Part III

Reinforcement Learning

CHAPTER

12 Decision Processes

Supervised classification problems can be viewed as a simple type of decision-making, in which we observe a feature x and decide on an action y, based on examples from an expert. Reinforcement learning extends this concept of decisions and actions in several important ways, enabling an agent to determine what to do without the benefit of an explicit, "right answer" label from an expert. Instead, we are provided with feedback about the quality of our performance, but without the luxury of being told what actions, if any, might have been better. Although this framework presents some difficulties, it also encompasses a large spectrum of important tasks.

Consider an AI learning to play cards. A supervised learning agent might try to learn what actions a professional player would make from any given situtation. This would not only need a large amount of difficult-to-acquire data, but even if successful, could at best reproduce the performance of the expert. In contrast, it is easy for us to provide feedback – we can simply reward the agent for winning games. Our AI is then not limited by strategies it has seen; it is free to try anything in pursuit of success.

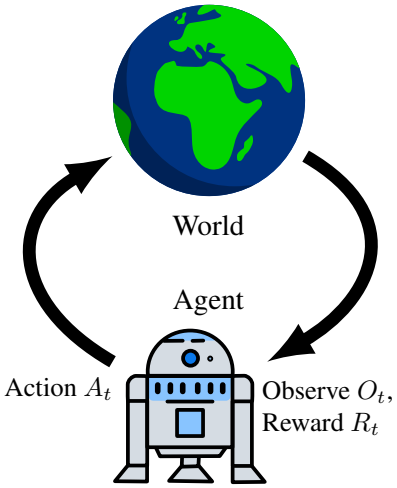
Reinforcement learning is usually, by its nature, temporal. Often the feedback we are able to provide (e.g., rewards for winning) are not immediately apparent at the time a decision is made, but only become clear after a sequence of many decisions. Our decisions are also not independent – the actions we take affect our situation, which then influences future decisions and rewards.

One useful distinction in types of reinforcement learning is whether our environment is *episodic* or *continuing*. Episodic settings are problems in which there is a beginning and an end, for example, a hand of poker or a single game of chess. In these settings, each episode will eventually complete, and we can review a collection of episodes to understand our performance. Typically, episodes are short enough that we can get useful feedback by waiting until completion and then analyzing the overall outcome of an episode. In contrast, continuing settings are those in which there is just a single, ongoing experience, for example, an autonomous vehicle driving around a city. In such settings, we cannot wait for "the end" to obtain our feedback; instead, we expect to be gathering meaningful feedback about how to adjust our behavior on the fly.

Examples of RL?

We begin by considering simpler settings, and build up over several sections to the general problem of reinforcement learning. In particular, we first consider how to take a known model of the world and use it to characterize our current performance over time, and how to optimize our strategy to improve it. We then turn to learning from data, first studying how to use experience to estimate the quality of a particular strategy, and finally how we can optimize our strategy using feedback from our experience.

12.1 Basics of Reinforcement Learning



Consider an agent interacting with the world. By its nature, this interaction is temporal – our agent observes the world with its sensors, and takes some action, which may change some aspect of the world, while other aspects of the world may simply change on their own over time. Sensing again can reveal what effect the action had, and inform us of what action it should perform next. We also recieve rewards, which provide feedback about the quality of our performance. Let us assume that our agent acts on some discrete time interval, which we will index by t. Then, we can denote the observation at time t by O_t, our reward as R_t, and our subsequent action as A_t.

Examples of O,R,A?

What information should our agent use to determine its action A_t? Clearly, it can only use the information it has access to, which we refer to as the *history* at time t:

$$H_t = [O_1, R_1, A_1, O_2, R_2, A_2, \ldots, O_t, R_t].$$

However, not all of this information is necessarily important to determine our action. Let us call the relevant portion of the history the *state*, S_t, which we can think of as a sufficient statistic[1] of the history, $S_t = f(H_t)$.

Our agent then takes this information, S_t, and decides what action to take using a *policy* π. Our policy can be either *deterministic*, mapping each state S_t to a single action a_t, or *stochastic*, meaning that S_t determines a distribution of possible actions A_t which we select from randomly:

$$a_t = \pi(S_t) \qquad\qquad a_t \sim \pi(A_t|S_t)$$

We assume that our policy is *stationary*[2], meaning that the action or action distribution depends only on the value of the state S_t, and not on the time step t.

What makes a good policy π? Many environments involve considerable randomness, so that even following the best possible strategy is not guaranteed to result in success. For example, in poker or other card games, even an expert player can lose any given game; it is only in the long run, as many games are played, that a player's skill becomes apparent. This suggests that to evaluate a strategy, we should be interested in the *expected* rewards that we obtain by following it.

We also differentiate between two types of tasks that our agent needs in order to act. For the first, *decision making*, we will assume that our agent has a complete model of how the world works, i.e., how actions influence the world and thus future observations, states, and rewards; then, we do not need to learn from data, but only to identify a good policy π. This is an example of *inference* – taking a specified model and current observations (e.g., $S_t = f(H_t)$) and deciding what action to take (determining $\pi(S_t)$). The second, more general type of task is *reinforcement learning*, in we treat the model as unknown, so that identifying a good policy π also requires learning the relationship between actions, their results, and our rewards purely from observations.

[1] This definition corresponds to the assumption of an *observable state*. More generally, we may also define the state to include some information that we would like to know, but cannot directly measure (a "*partially observable*" state), in which case S_t will be uncertain, with a distribution $p(S_t|H_t)$ that depends on (some or all of) the history.

[2] Stationarity is not very restrictive, since if necessary we can make the state S_t depend on the time step; for example, keeping track of what stage of a game we are currently at (how many cards have been revealed, etc.).

12.2 Markov Chains & Markov Reward Processes

A *Markov process* or *Markov chain* is a distribution over a sequence of random variables S_t (here, indexed by time) in which each state S_t is independent of the rest of the history H_t given S_{t-1}, so that

$$p(S_1, \ldots, S_T) = p(S_1) \prod_{t=2}^{T} p(S_t \mid S_{t-1})$$

This means that, if we know the current state S_t of our system, its future evolution is independent of the past. We will assume that our model is *homogeneous*, meaning that the distribution $p(S_t \mid S_{t-1})$ does not depend on the time t, i.e.,

$$p(S_t = s \mid S_{t-1} = s') = p(S_{t'} = s \mid S_{t'-1} = s') \qquad \text{for all } t, t' \text{ , } s, s'$$

For discrete-valued states S_t, a Markov chain forms a kind of (stochastic) finite state machine. The initial state distribution $p(S_1)$ determines in what state we begin, and then the state transition distribution $p(S_t|S_{t-1})$ determines which state we move to at each subsequent time. For convenience, we can collect these probabilities into matrices $\mathbf{p}$ and $\mathbf{P}$, where

$$[\mathbf{p}]_j = p(S_1 = j) \qquad\qquad [\mathbf{P}]_{ij} = p(S_t = j|S_{t-1} = i).$$

We will use n to denote the number of possible values of any state S_t.

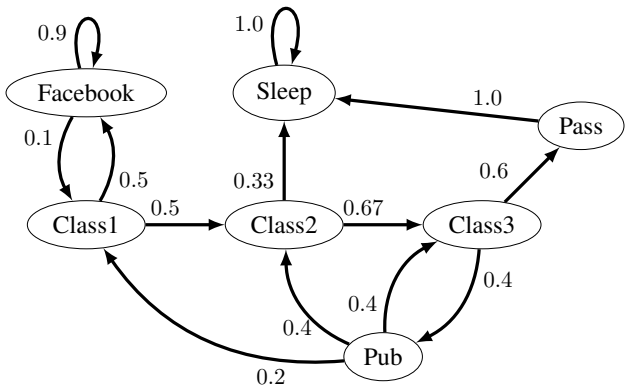
A Markov chain gives us a compact way to describe a probability distribution over sequences of arbitrary, possibily infinite length. **define episode?** If we observe a sequence of states, $[s_1, \ldots, s_T]$, we can evaluate its probability as,

$$p(s_1, \ldots, s_t) = p(S_1 = s_1) \prod_{t>1} p(S_t = s_t|S_{t-1} = s_{t-1}) = \mathbf{p}_{s_1} \prod_{t>1} \mathbf{P}_{s_{t-1}s_t}$$

which requires specifying only $O(n^2)$ probabilities (independent of T). Moreover, there are often only a few states s_t with non-zero probability $p(S_t = s_t|S_{t-1} = s_{t-1})$ (i.e., each row of the matrix $\mathbf{P}$ is sparse). We can often visualize which state transitions are possible using a *state transition diagram*, a graph with one node for each of the n state values, and directed edges to indicate which transitions have non-zero probability (which may also be annotated with the probability values themselves).

Example 12-1 : University life Markov chain

Consider a simple Markov chain model for the life of a university student (adapted from David Silver's lectures **cite?**):



$\mathbf{P}_{s',s} = p(S_t = s|S_{t-1} = s')$:

	Fa	C1	C2	C3	Pu	Pa	Sl
Fa	0.9	0.1					
C1	0.5		0.5				
C2				0.67			0.33
C3					0.4	0.6	
Pu		0.2	0.4	0.4			
Pa							1.0
Sl							1.0

We begin by studying our class, $S_1 = \text{C1}$, knowing very little. Then, with equal probability (0.5), we either learn enough to reach the second level of understanding for our class ("C2"), or end up messing around on Facebook ("Fa"). From Class-2 we might simply go to sleep, or continue studying and reach Class-3; from Class 3 we might lock in our knowledge to pass the course exam ("Pass"), or we might go to the pub, where we may end up losing some of our study progress.

Notice that "Sleep" is a *terminal state*; once we reach this state, we will never leave. We can view this as the end of an episode (one day of studying), in which case our model defines a distribution over sequences of finite[3] but variable length, all ending with "Sleep".

Sampling from our model gives episodes like:

C1	C2	C3	Pass	Sleep											
C1	Fa	Fa	C1	C2	Sleep										
C1	C2	C3	Pub	C2	C3	Pass	Sleep								
C1	Fa	Fa	C1	C2	C3	Pub	C1	Fa	Fa	Fa	C1	C2	C3	Pub	C2	Sleep

We can easily compute the probability of any of these sequences; for example,

$$\begin{aligned}\Pr\left[\text{ C1 C2 C3 Pass Sleep }\right] &= \mathbf{p}_{\text{C1}}\,\mathbf{P}_{\text{C1,C2}}\,\mathbf{P}_{\text{C2,C3}}\,\mathbf{P}_{\text{C3,Pass}}\,\mathbf{P}_{\text{Pass,Sleep}} \\ &= (1)\cdot(0.5)\cdot(0.67)\cdot(0.6)\cdot(1.0) \quad = 0.20\end{aligned}$$

A *Markov reward process* (MRP) is simply a Markov process in which we have added a reward R_t at each time t, with

$$p(R_{t+1}|S_1,\ldots S_t) = p(R_{t+1}|S_t),$$

i.e., our next reward R_{t+1} also exhibits a Markov property, in that it depends only on our current state S_t. We also define a *discount factor* $\gamma \in [0,1]$, and use it to define the *return* on being in state s_t, which is the sum of discounted rewards we receive after being in state s:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{\tau=1}^{\infty} \gamma^{\tau-1} R_{t+\tau}.$$

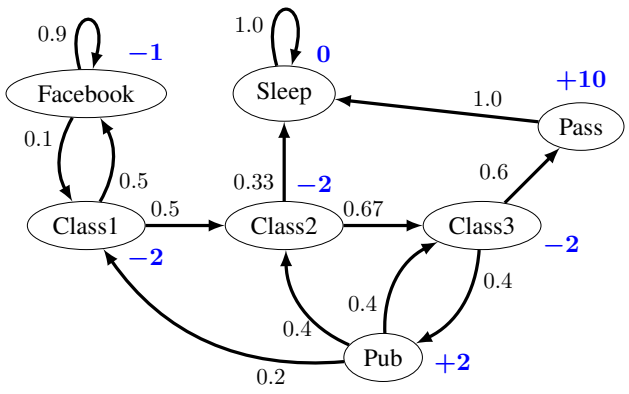
The discount factor γ tells us how much we care about rewards in the future, relative to immediate rewards. It is well-known in behavioral theory that humans prioritize immediate rewards over rewards in the distant future, to a degree that varies by the individual – as evidenced by the famous "marshmallow experiment", in which children were asked to defer eating a marshmallow in return for two later [Mischel and Ebbesen, 1970]. Such discounts arise naturally in part from uncertainty: interest rates on financial instruments like bonds reflect uncertainty in the value of the dollar in the future, and actuarial rates inform us of the financial implications of our own risk of mortality. First and foremost, however, the discount factor $\gamma < 1$ and geometric time dependence γ^τ give a mathematically convenient form, which is used to define recurrence relations for solving Markov reward processes of potentially infinite length[4].

Example 12-2 : University life MRP

We can extend Example 12-1 to include rewards associated with each state s (annotated in blue):

[3] In this case, it is possible to show that the total probability of all sequences longer than T decreases to zero as $T \to \infty$, so that the probability of not eventually reaching the terminal state is zero.

[4] In some settings, such as when all episodes are guaranteed to have finite length (blackjack, etc.) it is possible to use no discount, $\gamma = 1$.



$\mathbf{P}_{s',s} = p(S_t = s|S_{t-1} = s')$:

	Fa	C1	C2	C3	Pu	Pa	Sl
Fa	0.9	0.1					
C1	0.5		0.5				
C2				0.67			0.33
C3					0.4	0.6	
Pu		0.2	0.4	0.4			
Pa							1.0
Sl							1.0

$\mathbf{r}_s = \mathbb{E}[R_{t+1}|S_t = s]$:

Fa	C1	C2	C3	Pu	Pa	Sl
−1	−2	−2	−2	+2	+10	0

If we assume deterministic rewards $R_{t+1} = \mathbb{E}[R_{t+1}|S_t]$ for each state S_t, we can compute the return of any sequence, for example,Markov reward process.

$$\begin{aligned} G\big[\text{ C1 C2 C3 Pass Sleep }\big] &= \mathbf{r}_{\text{C1}} + \mathbf{r}_{\text{C2}} + \mathbf{r}_{\text{C3}} + \mathbf{r}_{\text{Pass}} + \mathbf{r}_{\text{Sleep}} \\ &= (-2) + (-2) + (-2) + (10) + (0) \quad = 4 \end{aligned}$$

We define the *value function* $v(s)$ of an MRP to be the expected return, or expected discounted sum of rewards, of sequences beginning in state s. The value function thus captures both the immediate reward of the state s, along with the potential rewards that can arise in the future due to our path passing through s. We can rewrite the function $v(s)$ in terms of itself, giving the recursion,

$$\begin{aligned} v(s) &= \mathbb{E}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \big| S_t = s\big] \\ &= \mathbb{E}\big[R_{t+1}\big|S_t = s\big] + \gamma \mathbb{E}\big[R_{t+2} + \gamma R_{t+3} + \dots |S_t = s\big] \\ &= \mathbb{E}\big[R_{t+1}\big|S_t = s\big] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s) \mathbb{E}\big[R_{t+2} + \gamma R_{t+3} + \dots |S_{t+1} = s'\big] \\ &= \mathbb{E}\big[R_{t+1}|S_t = s\big] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s)\ v(s'). \end{aligned}$$

This recursive definition of $v(s)$ is called the *Bellman equation*. Defining the vectors

$$[\mathbf{r}]_s = \mathbb{E}[R_{t+1}|S_t = s] \qquad \text{and} \qquad [\mathbf{v}]_s = v(s),$$

this can be rewritten as a matrix vector product, and solved for $\mathbf{v}$:

$$\mathbf{v} = \mathbf{r} + \gamma\, \mathbf{P} \odot \mathbf{v} \qquad \Rightarrow \qquad \mathbf{v} = (I - \gamma\, \mathbf{P})^{-1} \odot \mathbf{r} \tag{12.1}$$

Example 12-3 : Direct solution

Solving the Bellman equations directly, using matrix inversion, for $\mathbf{P}$ given in Example 12-2, yields:

	$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$\gamma = 0$	$v(s) =$	-1	-2	-2	-2	1	10	0
$\gamma = 0.5$	$v(s) =$	-2.08	-2.91	-1.55	1.33	1.67	10	0
$\gamma = 0.99$	$v(s) =$	-18.98	-10.80	1.20	4.83	2.25	10	0

We can see that, for very small discount factors ($\gamma \approx 0$), our expected return is simply the immediate reward of our state. However, as γ increases, the value of our state begins to reflect the rewards which can (eventually) be reached from that state. States which themselves have low immediate reward, but can lead to high-reward paths (such as Class-3, which often leads to passing the exam and its associated reward) become more promising. Meanwhile, states like Facebook, which offer slightly better immediate rewards but have low probability of taking us along the highest reward path, seem promising at short or even intermediate horizons but are less promising when we care about long-term reward.

Eq. 12.1 gives us an exact, closed form solution for $\mathbf{v}$; however, it is only practically useful in very small Markov systems (models with small numbers of states n). The inverse is of an $n \times n$ matrix, which may take $O(n^{2.8}) \approx O(n^3)$ time to compute. However, we may be able to solve the linear system more efficiently using an iterative method, such as dynamic programming.

Dynamic programming is a powerful technique for solving many computational problems in which the solutions to one or more smaller problems are cached and reused to compute the solution to the overall problem. In the case of Markov reward processes, we will solve a sequence of finite-horizon value functions, each of which uses the solution to the previous horizon. So, let us define $v^{(T)}(s)$ to be the value function over sequences of length (at most) T, i.e.,

$$v^{(T)}(s) = \mathbb{E}\Big[\sum_{\tau=1}^{T} \gamma^{\tau-1} R_{t+\tau} \;\Big|\; S_t = s\Big].$$

Clearly, $v^{(1)}(s) = \mathbb{E}[R_{t+1}|S_t = s]$, since we get the reward for being in state s, and then stop immediately. Then, we can see that for any $T > 1$, we have,

$$v^{(T)}(s) = \mathbb{E}[R_{t+1}|S_t = s] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s) v^{(T-1)}(s'),$$

i.e., we can compute the value of sequences of length T in terms of their immediate reward R_{t+1}, plus the expected value of the remaining sequence (of length $T-1$) starting in whichever state s' to which we might have transitioned. So, we can compute $v^{(2)}$ using $v^{(1)}$, then $v^{(3)}$ using $v^{(2)}$, and so on.

Example 12-4 : Dynamic programming

Let us solve the university life MRP using dynamic programming, for discount rate $\gamma = 0.5$. At each step, we compute $v^{(T)}(s)$ for each state s, using the previously computed values of $v^{(T-1)}(\cdot)$. For $T = 1$, $v^{(1)}(s) = \mathbb{E}[R_1|S_1 = s]$:

$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$T = 1$	-1	-2	-2	-2	2	10	0

Now, we can compute $v^{(2)}(s)$, since, for example,

$$\begin{aligned} v^{(2)}(\text{Fa}) &= \mathbb{E}\big[R_2|S_2 = \text{Fa}\big] + \gamma \cdot \big(.9 \cdot v^{(1)}(\text{Fa}) + .1 \cdot v^{(1)}(\text{C1})\big) \\ &= \quad (-1) \quad + (0.5) \cdot \big(.9 \cdot (-1) + .1 \cdot (-2)\big) \quad = -1.55. \end{aligned}$$

Continuing for each state s, we get:

$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$T = 2$	-1.55	-2.75	-2.67	1.4	1	10	0

We then use $v^{(2)}$ to compute $v^{(3)}$, and so on, giving:

$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$T = 1$	-1	-2	-2	-2	2	10	0
$T = 2$	-1.55	-2.75	-2.67	1.4	1	10	0
$T = 3$	-1.84	-3.06	-1.53	1.2	1.47	10	0
$\vdots$							
$T = 10$	-2.08	-2.91	-1.55	1.33	1.67	10	0

and we can see that, by $T = 10$, we have converged to the solution from Example 12-3.

How do we know that the sequence of finite-horizon value functions converges to the correct (infinite horizon) value function, $\mathbf{v}^{(T)} \to \mathbf{v}$? Due to the discount factor γ, the difference $\mathbf{v} - \mathbf{v}^{(T)}$ (the terms left out of the sum up to time T) is $\gamma^T \mathbb{E}_{S_T}[v(S_T)]$. Thus, if $\mathbf{v}_s$ is finite for all states s, $\mathbf{v}^{(T)}$ converges to $\mathbf{v}$, exponentially quickly in T. Moreover, our iterative computation only needs to store the function $v(s)$, with cost $O(n)$, and our update depends only on the number of possible transitions $p(S_{t+1}|S_t)$, for total cost $O(n\,e)$ where e is the number of edges in the state transition diagram (typically far fewer than n^2).

12.3 Markov Decision Processes

Having constructed a probability model over temporal sequences and associated rewards, we now consider what effect our agent's actions might have. We can create a *Markov decision process* by including a set of actions A_t at each time t, and having our action influence both the reward R_{t+1} and our transition to the next state S_{t+1}, so that

$$r_{t+1} \sim p(R_{t+1}|S_t, A_t) \qquad\qquad s_{t+1} \sim p(S_{t+1}|S_t, A_t)$$

Thus, our actions can be used to obtain immediate rewards (eat a marshmallow now), or to move our environment toward states with potentially higher future rewards (save the marshmallow and hopefully eat two later). For discrete states we can define a reward vector and transition probability matrix for each action a,

$$[\mathbf{r}^a]_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \qquad\qquad [\mathbf{P}^a]_{s',s} = p(S_{t+1} = s|S_t = s', A_t = a)$$

Our actions at each time are determined by a *policy* π. The policy is assumed to be stationary (independent of time t), and either maps our current state S_t to an action (a deterministic policy) or draws from a distribution of actions (a stochastic policy).

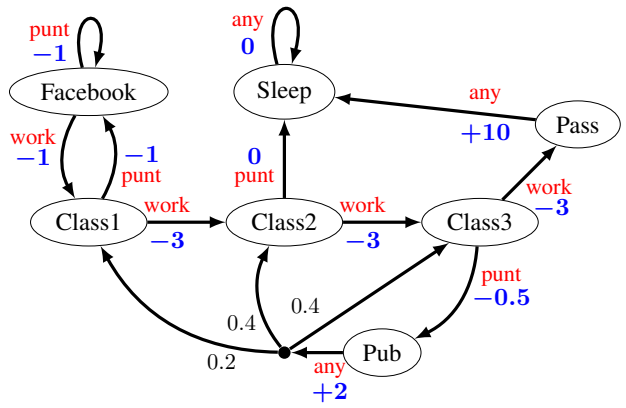
It is easy to see that a Markov decision process with fixed policy π is equivalent to a Markov reward process with

$$p(R_{t+1}|S_t) = \sum_a p(R_{t+1}|S_t, A_t = a)p_\pi(A_t = a|S_t)$$
$$p(S_{t+1}|S_t) = \sum_a p(S_{t+1}|S_t, A_t = a)p_\pi(A_t = a|S_t)$$

i.e., sampling an action a_t from policy π given state S_t induces an MRP defined by the marginal distribution over rewards and transitions. We can then also define the state value function $v_\pi(s)$ and state-action value function $q_\pi(s, a)$ for a Markov decision process with policy π to be analagous to the same functions on the induced Markov reward process.

Example 12-5 : University life MDP

We can construct a Markov decision process by modifying our Markov reward process from Example 12-2 to include actions. Specifically, for states "Facebook", "Class1", "Class2", and "Class3", we make the transition depend on a binary action, "work", or "punt" (give up). We associate a reward $\mathbf{r}_s^a$ with each state-action pair. States "Pass" and "Pub" do not depend on any action, and transition deterministically ("Pass") or stochastically ("Pub") with some reward.



$\mathbf{P}^{\text{work}}_{s',s} = p(S_t = s|S_{t-1} = s', A = \text{work})$:

	Fa	C1	C2	C3	Pu	Pa	Sl
Fa		1.0					
C1			1.0				
C2				1.0			
C3						1.0	
Pu		0.2	0.4	0.4			
Pa							1.0
Sl							1.0

$\mathbf{P}^{\text{punt}}_{s',s} = p(S_t = s|S_{t-1} = s', A = \text{punt})$:

	Fa	C1	C2	C3	Pu	Pa	Sl
Fa	1.0						
C1	1.0						
C2							1.0
C3					1.0		
Pu		0.2	0.4	0.4			
Pa							1.0
Sl							1.0

$\mathbf{r}_s^a = \mathbb{E}[R_t|S_t = s, A = a]$:

	Fa	C1	C2	C3	Pu	Pa	Sl
work	−1	−3	−3	−3	2	10	0
punt	−1	−1	0	−0.5	2	10	0

We can see that, with fixed stochastic policy π:

$$p_\pi(\text{work}|\text{Fa}) = 0.1 \qquad p_\pi(\text{work}|\text{C1}) = 0.5 \qquad p_\pi(\text{work}|\text{C2}) = 0.67 \qquad p_\pi(\text{work}|\text{C3}) = 0.6$$

our model is equivalent to the Markov reward process from Example 12-2, since for $A \sim p_\pi(A|s)$ we have that $\mathbb{E}_A[\mathbf{r}_s^A|s] = \mathbf{r}_s$ and $\mathbb{E}_A[\mathbf{P}^A_{s,s'}|s] = \mathbf{P}_{s,s'}$.

Optimal value functions and policies

How can we compare policies, and find the best one? The value function $v_\pi(s)$ tells us the expected return from following a policy π. Suppose that, for every state s, we followed the best possible policy from that state; then we can define the optimal state-value function $v_*(s)$, and optimal state-action value function $q_*(s, a)$, by the best value achievable from s:

$$v_*(s) = \max_\pi v_\pi(s) \qquad q_*(s, a) = \max_\pi q_\pi(s, a)$$

It is easy to see that v_* and q_* are directly related to one another, since

$$v_*(s) = \max_a q_*(s, a) \qquad q_*(s, a) = \mathbb{E}[R_{t+1}|s, a] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s)\, v_*(s')$$

i.e., the optimal state value $v_*(s)$ is achieved by taking the best action a and then following the best possible strategy, and the optimal state-action value $q_*(s, a)$ is given by taking the specified action a and then following the best strategy, which has value $v_*(s')$.

More generally, the value function v_π gives us a partial ordering on policies π, where we can say that a policy π dominates another, π', if

$$v_\pi(s) \geq v_{\pi'}(s) \quad \forall\, s$$

i.e., it provides a better expected return from every state s. In fact, there is at least one policy π with $v_\pi(s) = v_*(s)$ for all s, so that π dominates all other policies; one such policy is the *greedy policy* with respect to v_*:

$$\pi_*(s) = \arg\max_a q_*(s, a) = \arg\max_a \Big[\mathbb{E}[R_{t+1}|s, a] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s)\, v_*(s')\Big] \tag{12.2}$$

Any policy π with $v_\pi(s) = v_*(s)$ for all s is called an optimal policy. So, there may be multiple optimal policies for a given MDP, but all of them will have the same value function.

Just as with the Markov reward process, we can express v_* in terms of itself, since

$$v_*(s) = \max_a \Big[\mathbb{E}[R_{t+1}|s, a] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s)\, v_*(s')\Big], \tag{12.3}$$

which is called the Bellman equation for MDPs. A similar recursion defines q_*:

$$q_*(s, a) = \max_{a'} \Big[\mathbb{E}[R_{t+1}|s, a] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s)\, q_*(s', a')\Big]. \tag{12.4}$$

Now that we have defined optimality in a Markov decision process, we can see how to find an optimal policy. There are two common basic strategies for doing so: *policy iteration*, which iteratively updates a policy for the MDP, and *value iteration*, which iteratively computes the optimal value function, and then uses Eq. (12.2) to define the policy.

Policy Iteration

Policy iteration is a technique in which we start with an initial policy π, then use π to find an updated policy π' that improves upon it. The basic idea of policy iteration is to use the greedy policy from Eq. (12.2), but substitute the value function of π, v_π, for the optimal value function v_*:

$$\pi'(s) = \arg\max_a \Big[\mathbb{E}[R_{t+1}|s, a] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s)\, v_\pi(s')\Big]$$

This can be viewed as optimizing a lower bound of the best expected return, since $v_\pi(s) \leq v_*(s)$. Alternatively, we can interpret it as taking the best possible action now, knowing that from $t + 1$ onward we will be following policy π. It is easy to see that we can guarantee $v_{\pi'}(s) \geq v_\pi(s)$ for all s, since selecting action $a = \pi(s)$ would simply produce the original policy π and its associated value function; thus the maximum over a must be at least as large if not larger.

After selecting policy π', we then evaluate the value function $v_{\pi'}$ (either in closed form, or iteratively), and repeat the process to select an even better policy π''. Convergence is achieved if $v_\pi = v_{\pi'}$, in which case our policy's value function satisfies the Bellman equation (12.3) and is thus optimal.

Example 12-6 : Policy iteration

Consider the MDP in Example 12-5 with $\gamma = 0.5$, and suppose that we initialize our policy π to

$$p_\pi(\text{work}|\text{Fa}) = 0.1 \qquad p_\pi(\text{work}|\text{C1}) = 0.5 \qquad p_\pi(\text{work}|\text{C2}) = 0.67 \qquad p_\pi(\text{work}|\text{C3}) = 0.6.$$

This gives the MRP from Example 12-2, with value function

$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$v(s) =$	-2.08	-2.91	-1.55	1.33	1.67	10	0

as previously computed. Now, for each state, we compute the greedy policy according to Eq. 12.2; for example, for $s =$ Fa:

$$\pi'(\text{Fa}) = \arg\max_A \begin{cases} -1 + \gamma(-2.08) & (A = \text{punt}) \\ -1 + \gamma(-2.91) & (A = \text{work}) \end{cases} = \begin{cases} -2.04 \\ -2.45 \end{cases} \quad \Rightarrow \quad \pi'(\text{Fa}) = \text{punt}.$$

Repeating for each state s gives,

$\mathbf{r}_s^a + \gamma\,\mathbb{E}[v_\pi(S_{t+1})|S_t = s]$:

	Fa	C1	C2	C3	Pu	Pa	Sl
work	-2.45	-3.78	-2.33	2	1.66	10	0
punt	-2.04	-2.04	0	0.33			

$\Rightarrow$ $\pi'(s)$:

Fa	C1	C2	C3
punt	punt	punt	work

Clearly, there will always be some optimal policy that is deterministic, since the $\arg\max$ will be achievable by some value a. Now, we compute $v_{\pi'}$ for our new policy:

$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$v_{\pi'}(s) =$	-2	-2	0	0.6	2.2	10	0

If we repeat this procedure with the new policy π', we will find

$\mathbf{r}_s^a + \gamma\,\mathbb{E}[v_{\pi'}(S_{t+1})|S_t = s]$:

	Fa	C1	C2	C3	Pu	Pa	Sl
work	-2	-3	-2	2	2.2	10	0
punt	-2	-2	0	0.6			

$\Rightarrow$ $\pi''(s)$:

Fa	C1	C2	C3
punt	punt	punt	work

so that $v_{\pi''} = v_{\pi'}$ and thus π' is an optimal policy. Although in this case, we obtained an optimal policy in one update, in general policy iteration may require more steps to converge.

We can see that, for $s =$ Fa, both actions have equal expected return, since

$$G[\text{Fa Fa Fa}\ldots] = -1 - \frac{1}{2} - \frac{1}{4} - \ldots = -2$$
$$G[\text{Fa C1 C2 Sleep}] = -1 + \frac{1}{2}(-2) + \frac{1}{4}(0) = -2$$

which tells us that any distribution of actions at Fa is also potentially optimal. However, the value function depends on our discount factor γ: if we increase γ so that future rewards are more important, the value function will change, and we will find that "work" (so that Fa $\to$ C1) becomes the only optimal action.

Value Iteration

Another technique for finding an optimal policy π_* is to first solve for the optimal value function v_* using dynamic programming, and then create a policy that achieves this value. As with dynamic programming for computing the value function of a Markov reward process, we will define a "finite horizon" value function $v_*^{(T)}$, representing the value (expected return) of the optimal policy over at most T steps.

Then, we can compute $v_*^{(T)}$ iteratively, using $v_*^{(T-1)}$. We first note that for $T = 1$, we can only take one action, and receive one reward, before the sequence terminates. Thus $v_*^{(1)}$ is simply the maximum expected return from a single action, $v_*^{(1)}(s) = \max_a \mathbb{E}[R_{t+1}|S_t = s, A = a]$.

Now, suppose that we have T time steps left to perform, but we know the value of the best strategy to follow for the final $T-1$ steps. Then we only need to select the first action, a, and can subsequently follow our best strategy to receive return $v_*^{(T-1)}$. Thus we have,

$$v_*^{(T)}(s) = \max_a \left[\mathbb{E}[R_{t+1}|S_t = s, A = a] + \gamma \sum_{s'} p(S_{t+1} = s'|S_t = s) v_*^{(T-1)}(s')\right]$$

This exactly matches our dynamic programming solution for MRPs, except that we are now also optimizing over the immediate action a. As $T \to \infty$, the influence of terminating the sequence after T steps decreases to zero (at rate γ^T), and our value function converges to the optimal, $v_*^{(T)} \to v_*$.

Example 12-7 : Value iteration

For $T = 1$, it is easy to read out $v_*^{(1)}(s) = \max_A \mathbf{r}_s^A$:

$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$v_*^{(1)}(s)$	-1	-1	0	-0.5	2	10	0

We then use $v_*^{(1)}$ to compute $v_*^{(2)}$; for example,

$$v_*^{(2)}(\text{Fa}) = \max \begin{cases} -1 + \gamma\, v_*^{(1)}(\text{Fa}) & (A = \text{punt}) \\ -1 + \gamma\, v_*^{(1)}(\text{C1}) & (A = \text{work}) \end{cases} = \max \begin{cases} -1 + (0.5)\,(-1) \\ -1 + (0.5)\,(-1) \end{cases} = -1.5$$

$$v_*^{(2)}(\text{C1}) = \max \begin{cases} -1 + \gamma\, v_*^{(1)}(\text{Fa}) & (A = \text{punt}) \\ -3 + \gamma\, v_*^{(1)}(\text{C2}) & (A = \text{work}) \end{cases} = \max \begin{cases} -1 + (0.5)\,(-1) \\ -3 + (0.5)\,(0) \end{cases} = -1.5$$

$$\vdots$$

Computing for all the states yields,

$\Rightarrow$

$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$v_*^{(2)}(s)$	-1.5	-1.5	0	2	1.8	10	0

Continuing, we have,

$s =$	Fa	C1	C2	C3	Pu	Pa	Sl
$v_*^{(3)}(s)$	-1.75	-1.75	0	2	2.25	10	0
$v_*^{(4)}(s)$	-1.88	-1.88	0	2	2.23	10	0
$\vdots$	$\vdots$						$\vdots$
$v_*^{(10)}(s)$	-2.0	-2.0	0	2	2.2	10	0

We can read off the policy by simply computing which action is selected by the $\max$ operation, e.g.,

$$v_*^{(11)}(\text{C1}) = \max \begin{cases} -1 + (0.5)\,(-2) & (A = \text{punt}) \\ -3 + (0.5)\,(0) & (A = \text{work}) \end{cases} = \max \begin{cases} -2 \\ -3 \end{cases} \quad \Rightarrow \pi_*(\text{C1}) = \text{punt}.$$

CHAPTER

13 Reinforcement Learning

Having now studied how to identify optimal policies and their value in systems with a known model (inference), let us now turn to the *learning* task. Suppose that, instead of a concrete model, we have only a mechanism for gathering experience. These data could be pre-collected training examples, as obtained by (say) watching an expert perform a task, or they could be the result of our own agent being able to act and see the results of its actions in the world (or in a simulator).

13.1 Learning value functions

As previously, let us first consider only the *policy evaluation* component of the problem. In this setting, we imagine that a policy π has been selected (either our current one, or an experts), and our only task is to estimate the value of this policy, i.e., to evaluate $v_\pi(s)$ for all states s.

Monte Carlo Learning

The value function $v_\pi(s)$ tells us the expected return achieved by following policy π from state s. If we want to estimate this quantity, perhaps the simplest way to do so is to use an empirical expectation, i.e., to average over the return that we see in practice in this situation.

Suppose that our environment is episodic, so that (while following policy π) we observe a collection of episodes $\{x^{(i)}\}$, where

$$x^{(i)} = \left(s_1^{(i)}, a_1^{(i)}, r_2^{(i)}, s_2^{(i)}, a_2^{(i)}, \ldots, s_{T_i}^{(i)}, a_{T_i}^{(i)}, r_{T_i+1}^{(i)}\right),$$

each of which eventually terminates at some time T_i, allowing us to calculate their return. Then, for any state s, each episode i and time t for which $s_t^{(i)} = s$ gives us a trajectory – the rest of the episode – that we can use to estimate $v_\pi(s)$, based on the return $G_t^{(i)}$ from that point onward:

$$\hat{v}_\pi(s) = \frac{1}{m_s} \sum_{(i,t):s_t^{(i)}=s} G_t^{(i)} \qquad \text{where} \qquad G_t^{(i)} = \sum_{\tau=1}^{T_i-t} \gamma^{\tau-1} r_{t+\tau}^{(i)}, \qquad m_s = \#\{(i,t) : s_t^{(i)} = s\}.$$

This procedure is called the *every-visit* Monte Carlo estimator of the value function, since we use each visit to state s to provide a return for our average. We can calculate the return $G_t^{(i)}$ from each time t by running backwards through each episode, and update a running average to estimate the value function $\hat{v}(s)$; see Algorithm 13.1.

This simple Monte Carlo estimator is easy to compute, and provides an unbiased estimate of $v_\pi(s)$ for each s. However, its quality is very data dependent – if we have few episodes, or if there are states that we observe very rarely, our estimates may have high variance. As we observe more and more data, the quality of our estimator will improve.

Algorithm 13.1 Every-visit Monte Carlo

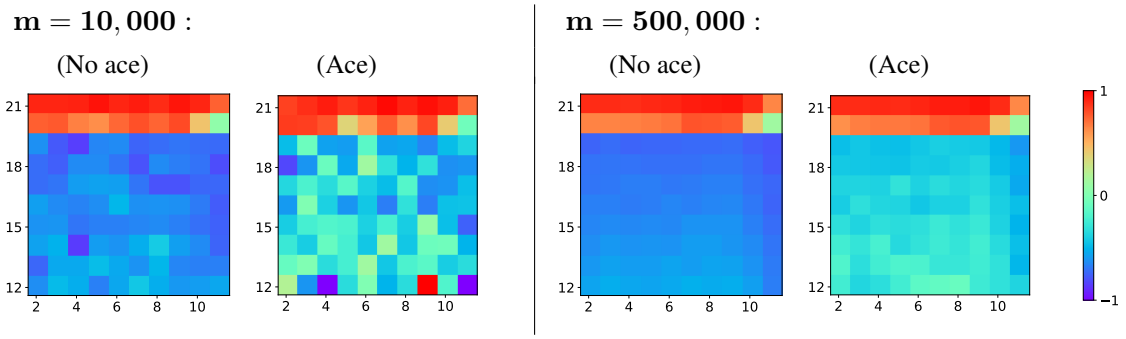
Initialize $\hat{v}(s) = 0$ and $m_s = 0$ for all s
for each episode i: **do**
  $G_{T_i+1}^{(i)} = 0$
  for each time $t \in \{T_i, \ldots, 1\}$: **do**
    $G_t^{(i)} = r_{t+1}^{(i)} + \gamma G_{t+1}^{(i)}$
    $\hat{v}(s_t^{(i)}) \leftarrow \hat{v}(s_t^{(i)}) + \frac{G_t^{(i)} - \hat{v}(s_t^{(i)})}{m_{s_t^{(i)}} + 1}$
    $m_{s_t^{(i)}} \leftarrow m_{s_t^{(i)}} + 1$
  end for
end for

Example 13-1 : Blackjack

Consider a simplified[1] version of Blackjack. We recieve two cards, and the dealer shows one card face up. We can choose to "hit" (collect another card), or "stand" (end our turn). Our goal is to get our sum of card values as close to 21 as possible without going over; aces may be counted as either 1 or 11, at the choice of the player. After our "stand" action, the dealer takes cards until their total is 17 or greater. If we exceed 21, or the dealer's hand is larger than ours and not over 21, we lose (reward -1); if our hands are the same, we tie (reward 0), and otherwise we win (reward 1).

Our state definition is fairly simple – we have our card total ($\{12, \ldots, 21\}$; below value 12 we should always take another card), an indicator of whether we have an ace which we can convert from value 11 to value 1 (we ignore the possibility of multiple aces), and the dealer's visible card value ($\{2 \ldots 11\}$), plus the extra terminal state "bust", giving a total of 201 states. Our reward comes only at the end of the game, which is always finite, so we take $\gamma = 1$.

Suppose we fix our policy π to be "hit" if our current total is less than 20, and "stand" otherwise. We can compute a Monte Carlo estimate of $v_\pi(s)$ by simulating m games, and averaging the outcome (payoff) over all games in which state s occurred. The larger the number of samples m, the better our estimate of the value function:



We can see that states that we encounter less often (such as those in which we have an ace) have more noise in the estimated value function than states that are more common; as m increases, even rare states are seen often enough to provide a good estimate of $vi_\pi(s)$.

[1] We treat "naturals" (immediate values of 21) the same as other hands, and do not allow splitting or doubling down.

Temporal-Difference Learning

Monte Carlo estimation of the value function is elegantly simple, but it does not make use of any of our understanding of how Markov processes work – it simply estimates the empirical return given the current observable state s, and must wait until the epsiode ends to evaluate the return. The goal of *temporal difference* estimators is two-fold: first, to enable us to update our estimates in a continuous (rather than episodic) manner, and second, to bring to bear the temporal structure of the Markov process, which tells us that the value function should be consistent across states, according to the Bellman equation.

The simplest temporal difference algorithm is denoted TD(0). Rather than estimating the expected return $\mathbb{E}[G_t|S_t = s]$ using the actual, empirical return from s, we can note that the Bellman equation tells us that $G_t = R_t + \mathbb{E}_{s'|s} v_\pi(s')$. If we already knew the value function $v_\pi(s')$ at other states s', we could estimate the value function at s using only the immediate return R_t. The TD(0) algorithm simply plugs in our current estimates $\hat{v}_\pi(s')$, and uses this as a target toward which to update $v_\pi(s)$, using a learning rate (step size) α. The TD(0) estimator is given in Algorithm 13.2.

Structurally, we can see that the TD(0) algorithm is very similar to the Monte Carlo estimate, except that it uses a different estimate $\hat{G}_t$ for the target return, and that the update's step size α is arbitrary, rather than a function of the number of data. A practical implication of this is that while Monte Carlo estimation needs only a single pass through the data, since the TD(0) estimate of v depends on its own past values, it can be different if we traverse the data in a different order, or use multiple passes.

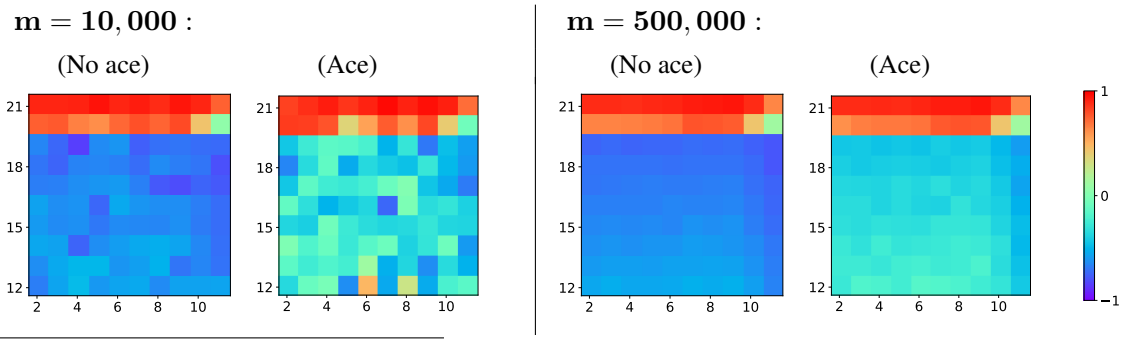
Algorithm 13.2 TD(0)

Initialize $\hat{v}(s) = 0$ for all s, step α
while not converged: **do**
  for each episode & time i, t: **do**
    $\hat{G}_t^{(i)} = r_{t+1}^{(i)} + \gamma\, \hat{v}(s_{t+1}^{(i)})$
    $\hat{v}(s_t^{(i)}) \leftarrow \hat{v}(s_t^{(i)}) + \alpha\big(\hat{G}_t^{(i)} - \hat{v}(s_t^{(i)})\big)$
  end for
end while

We can understand the usefulness of TD(0) in terms of our usual bias/variance tradeoff. The Monte Carlo estimator is unbiased, but can have high variance, especially for rare states s, since our estimate of $v(s)$ stems only from episodes in which we pass through s. In contrast, the TD(0) estimate is biased – each update uses its own current estimates of $v(s')$, and thus $v(s)$ will tend to stay close to its initial value given only a few data points[2]. However, this reduces the variance, since now when we visit a rare state s, its estimate uses the current estimate for $v(s')$, which (if s' is not rare) may have significantly more data and thus be more accurate. We can illustrate this point with an example.

Example 13-2 : Blackjack with TD(0)

We can apply TD(0) learning to estimate v_π, using the same blackjack episodes from Example 13-1. Then, our estimates will be slightly less noisy at low values of m (less variance, but more bias), with similar estimates for large m:



[2] Alternatively, we can see that our estimate of the target value is not an unbiased estimate of the correct update rule, i.e., $\mathbb{E}[r_{t+1}^{(i)} + \gamma\, \hat{v}(s^{(i)} a_{t+1})] \neq \mathbb{E}[r + \gamma v(s')]$, because our current estimate $\hat{v}(s') \neq v(s')$.

To highlight a difference between the Monte Carlo and TD(0) estimates, when we take $m = 10$k episodes, we observe only one episode in which we visit the state [Player=12 (ace), Dealer=9], which is:

$$\text{[12 (ace), 9]} \quad \Rightarrow +10 \Rightarrow \quad \text{[12 (no ace), 9]} \quad \Rightarrow +8 \Rightarrow \quad \text{[20 (no ace), 9]} \quad \Rightarrow \quad \text{win}$$

We happened to win that episode, so the Monte Carlo estimate is $v_\pi(s) = 1$. However, TD(0) observes that the episode transitioned to [12 (no ace), 9], a state from which we have quite a lot of experience and often lose; thus its estimate of $v_\pi(s)$ is much lower (and more accurate).

Constructing policies from estimated value functions

Although we have framed our discussion in terms of estimating the state-value function $v_\pi(s)$, in fact, it is often more useful in practice to work with the state-action value function, $q_\pi(s, a)$. When dealing with a known model, they are effectively interchangable; but when working with and learning an unknown model, the difference becomes important.

The key observation is that, when the model is known, either form can be easily converted into a policy using Eq. (12.2), the greedy policy with respect to the value function. However, for the state-action value form, this conversion is a trivial maximization of $q(s, a)$ over actions a, while for the state value form, it requires knowledge of the model, in the form of the expected reward $\mathbf{r}_s^a$ and transisition probabilities $\mathbf{P}_{s',s}^a$. These can be as difficult to estimate as v or q, and even difficult to represent for large numbers of states n. By estimating $q(s, a)$ instead, we can convert our estimated value function into a policy without constructing a model of the MDP itself; the effects of the reward and state transition distributions are already "folded in" to the estimate.

13.2 Model-free policy optimization

To perform full reinforcement learning, we need to not only be able to estimate the value of a policy π, but optimize over the policy as well. While this may seem like a straightforward extension of our decision-making analysis, the combination of uncertainty in our model and our observations' dependence on our policy and actions creates some difficulty.

Multi-armed bandits

Consider an extremely simple decision process, in which we make a single decision (for example, which of a number of advertisements to display on a webpage), and then receive a reward (money, if the viewer clicks on the ad). This system has very few parameters to estimate – only the expected reward, $\mathbb{E}[R|A = a]$, for each ad a. However, it is still not trivial to solve.

This type of problem is called a *multi-armed bandit* problem [**?**]; the name derives from an analogy of gambling at a casino, and trying to maximize our winnings by selecting the slot machine (or, "one-armed bandit") that has the best payoff. The issue is that each time we select an action a, we learn a bit about the payoff of that action, but nothing about any other actions. Thus there is a tension between wanting to select the best action a given the information so far, versus selecting an action that will tell us more about the payoff of an action we have not tried as much. If we play a slot machine only a few times and lose, we cannot conclude that its payoff rate is low; we may have just been unlucky. This fundamental balance is sometimes called the *exploration-exploitation* tradeoff – we would like to exploit the current best action (to increase our winnings as much as possible), but we must balance it with exploring, in case another action is actually better in the long run.

Regret definition?

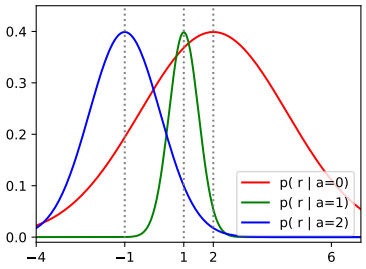
ϵ-Greedy Policy

There are many popular algorithms for multi-armed bandit problems, but let us consider only a very simple one for now. The ϵ-greedy exploration algorithm selects the action currently estimated to have the highest reward with probability $(1-\epsilon)$, and selects an action at random (say, uniformly) with probability ϵ. Although trivial, this approach guarantees that every action will eventually be explored infinitely often.

Example 13-3 : Three-Armed Bandit

Consider a simple multi-armed bandit with three possible actions, where each action yields a Gaussian-distributed reward:

$$p(r|a=0) = \mathcal{N}(r\,;\,2\,,\,2.5^2)$$
$$p(r|a=1) = \mathcal{N}(r\,;\,1\,,\,0.5^2)$$
$$p(r|a=2) = \mathcal{N}(r\,;\,-1\,,\,1.2^2)$$

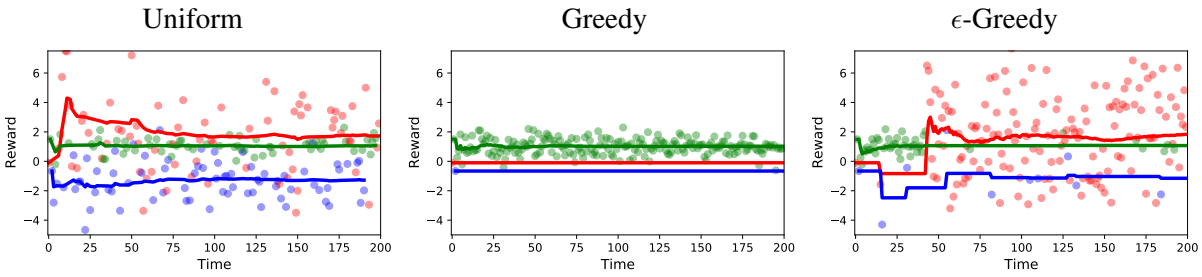


(illustrated at right). Then, the best action, on average, is $a = 0$ (red), with expected reward 0.2; next is $a = 1$ (green), and worst is $a = 2$ (blue). However, we also see that $a = 0$ (red) is considerably more random, i.e., has higher variance, than the other two actions.

If we try following a simple, fixed strategy – say, picking our actions randomly and with equal probability – we will improve our estimates of the expected reward for each action over time, but will not benefit from these estimates; we will obtain average reward $\frac{1}{3}(2+1-1) = \frac{2}{3}$.

On the other hand, if we behave in a *greedy* fashion and pick the action which has the highest current estimate of its expected reward, we can get unlucky – if we see a poor return for $a = 0$, we may never try it again, and only ever select $a = 1$, for average reward 1. By following an ϵ-greedy policy, though we may initially select $a = 1$, we will always eventually try $a = 0$ and $a = 2$ again, so that given sufficient time, our estimates improve and our greedy action converges to the optimal, $a = 0$, and asymptotic reward $2 \cdot (1-\epsilon) + \frac{2}{3} \cdot \epsilon$.

If we look at the actions selected over time, and the estimated reward of each action given the data so far, we can see these behaviors illustrated:



Upper confidence bounds

How do we decide how much evidence do we need before we decide that a given action is really better than the others? One principle is to try to reason concretely about the degree of uncertainty in our esti-

mates; this enables us to quantify whether our number of observations is sufficient to support a particular decision.

A simple instantiation of this principle are upper confidence bound methods. Suppose that we have interacted with our multi-armed bandit t times so far, and that at this point, we have taken each action a some number of times, $m_a^{(t)}$. If $m_a^{(t)}$ is large, our empirical rewards for that action should give a pretty accurate estimate of the true expected reward $\mathbb{E}[r|a]$, while if $m_a^{(t)}$ is small, our estimate may not be accurate, and we may want to collect more data in case the true expected reward is higher.

We can quantify this behavior by computing an upper confidence bound on the true reward; for example, we find a value U_a such that with 95% confidence, the true reward $\mathbb{E}[r|a] \leq U_a$. Then, when we have not seen an action very often, its bound may be much higher than its current average; as we take the action more and more often, our confidence interval tightens and we begin to only select the action if it appears better than the others. If we know that the reward is sub-Gaussian (i.e., it has thin probability tails), we can apply the bound,

$$U_a^{(t)} = \hat{\mu}_a^{(t)} + \sqrt{2\log(\delta^{-1})/m_a^{(t)}}$$

where δ is the probability of exceeding the bound, i.e., $\delta = 0.05$ for a 95% confidence bound. Selecting the action a with the highest "optimistic estimate" (upper bound) $U_a^{(t)}$ will prefer actions with good historical returns (high $\hat{\mu}_a$), or actions with few attempts so far (small m_a).

Notation; subGauss add rate σ^2; related to c later

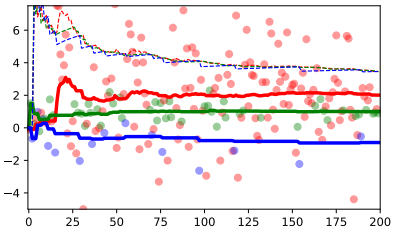
As we are able to play the bandit more often (i.e., as t increases), we may want to adjust δ. The standard UCB method [**?**] selects $\delta = 1/t$, which ensures that each action will be selected infinitely often, and leads to relatively strong bounds on the possible regret of the procedure. In this case, the quantity $U_a^{(t)}$ can be expressed as,

$$U_a^{(t)} = \hat{\mu}_a^{(t)} + c\sqrt{\log(t)/m_a^{(t)}}$$

where c is a constant scaling coefficient that influences how much exploration will be performed.

Example 13-4 : Three-armed UCB

For our simple three-armed bandit, we can compute the upper confidence bounds, taking $\delta = 1/t$. As we perform each action, we get more information about whichever action we took, and our confidence bound (dashed line) for that action decreases. This means that we select actions that appear good (such as red) more often, while actions with lower average reward (such as blue) are selected only often enough to verify that they remain unlikely to be the best action.



Thompson sampling

todo

- easiest to understand from Bayesian perspective

- Start with a model of rewards given unknown parameters (e.g., mean and variance of reward), and a prior over the parameters
- Sample a reward for each action from the model, and choose the action a_t with highest reward
- Execute that action and observe its actual reward R_t
- Replace the prior with the inew posterior, $p(\theta|R_t) \propto p(\theta)p(R|\theta, a_t)$
- more stochastic than UCB; takes an action a with probability equal to $Pr[R_a \geq R_{a'} \; \forall a']$

Algorithm 13.3 Thompson sampling

Initialize $p(\theta_a)$ for all a
for each time t: **do**
  Sample $\tilde{R}_a \sim p(R|\theta_a)p(\theta_a)$ for each a
  Select action $a_t = \arg\max_a \tilde{R}_a$ and observe reward R_t
  Update $p(\theta_{a_t}) \leftarrow p(\theta_{a_t}|R_t)$
end for

SARSA

A natural idea for more general reinforcement learning, then, is to apply an ϵ-greedy policy with temporal difference estimation of the state-action value function. Specifically, we can express the state-action value function, $q_\pi(s, a)$, as

$$q_\pi(s, a) = \mathbb{E}[R_{t+1}|S_t = s] + \gamma \mathbb{E}_{S_{t+1}, A_{t+1}}\left[q_\pi(S_{t+1}, A_{t+1})\right]$$

As we proceed in an online manner according to policy π, at time $t+1$ we can use the snippet, $[s_t^{(i)}, a_t^{(i)}, r_{t+1}^{(i)}, s_{t+1}^{(i)}, a_{t+1}^{(i)}]$ as a stochastic estimate of the expectations given $S_t = s_t$ and $A_t = a_t$, to update $q(s_t, a_t)$. We can then update π to follow an ϵ-greedy policy for our current estimated $q(s, a)$. (The form of the snippet gives rise to the name, "SARSA", or "state, action, reward, state, action"). Taking a learning rate $\alpha \in [0, 1]$, our update takes the form,

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha\Big(\underbrace{r_{t+1} + \gamma q(s_{t+1}, a_{t+1})}_{\text{(Estimated return)}} - \underbrace{q(s_t, a_t)}_{\text{(Old value)}}\Big)$$

i.e., we update our estimated state-action value to move slightly toward the estimate given by the observed reward and state transition, followed by our estimate of the (discounted) value of our observed next state and selected action.

This algorithm gives a simple mechanism for *on-policy* learning, i.e., simultaneously optimizing and learning the value function of a policy that we are currently following. In other words, our policy is "learning by doing". However, on-policy learning can be inefficient – we are relying on our policy trying good actions often enough to discover their value. *Off-policy* learning can allow us to gain experience by watching other agents' behavior – including humans or other already-trained systems. It also allows us to reuse our own experience, in the form of episodes generated by following old policies. If we initially follow a more exploratory policy, we can still use these episodes to teach ourselves about our current, more optimal policy.

Q-Learning

- Can be used for off-policy learning
- Directly estimate state-action value function $q(s, a)$

- Stochastic update, similar to SGD

$$q(s_t^{(i)}, a_t^{(i)}) \leftarrow q(s_t^{(i)}, a_t^{(i)}) + \alpha \left(\underbrace{r_{t+1}^{(i)} + q(s_{t+1}^{(i)}, a')}_{\text{(Estimated return)}} - \underbrace{q(s_t^{(i)}, a_t^{(i)})}_{\text{(Old value)}} \right)$$

  where

$$a' \sim \pi(s_t^{(i)})$$

  is an action selected via our policy.

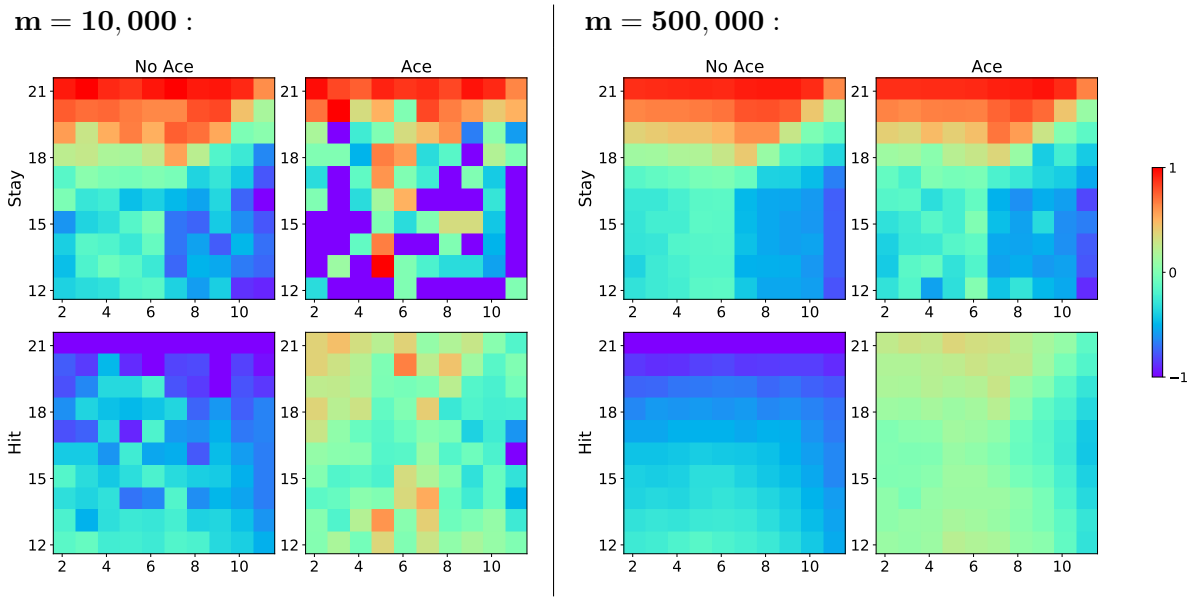
- Use observed action, reward, and state transition, but insert our policy for last action (est'd value of new state)
- Demo policy tells us what will happen from s when we do a, but we should come to our own conclusions about the value of where we end up (maybe they followed the wrong policy after that)
- Simplifies if we use e.g. greedy for our policy, epsilon greedy for demo policy

Example 13-5 : Blackjack Q-Learning

We can learn a policy for our blackjack simulation using q-learning updates and any policy; here we will use the simple epsilon-greedy policy for training. For $q(s, a)$, we have two actions, $a \in \{\text{stay}, \text{hit}\}$, and as before, our states s consist of the player's card total, the dealer's showing card, and whether the player has an ace that can be converted from value 11 to 1.

details

After 10k episodes, our $q(\cdot)$ estimate is fairly rough; but after a longer period of play, it begins to converge:



We can still see some noise from incomplete convergence (for example, the value of action "stay" should the same whether we have an ace or not), but the value functions are clearly close to convergence.