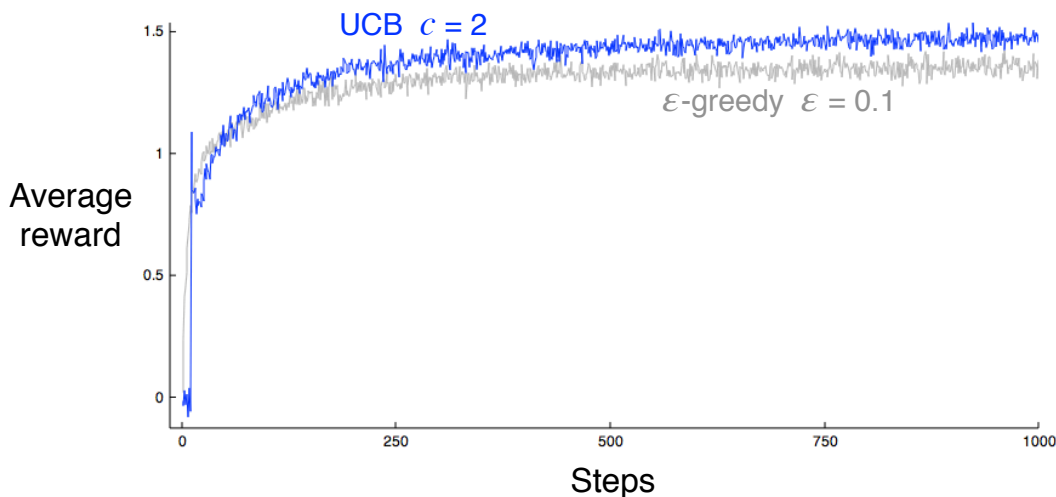


Figure 2.3: Average performance of UCB action selection on the 10-armed testbed. As shown, UCB generally performs better than  $\epsilon$ -greedy action selection, except in the first  $n$  plays, when it selects randomly among the as-yet-unplayed actions. UCB with  $c = 1$  would perform even better but would not show the prominent spike in performance on the 11th play. Can you think of an explanation of this spike?

as time goes by it will be a longer wait, and thus a lower selection frequency, for actions with a lower value estimate or that have already been selected more times.

Results with UCB on the 10-armed testbed are shown in Figure 2.3. UCB will often perform well, as shown here, but is more difficult than  $\epsilon$ -greedy to extend beyond bandits to the more general reinforcement learning settings considered in the rest of this book. One difficulty is in dealing with nonstationary problems; something more complex than the methods presented in Section 2.4 would be needed. Another difficulty is dealing with large state spaces, particularly function approximation as developed in Part III of this book. In these more advanced settings there is currently no known practical way of utilizing the idea of UCB action selection.

## 2.7 Gradient Bandits

So far in this chapter we have considered methods that estimate action values and use those estimates to select actions. This is often a good approach, but it is not the only one possible. In this section we consider learning a numerical *preference*  $H_t(a)$  for each action  $a$ . The larger the preference, the

more often that action is taken, but the preference has no interpretation in terms of reward. Only the relative preference of one action over another is important; if we add 1000 to all the preferences there is no affect on the action probabilities, which are determined according to a soft-max distribution (i.e., Gibbs or Boltzmann distribution) as follows:

$$\Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_{b=1}^n e^{H_t(b)}} = \pi_t(a), \quad (2.9)$$

where here we have also introduced a useful new notation  $\pi_t(a)$  for the probability of taking action  $a$  at time  $t$ . Initially all preferences are the same (e.g.,  $H_1(a) = 0, \forall a$ ) so that all actions have an equal probability of being selected.

There is a natural learning algorithm for this setting based on the idea of stochastic gradient ascent. On each step, after selecting the action  $A_t$  and receiving the reward  $R_t$ , the preferences are updated by:

$$\begin{aligned} H_{t+1}(A_t) &= H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), & \text{and} \\ H_{t+1}(a) &= H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a), & \forall a \neq A_t, \end{aligned} \quad (2.10)$$

where  $\alpha > 0$  is a step-size parameter, and  $\bar{R}_t \in \mathbb{R}$  is the average of all the rewards up through and including time  $t$ , which can be computed incrementally as described in Section 2.3 (or Section 2.4 if the problem is nonstationary). The  $\bar{R}_t$  term serves as a baseline with which the reward is compared. If the reward is higher than the baseline, then the probability of taking  $A_t$  in the future is increased, and if the reward is below baseline, then probability is decreased. The non-selected actions move in the opposite direction.

Figure 2.4 shows results with the gradient-bandit algorithm on a variant of the 10-armed testbed in which the true expected rewards were selected according to a normal distribution with a mean of +4 instead of zero (and with unit variance as before). This shifting up of all the rewards has absolutely no affect on the gradient-bandit algorithm because of the reward baseline term, which instantaneously adapts to the new level. But if the baseline were omitted (that is, if  $\bar{R}_t$  was taken to be constant zero in (2.10)), then performance would be significantly degraded, as shown in the figure.

One can gain a deeper insight into this algorithm by understanding it as a stochastic approximation to gradient ascent. In exact *gradient ascent*, each preference  $H_t(a)$  would be incrementing proportional to the increment's effect on performance:

$$H_{t+1}(a) = H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \quad (2.11)$$

where the measure of performance here is the expected reward:

$$\mathbb{E}[R_t] = \sum_b \pi_t(b)q(b).$$

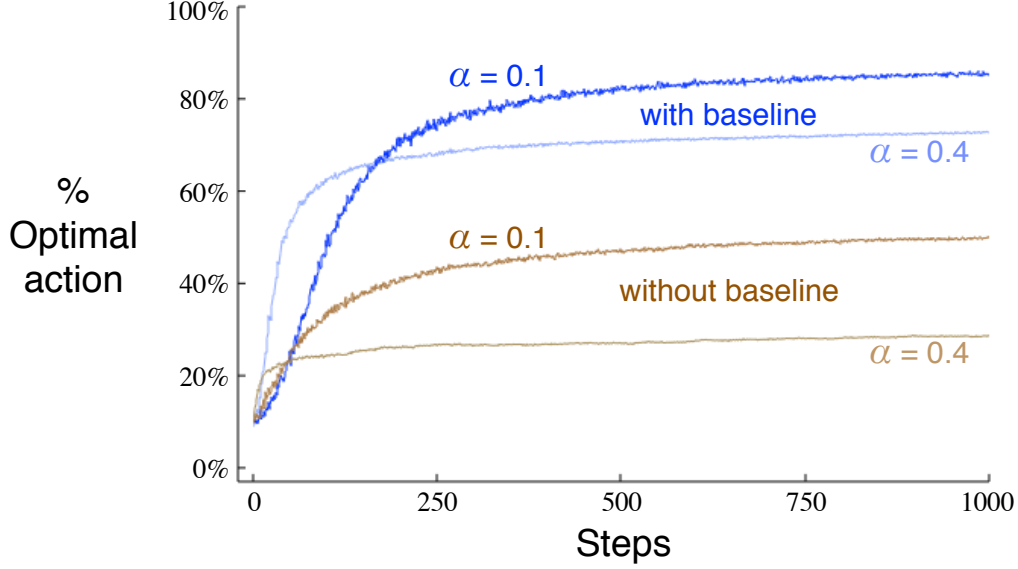


Figure 2.4: Average performance of the gradient-bandit algorithm with and without a reward baseline on the 10-armed testbed with  $\mathbb{E}[q(a)] = 4$ .

Of course, it is not possible to implement gradient ascent exactly in our case because by assumption we do not know the  $q(b)$ , but in fact the updates of our algorithm (2.10) are equal to (2.11) in expected value, making the algorithm an instance of *stochastic gradient ascent*.

The calculations showing this require only beginning calculus, but take several steps. If you are mathematically inclined, then you will enjoy the rest of this section in which we go through these steps. First we take a closer look at the exact performance gradient:

$$\begin{aligned}
 \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \left[ \sum_b \pi_t(b) q(b) \right] \\
 &= \sum_b q(b) \frac{\partial \pi_t(b)}{\partial H_t(a)} \\
 &= \sum_b (q(b) - X_t) \frac{\partial \pi_t(b)}{\partial H_t(a)},
 \end{aligned}$$

where  $X_t$  can be any scalar that does not depend on  $b$ . We can include it here because the gradient sums to zero over all the actions,  $\sum_b \frac{\partial \pi_t(b)}{\partial H_t(a)} = 0$ . As  $H_t(a)$  is changed, some actions' probabilities go up and some down, but the sum of the changes must be zero because the sum of the probabilities must

remain one.

$$= \sum_b \pi_t(b) (q(b) - X_t) \frac{\partial \pi_t(b)}{\partial H_t(a)} / \pi_t(b)$$

The equation is now in the form of an expectation, summing over all possible values  $b$  of the random variable  $A_t$ , then multiplying by the probability of taking those values. Thus:

$$\begin{aligned} &= \mathbb{E} \left[ (q(A_t) - X_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t) \right] \\ &= \mathbb{E} \left[ (R_t - \bar{R}_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t) \right], \end{aligned}$$

where here we have chosen  $X_t = \bar{R}_t$  and substituted  $R_t$  for  $q(A_t)$ , which is permitted because  $\mathbb{E}[R_t] = q(A_t)$  and because all the other factors are non-random. Shortly we will establish that  $\frac{\partial \pi_t(b)}{\partial H_t(a)} = \pi_t(b) (\mathbb{I}_{a=b} - \pi_t(a))$ , where  $\mathbb{I}_{a=b}$  is defined to be 1 if  $a = b$ , else 0. Assuming that for now we have

$$\begin{aligned} &= \mathbb{E} \left[ (R_t - \bar{R}_t) \pi_t(A_t) (\mathbb{I}_{a=A_t} - \pi_t(a)) / \pi_t(A_t) \right] \\ &= \mathbb{E} \left[ (R_t - \bar{R}_t) (\mathbb{I}_{a=A_t} - \pi_t(a)) \right]. \end{aligned}$$

Recall that our plan has been to write the performance gradient as an expectation of something that we can sample on each step, as we have just done, and then update on each step proportional to the sample. Substituting a sample of the expectation above for the performance gradient in (2.11) yields:

$$H_{t+1}(a) = H_t(a) + \alpha (R_t - \bar{R}_t) (\mathbb{I}_{a=A_t} - \pi_t(a)), \quad \forall a,$$

which you will recognize as being equivalent to our original algorithm (2.10).

Thus it remains only to show that  $\frac{\partial \pi_t(b)}{\partial H_t(a)} = \pi_t(b) (\mathbb{I}_{a=b} - \pi_t(a))$ , as we assumed earlier. Recall the standard quotient rule for derivatives:

$$\frac{\partial}{\partial x} \left[ \frac{f(x)}{g(x)} \right] = \frac{\frac{\partial f(x)}{\partial x} g(x) - f(x) \frac{\partial g(x)}{\partial x}}{g(x)^2}.$$

Using this, we can write

$$\begin{aligned}
\frac{\partial \pi_t(b)}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \pi_t(b) \\
&= \frac{\partial}{\partial H_t(a)} \left[ \frac{e^{H_t(b)}}{\sum_{c=1}^n e^{H_t(c)}} \right] \\
&= \frac{\frac{\partial e^{H_t(b)}}{\partial H_t(a)} \sum_{c=1}^n e^{H_t(c)} - e^{H_t(b)} \frac{\partial \sum_{c=1}^n e^{H_t(c)}}{\partial H_t(a)}}{\left( \sum_{c=1}^n e^{H_t(c)} \right)^2} && \text{(by the quotient rule)} \\
&= \frac{\mathbb{I}_{a=b} e^{H_t(a)} \sum_{c=1}^n e^{H_t(c)} - e^{H_t(b)} e^{H_t(a)}}{\left( \sum_{c=1}^n e^{H_t(c)} \right)^2} && \text{(because } \frac{\partial e^x}{\partial x} = e^x \text{)} \\
&= \frac{\mathbb{I}_{a=b} e^{H_t(b)}}{\sum_{c=1}^n e^{H_t(c)}} - \frac{e^{H_t(b)} e^{H_t(a)}}{\left( \sum_{c=1}^n e^{H_t(c)} \right)^2} \\
&= \mathbb{I}_{a=b} \pi_t(b) - \pi_t(b) \pi_t(a) \\
&= \pi_t(b) (\mathbb{I}_{a=b} - \pi_t(a)).
\end{aligned}$$

Q.E.D.

We have just shown that the expected update of the gradient-bandit algorithm is equal to the gradient of expected reward, and thus that the algorithm is an instance of stochastic gradient ascent. This assures us that the algorithm has robust convergence properties.

Note that we did not require anything of the reward baseline other than that it not depend on the selected action. For example, we could have set it to zero, or to 1000, and the algorithm would still have been an instance of stochastic gradient ascent. The choice of the baseline does not affect the expected update of the algorithm, but it does affect the variance of the update and thus the rate of convergence (as shown, e.g., in Figure 2.4). Choosing it as the average of the rewards may not be the very best, but it is simple and works well in practice.

## 2.8 Associative Search (Contextual Bandits)

So far in this chapter we have considered only nonassociative tasks, in which there is no need to associate different actions with different situations. In these tasks the learner either tries to find a single best action when the task is stationary, or tries to track the best action as it changes over time when the task is nonstationary. However, in a general reinforcement learning task there is more than one situation, and the goal is to learn a policy: a mapping from situations to the actions that are best in those situations. To set the stage for



the full problem, we briefly discuss the simplest way in which nonassociative tasks extend to the associative setting.

As an example, suppose there are several different  $n$ -armed bandit tasks, and that on each play you confront one of these chosen at random. Thus, the bandit task changes randomly from play to play. This would appear to you as a single, nonstationary  $n$ -armed bandit task whose true action values change randomly from play to play. You could try using one of the methods described in this chapter that can handle nonstationarity, but unless the true action values change slowly, these methods will not work very well. Now suppose, however, that when a bandit task is selected for you, you are given some distinctive clue about its identity (but not its action values). Maybe you are facing an actual slot machine that changes the color of its display as it changes its action values. Now you can learn a policy associating each task, signaled by the color you see, with the best action to take when facing that task—for instance, if red, play arm 1; if green, play arm 2. With the right policy you can usually do much better than you could in the absence of any information distinguishing one bandit task from another.

This is an example of an *associative search* task, so called because it involves both trial-and-error learning in the form of *search* for the best actions and *association* of these actions with the situations in which they are best.<sup>2</sup> Associative search tasks are intermediate between the  $n$ -armed bandit problem and the full reinforcement learning problem. They are like the full reinforcement learning problem in that they involve learning a policy, but like our version of the  $n$ -armed bandit problem in that each action affects only the immediate reward. If actions are allowed to affect the *next situation* as well as the reward, then we have the full reinforcement learning problem. We present this problem in the next chapter and consider its ramifications throughout the rest of the book.

## 2.9 Summary

We have presented in this chapter several simple ways of balancing exploration and exploitation. The  $\varepsilon$ -greedy methods choose randomly a small fraction of the time, whereas UCB methods choose deterministically but achieve exploration by subtly favoring at each step the actions that have so far received fewer samples. Gradient-bandit algorithms estimate not action values, but action preferences, and favor the more preferred actions in a graded, probabilistic manner using a soft-max distribution. The simple expedient of initializing

---

<sup>2</sup>Associative search tasks are often now termed *contextual bandits* in the literature.

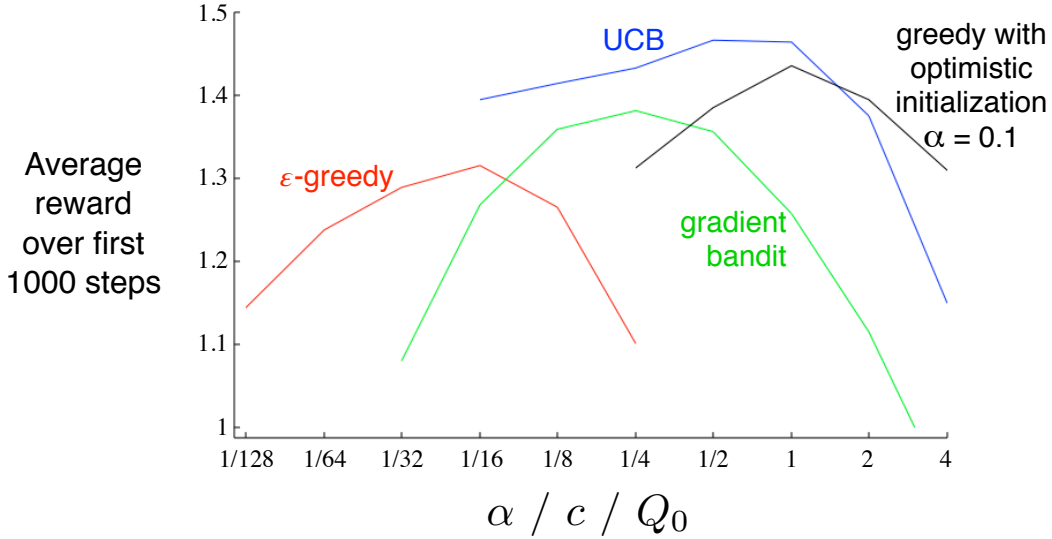


Figure 2.5: A parameter study of the various bandit algorithms presented in this chapter. Each point is the average reward obtained over 1000 steps with a particular algorithm at a particular setting of its parameter.

estimates optimistically causes even greedy methods to explore significantly.

It is natural to ask which of these methods is best. Although this is a difficult question to answer in general, we can certainly run them all on the 10-armed testbed that we have used throughout this chapter and compare their performances. A complication is that they all have a parameter; to get a meaningful comparison we will have to consider their performance as a function of their parameter. Our graphs so far have shown the course of learning over time for each algorithm and parameter setting, but it would be too visually confusing to show such a *learning curve* for each algorithm and parameter value. Instead we summarize a complete learning curve by its average value over the 1000 steps; this value is proportional to the area under the learning curves we have shown up to now. Figure 2.5 shows this measure for the various bandit algorithms from this chapter, each as a function of its own parameter shown on a single scale on the x-axis. Note that the parameter values are varied by factors of two and presented on a log scale. Note also the characteristic inverted-U shapes of each algorithm's performance; all the algorithms perform best at an intermediate value of their parameter, neither too large nor too big. In assessing an method, we should attend not just to how well it does at its best parameter setting, but also to how sensitive it is to its parameter value. All of these algorithms are fairly insensitive, performing well over a range of parameter values varying by about an order of magnitude. Overall, on this problem, UCB seems to perform best.

Despite their simplicity, in our opinion the methods presented in this chapter can fairly be considered the state of the art. There are more sophisticated methods, but their complexity and assumptions make them impractical for the full reinforcement learning problem that is our real focus. Starting in Chapter 5 we present learning methods for solving the full reinforcement learning problem that use in part the simple methods explored in this chapter.

Although the simple methods explored in this chapter may be the best we can do at present, they are far from a fully satisfactory solution to the problem of balancing exploration and exploitation.

The classical solution to balancing exploration and exploitation in  $n$ -armed bandit problems is to compute special functions called *Gittins indices*. These provide an optimal solution to a certain kind of bandit problem more general than that considered here but that assumes the prior distribution of possible problems is known. Unfortunately, neither the theory nor the computational tractability of this method appear to generalize to the full reinforcement learning problem that we consider in the rest of the book.

There is also a well-known algorithm for computing the Bayes optimal way to balance exploration and exploitation. This method is computationally intractable when done exactly, but there may be efficient ways to approximate it. In this method we assume that we know the distribution of problem instances, that is, the probability of each possible set of true action values. Given any action selection, we can then compute the probability of each possible immediate reward and the resultant posterior probability distribution over action values. This evolving distribution becomes the *information state* of the problem. Given a horizon, say 1000 plays, one can consider all possible actions, all possible resulting rewards, all possible next actions, all next rewards, and so on for all 1000 plays. Given the assumptions, the rewards and probabilities of each possible chain of events can be determined, and one need only pick the best. But the tree of possibilities grows extremely rapidly; even if there are only two actions and two rewards, the tree will have  $2^{2000}$  leaves. This approach effectively turns the bandit problem into an instance of the full reinforcement learning problem. In the end, we may be able to use reinforcement learning methods to approximate this optimal solution. But that is a topic for current research and beyond the scope of this book.

## Bibliographical and Historical Remarks

- 2.1** Bandit problems have been studied in statistics, engineering, and psychology. In statistics, bandit problems fall under the heading “sequential design of experiments,” introduced by Thompson (1933, 1934) and