Contents

Summary of Notation

Capital letters are used for random variables, whereas lower case letters are used for the values of random variables and for scalar functions. Quantities that are required to be real-valued vectors are written in bold and in lower case (even if random variables). Matrices are bold capitals.

$\doteq$	equality relationship that is true by definition
$\approx$	approximately equal
$\propto$	proportional to
$\Pr\{X=x\}$	probability that a random variable X takes on the value x
$X \sim p$	random variable X selected from distribution $p(x) \doteq \Pr\{X=x\}$
$\mathbb{E}[X]$	expectation of a random variable X, i.e., $\mathbb{E}[X] \doteq \sum_x p(x)x$
$\arg\max_a f(a)$	a value of a at which $f(a)$ takes its maximal value
$\ln x$	natural logarithm of x
e^x	the base of the natural logarithm, $e \approx 2.71828$, carried to power x; $e^{\ln x} =$
$\mathbb{R}$	set of real numbers
$f : \mathcal{X} \to \mathcal{Y}$	function f from elements of set $\mathcal{X}$ to elements of set $\mathcal{Y}$
$\leftarrow$	assignment
$(a, b]$	the real interval between a and b including b but not including a
ε	probability of taking a random action in an ε-greedy policy
α, β	step-size parameters
γ	discount-rate parameter
λ	decay-rate parameter for eligibility traces
$\mathbb{1}_{predicate}$	indicator function ($\mathbb{1}_{predicate} \doteq 1$ if the *predicate* is true, else 0)

In a multi-arm bandit problem:

k	number of actions (arms)
t	discrete time step or play number
$q_*(a)$	true value (expected reward) of action a
$Q_t(a)$	estimate at time t of $q_*(a)$
$N_t(a)$	number of times action a has been selected up prior to time t
$H_t(a)$	learned preference for selecting action a at time t
$\pi_t(a)$	probability of selecting action a at time t
$\bar{R}_t$	estimate at time t of the expected reward given π_t

In a Markov Decision Process:

s, s'	states
a	an action
r	a reward
$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{A}(s)$	set of all actions available in state s
$\mathcal{R}$	set of all possible rewards, a finite subset of $\mathbb{R}$
$\subset$	subset of; e.g., $\mathcal{R} \subset \mathbb{R}$
$\in$	is an element of; e.g., $s \in \mathcal{S}$, $r \in \mathcal{R}$
$\lvert\mathcal{S}\rvert$	number of elements in set $\mathcal{S}$
t	discrete time step
$T, T(t)$	final time step of an episode, or of the episode including time step t
A_t	action at time t
S_t	state at time t, typically due, stochastically, to S_{t-1} and A_{t-1}
R_t	reward at time t, typically due, stochastically, to S_{t-1} and A_{t-1}
π	policy (decision-making rule)
$\pi(s)$	action taken in state s under *deterministic* policy π
$\pi(a\lvert s)$	probability of taking action a in state s under *stochastic* policy π
G_t	return following time t
h	horizon, the time step one looks up to in a forward view
$G_{t:t+n}, G_{t:h}$	n-step return from $t+1$ to $t+n$, or to h (discounted and corrected)
$\bar{G}_{t:h}$	flat return (undiscounted and uncorrected) from $t+1$ to h (Section 5.8)
G_t^λ	λ-return (Section 12.1)
$G_{t:h}^\lambda$	truncated, corrected λ-return (Section 12.3)
$G_t^{\lambda s}, G_t^{\lambda a}$	λ-return, corrected by estimated state, or action, values (Section 12.8)
$p(s', r \mid s, a)$	probability of transition to state s' with reward r, from state s and actior
$p(s' \mid s, a)$	probability of transition to state s', from state s taking action a
$r(s, a)$	expected immediate reward from state s after action a
$r(s, a, s')$	expected immediate reward on transition from s to s' under action a
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy
$q_\pi(s, a)$	value of taking action a in state s under policy π
$q_*(s, a)$	value of taking action a in state s under the optimal policy
V, V_t	array estimates of state-value function v_π or v_*
Q, Q_t	array estimates of action-value function q_π or q_*
$\bar{V}_t(s)$	expected approximate action value, e.g., $\bar{V}_t(s) \doteq \sum_a \pi(a\lvert s)Q_t(s, a)$
U_t	target for estimate at time t

δ_t	temporal-difference (TD) error at t (a random variable) (Section 6.1)
δ_t^s, δ_t^a	state- and action-specific forms of the TD error (Section 12.9)
n	in n-step methods, n is the number of steps of bootstrapping
d	dimensionality—the number of components of $\mathbf{w}$
d'	alternate dimensionality—the number of components of $\boldsymbol{\theta}$
$\mathbf{w}, \mathbf{w}_t$	d-vector of weights underlying an approximate value function
$w_i, w_{t,i}$	ith component of learnable weight vector
$\hat{v}(s,\mathbf{w})$	approximate value of state s given weight vector $\mathbf{w}$
$v_{\mathbf{w}}(s)$	alternate notation for $\hat{v}(s,\mathbf{w})$
$\hat{q}(s,a,\mathbf{w})$	approximate value of state–action pair s, a given weight vector $\mathbf{w}$
$\nabla\hat{v}(s,\mathbf{w})$	column vector of partial derivatives of $\hat{v}(s,\mathbf{w})$ with respect to $\mathbf{w}$
$\nabla\hat{q}(s,a,\mathbf{w})$	column vector of partial derivatives of $\hat{q}(s,a,\mathbf{w})$ with respect to $\mathbf{w}$
$\mathbf{x}(s)$	vector of features visible when in state s
$\mathbf{x}(s,a)$	vector of features visible when in state s taking action a
$x_i(s), x_i(s,a)$	ith component of vector $\mathbf{x}(s)$ or $\mathbf{x}(s,a)$
$\mathbf{x}_t$	shorthand for $\mathbf{x}(S_t)$ or $\mathbf{x}(S_t,A_t)$
$\mathbf{w}^\top\mathbf{x}$	inner product of vectors, $\mathbf{w}^\top\mathbf{x} \doteq \sum_i w_i x_i$; e.g., $\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top\mathbf{x}(s)$
$\mathbf{v}, \mathbf{v}_t$	secondary d-vector of weights, used to learn $\mathbf{w}$ (Chapter 11)
$\mathbf{z}_t$	d-vector of eligibility traces at time t (Chapter 12)
$\boldsymbol{\theta}, \boldsymbol{\theta}_t$	parameter vector of target policy (Chapter 13)
$\pi(a\|s,\boldsymbol{\theta})$	probability of taking action a in state s given parameter vector $\boldsymbol{\theta}$
$\pi_{\boldsymbol{\theta}}$	policy corresponding to parameter $\boldsymbol{\theta}$
$\nabla\pi(a\|s,\boldsymbol{\theta})$	column vector of partial derivatives of $\pi(a\|s,\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$
$J(\boldsymbol{\theta})$	performance measure for the policy $\pi_{\boldsymbol{\theta}}$
$\nabla J(\boldsymbol{\theta})$	column vector of partial derivatives of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$
$h(s,a,\boldsymbol{\theta})$	preference for selecting action a in state s based on $\boldsymbol{\theta}$
$b(a\|s)$	behavior policy used to select actions while learning about target policy π
$b(s)$	a baseline function $b : \mathcal{S} \mapsto \mathbb{R}$ for policy-gradient methods
b	branching factor for an MDP or search tree
$\rho_{t:h}$	importance sampling ratio for time t through time h (Section 5.5)
ρ_t	importance sampling ratio for time t alone, $\rho_t \doteq \rho_{t:t}$
$r(\pi)$	average reward (reward rate) for policy π (Section 10.3)
$\bar{R}_t$	estimate of $r(\pi)$ at time t
$\mu(s)$	on-policy distribution over states (Section 9.2)
$\boldsymbol{\mu}$	$\|\mathcal{S}\|$-vector of the $\mu(s)$ for all $s \in \mathcal{S}$
$\|v\|_\mu^2$	μ-weighted squared norm of value function v, i.e., $\|v\|_\mu^2 \doteq \sum_{s\in\mathcal{S}} \mu(s)v(s)^2$
$\eta(s)$	expected number of visits to state s per episode (page 199)
Π	projection operator for value functions (page 268)
B_π	Bellman operator for value functions (Section 11.4)

$\mathbf{A}$	$d \times d$ matrix $\mathbf{A} \doteq \mathbb{E}\Big[\mathbf{x}_t\big(\mathbf{x}_t - \gamma\mathbf{x}_{t+1}\big)^\top\Big]$
$\mathbf{b}$	d-dimensional vector $\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]$
$\mathbf{w}_{\text{TD}}$	TD fixed point $\mathbf{w}_{\text{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}$ (a d-vector, Section 9.4)
$\mathbf{I}$	identity matrix
$\mathbf{P}$	$\|\mathcal{S}\| \times \|\mathcal{S}\|$ matrix of state-transition probabilities under π
$\mathbf{D}$	$\|\mathcal{S}\| \times \|\mathcal{S}\|$ diagonal matrix with $\boldsymbol{\mu}$ on its diagonal
$\mathbf{X}$	$\|\mathcal{S}\| \times d$ matrix with the $\mathbf{x}(s)$ as its rows
$\bar{\delta}_{\mathbf{w}}(s)$	Bellman error (expected TD error) for $v_{\mathbf{w}}$ at state s (Section 11.4)
$\bar{\boldsymbol{\delta}}_{\mathbf{w}}$, BE	Bellman error vector, with components $\bar{\delta}_{\mathbf{w}}(s)$
$\overline{\text{VE}}(\mathbf{w})$	mean square value error $\overline{\text{VE}}(\mathbf{w}) \doteq \|v_{\mathbf{w}} - v_\pi\|_\mu^2$ (Section 9.2)
$\overline{\text{BE}}(\mathbf{w})$	mean square Bellman error $\overline{\text{BE}}(\mathbf{w}) \doteq \big\|\bar{\delta}_{\mathbf{w}}\big\|_\mu^2$
$\overline{\text{PBE}}(\mathbf{w})$	mean square projected Bellman error $\overline{\text{PBE}}(\mathbf{w}) \doteq \big\|\Pi\bar{\delta}_{\mathbf{w}}\big\|_\mu^2$
$\overline{\text{TDE}}(\mathbf{w})$	mean square temporal-difference error $\overline{\text{TDE}}(\mathbf{w}) \doteq \mathbb{E}_b\big[\rho_t\delta_t^2\big]$ (Section 11.5)
$\overline{\text{RE}}(\mathbf{w})$	mean square return error (Section 11.6)

Chapter 1

Introduction

The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

In this book we explore a *computational* approach to learning from interaction. Rather than directly theorizing about how people or animals learn, we primarily explore idealized learning situations and evaluate the effectiveness of various learning methods.[1] That is, we adopt the perspective of an artificial intelligence researcher or engineer. We explore designs for machines that are effective in solving learning problems of scientific or economic interest, evaluating the designs through mathematical analysis or computational experiments. The approach we explore, called *reinforcement learning*, is much more focused on goal-directed learning from interaction than are other approaches to machine learning.

1.1 Reinforcement Learning

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate

[1] The relationships to psychology and neuroscience are summarized in Chapters 14 and 15.

reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

Reinforcement learning, like many topics whose names end with "ing," such as machine learning and mountaineering, is simultaneously a problem, a class of solution methods that work well on the problem, and the field that studies this problem and its solution methods. It is convenient to use a single name for all three things, but at the same time essential to keep the three conceptually separate. In particular, the distinction between problems and solution methods is very important in reinforcement learning; failing to make this distinction is the source of many confusions.

We formalize the problem of reinforcement learning using ideas from dynamical systems theory, specifically, as the optimal control of incompletely-known Markov decision processes. The details of this formalization must wait until Chapter 3, but the basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting over time with its environment to achieve a goal. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. Markov decision processes are intended to include just these three aspects—sensation, action, and goal—in their simplest possible forms without trivializing any of them. Any method that is well suited to solving such problems we consider to be a reinforcement learning method.

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

Reinforcement learning is also different from what machine learning researchers call *unsupervised learning*, which is typically about finding structure hidden in collections of unlabeled data. The terms supervised learning and unsupervised learning would seem to exhaustively classify machine learning paradigms, but they do not. Although one might be tempted to think of reinforcement learning as a kind of unsupervised learning because it does not rely on examples of correct behavior, reinforcement learning is trying to maximize a reward signal instead of trying to find hidden structure. Uncovering structure in an agent's experience can certainly be useful in reinforcement learning, but by itself does not address the reinforcement learning problem of maximizing a reward signal. We therefore consider reinforcement learning to be a third machine learning paradigm, alongside supervised learning and unsupervised learning and perhaps other paradigms.

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to *exploit* what it has already experienced in order to obtain reward, but it also has to *explore* in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions *and* progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate of its expected reward. The exploration–exploitation dilemma has been intensively studied by mathematicians for many decades, yet remains unresolved. For now, we simply note that the entire issue of balancing exploration and exploitation does not even arise in supervised and unsupervised learning, at least in the purest forms of these paradigms.

Another key feature of reinforcement learning is that it explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment. This is in contrast to many approaches that consider subproblems without addressing how they might fit into a larger picture. For example, we have mentioned that much of machine learning research is concerned with supervised learning without explicitly specifying how such an ability would finally be useful. Other researchers have developed theories of planning with general goals, but without considering planning's role in real-time decision making, or the question of where the predictive models necessary for planning would come from. Although these approaches have yielded many useful results, their focus on isolated subproblems is a significant limitation.

Reinforcement learning takes the opposite tack, starting with a complete, interactive, goal-seeking agent. All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. Moreover, it is usually assumed from the beginning that the agent has to operate despite significant uncertainty about the environment it faces. When reinforcement learning involves planning, it has to address the interplay between planning and real-time action selection, as well as the question of how environment models are acquired and improved. When reinforcement learning involves supervised learning, it does so for specific reasons that determine which capabilities are critical and which are not. For learning research to make progress, important subproblems have to be isolated and studied, but they should be subproblems that play clear roles in complete, interactive, goal-seeking agents, even if all the details of the complete agent cannot yet be filled in.

By a complete, interactive, goal-seeking agent we do not always mean something like a complete organism or robot. These are clearly examples, but a complete, interactive, goal-seeking agent can also be a component of a larger behaving system. In this case, the agent directly interacts with the rest of the larger system and indirectly interacts with the larger system's environment. A simple example is an agent that monitors the charge level of robot's battery and sends commands to the robot's control architecture. This agent's environment is the rest of the robot together with the robot's environment. One must look beyond the most obvious examples of agents and their environments to

appreciate the generality of the reinforcement learning framework.

One of the most exciting aspects of modern reinforcement learning is its substantive and fruitful interactions with other engineering and scientific disciplines. Reinforcement learning is part of a decades-long trend within artificial intelligence and machine learning toward greater integration with statistics, optimization, and other mathematical subjects. For example, the ability of some reinforcement learning methods to learn with parameterized approximators addresses the classical "curse of dimensionality" in operations research and control theory. More distinctively, reinforcement learning has also interacted strongly with psychology and neuroscience, with substantial benefits going both ways. Of all the forms of machine learning, reinforcement learning is the closest to the kind of learning that humans and other animals do, and many of the core algorithms of reinforcement learning were originally inspired by biological learning systems. Reinforcement learning has also given back, both through a psychological model of animal learning that better matches some of the empirical data, and through an influential model of parts of the brain's reward system. The body of this book develops the ideas of reinforcement learning that pertain to engineering and artificial intelligence, with connections to psychology and neuroscience summarized in Chapters 14 and 15.

Finally, reinforcement learning is also part of a larger trend in artificial intelligence back toward simple general principles. Since the late 1960's, many artificial intelligence researchers presumed that there are no general principles to be discovered, that intelligence is instead due to the possession of a vast number of special purpose tricks, procedures, and heuristics. It was sometimes said that if we could just get enough relevant facts into a machine, say one million, or one billion, then it would become intelligent. Methods based on general principles, such as search or learning, were characterized as "weak methods," whereas those based on specific knowledge were called "strong methods." This view is still common today, but not dominant. From our point of view, it was simply premature: too little effort had been put into the search for general principles to conclude that there were none. Modern artificial intelligence now includes much research looking for general principles of learning, search, and decision making. It is not clear how far back the pendulum will swing, but reinforcement learning research is certainly part of the swing back toward simpler and fewer general principles of artificial intelligence.

1.2 Examples

A good way to understand reinforcement learning is to consider some of the examples and possible applications that have guided its development.

- A master chess player makes a move. The choice is informed both by planning—anticipating possible replies and counterreplies—and by immediate, intuitive judgments of the desirability of particular positions and moves.
- An adaptive controller adjusts parameters of a petroleum refinery's operation in real time. The controller optimizes the yield/cost/quality trade-off on the basis of specified marginal costs without sticking strictly to the set points originally suggested by engineers.

- A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 20 miles per hour.

- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station. It makes its decision based on the current charge level of its battery and how quickly and easily it has been able to find the recharger in the past.

- Phil prepares his breakfast. Closely examined, even this apparently mundane activity reveals a complex web of conditional behavior and interlocking goal–subgoal relationships: walking to the cupboard, opening it, selecting a cereal box, then reaching for, grasping, and retrieving the box. Other complex, tuned, interactive sequences of behavior are required to obtain a bowl, spoon, and milk carton. Each step involves a series of eye movements to obtain information and to guide reaching and locomotion. Rapid judgments are continually made about how to carry the objects or whether it is better to ferry some of them to the dining table before obtaining others. Each step is guided by goals, such as grasping a spoon or getting to the refrigerator, and is in service of other goals, such as having the spoon to eat with once the cereal is prepared and ultimately obtaining nourishment. Whether he is aware of it or not, Phil is accessing information about the state of his body that determines his nutritional needs, level of hunger, and food preferences.

These examples share features that are so basic that they are easy to overlook. All involve *interaction* between an active decision-making agent and its environment, within which the agent seeks to achieve a *goal* despite *uncertainty* about its environment. The agent's actions are permitted to affect the future state of the environment (e.g., the next chess position, the level of reservoirs of the refinery, the robot's next location and the future charge level of its battery), thereby affecting the actions and opportunities available to the agent at later times. Correct choice requires taking into account indirect, delayed consequences of actions, and thus may require foresight or planning.

At the same time, in all of these examples the effects of actions cannot be fully predicted; thus the agent must monitor its environment frequently and react appropriately. For example, Phil must watch the milk he pours into his cereal bowl to keep it from overflowing. All these examples involve goals that are explicit in the sense that the agent can judge progress toward its goal based on what it can sense directly. The chess player knows whether or not he wins, the refinery controller knows how much petroleum is being produced, the gazelle calf knows when it falls, the mobile robot knows when its batteries run down, and Phil knows whether or not he is enjoying his breakfast.

In all of these examples the agent can use its experience to improve its performance over time. The chess player refines the intuition he uses to evaluate positions, thereby improving his play; the gazelle calf improves the efficiency with which it can run; Phil learns to streamline making his breakfast. The knowledge the agent brings to the task at the start—either from previous experience with related tasks or built into it by design or evolution—influences what is useful or easy to learn, but interaction with the environment is essential for adjusting behavior to exploit specific features of the task.

1.3 Elements of Reinforcement Learning

Beyond the agent and the environment, one can identify four main subelements of a reinforcement learning system: a *policy*, a *reward signal*, a *value function*, and, optionally, a *model* of the environment.

A *policy* defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. It corresponds to what in psychology would be called a set of stimulus–response rules or associations. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic, specifying probabilities for each action.

A *reward signal* defines the goal of a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the *reward*. The agent's sole objective is to maximize the total reward it receives over the long run. The reward signal thus defines what are the good and bad events for the agent. In a biological system, we might think of rewards as analogous to the experiences of pleasure or pain. They are the immediate and defining features of the problem faced by the agent. The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. In general, reward signals may be stochastic functions of the state of the environment and the actions taken.

Whereas the reward signal indicates what is good in an immediate sense, a *value function* specifies what is good in the long run. Roughly speaking, the *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the *long-term* desirability of states after taking into account the states that are likely to follow and the rewards available in those states. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards. Or the reverse could be true. To make a human analogy, rewards are somewhat like pleasure (if high) and pain (if low), whereas values correspond to a more refined and farsighted judgment of how pleased or displeased we are that our environment is in a particular state.

Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no values, and the only purpose of estimating values is to achieve more reward. Nevertheless, it is values with which we are most concerned when making and evaluating decisions. Action choices are made based on value judgments. We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run. Unfortunately, it is much harder to determine values than it is to determine rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime. In fact, the most important component of almost all reinforcement learning algorithms we consider is a

method for efficiently estimating values. The central role of value estimation is arguably the most important thing that has been learned about reinforcement learning over the last six decades.

The fourth and final element of some reinforcement learning systems is a *model* of the environment. This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for *planning*, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called *model-based* methods, as opposed to simpler *model-free* methods that are explicitly trial-and-error learners—viewed as almost the *opposite* of planning. In Chapter 8 we explore reinforcement learning systems that simultaneously learn by trial and error, learn a model of the environment, and use the model for planning. Modern reinforcement learning spans the spectrum from low-level, trial-and-error learning to high-level, deliberative planning.

1.4 Limitations and Scope

Reinforcement learning relies heavily on the concept of state—as input to the policy and value function, and as both input to and output from the model. Informally, we can think of the state as a signal conveying to the agent some sense of "how the environment is" at a particular time. The formal definition of state as we use it here is given by the framework of Markov decision processes presented in Chapter 3. More generally, however, we encourage the reader to follow the informal meaning and think of the state as whatever information is available to the agent about its environment. In effect, we assume that the state signal is produced by some preprocessing system that is nominally part of the agent's environment. We do not address the issues of constructing, changing, or learning the state signal in this book (other than briefly in Section 17.3). We take this approach not because we consider state representation to be unimportant, but in order to focus fully on the decision-making issues. In other words, our concern in this book is not with designing the state signal, but with deciding what action to take as a function of whatever state signal is available.

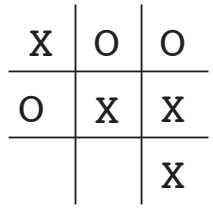
Most of the reinforcement learning methods we consider in this book are structured around estimating value functions, but it is not strictly necessary to do this to solve reinforcement learning problems. For example, solution methods such as genetic algorithms, genetic programming, simulated annealing, and other optimization methods never estimate value functions. These methods apply multiple static policies each interacting over an extended period of time with a separate instance of the environment. The policies that obtain the most reward, and random variations of them, are carried over to the next generation of policies, and the process repeats. We call these *evolutionary* methods because their operation is analogous to the way biological evolution produces organisms with skilled behavior even if they do not learn during their individual lifetimes. If the space of policies is sufficiently small, or can be structured so that good policies are

common or easy to find—or if a lot of time is available for the search—then evolutionary methods can be effective. In addition, evolutionary methods have advantages on problems in which the learning agent cannot sense the complete state of its environment.

Our focus is on reinforcement learning methods that learn while interacting with the environment, which evolutionary methods do not do. Methods able to take advantage of the details of individual behavioral interactions can be much more efficient than evolutionary methods in many cases. Evolutionary methods ignore much of the useful structure of the reinforcement learning problem: they do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects. In some cases this information can be misleading (e.g., when states are misperceived), but more often it should enable more efficient search. Although evolution and learning share many features and naturally work together, we do not consider evolutionary methods by themselves to be especially well suited to reinforcement learning problems and, accordingly, we do not cover them in this book.

1.5 An Extended Example: Tic-Tac-Toe

To illustrate the general idea of reinforcement learning and contrast it with other approaches, we next consider a single example in more detail.



Consider the familiar child's game of tic-tac-toe. Two players take turns playing on a three-by-three board. One player plays Xs and the other Os until one player wins by placing three marks in a row, horizontally, vertically, or diagonally, as the X player has in the game shown to the right. If the board fills up with neither player getting three in a row, then the game is a draw. Because a skilled player can play so as never to lose, let us assume that we are playing against an imperfect player, one whose play is sometimes incorrect and allows us to win. For the moment, in fact, let us consider draws and losses to be equally bad for us. How might we construct a player that will find the imperfections in its opponent's play and learn to maximize its chances of winning?

Although this is a simple problem, it cannot readily be solved in a satisfactory way through classical techniques. For example, the classical "minimax" solution from game theory is not correct here because it assumes a particular way of playing by the opponent. For example, a minimax player would never reach a game state from which it could lose, even if in fact it always won from that state because of incorrect play by the opponent. Classical optimization methods for sequential decision problems, such as dynamic programming, can *compute* an optimal solution for any opponent, but require as input a complete specification of that opponent, including the probabilities with which the opponent makes each move in each board state. Let us assume that this information is not available a priori for this problem, as it is not for the vast majority of problems of practical interest. On the other hand, such information can be estimated from experience, in this case by playing many games against the opponent. About the best one can do

on this problem is first to learn a model of the opponent's behavior, up to some level of confidence, and then apply dynamic programming to compute an optimal solution given the approximate opponent model. In the end, this is not that different from some of the reinforcement learning methods we examine later in this book.

An evolutionary method applied to this problem would directly search the space of possible policies for one with a high probability of winning against the opponent. Here, a policy is a rule that tells the player what move to make for every state of the game—every possible configuration of Xs and Os on the three-by-three board. For each policy considered, an estimate of its winning probability would be obtained by playing some number of games against the opponent. This evaluation would then direct which policy or policies were considered next. A typical evolutionary method would hill-climb in policy space, successively generating and evaluating policies in an attempt to obtain incremental improvements. Or, perhaps, a genetic-style algorithm could be used that would maintain and evaluate a population of policies. Literally hundreds of different optimization methods could be applied.

Here is how the tic-tac-toe problem would be approached with a method making use of a value function. First we would set up a table of numbers, one for each possible state of the game. Each number will be the latest estimate of the probability of our winning from that state. We treat this estimate as the state's *value*, and the whole table is the learned value function. State A has higher value than state B, or is considered "better" than state B, if the current estimate of the probability of our winning from A is higher than it is from B. Assuming we always play Xs, then for all states with three Xs in a row the probability of winning is 1, because we have already won. Similarly, for all states with three Os in a row, or that are filled up, the correct probability is 0, as we cannot win from them. We set the initial values of all the other states to 0.5, representing a guess that we have a 50% chance of winning.

We then play many games against the opponent. To select our moves we examine the states that would result from each of our possible moves (one for each blank space on the board) and look up their current values in the table. Most of the time we move *greedily*, selecting the move that leads to the state with greatest value, that is, with the highest estimated probability of winning. Occasionally, however, we select randomly from among the other moves instead. These are called *exploratory* moves because they cause us to experience states that we might otherwise never see. A sequence of moves made and considered during a game can be diagrammed as in Figure 1.1.

While we are playing, we change the values of the states in which we find ourselves during the game. We attempt to make them more accurate estimates of the probabilities of winning. To do this, we "back up" the value of the state after each greedy move to the state before the move, as suggested by the arrows in Figure 1.1. More precisely, the current value of the earlier state is updated to be closer to the value of the later state. This can be done by moving the earlier state's value a fraction of the way toward the value of the later state. If we let S_t denote the state before the greedy move, and S_{t+1} the state after the move, then the update to the estimated value of S_t, denoted $V(S_t)$, can be written as

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[V(S_{t+1}) - V(S_t)\Big],$$

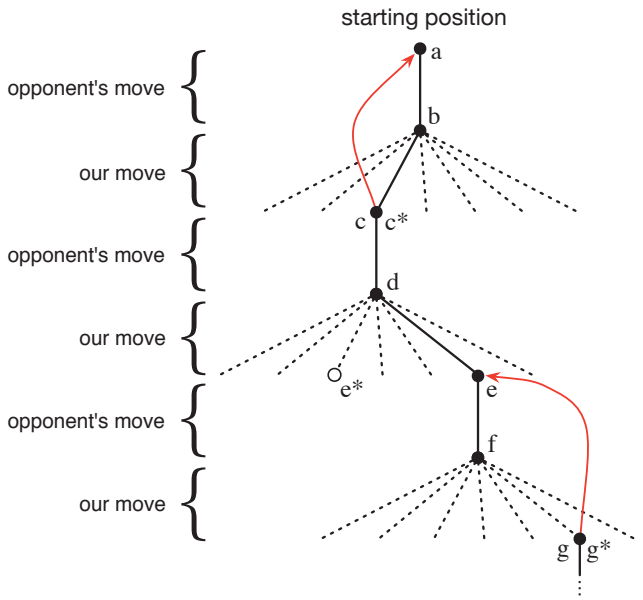


Figure 1.1: A sequence of tic-tac-toe moves. The solid black lines represent the moves taken during a game; the dashed lines represent moves that we (our reinforcement learning player) considered but did not make. Our second move was an exploratory move, meaning that it was taken even though another sibling move, the one leading to e^*, was ranked higher. Exploratory moves do not result in any learning, but each of our other moves does, causing updates as suggested by the red arrows in which estimated values are moved up the tree from later nodes to earlier nodes as detailed in the text.

where α is a small positive fraction called the *step-size parameter*, which influences the rate of learning. This update rule is an example of a *temporal-difference* learning method, so called because its changes are based on a difference, $V(S_{t+1}) - V(S_t)$, between estimates at two successive times.

The method described above performs quite well on this task. For example, if the step-size parameter is reduced properly over time, then this method converges, for any fixed opponent, to the true probabilities of winning from each state given optimal play by our player. Furthermore, the moves then taken (except on exploratory moves) are in fact the optimal moves against this (imperfect) opponent. In other words, the method converges to an optimal policy for playing the game against this opponent. If the step-size parameter is not reduced all the way to zero over time, then this player also plays well against opponents that slowly change their way of playing.

This example illustrates the differences between evolutionary methods and methods that learn value functions. To evaluate a policy an evolutionary method holds the policy fixed and plays many games against the opponent, or simulates many games using a model of the opponent. The frequency of wins gives an unbiased estimate of the probability

of winning with that policy, and can be used to direct the next policy selection. But each policy change is made only after many games, and only the final outcome of each game is used: what happens *during* the games is ignored. For example, if the player wins, then *all* of its behavior in the game is given credit, independently of how specific moves might have been critical to the win. Credit is even given to moves that never occurred! Value function methods, in contrast, allow individual states to be evaluated. In the end, evolutionary and value function methods both search the space of policies, but learning a value function takes advantage of information available during the course of play.

This simple example illustrates some of the key features of reinforcement learning methods. First, there is the emphasis on learning while interacting with an environment, in this case with an opponent player. Second, there is a clear goal, and correct behavior requires planning or foresight that takes into account delayed effects of one's choices. For example, the simple reinforcement learning player would learn to set up multi-move traps for a shortsighted opponent. It is a striking feature of the reinforcement learning solution that it can achieve the effects of planning and lookahead without using a model of the opponent and without conducting an explicit search over possible sequences of future states and actions.

While this example illustrates some of the key features of reinforcement learning, it is so simple that it might give the impression that reinforcement learning is more limited than it really is. Although tic-tac-toe is a two-person game, reinforcement learning also applies in the case in which there is no external adversary, that is, in the case of a "game against nature." Reinforcement learning also is not restricted to problems in which behavior breaks down into separate episodes, like the separate games of tic-tac-toe, with reward only at the end of each episode. It is just as applicable when behavior continues indefinitely and when rewards of various magnitudes can be received at any time. Reinforcement learning is also applicable to problems that do not even break down into discrete time steps like the plays of tic-tac-toe. The general principles apply to continuous-time problems as well, although the theory gets more complicated and we omit it from this introductory treatment.

Tic-tac-toe has a relatively small, finite state set, whereas reinforcement learning can be used when the state set is very large, or even infinite. For example, Gerry Tesauro (1992, 1995) combined the algorithm described above with an artificial neural network to learn to play backgammon, which has approximately 10^{20} states. With this many states it is impossible ever to experience more than a small fraction of them. Tesauro's program learned to play far better than any previous program and eventually better than the world's best human players (Section 16.1). The artificial neural network provides the program with the ability to generalize from its experience, so that in new states it selects moves based on information saved from similar states faced in the past, as determined by its network. How well a reinforcement learning system can work in problems with such large state sets is intimately tied to how appropriately it can generalize from past experience. It is in this role that we have the greatest need for supervised learning methods with reinforcement learning. Artificial neural networks and deep learning (Section 9.6) are not the only, or necessarily the best, way to do this.

In this tic-tac-toe example, learning started with no prior knowledge beyond the

rules of the game, but reinforcement learning by no means entails a tabula rasa view of learning and intelligence. On the contrary, prior information can be incorporated into reinforcement learning in a variety of ways that can be critical for efficient learning (e.g., see Sections 9.5, 17.4, and 13.1). We also have access to the true state in the tic-tac-toe example, whereas reinforcement learning can also be applied when part of the state is hidden, or when different states appear to the learner to be the same.

Finally, the tic-tac-toe player was able to look ahead and know the states that would result from each of its possible moves. To do this, it had to have a model of the game that allowed it to foresee how its environment would change in response to moves that it might never make. Many problems are like this, but in others even a short-term model of the effects of actions is lacking. Reinforcement learning can be applied in either case. A model is not required, but models can easily be used if they are available or can be learned (Chapter 8).

On the other hand, there are reinforcement learning methods that do not need any kind of environment model at all. Model-free systems cannot even think about how their environments will change in response to a single action. The tic-tac-toe player is model-free in this sense with respect to its opponent: it has no model of its opponent of any kind. Because models have to be reasonably accurate to be useful, model-free methods can have advantages over more complex methods when the real bottleneck in solving a problem is the difficulty of constructing a sufficiently accurate environment model. Model-free methods are also important building blocks for model-based methods. In this book we devote several chapters to model-free methods before we discuss how they can be used as components of more complex model-based methods.

Reinforcement learning can be used at both high and low levels in a system. Although the tic-tac-toe player learned only about the basic moves of the game, nothing prevents reinforcement learning from working at higher levels where each of the "actions" may itself be the application of a possibly elaborate problem-solving method. In hierarchical learning systems, reinforcement learning can work simultaneously on several levels.

Exercise 1.1: Self-Play Suppose, instead of playing against a random opponent, the reinforcement learning algorithm described above played against itself, with both sides learning. What do you think would happen in this case? Would it learn a different policy for selecting moves? □

Exercise 1.2: Symmetries Many tic-tac-toe positions appear different but are really the same because of symmetries. How might we amend the learning process described above to take advantage of this? In what ways would this change improve the learning process? Now think again. Suppose the opponent did not take advantage of symmetries. In that case, should we? Is it true, then, that symmetrically equivalent positions should necessarily have the same value? □

Exercise 1.3: Greedy Play Suppose the reinforcement learning player was *greedy*, that is, it always played the move that brought it to the position that it rated the best. Might it learn to play better, or worse, than a nongreedy player? What problems might occur? □

Exercise 1.4: Learning from Exploration Suppose learning updates occurred after *all* moves, including exploratory moves. If the step-size parameter is appropriately reduced

over time (but not the tendency to explore), then the state values would converge to a different set of probabilities. What (conceptually) are the two sets of probabilities computed when we do, and when we do not, learn from exploratory moves? Assuming that we do continue to make exploratory moves, which set of probabilities might be better to learn? Which would result in more wins? □

Exercise 1.5: Other Improvements Can you think of other ways to improve the reinforcement learning player? Can you think of any better way to solve the tic-tac-toe problem as posed? □

1.6 Summary

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment. In our opinion, reinforcement learning is the first field to seriously address the computational issues that arise when learning from interaction with an environment in order to achieve long-term goals.

Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards. This framework is intended to be a simple way of representing essential features of the artificial intelligence problem. These features include a sense of cause and effect, a sense of uncertainty and nondeterminism, and the existence of explicit goals.

The concepts of value and value function are key to most of the reinforcement learning methods that we consider in this book. We take the position that value functions are important for efficient search in the space of policies. The use of value functions distinguishes reinforcement learning methods from evolutionary methods that search directly in policy space guided by evaluations of entire policies.

1.7 Early History of Reinforcement Learning

The early history of reinforcement learning has two main threads, both long and rich, that were pursued independently before intertwining in modern reinforcement learning. One thread concerns learning by trial and error, and originated in the psychology of animal learning. This thread runs through some of the earliest work in artificial intelligence and led to the revival of reinforcement learning in the early 1980s. The second thread concerns the problem of optimal control and its solution using value functions and dynamic programming. For the most part, this thread did not involve learning. The two threads were mostly independent, but became interrelated to some extent around a third, less distinct thread concerning temporal-difference methods such as that used in the tic-tac-toe example in this chapter. All three threads came together in the late 1980s to produce the modern field of reinforcement learning as we present it in this book.

The thread focusing on trial-and-error learning is the one with which we are most familiar and about which we have the most to say in this brief history. Before doing that, however, we briefly discuss the optimal control thread.

The term "optimal control" came into use in the late 1950s to describe the problem of designing a controller to minimize or maximize a measure of a dynamical system's behavior over time. One of the approaches to this problem was developed in the mid-1950s by Richard Bellman and others through extending a nineteenth century theory of Hamilton and Jacobi. This approach uses the concepts of a dynamical system's state and of a value function, or "optimal return function," to define a functional equation, now often called the Bellman equation. The class of methods for solving optimal control problems by solving this equation came to be known as dynamic programming (Bellman, 1957a). Bellman (1957b) also introduced the discrete stochastic version of the optimal control problem known as Markov decision processes (MDPs). Ronald Howard (1960) devised the policy iteration method for MDPs. All of these are essential elements underlying the theory and algorithms of modern reinforcement learning.

Dynamic programming is widely considered the only feasible way of solving general stochastic optimal control problems. It suffers from what Bellman called "the curse of dimensionality," meaning that its computational requirements grow exponentially with the number of state variables, but it is still far more efficient and more widely applicable than any other general method. Dynamic programming has been extensively developed since the late 1950s, including extensions to partially observable MDPs (surveyed by Lovejoy, 1991), many applications (surveyed by White, 1985, 1988, 1993), approximation methods (surveyed by Rust, 1996), and asynchronous methods (Bertsekas, 1982, 1983). Many excellent modern treatments of dynamic programming are available (e.g., Bertsekas, 2005, 2012; Puterman, 1994; Ross, 1983; and Whittle, 1982, 1983). Bryson (1996) provides an authoritative history of optimal control.

Connections between optimal control and dynamic programming, on the one hand, and learning, on the other, were slow to be recognized. We cannot be sure about what accounted for this separation, but its main cause was likely the separation between the disciplines involved and their different goals. Also contributing may have been the prevalent view of dynamic programming as an offline computation depending essentially on accurate system models and analytic solutions to the Bellman equation. Further, the simplest form of dynamic programming is a computation that proceeds backwards in time, making it difficult to see how it could be involved in a learning process that must proceed in a forward direction. Some of the earliest work in dynamic programming, such as that by Bellman and Dreyfus (1959), might now be classified as following a learning approach. Witten's (1977) work (discussed below) certainly qualifies as a combination of learning and dynamic-programming ideas. Werbos (1987) argued explicitly for greater interrelation of dynamic programming and learning methods and for dynamic programming's relevance to understanding neural and cognitive mechanisms. For us the full integration of dynamic programming methods with online learning did not occur until the work of Chris Watkins in 1989, whose treatment of reinforcement learning using the MDP formalism has been widely adopted. Since then these relationships have been extensively developed by many researchers, most particularly by Dimitri Bertsekas

and John Tsitsiklis (1996), who coined the term "neurodynamic programming" to refer to the combination of dynamic programming and artificial neural networks. Another term currently in use is "approximate dynamic programming." These various approaches emphasize different aspects of the subject, but they all share with reinforcement learning an interest in circumventing the classical shortcomings of dynamic programming.

We consider all of the work in optimal control also to be, in a sense, work in reinforcement learning. We define a reinforcement learning method as any effective way of solving reinforcement learning problems, and it is now clear that these problems are closely related to optimal control problems, particularly stochastic optimal control problems such as those formulated as MDPs. Accordingly, we must consider the solution methods of optimal control, such as dynamic programming, also to be reinforcement learning methods. Because almost all of the conventional methods require complete knowledge of the system to be controlled, it feels a little unnatural to say that they are part of reinforcement *learning*. On the other hand, many dynamic programming algorithms are incremental and iterative. Like learning methods, they gradually reach the correct answer through successive approximations. As we show in the rest of this book, these similarities are far more than superficial. The theories and solution methods for the cases of complete and incomplete knowledge are so closely related that we feel they must be considered together as part of the same subject matter.

Let us return now to the other major thread leading to the modern field of reinforcement learning, the thread centered on the idea of trial-and-error learning. We only touch on the major points of contact here, taking up this topic in more detail in Section 14.3. According to American psychologist R. S. Woodworth (1938) the idea of trial-and-error learning goes as far back as the 1850s to Alexander Bain's discussion of learning by "groping and experiment" and more explicitly to the British ethologist and psychologist Conway Lloyd Morgan's 1894 use of the term to describe his observations of animal behavior. Perhaps the first to succinctly express the essence of trial-and-error learning as a principle of learning was Edward Thorndike:

> Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. (Thorndike, 1911, p. 244)

Thorndike called this the "Law of Effect" because it describes the effect of reinforcing events on the tendency to select actions. Thorndike later modified the law to better account for subsequent data on animal learning (such as differences between the effects of reward and punishment), and the law in its various forms has generated considerable controversy among learning theorists (e.g., see Gallistel, 2005; Herrnstein, 1970; Kimble, 1961, 1967; Mazur, 1994). Despite this, the Law of Effect—in one form or another—is widely regarded as a basic principle underlying much behavior (e.g., Hilgard and Bower, 1975; Dennett, 1978; Campbell, 1960; Cziko, 1995). It is the basis of the influential

learning theories of Clark Hull (1943, 1952) and the influential experimental methods of B. F. Skinner (1938).

The term "reinforcement" in the context of animal learning came into use well after Thorndike's expression of the Law of Effect, first appearing in this context (to the best of our knowledge) in the 1927 English translation of Pavlov's monograph on conditioned reflexes. Pavlov described reinforcement as the strengthening of a pattern of behavior due to an animal receiving a stimulus—a reinforcer—in an appropriate temporal relationship with another stimulus or with a response. Some psychologists extended the idea of reinforcement to include weakening as well as strengthening of behavior, and extended the idea of a reinforcer to include possibly the omission or termination of stimulus. To be considered reinforcer, the strengthening or weakening must persist after the reinforcer is withdrawn; a stimulus that merely attracts an animal's attention or that energizes its behavior without producing lasting changes would not be considered a reinforcer.

The idea of implementing trial-and-error learning in a computer appeared among the earliest thoughts about the possibility of artificial intelligence. In a 1948 report, Alan Turing described a design for a "pleasure-pain system" that worked along the lines of the Law of Effect:

> When a configuration is reached for which the action is undetermined, a random choice for the missing data is made and the appropriate entry is made in the description, tentatively, and is applied. When a pain stimulus occurs all tentative entries are cancelled, and when a pleasure stimulus occurs they are all made permanent. (Turing, 1948)

Many ingenious electro-mechanical machines were constructed that demonstrated trial-and-error learning. The earliest may have been a machine built by Thomas Ross (1933) that was able to find its way through a simple maze and remember the path through the settings of switches. In 1951 W. Grey Walter built a version of his "mechanical tortoise" (Walter, 1950) capable of a simple form of learning. In 1952 Claude Shannon demonstrated a maze-running mouse named Theseus that used trial and error to find its way through a maze, with the maze itself remembering the successful directions via magnets and relays under its floor (see also Shannon, 1951). J. A. Deutsch (1954) described a maze-solving machine based on his behavior theory (Deutsch, 1953) that has some properties in common with model-based reinforcement learning (Chapter 8). In his Ph.D. dissertation, Marvin Minsky (1954) discussed computational models of reinforcement learning and described his construction of an analog machine composed of components he called SNARCs (Stochastic Neural-Analog Reinforcement Calculators) meant to resemble modifiable synaptic connections in the brain (Chapter 15). The web site `cyberneticzoo.com` contains a wealth of information on these and many other electro-mechanical learning machines.

Building electro-mechanical learning machines gave way to programming digital computers to perform various types of learning, some of which implemented trial-and-error learning. Farley and Clark (1954) described a digital simulation of a neural-network learning machine that learned by trial and error. But their interests soon shifted from trial-and-error learning to generalization and pattern recognition, that is, from reinforcement learning to supervised learning (Clark and Farley, 1955). This began a pattern

of confusion about the relationship between these types of learning. Many researchers seemed to believe that they were studying reinforcement learning when they were actually studying supervised learning. For example, artificial neural network pioneers such as Rosenblatt (1962) and Widrow and Hoff (1960) were clearly motivated by reinforcement learning—they used the language of rewards and punishments—but the systems they studied were supervised learning systems suitable for pattern recognition and perceptual learning. Even today, some researchers and textbooks minimize or blur the distinction between these types of learning. For example, some artificial neural network textbooks have used the term "trial-and-error" to describe networks that learn from training examples. This is an understandable confusion because these networks use error information to update connection weights, but this misses the essential character of trial-and-error learning as selecting actions on the basis of evaluative feedback that does not rely on knowledge of what the correct action should be.

Partly as a result of these confusions, research into genuine trial-and-error learning became rare in the 1960s and 1970s, although there were notable exceptions. In the 1960s the terms "reinforcement" and "reinforcement learning" were used in the engineering literature for the first time to describe engineering uses of trial-and-error learning (e.g., Waltz and Fu, 1965; Mendel, 1966; Fu, 1970; Mendel and McClaren, 1970). Particularly influential was Minsky's paper "Steps Toward Artificial Intelligence" (Minsky, 1961), which discussed several issues relevant to trial-and-error learning, including prediction, expectation, and what he called the *basic credit-assignment problem for complex reinforcement learning systems*: How do you distribute credit for success among the many decisions that may have been involved in producing it? All of the methods we discuss in this book are, in a sense, directed toward solving this problem. Minsky's paper is well worth reading today.

In the next few paragraphs we discuss some of the other exceptions and partial exceptions to the relative neglect of computational and theoretical study of genuine trial-and-error learning in the 1960s and 1970s.

One exception was the work of the New Zealand researcher John Andreae, who developed a system called STeLLA that learned by trial and error in interaction with its environment. This system included an internal model of the world and, later, an "internal monologue" to deal with problems of hidden state (Andreae, 1963, 1969a,b). Andreae's later work (1977) placed more emphasis on learning from a teacher, but still included learning by trial and error, with the generation of novel events being one of the system's goals. A feature of this work was a "leakback process," elaborated more fully in Andreae (1998), that implemented a credit-assignment mechanism similar to the backing-up update operations that we describe. Unfortunately, his pioneering research was not well known and did not greatly impact subsequent reinforcement learning research. Recent summaries are available (Andreae, 2017a,b).

More influential was the work of Donald Michie. In 1961 and 1963 he described a simple trial-and-error learning system for learning how to play tic-tac-toe (or naughts and crosses) called MENACE (for Matchbox Educable Naughts and Crosses Engine). It consisted of a matchbox for each possible game position, each matchbox containing a number of colored beads, a different color for each possible move from that position. By

drawing a bead at random from the matchbox corresponding to the current game position, one could determine MENACE's move. When a game was over, beads were added to or removed from the boxes used during play to reward or punish MENACE's decisions. Michie and Chambers (1968) described another tic-tac-toe reinforcement learner called GLEE (Game Learning Expectimaxing Engine) and a reinforcement learning controller called BOXES. They applied BOXES to the task of learning to balance a pole hinged to a movable cart on the basis of a failure signal occurring only when the pole fell or the cart reached the end of a track. This task was adapted from the earlier work of Widrow and Smith (1964), who used supervised learning methods, assuming instruction from a teacher already able to balance the pole. Michie and Chambers's version of pole-balancing is one of the best early examples of a reinforcement learning task under conditions of incomplete knowledge. It influenced much later work in reinforcement learning, beginning with some of our own studies (Barto, Sutton, and Anderson, 1983; Sutton, 1984). Michie consistently emphasized the role of trial and error and learning as essential aspects of artificial intelligence (Michie, 1974).

Widrow, Gupta, and Maitra (1973) modified the Least-Mean-Square (LMS) algorithm of Widrow and Hoff (1960) to produce a reinforcement learning rule that could learn from success and failure signals instead of from training examples. They called this form of learning "selective bootstrap adaptation" and described it as "learning with a critic" instead of "learning with a teacher." They analyzed this rule and showed how it could learn to play blackjack. This was an isolated foray into reinforcement learning by Widrow, whose contributions to supervised learning were much more influential. Our use of the term "critic" is derived from Widrow, Gupta, and Maitra's paper. Buchanan, Mitchell, Smith, and Johnson (1978) independently used the term critic in the context of machine learning (see also Dietterich and Buchanan, 1984), but for them a critic is an expert system able to do more than evaluate performance.

Research on *learning automata* had a more direct influence on the trial-and-error thread leading to modern reinforcement learning research. These are methods for solving a nonassociative, purely selectional learning problem known as the *k-armed bandit* by analogy to a slot machine, or "one-armed bandit," except with k levers (see Chapter 2). Learning automata are simple, low-memory machines for improving the probability of reward in these problems. Learning automata originated with work in the 1960s of the Russian mathematician and physicist M. L. Tsetlin and colleagues (published posthumously in Tsetlin, 1973) and has been extensively developed since then within engineering (see Narendra and Thathachar, 1974, 1989). These developments included the study of *stochastic learning automata*, which are methods for updating action probabilities on the basis of reward signals. Although not developed in the tradition of stochastic learning automata, Harth and Tzanakou's (1974) Alopex algorithm (for *Al*gorithm *o*f *p*attern *ex*traction) is a stochastic method for detecting correlations between actions and reinforcement that influenced some of our early research (Barto, Sutton, and Brouwer, 1981). Stochastic learning automata were foreshadowed by earlier work in psychology, beginning with William Estes' (1950) effort toward a statistical theory of learning and further developed by others (e.g., Bush and Mosteller, 1955; Sternberg, 1963).

The statistical learning theories developed in psychology were adopted by researchers in

economics, leading to a thread of research in that field devoted to reinforcement learning. This work began in 1973 with the application of Bush and Mosteller's learning theory to a collection of classical economic models (Cross, 1973). One goal of this research was to study artificial agents that act more like real people than do traditional idealized economic agents (Arthur, 1991). This approach expanded to the study of reinforcement learning in the context of game theory. Reinforcement learning in economics developed largely independently of the early work in reinforcement learning in artificial intelligence, though game theory remains a topic of interest in both fields (beyond the scope of this book). Camerer (2011) discusses the reinforcement learning tradition in economics, and Nowé, Vrancx, and De Hauwere (2012) provide an overview of the subject from the point of view of multi-agent extensions to the approach that we introduce in this book. Reinforcement in the context of game theory is a much different subject than reinforcement learning used in programs to play tic-tac-toe, checkers, and other recreational games. See, for example, Szita (2012) for an overview of this aspect of reinforcement learning and games.

John Holland (1975) outlined a general theory of adaptive systems based on selectional principles. His early work concerned trial and error primarily in its nonassociative form, as in evolutionary methods and the k-armed bandit. In 1976 and more fully in 1986, he introduced *classifier systems*, true reinforcement learning systems including association and value functions. A key component of Holland's classifier systems was the "bucket-brigade algorithm" for credit assignment, which is closely related to the temporal difference algorithm used in our tic-tac-toe example and discussed in Chapter 6. Another key component was a *genetic algorithm*, an evolutionary method whose role was to evolve useful representations. Classifier systems have been extensively developed by many researchers to form a major branch of reinforcement learning research (reviewed by Urbanowicz and Moore, 2009), but genetic algorithms—which we do not consider to be reinforcement learning systems by themselves—have received much more attention, as have other approaches to evolutionary computation (e.g., Fogel, Owens and Walsh, 1966, and Koza, 1992).

The individual most responsible for reviving the trial-and-error thread to reinforcement learning within artificial intelligence was Harry Klopf (1972, 1975, 1982). Klopf recognized that essential aspects of adaptive behavior were being lost as learning researchers came to focus almost exclusively on supervised learning. What was missing, according to Klopf, were the hedonic aspects of behavior, the drive to achieve some result from the environment, to control the environment toward desired ends and away from undesired ends (see Section 15.9). This is the essential idea of trial-and-error learning. Klopf's ideas were especially influential on the authors because our assessment of them (Barto and Sutton, 1981a) led to our appreciation of the distinction between supervised and reinforcement learning, and to our eventual focus on reinforcement learning. Much of the early work that we and colleagues accomplished was directed toward showing that reinforcement learning and supervised learning were indeed different (Barto, Sutton, and Brouwer, 1981; Barto and Sutton, 1981b; Barto and Anandan, 1985). Other studies showed how reinforcement learning could address important problems in artificial neural network learning, in particular, how it could produce learning algorithms for multilayer networks (Barto, Anderson, and Sutton, 1982; Barto and Anderson, 1985; Barto, 1985, 1986; Barto and Jordan, 1987).

We turn now to the third thread to the history of reinforcement learning, that concerning temporal-difference learning. Temporal-difference learning methods are distinctive in being driven by the difference between temporally successive estimates of the same quantity—for example, of the probability of winning in the tic-tac-toe example. This thread is smaller and less distinct than the other two, but it has played a particularly important role in the field, in part because temporal-difference methods seem to be new and unique to reinforcement learning.

The origins of temporal-difference learning are in part in animal learning psychology, in particular, in the notion of *secondary reinforcers*. A secondary reinforcer is a stimulus that has been paired with a primary reinforcer such as food or pain and, as a result, has come to take on similar reinforcing properties. Minsky (1954) may have been the first to realize that this psychological principle could be important for artificial learning systems. Arthur Samuel (1959) was the first to propose and implement a learning method that included temporal-difference ideas, as part of his celebrated checkers-playing program (Section 16.2).

Samuel made no reference to Minsky's work or to possible connections to animal learning. His inspiration apparently came from Claude Shannon's (1950) suggestion that a computer could be programmed to use an evaluation function to play chess, and that it might be able to improve its play by modifying this function online. (It is possible that these ideas of Shannon's also influenced Bellman, but we know of no evidence for this.) Minsky (1961) extensively discussed Samuel's work in his "Steps" paper, suggesting the connection to secondary reinforcement theories, both natural and artificial.

As we have discussed, in the decade following the work of Minsky and Samuel, little computational work was done on trial-and-error learning, and apparently no computational work at all was done on temporal-difference learning. In 1972, Klopf brought trial-and-error learning together with an important component of temporal-difference learning. Klopf was interested in principles that would scale to learning in large systems, and thus was intrigued by notions of local reinforcement, whereby subcomponents of an overall learning system could reinforce one another. He developed the idea of "generalized reinforcement," whereby every component (nominally, every neuron) views all of its inputs in reinforcement terms: excitatory inputs as rewards and inhibitory inputs as punishments. This is not the same idea as what we now know as temporal-difference learning, and in retrospect it is farther from it than was Samuel's work. On the other hand, Klopf linked the idea with trial-and-error learning and related it to the massive empirical database of animal learning psychology.

Sutton (1978a,b,c) developed Klopf's ideas further, particularly the links to animal learning theories, describing learning rules driven by changes in temporally successive predictions. He and Barto refined these ideas and developed a psychological model of classical conditioning based on temporal-difference learning (Sutton and Barto, 1981a; Barto and Sutton, 1982). There followed several other influential psychological models of classical conditioning based on temporal-difference learning (e.g., Klopf, 1988; Moore et al., 1986; Sutton and Barto, 1987, 1990). Some neuroscience models developed at this time are well interpreted in terms of temporal-difference learning (Hawkins and Kandel, 1984; Byrne, Gingrich, and Baxter, 1990; Gelperin, Hopfield, and Tank, 1985; Tesauro,

1986; Friston et al., 1994), although in most cases there was no historical connection.

Our early work on temporal-difference learning was strongly influenced by animal learning theories and by Klopf's work. Relationships to Minsky's "Steps" paper and to Samuel's checkers players were recognized only afterward. By 1981, however, we were fully aware of all the prior work mentioned above as part of the temporal-difference and trial-and-error threads. At this time we developed a method for using temporal-difference learning combined with trial-and-error learning, known as the *actor–critic architecture*, and applied this method to Michie and Chambers's pole-balancing problem (Barto, Sutton, and Anderson, 1983). This method was extensively studied in Sutton's (1984) Ph.D. dissertation and extended to use backpropagation neural networks in Anderson's (1986) Ph.D. dissertation. Around this time, Holland (1986) incorporated temporal-difference ideas explicitly into his classifier systems in the form of his bucket-brigade algorithm. A key step was taken by Sutton (1988) by separating temporal-difference learning from control, treating it as a general prediction method. That paper also introduced the TD(λ) algorithm and proved some of its convergence properties.

As we were finalizing our work on the actor–critic architecture in 1981, we discovered a paper by Ian Witten (1977, 1976a) which appears to be the earliest publication of a temporal-difference learning rule. He proposed the method that we now call tabular TD(0) for use as part of an adaptive controller for solving MDPs. This work was first submitted for journal publication in 1974 and also appeared in Witten's 1976 PhD dissertation. Witten's work was a descendant of Andreae's early experiments with STeLLA and other trial-and-error learning systems. Thus, Witten's 1977 paper spanned both major threads of reinforcement learning research—trial-and-error learning and optimal control—while making a distinct early contribution to temporal-difference learning.

The temporal-difference and optimal control threads were fully brought together in 1989 with Chris Watkins's development of Q-learning. This work extended and integrated prior work in all three threads of reinforcement learning research. Paul Werbos (1987) contributed to this integration by arguing for the convergence of trial-and-error learning and dynamic programming since 1977. By the time of Watkins's work there had been tremendous growth in reinforcement learning research, primarily in the machine learning subfield of artificial intelligence, but also in artificial neural networks and artificial intelligence more broadly. In 1992, the remarkable success of Gerry Tesauro's backgammon playing program, TD-Gammon, brought additional attention to the field.

In the time since publication of the first edition of this book, a flourishing subfield of neuroscience developed that focuses on the relationship between reinforcement learning algorithms and reinforcement learning in the nervous system. Most responsible for this is an uncanny similarity between the behavior of temporal-difference algorithms and the activity of dopamine producing neurons in the brain, as pointed out by a number of researchers (Friston et al., 1994; Barto, 1995a; Houk, Adams, and Barto, 1995; Montague, Dayan, and Sejnowski, 1996; and Schultz, Dayan, and Montague, 1997). Chapter 15 provides an introduction to this exciting aspect of reinforcement learning. Other important contributions made in the recent history of reinforcement learning are too numerous to mention in this brief account; we cite many of these at the end of the individual chapters in which they arise.

Bibliographical Remarks

For additional general coverage of reinforcement learning, we refer the reader to the books by Szepesvári (2010), Bertsekas and Tsitsiklis (1996), Kaelbling (1993a), and Sugiyama, Hachiya, and Morimura (2013). Books that take a control or operations research perspective include those of Si, Barto, Powell, and Wunsch (2004), Powell (2011), Lewis and Liu (2012), and Bertsekas (2012). Cao's (2009) review places reinforcement learning in the context of other approaches to learning and optimization of stochastic dynamic systems. Three special issues of the journal *Machine Learning* focus on reinforcement learning: Sutton (1992a), Kaelbling (1996), and Singh (2002). Useful surveys are provided by Barto (1995b); Kaelbling, Littman, and Moore (1996); and Keerthi and Ravindran (1997). The volume edited by Weiring and van Otterlo (2012) provides an excellent overview of recent developments.

1.2 The example of Phil's breakfast in this chapter was inspired by Agre (1988).

1.5 The temporal-difference method used in the tic-tac-toe example is developed in Chapter 6.

Part I: Tabular Solution Methods

In this part of the book we describe almost all the core ideas of reinforcement learning algorithms in their simplest forms: that in which the state and action spaces are small enough for the approximate value functions to be represented as arrays, or *tables*. In this case, the methods can often find exact solutions, that is, they can often find exactly the optimal value function and the optimal policy. This contrasts with the approximate methods described in the next part of the book, which only find approximate solutions, but which in return can be applied effectively to much larger problems.

The first chapter of this part of the book describes solution methods for the special case of the reinforcement learning problem in which there is only a single state, called bandit problems. The second chapter describes the general problem formulation that we treat throughout the rest of the book—finite Markov decision processes—and its main ideas including Bellman equations and value functions.

The next three chapters describe three fundamental classes of methods for solving finite Markov decision problems: dynamic programming, Monte Carlo methods, and temporal-difference learning. Each class of methods has its strengths and weaknesses. Dynamic programming methods are well developed mathematically, but require a complete and accurate model of the environment. Monte Carlo methods don't require a model and are conceptually simple, but are not well suited for step-by-step incremental computation. Finally, temporal-difference methods require no model and are fully incremental, but are more complex to analyze. The methods also differ in several ways with respect to their efficiency and speed of convergence.

The remaining two chapters describe how these three classes of methods can be combined to obtain the best features of each of them. In one chapter we describe how the strengths of Monte Carlo methods can be combined with the strengths of temporal-difference methods via multi-step bootstrapping methods. In the final chapter of this part of the book we show how temporal-difference learning methods can be combined with model learning and planning methods (such as dynamic programming) for a complete and unified solution to the tabular reinforcement learning problem.

Chapter 2

Multi-armed 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 search for good behavior. Purely evaluative feedback indicates how good the action taken was, but not whether it was the best or the worst action possible. 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.

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 enables 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 k-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 A k-armed Bandit Problem

Consider the following learning problem. You are faced repeatedly with a choice among k 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 *k-armed bandit problem*, so named by analogy to a slot machine, or "one-armed bandit," except that it has k 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 is the selection of a treatment, and each reward is the survival or well-being of the patient. Today the term "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 k-armed bandit problem, each of the k actions has an expected or mean reward given that that action is selected; let us call this the *value* of that action. We denote the action selected on time step t as A_t, and the corresponding reward as R_t. The value then of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a] .$$

If you knew the value of each action, then it would be trivial to solve the k-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. We denote the estimated value of action a at time step t as $Q_t(a)$. We would like $Q_t(a)$ to be close to $q_*(a)$.

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 these the *greedy* actions. When you select one of these actions, 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 a 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 k-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 k-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 our version of the k-armed bandit problem enables us to show this in a particularly clear form.

2.2 Action-value Methods

We begin by looking more closely at methods for estimating the values of actions and for using the estimates to make action selection decisions, which we collectively call *action-value methods*. Recall that the true value of an action is the mean reward when that action is selected. One natural way to estimate this is by averaging the rewards actually received:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}, \tag{2.1}$$

where $\mathbb{1}_{predicate}$ denotes the random variable that is 1 if *predicate* is true and 0 if it is not. If the denominator is zero, then we instead define $Q_t(a)$ as some default value, such as 0. As the denominator goes to infinity, 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 an 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 one of the actions with the highest estimated value, that is, one of the greedy actions as defined in the previous section. If there is more than one greedy action, then a selection is made among them in some arbitrary way, perhaps randomly. We write this *greedy* action selection method as

$$A_t \doteq \arg\max_a Q_t(a), \tag{2.2}$$

where $\arg\max_a$ denotes the action a for which the expression that follows is maximized (again, 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 ε, instead

select randomly from among all the actions with equal probability, independently of the action-value estimates. We call methods using this near-greedy action selection rule *ε-greedy* methods. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, 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.

Exercise 2.1 In ε-greedy action selection, for the case of two actions and $\varepsilon = 0.5$, what is the probability that the greedy action is selected? □

2.3 The 10-armed Testbed

To roughly assess the relative effectiveness of the greedy and ε-greedy action-value methods, we compared them numerically on a suite of test problems. This was a set of 2000 randomly generated k-armed bandit problems with $k = 10$. For each bandit problem, such as the one shown in Figure 2.1, the action values, $q_*(a)$, $a = 1, \ldots, 10$,

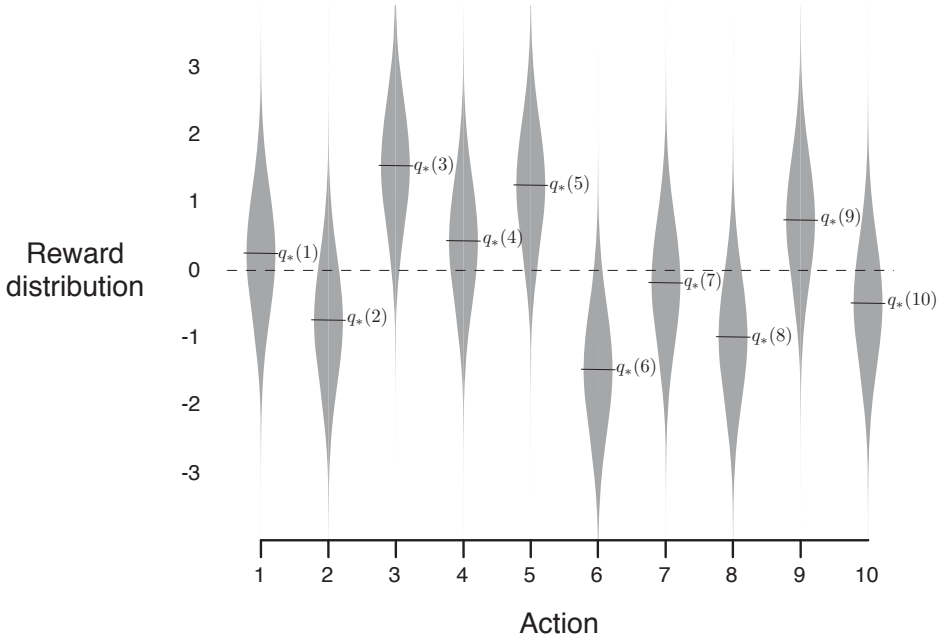


Figure 2.1: An example bandit problem from the 10-armed testbed. The true value $q_*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean $q_*(a)$ unit variance normal distribution, as suggested by these gray distributions.

were selected according to a normal (Gaussian) distribution with mean 0 and variance 1. Then, when a learning method applied to that problem selected action A_t at time step t, the actual reward, R_t, was selected from a normal distribution with mean $q_*(A_t)$ and variance 1. These distributions are shown in gray in Figure 2.1. We call this suite of test tasks the *10-armed testbed*. For any learning method, we can measure its performance and behavior as it improves with experience over 1000 time steps when applied to one of the bandit problems. This makes up one *run*. Repeating this for 2000 independent runs, each with a different bandit problem, we obtained measures of the learning algorithm's average behavior.

Figure 2.2 compares a greedy method with two ε-greedy methods ($\varepsilon = 0.01$ and $\varepsilon = 0.1$), as described above, on the 10-armed testbed. All the methods 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 performed significantly worse in the long run because it

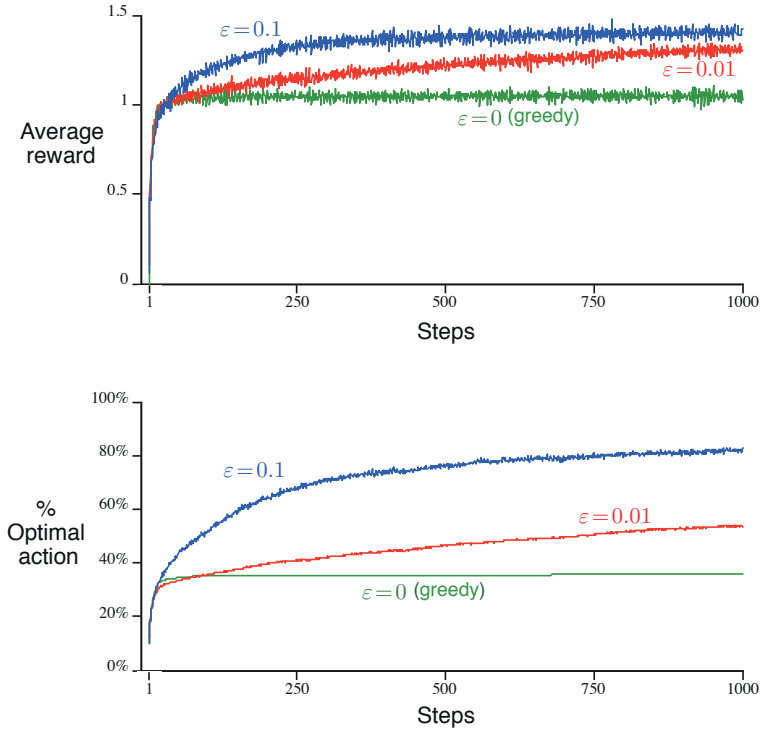


Figure 2.2: Average performance of ε-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates.

often got 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 ε-greedy methods eventually performed better because they continued to explore and to improve their chances of recognizing the optimal action. The $\varepsilon = 0.1$ method explored more, and usually found the optimal action earlier, but it never selected that action more than 91% of the time. The $\varepsilon = 0.01$ method improved more slowly, but eventually would perform better than the $\varepsilon = 0.1$ method on both performance measures shown in the figure. It is also possible to reduce ε over time to try to get the best of both high and low values.

The advantage of ε-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 ε-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, 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 shall see in the next few chapters, 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 as learning proceeds and the agent's decision-making policy changes. Reinforcement learning requires a balance between exploration and exploitation.

Exercise 2.2: Bandit example Consider a k-armed bandit problem with $k = 4$ actions, denoted 1, 2, 3, and 4. Consider applying to this problem a bandit algorithm using ε-greedy action selection, sample-average action-value estimates, and initial estimates of $Q_1(a) = 0$, for all a. Suppose the initial sequence of actions and rewards is $A_1 = 1$, $R_1 = 1$, $A_2 = 2$, $R_2 = 1$, $A_3 = 2$, $R_3 = 2$, $A_4 = 2$, $R_4 = 2$, $A_5 = 3$, $R_5 = 0$. On some of these time steps the ε case may have occurred, causing an action to be selected at random. On which time steps did this definitely occur? On which time steps could this possibly have occurred? □

Exercise 2.3 In the comparison shown in Figure 2.2, which method will perform best in the long run in terms of cumulative reward and probability of selecting the best action? How much better will it be? Express your answer quantitatively. □

2.4 Incremental Implementation

The action-value methods we have discussed so far all estimate action values as sample averages of observed rewards. We now turn to the question of how these averages can be computed in a computationally efficient manner, in particular, with constant memory

and constant per-time-step computation.

To simplify notation we concentrate on a single action. Let R_i now denote the reward received after the ith selection *of this action*, and let Q_n denote the estimate of its action value after it has been selected $n-1$ times, which we can now write simply as

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}.$$

The obvious implementation would be to maintain a record of all the rewards and then perform this computation whenever the estimated value was needed. However, if this is done, then the memory and computational requirements would grow over time as more rewards are seen. Each additional reward would require additional memory to store it and additional computation to compute the sum in the numerator.

As you might suspect, this is not really necessary. It is easy to devise incremental formulas for updating averages with small, constant computation required to process each new reward. Given Q_n and the nth reward, R_n, the new average of all n rewards can be computed by

$$\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\Big(R_n + (n-1)Q_n\Big) \\
&= \frac{1}{n}\Big(R_n + nQ_n - Q_n\Big) \\
&= Q_n + \frac{1}{n}\Big[R_n - Q_n\Big], \qquad (2.3)
\end{aligned}$$

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

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

$$\textit{NewEstimate} \leftarrow \textit{OldEstimate} \;+\; \textit{StepSize}\,\Big[\textit{Target} - \textit{OldEstimate}\Big]. \qquad (2.4)$$

The expression $\big[\textit{Target} - \textit{OldEstimate}\big]$ 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 nth reward.

Note that the step-size parameter (*StepSize*) used in the incremental method (2.3) changes from time step to time step. In processing the nth reward for action a, the method uses the step-size parameter $\frac{1}{n}$. In this book we denote the step-size parameter by α or, more generally, by $\alpha_t(a)$.

Pseudocode for a complete bandit algorithm using incrementally computed sample averages and ε-greedy action selection is shown in the box below. The function $bandit(a)$ is assumed to take an action and return a corresponding reward.

A simple bandit algorithm

Initialize, for $a = 1$ to k:
  $Q(a) \leftarrow 0$
  $N(a) \leftarrow 0$

Loop forever:
  $A \leftarrow \begin{cases} \operatorname{argmax}_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
  $R \leftarrow bandit(A)$
  $N(A) \leftarrow N(A) + 1$
  $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\big[R - Q(A)\big]$

2.5 Tracking a Nonstationary Problem

The averaging methods discussed so far are appropriate for stationary bandit problems, that is, for bandit problems in which the reward probabilities do not change over time. As noted earlier, we often encounter reinforcement learning problems that are effectively nonstationary. In such cases it makes sense to give more weight to recent rewards than to long-past rewards. 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_n of the $n-1$ past rewards is modified to be

$$Q_{n+1} \doteq Q_n + \alpha\Big[R_n - Q_n\Big], \qquad (2.5)$$

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

$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha\Big[R_n - Q_n\Big] \\
&= \alpha R_n + (1-\alpha)Q_n \\
&= \alpha R_n + (1-\alpha)\left[\alpha R_{n-1} + (1-\alpha)Q_{n-1}\right] \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2\alpha R_{n-2} + \\
&\qquad\qquad \cdots + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1 \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i. \qquad (2.6)
\end{aligned}$$

We call this a weighted average because the sum of the weights is $(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} = 1$, as you can check for yourself. Note that the weight, $\alpha(1-\alpha)^{n-i}$, given to the reward R_i depends on how many rewards ago, $n-i$, it was 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_n, 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 $\alpha_n(a)$ denote the step-size parameter used to process the reward received after the nth selection of action a. As we have noted, the choice $\alpha_n(a) = \frac{1}{n}$ 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 $\{\alpha_n(a)\}$. A well-known result in stochastic approximation theory gives us the conditions required to assure convergence with probability 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty. \tag{2.7}$$

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, $\alpha_n(a) = \frac{1}{n}$, but not for the case of constant step-size parameter, $\alpha_n(a) = \alpha$. 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 most common 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.

Exercise 2.4 If the step-size parameters, α_n, are not constant, then the estimate Q_n is a weighted average of previously received rewards with a weighting different from that given by (2.6). What is the weighting on each prior reward for the general case, analogous to (2.6), in terms of the sequence of step-size parameters? □

Exercise 2.5 (programming) Design and conduct an experiment to demonstrate the difficulties that sample-average methods have for nonstationary problems. Use a modified version of the 10-armed testbed in which all the $q_*(a)$ start out equal and then take independent random walks (say by adding a normally distributed increment with mean zero and standard deviation 0.01 to all the $q_*(a)$ on each step). Prepare plots like Figure 2.2 for an action-value method using sample averages, incrementally computed, and another action-value method using a constant step-size parameter, $\alpha = 0.1$. Use $\varepsilon = 0.1$ and longer runs, say of 10,000 steps. □

2.6 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, the bias disappears once all actions have been selected at least once, but for methods with constant α, 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 to encourage 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 2.3 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 ε-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

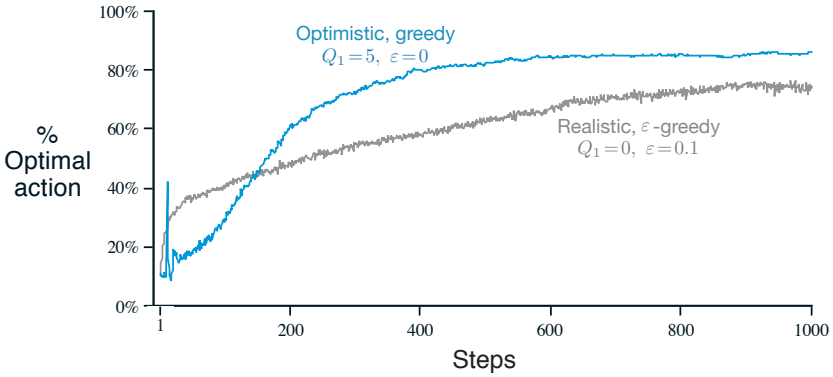


Figure 2.3: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$.

temporary. If the task changes, creating a renewed need for exploration, this method cannot help. Indeed, any method that focuses on the initial conditions 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.

Exercise 2.6: Mysterious Spikes The results shown in Figure 2.3 should be quite reliable because they are averages over 2000 individual, randomly chosen 10-armed bandit tasks. Why, then, are there oscillations and spikes in the early part of the curve for the optimistic method? In other words, what might make this method perform particularly better or worse, on average, on particular early steps? □

Exercise 2.7: Unbiased Constant-Step-Size Trick In most of this chapter we have used sample averages to estimate action values because sample averages do not produce the initial bias that constant step sizes do (see the analysis leading to (2.6)). However, sample averages are not a completely satisfactory solution because they may perform poorly on nonstationary problems. Is it possible to avoid the bias of constant step sizes while retaining their advantages on nonstationary problems? One way is to use a step size of

$$\beta_n \doteq \alpha/\bar{o}_n, \tag{2.8}$$

to process the nth reward for a particular action, where $\alpha > 0$ is a conventional constant step size, and $\bar{o}_n$ is a trace of one that starts at 0:

$$\bar{o}_n \doteq \bar{o}_{n-1} + \alpha(1 - \bar{o}_{n-1}), \quad \text{for } n \geq 0, \quad \text{with } \bar{o}_0 \doteq 0. \tag{2.9}$$

Carry out an analysis like that in (2.6) to show that Q_n is an exponential recency-weighted average *without initial bias*. □

2.7 Upper-Confidence-Bound Action Selection

Exploration is needed because there is always uncertainty about the accuracy of the action-value estimates. The greedy actions are those that look best at present, but some of the other actions may actually be better. ε-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 according to

$$A_t \doteq \operatorname*{arg\,max}_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right], \tag{2.10}$$

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), $N_t(a)$ denotes the number of times that action a has been selected prior to time t (the denominator in (2.1)), 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 c determining the confidence level. Each time a is selected the uncertainty is presumably reduced: $N_t(a)$ increments, and, as it appears in the denominator, the uncertainty term decreases. On the other hand, each time an action other than a is selected, t increases but $N_t(a)$ does not; because t appears in the numerator, the uncertainty estimate increases. The use of the natural logarithm means that the increases get smaller over time, but are unbounded; all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time.

Results with UCB on the 10-armed testbed are shown in Figure 2.4. UCB often performs well, as shown here, but is more difficult than ε-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; methods more complex than those presented in Section 2.5 would be needed. Another difficulty is dealing with large state spaces, particularly when using function approximation as developed in Part II of this book. In these more advanced settings the idea of UCB action selection is usually not practical.

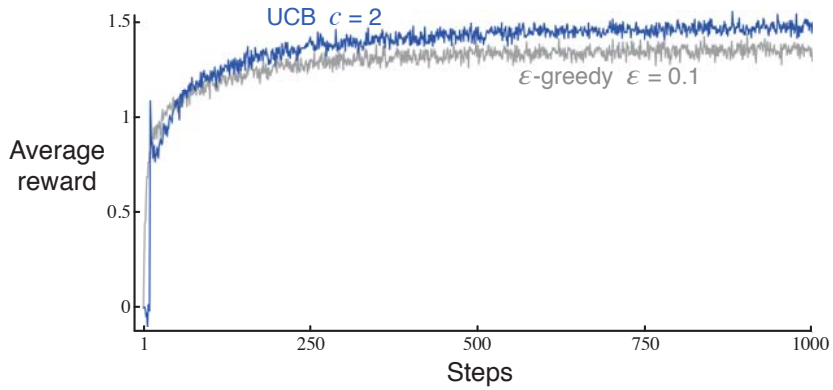


Figure 2.4: Average performance of UCB action selection on the 10-armed testbed. As shown, UCB generally performs better than ε-greedy action selection, except in the first k steps, when it selects randomly among the as-yet-untried actions.

Exercise 2.8: UCB Spikes In Figure 2.4 the UCB algorithm shows a distinct spike in performance on the 11th step. Why is this? Note that for your answer to be fully satisfactory it must explain both why the reward increases on the 11th step and why it decreases on the subsequent steps. Hint: if $c = 1$, then the spike is less prominent. □

2.8 Gradient Bandit Algorithms

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* for each action a, which we denote $H_t(a)$. 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 action preferences there is no effect on the action probabilities, which are determined according to a *soft-max distribution* (i.e., Gibbs or Boltzmann distribution) as follows:

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a), \tag{2.11}$$

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 action preferences are the same (e.g., $H_1(a) = 0$, for all a) so that all actions have an equal probability of being selected.

Exercise 2.9 Show that in the case of two actions, the soft-max distribution is the same as that given by the logistic, or sigmoid, function often used in statistics and artificial neural networks. □

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

$$\begin{aligned} H_{t+1}(A_t) &\doteq H_t(A_t) + \alpha\big(R_t - \bar{R}_t\big)\big(1 - \pi_t(A_t)\big), && \text{and} \\ H_{t+1}(a) &\doteq H_t(a) - \alpha\big(R_t - \bar{R}_t\big)\pi_t(a), && \text{for all } a \neq A_t, \end{aligned} \tag{2.12}$$

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 2.4 (or Section 2.5 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 2.5 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 effect 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 (2.12)), then performance would be significantly degraded, as shown in the figure.

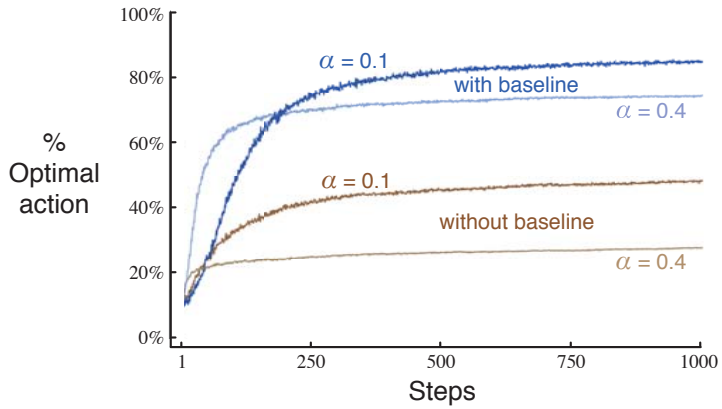


Figure 2.5: Average performance of the gradient bandit algorithm with and without a reward baseline on the 10-armed testbed when the $q_*(a)$ are chosen to be near +4 rather than near zero.

The Bandit Gradient Algorithm as Stochastic Gradient Ascent

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

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \tag{2.13}$$

where the measure of performance here is the expected reward:

$$\mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x),$$

and the measure of the increment's effect is the *partial derivative* of this performance measure with respect to the action preference. Of course, it is not possible to implement gradient ascent exactly in our case because by assumption we do not know the $q_*(x)$, but in fact the updates of our algorithm (2.12) are equal to (2.13) in expected value, making the algorithm an instance of *stochastic gradient ascent*. The calculations showing this require only beginning calculus, but take several

steps. First we take a closer look at the exact performance gradient:

$$\begin{aligned}\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)}\left[\sum_x \pi_t(x) q_*(x)\right] \\ &= \sum_x q_*(x)\frac{\partial \pi_t(x)}{\partial H_t(a)} \\ &= \sum_x \big(q_*(x) - B_t\big)\frac{\partial \pi_t(x)}{\partial H_t(a)},\end{aligned}$$

where B_t, called the *baseline* , can be any scalar that does not depend on x. We can include a baseline here without changing the equality because the gradient sums to zero over all the actions, $\sum_x \frac{\partial \pi_t(x)}{\partial H_t(a)} = 0$—as $H_t(a)$ is changed, some actions' probabilities go up and some go down, but the sum of the changes must be zero because the sum of the probabilities is always one.

Next we multiply each term of the sum by $\pi_t(x)/\pi_t(x)$:

$$\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} = \sum_x \pi_t(x)\big(q_*(x) - B_t\big)\frac{\partial \pi_t(x)}{\partial H_t(a)}/\pi_t(x).$$

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

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

where here we have chosen the baseline $B_t = \bar{R}_t$ and substituted R_t for $q_*(A_t)$, which is permitted because $\mathbb{E}[R_t|A_t] = q_*(A_t)$. Shortly we will establish that $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)\big(\mathbb{1}_{a=x} - \pi_t(a)\big)$, where $\mathbb{1}_{a=x}$ is defined to be 1 if $a = x$, else 0. Assuming that for now, we have

$$\begin{aligned}&= \mathbb{E}\big[\big(R_t - \bar{R}_t\big)\pi_t(A_t)\big(\mathbb{1}_{a=A_t} - \pi_t(a)\big)/\pi_t(A_t)\big] \\ &= \mathbb{E}\big[\big(R_t - \bar{R}_t\big)\big(\mathbb{1}_{a=A_t} - \pi_t(a)\big)\big].\end{aligned}$$

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 (2.13) yields:

$$H_{t+1}(a) = H_t(a) + \alpha\big(R_t - \bar{R}_t\big)\big(\mathbb{1}_{a=A_t} - \pi_t(a)\big), \qquad \text{for all } a,$$

which you may recognize as being equivalent to our original algorithm (2.12).

Thus it remains only to show that $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)\big(\mathbb{1}_{a=x} - \pi_t(a)\big)$, as we assumed. 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

$$\begin{aligned}\frac{\partial \pi_t(x)}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)}\pi_t(x) \\ &= \frac{\partial}{\partial H_t(a)}\left[\frac{e^{H_t(x)}}{\sum_{y=1}^k e^{H_t(y)}}\right] \\ &= \frac{\frac{\partial e^{H_t(x)}}{\partial H_t(a)}\sum_{y=1}^k e^{H_t(y)} - e^{H_t(x)}\frac{\partial \sum_{y=1}^k e^{H_t(y)}}{\partial H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)}\right)^2} \qquad \text{(by the quotient rule)} \\ &= \frac{\mathbb{1}_{a=x}e^{H_t(x)}\sum_{y=1}^k e^{H_t(y)} - e^{H_t(x)}e^{H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)}\right)^2} \qquad \text{(because } \tfrac{\partial e^x}{\partial x} = e^x\text{)} \\ &= \frac{\mathbb{1}_{a=x}e^{H_t(x)}}{\sum_{y=1}^k e^{H_t(y)}} - \frac{e^{H_t(x)}e^{H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)}\right)^2} \\ &= \mathbb{1}_{a=x}\pi_t(x) - \pi_t(x)\pi_t(a) \\ &= \pi_t(x)\big(\mathbb{1}_{a=x} - \pi_t(a)\big). \qquad \text{Q.E.D.}\end{aligned}$$

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 any properties of the reward baseline other than that it does not depend on the selected action. For example, we could have set it to zero, or to 1000, and the algorithm would still be 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 2.5). Choosing it as the average of the rewards may not be the very best, but it is simple and works well in practice.

2.9 Associative Search (Contextual Bandits)

So far in this chapter we have considered only nonassociative tasks, that is, 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 k-armed bandit tasks, and that on each step you confront one of these chosen at random. Thus, the bandit task changes randomly from step to step. This would appear to you as a single, nonstationary k-armed bandit task whose true action values change randomly from step to step. 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, select arm 1; if green, select 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 to *search* for the best actions, and *association* of these actions with the situations in which they are best. Associative search tasks are often now called *contextual bandits* in the literature. Associative search tasks are intermediate between the k-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 k-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.

Exercise 2.10 Suppose you face a 2-armed bandit task whose true action values change randomly from time step to time step. Specifically, suppose that, for any time step, the true values of actions 1 and 2 are respectively 0.1 and 0.2 with probability 0.5 (case A), and 0.9 and 0.8 with probability 0.5 (case B). If you are not able to tell which case you face at any step, what is the best expectation of success you can achieve and how should you behave to achieve it? Now suppose that on each step you are told whether you are facing case A or case B (although you still don't know the true action values). This is an associative search task. What is the best expectation of success you can achieve in this task, and how should you behave to achieve it? □

2.10 Summary

We have presented in this chapter several simple ways of balancing exploration and exploitation. The ε-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 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 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, to produce a *learning curve* for that algorithm and parameter setting. If we plotted learning curves for all algorithms and all parameter settings, then the graph would be too complex and crowded to make clear comparisons. 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 curve. Figure 2.6 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. This kind of graph is called a *parameter study*. 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 small. In assessing

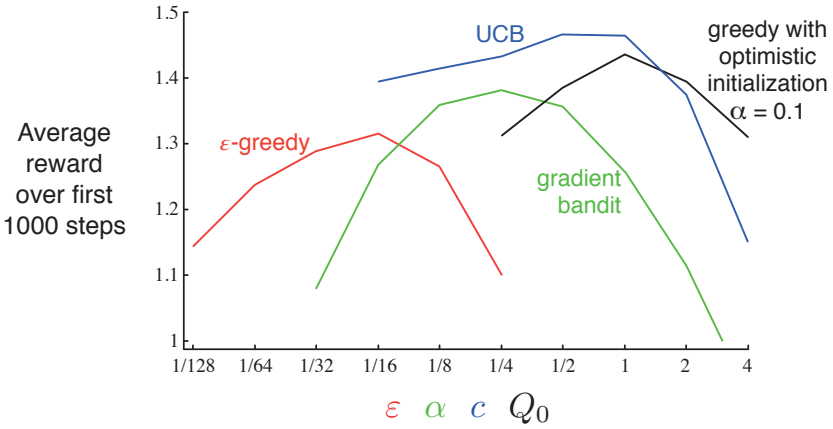


Figure 2.6: A parameter study of the various bandit algorithms presented in this chapter. Each point is the average reward obtained over 1000 steps with a particular algorithm at a particular setting of its parameter.

a 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.

One well-studied approach to balancing exploration and exploitation in k-armed bandit problems is to compute a special kind of action value called a *Gittins index*. In certain important special cases, this computation is tractable and leads directly to optimal solutions, although it does require complete knowledge of the prior distribution of possible problems, which we generally assume is not available. In addition, neither the theory nor the computational tractability of this approach appear to generalize to the full reinforcement learning problem that we consider in the rest of the book.

The Gittins-index approach is an instance of *Bayesian* methods, which assume a known initial distribution over the action values and then update the distribution exactly after each step (assuming that the true action values are stationary). In general, the update computations can be very complex, but for certain special distributions (called *conjugate priors*) they are easy. One possibility is to then select actions at each step according to their posterior probability of being the best action. This method, sometimes called *posterior sampling* or *Thompson sampling*, often performs similarly to the best of the distribution-free methods we have presented in this chapter.

In the Bayesian setting it is even conceivable to compute the *optimal* balance between exploration and exploitation. One can compute for any possible action the probability of each possible immediate reward and the resultant posterior distributions over action values. This evolving distribution becomes the *information state* of the problem. Given a horizon, say of 1000 steps, one can consider all possible actions, all possible resulting rewards, all possible next actions, all next rewards, and so on for all 1000 steps. 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 were only two actions and two rewards, the tree would have 2^{2000} leaves. It is generally not feasible to perform this immense computation exactly, but perhaps it could be approximated efficiently. This approach would effectively turn the bandit problem into an instance of the full reinforcement learning problem. In the end, we may be able to use approximate reinforcement learning methods such as those presented in Part II of this book to approach this optimal solution. But that is a topic for research and beyond the scope of this introductory book.

Exercise 2.11 (programming) Make a figure analogous to Figure 2.6 for the nonstationary case outlined in Exercise 2.5. Include the constant-step-size ε-greedy algorithm with $\alpha = 0.1$. Use runs of 200,000 steps and, as a performance measure for each algorithm and parameter setting, use the average reward over the last 100,000 steps. □

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 Robbins (1952), and studied by Bellman (1956). Berry and Fristedt (1985) provide an extensive treatment of bandit problems from the perspective of statistics. Narendra and Thathachar (1989) treat bandit problems from the engineering perspective, providing a good discussion of the various theoretical traditions that have focused on them. In psychology, bandit problems have played roles in statistical learning theory (e.g., Bush and Mosteller, 1955; Estes, 1950).

The term *greedy* is often used in the heuristic search literature (e.g., Pearl, 1984). The conflict between exploration and exploitation is known in control engineering as the conflict between identification (or estimation) and control (e.g., Witten, 1976b). Feldbaum (1965) called it the *dual control* problem, referring to the need to solve the two problems of identification and control simultaneously when trying to control a system under uncertainty. In discussing aspects of genetic algorithms, Holland (1975) emphasized the importance of this conflict, referring to it as the conflict between the need to exploit and the need for new information.

2.2 Action-value methods for our k-armed bandit problem were first proposed by Thathachar and Sastry (1985). These are often called *estimator algorithms* in the learning automata literature. The term *action value* is due to Watkins (1989). The first to use ε-greedy methods may also have been Watkins (1989, p. 187), but the idea is so simple that some earlier use seems likely.

2.4–5 This material falls under the general heading of stochastic iterative algorithms, which is well covered by Bertsekas and Tsitsiklis (1996).

2.6 Optimistic initialization was used in reinforcement learning by Sutton (1996).

2.7 Early work on using estimates of the upper confidence bound to select actions was done by Lai and Robbins (1985), Kaelbling (1993b), and Agrawal (1995). The UCB algorithm we present here is called UCB1 in the literature and was first developed by Auer, Cesa-Bianchi and Fischer (2002).

2.8 Gradient bandit algorithms are a special case of the gradient-based reinforcement learning algorithms introduced by Williams (1992), and that later developed into the actor–critic and policy-gradient algorithms that we treat later in this book. Our development here was influenced by that by Balaraman Ravindran (personal

communication). Further discussion of the choice of baseline is provided there and by Greensmith, Bartlett, and Baxter (2002, 2004) and Dick (2015). Early systematic studies of algorithms like this were done by Sutton (1984).

The term *soft-max* for the action selection rule (2.11) is due to Bridle (1990). This rule appears to have been first proposed by Luce (1959).

2.9 The term *associative search* and the corresponding problem were introduced by Barto, Sutton, and Brouwer (1981). The term *associative reinforcement learning* has also been used for associative search (Barto and Anandan, 1985), but we prefer to reserve that term as a synonym for the full reinforcement learning problem (as in Sutton, 1984). (And, as we noted, the modern literature also uses the term "contextual bandits" for this problem.) We note that Thorndike's Law of Effect (quoted in Chapter 1) describes associative search by referring to the formation of associative links between situations (states) and actions. According to the terminology of operant, or instrumental, conditioning (e.g., Skinner, 1938), a discriminative stimulus is a stimulus that signals the presence of a particular reinforcement contingency. In our terms, different discriminative stimuli correspond to different states.

2.10 Bellman (1956) was the first to show how dynamic programming could be used to compute the optimal balance between exploration and exploitation within a Bayesian formulation of the problem. The Gittins index approach is due to Gittins and Jones (1974). Duff (1995) showed how it is possible to learn Gittins indices for bandit problems through reinforcement learning. The survey by Kumar (1985) provides a good discussion of Bayesian and non-Bayesian approaches to these problems. The term *information state* comes from the literature on partially observable MDPs; see, e.g., Lovejoy (1991).

Other theoretical research focuses on the efficiency of exploration, usually expressed as how quickly an algorithm can approach an optimal decision-making policy. One way to formalize exploration efficiency is by adapting to reinforcement learning the notion of *sample complexity* for a supervised learning algorithm, which is the number of training examples the algorithm needs to attain a desired degree of accuracy in learning the target function. A definition of the sample complexity of exploration for a reinforcement learning algorithm is the number of time steps in which the algorithm does not select near-optimal actions (Kakade, 2003). Li (2012) discusses this and several other approaches in a survey of theoretical approaches to exploration efficiency in reinforcement learning. A thorough modern treatment of Thompson sampling is provided by Russo, Van Roy, Kazerouni, Osband, and Wen (2018).

Chapter 3

Finite Markov Decision Processes

In this chapter we introduce the formal problem of finite Markov decision processes, or finite MDPs, which we try to solve in the rest of the book. This problem involves evaluative feedback, as in bandits, but also an associative aspect—choosing different actions in different situations. MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward. Whereas in bandit problems we estimated the value $q_*(a)$ of each action a, in MDPs we estimate the value $q_*(s,a)$ of each action a in each state s, or we estimate the value $v_*(s)$ of each state given optimal action selections. These state-dependent quantities are essential to accurately assigning credit for long-term consequences to individual action selections.

MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. We introduce key elements of the problem's mathematical structure, such as returns, value functions, and Bellman equations. We try to convey the wide range of applications that can be formulated as finite MDPs. As in all of artificial intelligence, there is a tension between breadth of applicability and mathematical tractability. In this chapter we introduce this tension and discuss some of the trade-offs and challenges that it implies. Some ways in which reinforcement learning can be taken beyond MDPs are treated in Chapter 17.

3.1 The Agent–Environment Interface

MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to

these actions and presenting new situations to the agent.[1] The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions.

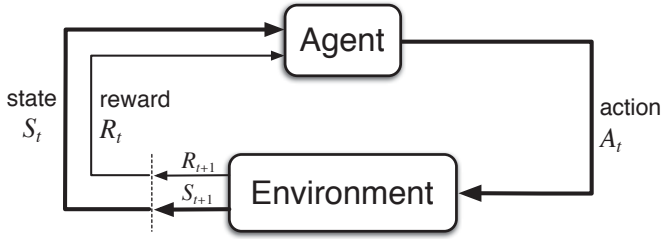


Figure 3.1: The agent–environment interaction in a Markov decision process.

More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$.[2] At each time step t, the agent receives some representation of the environment's *state*, $S_t \in \mathcal{S}$, and on that basis selects an *action*, $A_t \in \mathcal{A}(s)$.[3] One time step later, in part as a consequence of its action, the agent receives a numerical *reward*, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, S_{t+1}.[4] The MDP and agent together thereby give rise to a sequence or *trajectory* that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{3.1}$$

In a *finite* MDP, the sets of states, actions, and rewards ($\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$) all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p(s', r \,|\, s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}, \tag{3.2}$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The function p defines the *dynamics* of the MDP. The dot over the equals sign in the equation reminds us that it is a definition (in this case of the function p) rather than a fact that follows from previous definitions. The dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is an ordinary deterministic function of four arguments. The '|' in the middle of it comes from the notation for conditional probability,

[1] We use the terms *agent*, *environment*, and *action* instead of the engineers' terms *controller*, *controlled system* (or *plant*), and *control signal* because they are meaningful to a wider audience.

[2] We restrict attention to discrete time to keep things as simple as possible, even though many of the ideas can be extended to the continuous-time case (e.g., see Bertsekas and Tsitsiklis, 1996; Doya, 1996).

[3] To simplify notation, we sometimes assume the special case in which the action set is the same in all states and write it simply as $\mathcal{A}$.

[4] We use R_{t+1} instead of R_t to denote the reward due to A_t because it emphasizes that the next reward and next state, R_{t+1} and S_{t+1}, are jointly determined. Unfortunately, both conventions are widely used in the literature.

but here it just reminds us that p specifies a probability distribution for each choice of s and a, that is, that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s). \tag{3.3}$$

In a *Markov* decision process, the probabilities given by p completely characterize the environment's dynamics. That is, the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, S_{t-1} and A_{t-1}, and, given them, not at all on earlier states and actions. This is best viewed a restriction not on the decision process, but on the *state*. The state must include information about all aspects of the past agent–environment interaction that make a difference for the future. If it does, then the state is said to have the *Markov property*. We will assume the Markov property throughout this book, though starting in Part II we will consider approximation methods that do not rely on it, and in Chapter 17 we consider how a Markov state can be learned and constructed from non-Markov observations.

From the four-argument dynamics function, p, one can compute anything else one might want to know about the environment, such as the *state-transition probabilities* (which we denote, with a slight abuse of notation, as a three-argument function $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$),

$$p(s' \,|\, s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a). \tag{3.4}$$

We can also compute the expected rewards for state–action pairs as a two-argument function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$:

$$r(s, a) \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \,|\, s, a), \tag{3.5}$$

and the expected rewards for state–action–next-state triples as a three-argument function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$,

$$r(s, a, s') \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r \,|\, s, a)}{p(s' \,|\, s, a)}. \tag{3.6}$$

In this book, we usually use the four-argument p function (3.2), but each of these other notations are also occasionally convenient.

The MDP framework is abstract and flexible and can be applied to many different problems in many different ways. For example, the time steps need not refer to fixed intervals of real time; they can refer to arbitrary successive stages of decision making and acting. The actions can be low-level controls, such as the voltages applied to the motors of a robot arm, or high-level decisions, such as whether or not to have lunch or to go to graduate school. Similarly, the states can take a wide variety of forms. They can be completely determined by low-level sensations, such as direct sensor readings, or they can be more high-level and abstract, such as symbolic descriptions of objects in a room. Some of what makes up a state could be based on memory of past sensations or

even be entirely mental or subjective. For example, an agent could be in the state of not being sure where an object is, or of having just been surprised in some clearly defined sense. Similarly, some actions might be totally mental or computational. For example, some actions might control what an agent chooses to think about, or where it focuses its attention. In general, actions can be any decisions we want to learn how to make, and the states can be anything we can know that might be useful in making them.

In particular, the boundary between agent and environment is typically not the same as the physical boundary of a robot's or animal's body. Usually, the boundary is drawn closer to the agent than that. For example, the motors and mechanical linkages of a robot and its sensing hardware should usually be considered parts of the environment rather than parts of the agent. Similarly, if we apply the MDP framework to a person or animal, the muscles, skeleton, and sensory organs should be considered part of the environment. Rewards, too, presumably are computed inside the physical bodies of natural and artificial learning systems, but are considered external to the agent.

The general rule we follow is that anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment. We do not assume that everything in the environment is unknown to the agent. For example, the agent often knows quite a bit about how its rewards are computed as a function of its actions and the states in which they are taken. But we always consider the reward computation to be external to the agent because it defines the task facing the agent and thus must be beyond its ability to change arbitrarily. In fact, in some cases the agent may know *everything* about how its environment works and still face a difficult reinforcement learning task, just as we may know exactly how a puzzle like Rubik's cube works, but still be unable to solve it. The agent–environment boundary represents the limit of the agent's *absolute control*, not of its knowledge.

The agent–environment boundary can be located at different places for different purposes. In a complicated robot, many different agents may be operating at once, each with its own boundary. For example, one agent may make high-level decisions which form part of the states faced by a lower-level agent that implements the high-level decisions. In practice, the agent–environment boundary is determined once one has selected particular states, actions, and rewards, and thus has identified a specific decision making task of interest.

The MDP framework is a considerable abstraction of the problem of goal-directed learning from interaction. It proposes that whatever the details of the sensory, memory, and control apparatus, and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal to represent the choices made by the agent (the actions), one signal to represent the basis on which the choices are made (the states), and one signal to define the agent's goal (the rewards). This framework may not be sufficient to represent all decision-learning problems usefully, but it has proved to be widely useful and applicable.

Of course, the particular states and actions vary greatly from task to task, and how they are represented can strongly affect performance. In reinforcement learning, as in other kinds of learning, such representational choices are at present more art than science.

In this book we offer some advice and examples regarding good ways of representing states and actions, but our primary focus is on general principles for learning how to behave once the representations have been selected.

Example 3.1: Bioreactor Suppose reinforcement learning is being applied to determine moment-by-moment temperatures and stirring rates for a bioreactor (a large vat of nutrients and bacteria used to produce useful chemicals). The actions in such an application might be target temperatures and target stirring rates that are passed to lower-level control systems that, in turn, directly activate heating elements and motors to attain the targets. The states are likely to be thermocouple and other sensory readings, perhaps filtered and delayed, plus symbolic inputs representing the ingredients in the vat and the target chemical. The rewards might be moment-by-moment measures of the rate at which the useful chemical is produced by the bioreactor. Notice that here each state is a list, or vector, of sensor readings and symbolic inputs, and each action is a vector consisting of a target temperature and a stirring rate. It is typical of reinforcement learning tasks to have states and actions with such structured representations. Rewards, on the other hand, are always single numbers. ■

Example 3.2: Pick-and-Place Robot Consider using reinforcement learning to control the motion of a robot arm in a repetitive pick-and-place task. If we want to learn movements that are fast and smooth, the learning agent will have to control the motors directly and have low-latency information about the current positions and velocities of the mechanical linkages. The actions in this case might be the voltages applied to each motor at each joint, and the states might be the latest readings of joint angles and velocities. The reward might be +1 for each object successfully picked up and placed. To encourage smooth movements, on each time step a small, negative reward can be given as a function of the moment-to-moment "jerkiness" of the motion. ■

Exercise 3.1 Devise three example tasks of your own that fit into the MDP framework, identifying for each its states, actions, and rewards. Make the three examples as *different* from each other as possible. The framework is abstract and flexible and can be applied in many different ways. Stretch its limits in some way in at least one of your examples. □

Exercise 3.2 Is the MDP framework adequate to usefully represent *all* goal-directed learning tasks? Can you think of any clear exceptions? □

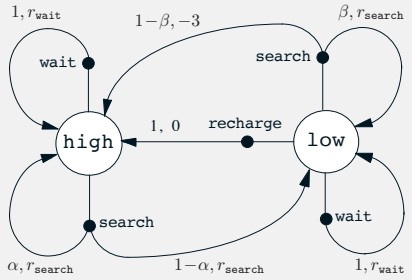
Exercise 3.3 Consider the problem of driving. You could define the actions in terms of the accelerator, steering wheel, and brake, that is, where your body meets the machine. Or you could define them farther out—say, where the rubber meets the road, considering your actions to be tire torques. Or you could define them farther in—say, where your brain meets your body, the actions being muscle twitches to control your limbs. Or you could go to a really high level and say that your actions are your choices of *where* to drive. What is the right level, the right place to draw the line between agent and environment? On what basis is one location of the line to be preferred over another? Is there any fundamental reason for preferring one location over another, or is it a free choice? □

Example 3.3 Recycling Robot

A mobile robot has the job of collecting empty soda cans in an office environment. It has sensors for detecting cans, and an arm and gripper that can pick them up and place them in an onboard bin; it runs on a rechargeable battery. The robot's control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper. High-level decisions about how to search for cans are made by a reinforcement learning agent based on the current charge level of the battery. To make a simple example, we assume that only two charge levels can be distinguished, comprising a small state set $\mathcal{S} = \{\texttt{high}, \texttt{low}\}$. In each state, the agent can decide whether to (1) actively `search` for a can for a certain period of time, (2) remain stationary and `wait` for someone to bring it a can, or (3) head back to its home base to `recharge` its battery. When the energy level is `high`, recharging would always be foolish, so we do not include it in the action set for this state. The action sets are then $\mathcal{A}(\texttt{high}) = \{\texttt{search}, \texttt{wait}\}$ and $\mathcal{A}(\texttt{low}) = \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}$.

The rewards are zero most of the time, but become positive when the robot secures an empty can, or large and negative if the battery runs all the way down. The best way to find cans is to actively search for them, but this runs down the robot's battery, whereas waiting does not. Whenever the robot is searching, the possibility exists that its battery will become depleted. In this case the robot must shut down and wait to be rescued (producing a low reward). If the energy level is `high`, then a period of active search can always be completed without risk of depleting the battery. A period of searching that begins with a `high` energy level leaves the energy level `high` with probability α and reduces it to `low` with probability $1 - \alpha$. On the other hand, a period of searching undertaken when the energy level is `low` leaves it `low` with probability β and depletes the battery with probability $1 - \beta$. In the latter case, the robot must be rescued, and the battery is then recharged back to `high`. Each can collected by the robot counts as a unit reward, whereas a reward of -3 results whenever the robot has to be rescued. Let $r_{\texttt{search}}$ and $r_{\texttt{wait}}$, with $r_{\texttt{search}} > r_{\texttt{wait}}$, respectively denote the expected number of cans the robot will collect (and hence the expected reward) while searching and while waiting. Finally, suppose that no cans can be collected during a run home for recharging, and that no cans can be collected on a step in which the battery is depleted. This system is then a finite MDP, and we can write down the transition probabilities and the expected rewards, with dynamics as indicated in the table on the left:

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1 - \alpha$	$r_{\texttt{search}}$
low	search	high	$1 - \beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	$r_{\texttt{wait}}$
low	wait	high	0	$r_{\texttt{wait}}$
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	0



Note that there is a row in the table for each possible combination of current state, s, action, $a \in \mathcal{A}(s)$, and next state, s'. Another useful way of summarizing the dynamics of a finite MDP is as a *transition graph* as shown above on the right. There are two kinds of

nodes: *state nodes* and *action nodes*. There is a state node for each possible state (a large open circle labeled by the name of the state), and an action node for each state–action pair (a small solid circle labeled by the name of the action and connected by a line to the state node). Starting in state s and taking action a moves you along the line from state node s to action node (s, a). Then the environment responds with a transition to the next state's node via one of the arrows leaving action node (s, a). Each arrow corresponds to a triple (s, s', a), where s' is the next state, and we label the arrow with the transition probability, $p(s'|s, a)$, and the expected reward for that transition, $r(s, a, s')$. Note that the transition probabilities labeling the arrows leaving an action node always sum to 1.

Exercise 3.4 Give a table analogous to that in Example 3.3, but for $p(s', r|s, a)$. It should have columns for s, a, s', r, and $p(s', r|s, a)$, and a row for every 4-tuple for which $p(s', r|s, a) > 0$. □

3.2 Goals and Rewards

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the *reward*, passing from the environment to the agent. At each time step, the reward is a simple number, $R_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run. We can clearly state this informal idea as the *reward hypothesis*:

> That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning.

Although formulating goals in terms of reward signals might at first appear limiting, in practice it has proved to be flexible and widely applicable. The best way to see this is to consider examples of how it has been, or could be, used. For example, to make a robot learn to walk, researchers have provided reward on each time step proportional to the robot's forward motion. In making a robot learn how to escape from a maze, the reward is often -1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible. To make a robot learn to find and collect empty soda cans for recycling, one might give it a reward of zero most of the time, and then a reward of $+1$ for each can collected. One might also want to give the robot negative rewards when it bumps into things or when somebody yells at it. For an agent to learn to play checkers or chess, the natural rewards are $+1$ for winning, -1 for losing, and 0 for drawing and for all nonterminal positions.

You can see what is happening in all of these examples. The agent always learns to maximize its reward. If we want it to do something for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It is thus critical that the rewards we set up truly indicate what we want accomplished.

In particular, the reward signal is not the place to impart to the agent prior knowledge about *how* to achieve what we want it to do.[5] For example, a chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponent's pieces or gaining control of the center of the board. If achieving these sorts of subgoals were rewarded, then the agent might find a way to achieve them without achieving the real goal. For example, it might find a way to take the opponent's pieces even at the cost of losing the game. The reward signal is your way of communicating to the robot *what* you want it to achieve, not *how* you want it achieved.[6]

3.3 Returns and Episodes

So far we have discussed the objective of learning informally. We have said that the agent's goal is to maximize the cumulative reward it receives in the long run. How might this be defined formally? If the sequence of rewards received after time step t is denoted $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$, then what precise aspect of this sequence do we wish to maximize? In general, we seek to maximize the *expected return*, where the return, denoted G_t, is defined as some specific function of the reward sequence. In the simplest case the return is the sum of the rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{3.7}$$

where T is a final time step. This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent–environment interaction breaks naturally into subsequences, which we call *episodes*,[7] such as plays of a game, trips through a maze, or any sort of repeated interaction. Each episode ends in a special state called the *terminal state*, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. Even if you think of episodes as ending in different ways, such as winning and losing a game, the next episode begins independently of how the previous one ended. Thus the episodes can all be considered to end in the same terminal state, with different rewards for the different outcomes. Tasks with episodes of this kind are called *episodic tasks*. In episodic tasks we sometimes need to distinguish the set of all nonterminal states, denoted $\mathcal{S}$, from the set of all states plus the terminal state, denoted $\mathcal{S}^+$. The time of termination, T, is a random variable that normally varies from episode to episode.

On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. For example, this would be the natural way to formulate an on-going process-control task, or an application to a robot with a long life span. We call these *continuing tasks*. The return formulation (3.7) is problematic for continuing tasks because the final time step would be $T = \infty$, and the return, which is what we are trying to maximize, could itself easily

[5] Better places for imparting this kind of prior knowledge are the initial policy or initial value function, or in influences on these.

[6] Section 17.4 delves further into the issue of designing effective reward signals.

[7] Episodes are sometimes called "trials" in the literature.

be infinite. (For example, suppose the agent receives a reward of +1 at each time step.) Thus, in this book we usually use a definition of return that is slightly more complex conceptually but much simpler mathematically.

The additional concept that we need is that of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses A_t to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{3.8}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.

The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the infinite sum in (3.8) has a finite value as long as the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose A_t so as to maximize only R_{t+1}. If each of the agent's actions happened to influence only the immediate reward, not future rewards as well, then a myopic agent could maximize (3.8) by separately maximizing each immediate reward. But in general, acting to maximize immediate reward can reduce access to future rewards so that the return is reduced. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma\big(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots\big) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.9}$$

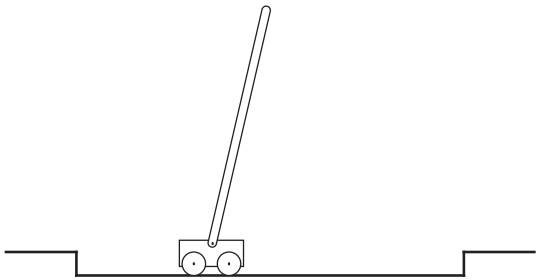
Note that this works for all time steps $t < T$, even if termination occurs at $t+1$, if we define $G_T = 0$. This often makes it easy to compute returns from reward sequences.

Note that although the return (3.8) is a sum of an infinite number of terms, it is still finite if the reward is nonzero and constant—if $\gamma < 1$. For example, if the reward is a constant +1, then the return is

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}. \tag{3.10}$$

Exercise 3.5 The equations in Section 3.1 are for the continuing case and need to be modified (very slightly) to apply to episodic tasks. Show that you know the modifications needed by giving the modified version of (3.3). □

Example 3.4: Pole-Balancing



The objective in this task is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over: A failure is said to occur if the pole falls past a given angle from vertical or if the cart runs off the track. The pole is reset to vertical after each failure. This task could be treated as episodic, where the natural episodes are the repeated attempts to balance the pole. The reward in this case could be +1 for every time step on which failure did not occur, so that the return at each time would be the number of steps until failure. In this case, successful balancing forever would mean a return of infinity. Alternatively, we could treat pole-balancing as a continuing task, using discounting. In this case the reward would be −1 on each failure and zero at all other times. The return at each time would then be related to $-\gamma^K$, where K is the number of time steps before failure. In either case, the return is maximized by keeping the pole balanced for as long as possible. ■

Exercise 3.6 Suppose you treated pole-balancing as an episodic task but also used discounting, with all rewards zero except for −1 upon failure. What then would the return be at each time? How does this return differ from that in the discounted, continuing formulation of this task? □

Exercise 3.7 Imagine that you are designing a robot to run a maze. You decide to give it a reward of +1 for escaping from the maze and a reward of zero at all other times. The task seems to break down naturally into episodes—the successive runs through the maze—so you decide to treat it as an episodic task, where the goal is to maximize expected total reward (3.7). After running the learning agent for a while, you find that it is showing no improvement in escaping from the maze. What is going wrong? Have you effectively communicated to the agent what you want it to achieve? □

Exercise 3.8 Suppose $\gamma = 0.5$ and the following sequence of rewards is received $R_1 = -1$, $R_2 = 2$, $R_3 = 6$, $R_4 = 3$, and $R_5 = 2$, with $T = 5$. What are G_0, G_1, ..., G_5? Hint: Work backwards. □

Exercise 3.9 Suppose $\gamma = 0.9$ and the reward sequence is $R_1 = 2$ followed by an infinite sequence of 7s. What are G_1 and G_0? □

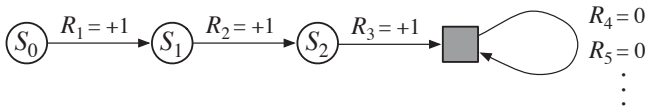
Exercise 3.10 Prove the second equality in (3.10). □

3.4 Unified Notation for Episodic and Continuing Task

In the preceding section we described two kinds of reinforcement learning tasks, one in which the agent–environment interaction naturally breaks down into a sequence of separate episodes (episodic tasks), and one in which it does not (continuing tasks). The former case is mathematically easier because each action affects only the finite number of rewards subsequently received during the episode. In this book we consider sometimes one kind of problem and sometimes the other, but often both. It is therefore useful to establish one notation that enables us to talk precisely about both cases simultaneously.

To be precise about episodic tasks requires some additional notation. Rather than one long sequence of time steps, we need to consider a series of episodes, each of which consists of a finite sequence of time steps. We number the time steps of each episode starting anew from zero. Therefore, we have to refer not just to S_t, the state representation at time t, but to $S_{t,i}$, the state representation at time t of episode i (and similarly for $A_{t,i}$, $R_{t,i}$, $\pi_{t,i}$, T_i, etc.). However, it turns out that when we discuss episodic tasks we almost never have to distinguish between different episodes. We are almost always considering a particular single episode, or stating something that is true for all episodes. Accordingly, in practice we almost always abuse notation slightly by dropping the explicit reference to episode number. That is, we write S_t to refer to $S_{t,i}$, and so on.

We need one other convention to obtain a single notation that covers both episodic and continuing tasks. We have defined the return as a sum over a finite number of terms in one case (3.7) and as a sum over an infinite number of terms in the other (3.8). These two can be unified by considering episode termination to be the entering of a special *absorbing state* that transitions only to itself and that generates only rewards of zero. For example, consider the state transition diagram:



Here the solid square represents the special absorbing state corresponding to the end of an episode. Starting from S_0, we get the reward sequence $+1, +1, +1, 0, 0, 0, \ldots$. Summing these, we get the same return whether we sum over the first T rewards (here $T = 3$) or over the full infinite sequence. This remains true even if we introduce discounting. Thus, we can define the return, in general, according to (3.8), using the convention of omitting episode numbers when they are not needed, and including the possibility that $\gamma = 1$ if the sum remains defined (e.g., because all episodes terminate). Alternatively, we can write

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{3.11}$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both). We use these conventions throughout the rest of the book to simplify notation and to express the close parallels between episodic and continuing tasks. (Later, in Chapter 10, we will introduce a formulation that is both continuing and undiscounted.)

3.5 Policies and Value Functions

Almost all reinforcement learning algorithms involve estimating *value functions*—functions of states (or of state–action pairs) that estimate *how good* it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.

Formally, a *policy* is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Like p, π is an ordinary function; the "|" in the middle of $\pi(a|s)$ merely reminds that it defines a probability distribution over $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$. Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.

Exercise 3.11 If the current state is S_t, and actions are selected according to stochastic policy π, then what is the expectation of R_{t+1} in terms of π and the four-argument function p (3.2)? □

The *value function* of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S}, \tag{3.12}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. Note that the value of the terminal state, if any, is always zero. We call the function v_π the *state-value function for policy* π.

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

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]. \tag{3.13}$$

We call q_π the *action-value function for policy* π.

Exercise 3.12 Give an equation for v_π in terms of q_π and π. □

Exercise 3.13 Give an equation for q_π in terms of v_π and the four-argument p. □

The value functions v_π and q_π can be estimated from experience. For example, if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state's value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity. If separate averages are kept for each action taken in each state, then these averages will similarly converge to the action values, $q_\pi(s, a)$. We call estimation methods of this kind *Monte Carlo methods* because they involve averaging over many random samples of actual returns.

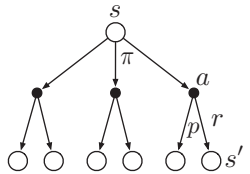
These kinds of methods are presented in Chapter 5. Of course, if there are very many states, then it may not be practical to keep separate averages for each state individually. Instead, the agent would have to maintain v_π and q_π as parameterized functions (with fewer parameters than states) and adjust the parameters to better match the observed returns. This can also produce accurate estimates, although much depends on the nature of the parameterized function approximator. These possibilities are discussed in Part II of the book.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return (3.9). For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \qquad \text{(by (3.9))} \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S}, \end{aligned} \tag{3.14}$$

where it is implicit that the actions, a, are taken from the set $\mathcal{A}(s)$, that the next states, s', are taken from the set $\mathcal{S}$ (or from $\mathcal{S}^+$ in the case of an episodic problem), and that the rewards, r, are taken from the set $\mathcal{R}$. Note also how in the last equation we have merged the two sums, one over all the values of s' and the other over all the values of r, into one sum over all the possible values of both. We use this kind of merged sum often to simplify formulas. Note how the final expression can be read easily as an expected value. It is really a sum over all values of the three variables, a, s', and r. For each triple, we compute its probability, $\pi(a|s)p(s', r \,|\, s, a)$, weight the quantity in brackets by that probability, then sum over all possibilities to get an expected value.

Equation (3.14) is the *Bellman equation for* v_π. It expresses a relationship between the value of a state and the values of its successor states. Think of looking ahead from a state to its possible successor states, as suggested by the diagram to the right. Each open circle represents a state and each solid circle represents a state–action pair. Starting from state s, the root node at the top, the agent could take any of some set of actions—three are shown in the diagram—based on its policy π. From each of these, the environment could respond with one of several next states, s' (two are shown in the figure), along with a reward, r, depending on its dynamics given by the function p. The Bellman equation (3.14) averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.

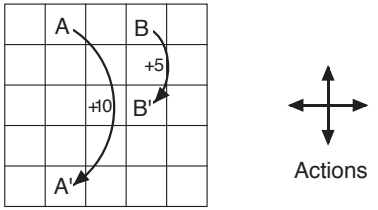


Backup diagram for v_π

The value function v_π is the unique solution to its Bellman equation. We show in subsequent chapters how this Bellman equation forms the basis of a number of ways to

compute, approximate, and learn v_π. We call diagrams like that above *backup diagrams* because they diagram relationships that form the basis of the update or *backup* operations that are at the heart of reinforcement learning methods. These operations transfer value information *back* to a state (or a state–action pair) from its successor states (or state–action pairs). We use backup diagrams throughout the book to provide graphical summaries of the algorithms we discuss. (Note that, unlike transition graphs, the state nodes of backup diagrams do not necessarily represent distinct states; for example, a state might be its own successor.)

Example 3.5: Gridworld Figure 3.2 (left) shows a rectangular gridworld representation of a simple finite MDP. The cells of the grid correspond to the states of the environment. At each cell, four actions are possible: `north`, `south`, `east`, and `west`, which deterministically cause the agent to move one cell in the respective direction on the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From state A, all four actions yield a reward of $+10$ and take the agent to A′. From state B, all actions yield a reward of $+5$ and take the agent to B′.



Actions

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Figure 3.2: Gridworld example: exceptional reward dynamics (left) and state-value function for the equiprobable random policy (right).

Suppose the agent selects all four actions with equal probability in all states. Figure 3.2 (right) shows the value function, v_π, for this policy, for the discounted reward case with $\gamma = 0.9$. This value function was computed by solving the system of linear equations (3.14). Notice the negative values near the lower edge; these are the result of the high probability of hitting the edge of the grid there under the random policy. State A is the best state to be in under this policy, but its expected return is less than 10, its immediate reward, because from A the agent is taken to A′, from which it is likely to run into the edge of the grid. State B, on the other hand, is valued more than 5, its immediate reward, because from B the agent is taken to B′, which has a positive value. From B′ the expected penalty (negative reward) for possibly running into an edge is more than compensated for by the expected gain for possibly stumbling onto A or B. ■

Exercise 3.14 The Bellman equation (3.14) must hold for each state for the value function v_π shown in Figure 3.2 (right) of Example 3.5. Show numerically that this equation holds for the center state, valued at $+0.7$, with respect to its four neighboring states, valued at $+2.3$, $+0.4$, -0.4, and $+0.7$. (These numbers are accurate only to one decimal place.) □

Exercise 3.15 In the gridworld example, rewards are positive for goals, negative for running into the edge of the world, and zero the rest of the time. Are the signs of these

rewards important, or only the intervals between them? Prove, using (3.8), that adding a constant c to all the rewards adds a constant, v_c, to the values of all states, and thus does not affect the relative values of any states under any policies. What is v_c in terms of c and γ? □

Exercise 3.16 Now consider adding a constant c to all the rewards in an episodic task, such as maze running. Would this have any effect, or would it leave the task unchanged as in the continuing task above? Why or why not? Give an example. □

Example 3.6: Golf To formulate playing a hole of golf as a reinforcement learning task, we count a penalty (negative reward) of -1 for each stroke until we hit the ball into the hole. The state is the location of the ball. The value of a state is the negative of the number of strokes to the hole from that location. Our actions are how we aim and swing at the ball, of course, and which club we select. Let us take the former as given and consider just the choice of club, which we assume is either a putter or a driver. The upper part of Figure 3.3 shows a possible state-value function, $v_{\texttt{putt}}(s)$, for the policy that always uses the putter. The terminal state *in-the-hole* has a value of 0. From anywhere on the green we assume we can make a putt; these states have value -1. Off the green we cannot reach the hole by putting, and the value is greater. If we can reach the green from a state by putting, then that state must have value one less than the green's value, that is, -2. For simplicity, let us assume we can putt very precisely and deterministically, but with a limited range. This gives us the sharp contour line labeled -2 in the figure; all locations between that line and the green require exactly two strokes to complete the hole. Similarly, any location within putting range of the -2 contour line must have a value of -3, and so on to get all the contour lines shown in the figure. Putting doesn't get us out of sand traps, so they have a value of $-\infty$. Overall, it takes us six strokes to get from the tee to the hole by putting.

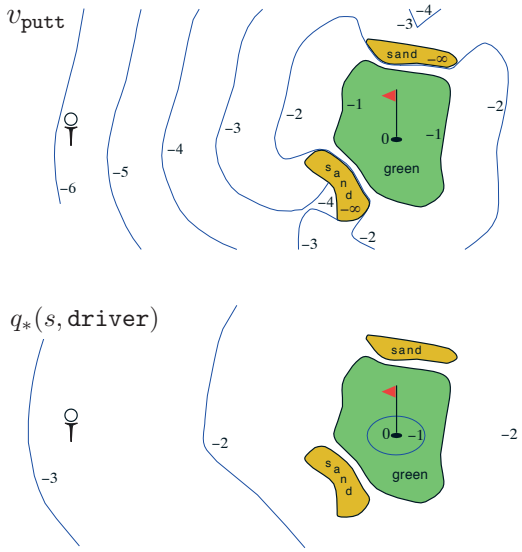
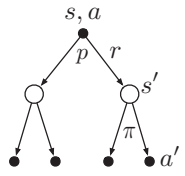


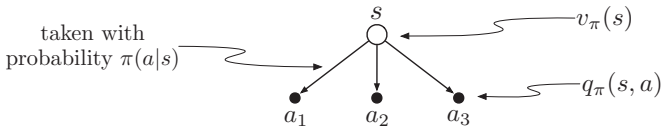
Figure 3.3: A golf example: the state-value function for putting (upper) and the optimal action-value function for using the driver (lower). ■

Exercise 3.17 What is the Bellman equation for action values, that is, for q_π? It must give the action value $q_\pi(s,a)$ in terms of the action values, $q_\pi(s',a')$, of possible successors to the state–action pair (s,a). Hint: the backup diagram to the right corresponds to this equation. Show the sequence of equations analogous to (3.14), but for action values. □



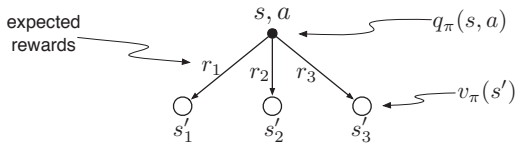
q_π backup diagram

Exercise 3.18 The value of a state depends on the values of the actions possible in that state and on how likely each action is to be taken under the current policy. We can think of this in terms of a small backup diagram rooted at the state and considering each possible action:



Give the equation corresponding to this intuition and diagram for the value at the root node, $v_\pi(s)$, in terms of the value at the expected leaf node, $q_\pi(s,a)$, given $S_t = s$. This equation should include an expectation conditioned on following the policy, π. Then give a second equation in which the expected value is written out explicitly in terms of $\pi(a|s)$ such that no expected value notation appears in the equation. □

Exercise 3.19 The value of an action, $q_\pi(s,a)$, depends on the expected next reward and the expected sum of the remaining rewards. Again we can think of this in terms of a small backup diagram, this one rooted at an action (state–action pair) and branching to the possible next states:



Give the equation corresponding to this intuition and diagram for the action value, $q_\pi(s,a)$, in terms of the expected next reward, R_{t+1}, and the expected next state value, $v_\pi(S_{t+1})$, given that $S_t = s$ and $A_t = a$. This equation should include an expectation but *not* one conditioned on following the policy. Then give a second equation, writing out the expected value explicitly in terms of $p(s', r \,|\, s, a)$ defined by (3.2), such that no expected value notation appears in the equation. □

3.6 Optimal Policies and Optimal Value Functions

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

$$v_*(s) \doteq \max_\pi v_\pi(s), \tag{3.15}$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action-value function*, denoted q_*, and defined as

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \tag{3.16}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For the state–action pair (s,a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]. \tag{3.17}$$

Example 3.7: Optimal Value Functions for Golf The lower part of Figure 3.3 shows the contours of a possible optimal action-value function $q_*(s, \texttt{driver})$. These are the values of each state if we first play a stroke with the driver and afterward select either the driver or the putter, whichever is better. The driver enables us to hit the ball farther, but with less accuracy. We can reach the hole in one shot using the driver only if we are already very close; thus the -1 contour for $q_*(s, \texttt{driver})$ covers only a small portion of the green. If we have two strokes, however, then we can reach the hole from much farther away, as shown by the -2 contour. In this case we don't have to drive all the way to within the small -1 contour, but only to anywhere on the green; from there we can use the putter. The optimal action-value function gives the values after committing to a particular *first* action, in this case, to the driver, but afterward using whichever actions are best. The -3 contour is still farther out and includes the starting tee. From the tee, the best sequence of actions is two drives and one putt, sinking the ball in three strokes. ■

Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values (3.14). Because it is the optimal value function, however, v_*'s consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for v_*, or the *Bellman optimality equation*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] && \text{(by (3.9))} \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] && (3.18) \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big]. && (3.19)
\end{aligned}$$

The last two equations are two forms of the Bellman optimality equation for v_*. The Bellman optimality equation for q_* is

$$\begin{aligned}
q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\
&= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big]. && (3.20)
\end{aligned}$$

The backup diagrams in the figure below show graphically the spans of future states and actions considered in the Bellman optimality equations for v_* and q_*. These are the same as the backup diagrams for v_π and q_π presented earlier except that arcs have been added at the agent's choice points to represent that the maximum over that choice is taken rather than the expected value given some policy. The backup diagram on the left graphically represents the Bellman optimality equation (3.19) and the backup diagram on the right graphically represents (3.20).

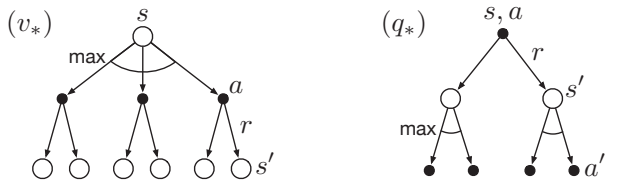


Figure 3.4: Backup diagrams for v_* and q_*

For finite MDPs, the Bellman optimality equation for v_π (3.19) has a unique solution independent of the policy. The Bellman optimality equation is actually a system of equations, one for each state, so if there are n states, then there are n equations in n unknowns. If the dynamics p of the environment are known, then in principle one can solve this system of equations for v_* using any one of a variety of methods for solving systems of nonlinear equations. One can solve a related set of equations for q_*.

Once one has v_*, it is relatively easy to determine an optimal policy. For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy. You can think of this as a one-step search. If you have the optimal value function, v_*, then the actions that appear best after a one-step search will be optimal actions. Another way of saying this is that any policy that is *greedy* with respect to the optimal evaluation function v_* is an optimal policy. The term greedy is used in computer science to describe any search or decision procedure that selects alternatives based only on local or immediate considerations, without considering the possibility that such a selection may prevent future access to even better alternatives. Consequently, it describes policies that select actions based only on their short-term consequences. The beauty of v_* is that if one uses it to evaluate the short-term consequences of actions—specifically, the one-step consequences—then a greedy policy is actually optimal in the long-term sense in which we are interested because v_* already takes into account the reward consequences of all possible future behavior. By means of v_*, the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state. Hence, a one-step-ahead search yields the long-term optimal actions.

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

representing a function of state–action pairs, instead of just of states, the optimal action-value function allows optimal actions to be selected without having to know anything about possible successor states and their values, that is, without having to know anything about the environment's dynamics.

Example 3.8: Solving the Gridworld Suppose we solve the Bellman equation for v_* for the simple grid task introduced in Example 3.5 and shown again in Figure 3.5 (left). Recall that state A is followed by a reward of +10 and transition to state A′, while state B is followed by a reward of +5 and transition to state B′. Figure 3.5 (middle) shows the optimal value function, and Figure 3.5 (right) shows the corresponding optimal policies. Where there are multiple arrows in a cell, all of the corresponding actions are optimal.

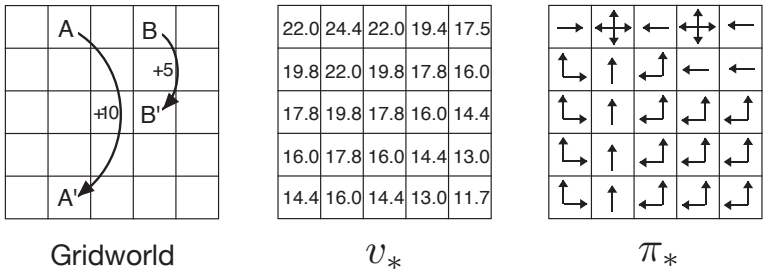


Figure 3.5: Optimal solutions to the gridworld example. ■

Example 3.9: Bellman Optimality Equations for the Recycling Robot Using (3.19), we can explicitly give the Bellman optimality equation for the recycling robot example. To make things more compact, we abbreviate the states `high` and `low`, and the actions `search`, `wait`, and `recharge` respectively by `h`, `l`, `s`, `w`, and `re`. Because there are only two states, the Bellman optimality equation consists of two equations. The equation for $v_*(\texttt{h})$ can be written as follows:

$$\begin{aligned} v_*(\texttt{h}) &= \max\left\{ \begin{array}{l} p(\texttt{h}|\texttt{h},\texttt{s})[r(\texttt{h},\texttt{s},\texttt{h}) + \gamma v_*(\texttt{h})] + p(\texttt{l}|\texttt{h},\texttt{s})[r(\texttt{h},\texttt{s},\texttt{l}) + \gamma v_*(\texttt{l})], \\ p(\texttt{h}|\texttt{h},\texttt{w})[r(\texttt{h},\texttt{w},\texttt{h}) + \gamma v_*(\texttt{h})] + p(\texttt{l}|\texttt{h},\texttt{w})[r(\texttt{h},\texttt{w},\texttt{l}) + \gamma v_*(\texttt{l})] \end{array} \right\} \\ &= \max\left\{ \begin{array}{l} \alpha[r_\texttt{s} + \gamma v_*(\texttt{h})] + (1-\alpha)[r_\texttt{s} + \gamma v_*(\texttt{l})], \\ 1[r_\texttt{w} + \gamma v_*(\texttt{h})] + 0[r_\texttt{w} + \gamma v_*(\texttt{l})] \end{array} \right\} \\ &= \max\left\{ \begin{array}{l} r_\texttt{s} + \gamma[\alpha v_*(\texttt{h}) + (1-\alpha) v_*(\texttt{l})], \\ r_\texttt{w} + \gamma v_*(\texttt{h}) \end{array} \right\}. \end{aligned}$$

Following the same procedure for $v_*(\texttt{l})$ yields the equation

$$v_*(\texttt{l}) = \max\left\{ \begin{array}{l} \beta r_\texttt{s} - 3(1-\beta) + \gamma[(1-\beta) v_*(\texttt{h}) + \beta v_*(\texttt{l})], \\ r_\texttt{w} + \gamma v_*(\texttt{l}), \\ \gamma v_*(\texttt{h}) \end{array} \right\}.$$

For any choice of $r_\texttt{s}$, $r_\texttt{w}$, α, β, and γ, with $0 \le \gamma < 1$, $0 \le \alpha, \beta \le 1$, there is exactly one pair of numbers, $v_*(\texttt{h})$ and $v_*(\texttt{l})$, that simultaneously satisfy these two nonlinear equations. ■

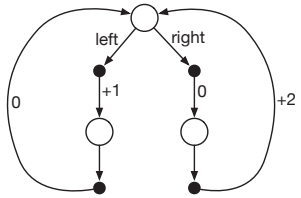
Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the reinforcement learning problem. However, this solution is rarely directly useful. It is akin to an exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. This solution relies on at least three assumptions that are rarely true in practice: (1) we accurately know the dynamics of the environment; (2) we have enough computational resources to complete the computation of the solution; and (3) the Markov property. For the kinds of tasks in which we are interested, one is generally not able to implement this solution exactly because various combinations of these assumptions are violated. For example, although the first and third assumptions present no problems for the game of backgammon, the second is a major impediment. Because the game has about 10^{20} states, it would take thousands of years on today's fastest computers to solve the Bellman equation for v_*, and the same is true for finding q_*. In reinforcement learning one typically has to settle for approximate solutions.

Many different decision-making methods can be viewed as ways of approximately solving the Bellman optimality equation. For example, heuristic search methods can be viewed as expanding the right-hand side of (3.19) several times, up to some depth, forming a "tree" of possibilities, and then using a heuristic evaluation function to approximate v_* at the "leaf" nodes. (Heuristic search methods such as A* are almost always based on the episodic case.) The methods of dynamic programming can be related even more closely to the Bellman optimality equation. Many reinforcement learning methods can be clearly understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions. We consider a variety of such methods in the following chapters.

Exercise 3.20 Draw or describe the optimal state-value function for the golf example. □

Exercise 3.21 Draw or describe the contours of the optimal action-value function for putting, $q_*(s, \texttt{putter})$, for the golf example. □

Exercise 3.22 Consider the continuing MDP shown on to the right. The only decision to be made is that in the top state, where two actions are available, left and right. The numbers show the rewards that are received deterministically after each action. There are exactly two deterministic policies, π_{left} and π_{right}. What policy is optimal if $\gamma = 0$? If $\gamma = 0.9$? If $\gamma = 0.5$? □



Exercise 3.23 Give the Bellman equation for q_* for the recycling robot. □

Exercise 3.24 Figure 3.5 gives the optimal value of the best state of the gridworld as 24.4, to one decimal place. Use your knowledge of the optimal policy and (3.8) to express this value symbolically, and then to compute it to three decimal places. □

Exercise 3.25 Give an equation for v_* in terms of q_*. □

Exercise 3.26 Give an equation for q_* in terms of v_* and the four-argument p. □

Exercise 3.27 Give an equation for π_* in terms of q_*. □

Exercise 3.28 Give an equation for π_* in terms of v_* and the four-argument p. □

Exercise 3.29 Rewrite the four Bellman equations for the four value functions (v_π, v_*, q_π, and q_*) in terms of the three argument function p (3.4) and the two-argument function r (3.5). □

3.7 Optimality and Approximation

We have defined optimal value functions and optimal policies. Clearly, an agent that learns an optimal policy has done very well, but in practice this rarely happens. For the kinds of tasks in which we are interested, optimal policies can be generated only with extreme computational cost. A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that agents can only approximate to varying degrees. As we discussed above, even if we have a complete and accurate model of the environment's dynamics, it is usually not possible to simply compute an optimal policy by solving the Bellman optimality equation. For example, board games such as chess are a tiny fraction of human experience, yet large, custom-designed computers still cannot compute the optimal moves. A critical aspect of the problem facing the agent is always the computational power available to it, in particular, the amount of computation it can perform in a single time step.

The memory available is also an important constraint. A large amount of memory is often required to build up approximations of value functions, policies, and models. In tasks with small, finite state sets, it is possible to form these approximations using arrays or tables with one entry for each state (or state–action pair). This we call the *tabular* case, and the corresponding methods we call tabular methods. In many cases of practical interest, however, there are far more states than could possibly be entries in a table. In these cases the functions must be approximated, using some sort of more compact parameterized function representation.

Our framing of the reinforcement learning problem forces us to settle for approximations. However, it also presents us with some unique opportunities for achieving useful approximations. For example, in approximating optimal behavior, there may be many states that the agent faces with such a low probability that selecting suboptimal actions for them has little impact on the amount of reward the agent receives. Tesauro's backgammon player, for example, plays with exceptional skill even though it might make very bad decisions on board configurations that never occur in games against experts. In fact, it is possible that TD-Gammon makes bad decisions for a large fraction of the game's state set. The online nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states. This is one key property that distinguishes reinforcement learning from other approaches to approximately solving MDPs.

3.8 Summary

Let us summarize the elements of the reinforcement learning problem that we have presented in this chapter. Reinforcement learning is about learning from interaction how to behave in order to achieve a goal. The reinforcement learning *agent* and its *environment* interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the *actions* are the choices made by the agent; the *states* are the basis for making the choices; and the *rewards* are the basis for evaluating the choices. Everything inside the agent is completely known and controllable by the agent; everything outside is incompletely controllable but may or may not be completely known. A *policy* is a stochastic rule by which the agent selects actions as a function of states. The agent's objective is to maximize the amount of reward it receives over time.

When the reinforcement learning setup described above is formulated with well defined transition probabilities it constitutes a *Markov decision process* (MDP). A *finite MDP* is an MDP with finite state, action, and (as we formulate it here) reward sets. Much of the current theory of reinforcement learning is restricted to finite MDPs, but the methods and ideas apply more generally.

The *return* is the function of future rewards that the agent seeks to maximize (in expected value). It has several different definitions depending upon the nature of the task and whether one wishes to *discount* delayed reward. The undiscounted formulation is appropriate for *episodic tasks*, in which the agent–environment interaction breaks naturally into *episodes*; the discounted formulation is appropriate for *continuing tasks*, in which the interaction does not naturally break into episodes but continues without limit. We try to define the returns for the two kinds of tasks such that one set of equations can apply to both the episodic and continuing cases.

A policy's *value functions* assign to each state, or state–action pair, the expected return from that state, or state–action pair, given that the agent uses the policy. The *optimal value functions* assign to each state, or state–action pair, the largest expected return achievable by any policy. A policy whose value functions are optimal is an *optimal policy*. Whereas the optimal value functions for states and state–action pairs are unique for a given MDP, there can be many optimal policies. Any policy that is *greedy* with respect to the optimal value functions must be an optimal policy. The *Bellman optimality equations* are special consistency conditions that the optimal value functions must satisfy and that can, in principle, be solved for the optimal value functions, from which an optimal policy can be determined with relative ease.

A reinforcement learning problem can be posed in a variety of different ways depending on assumptions about the level of knowledge initially available to the agent. In problems of *complete knowledge*, the agent has a complete and accurate model of the environment's dynamics. If the environment is an MDP, then such a model consists of the complete four-argument dynamics function p (3.2). In problems of *incomplete knowledge*, a complete and perfect model of the environment is not available.

Even if the agent has a complete and accurate environment model, the agent is typically unable to perform enough computation per time step to fully use it. The memory available is also an important constraint. Memory may be required to build

up accurate approximations of value functions, policies, and models. In most cases of practical interest there are far more states than could possibly be entries in a table, and approximations must be made.

A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that reinforcement learning agents can only approximate to varying degrees. In reinforcement learning we are very much concerned with cases in which optimal solutions cannot be found but must be approximated in some way.

Bibliographical and Historical Remarks

The reinforcement learning problem is deeply indebted to the idea of Markov decision processes (MDPs) from the field of optimal control. These historical influences and other major influences from psychology are described in the brief history given in Chapter 1. Reinforcement learning adds to MDPs a focus on approximation and incomplete information for realistically large problems. MDPs and the reinforcement learning problem are only weakly linked to traditional learning and decision-making problems in artificial intelligence. However, artificial intelligence is now vigorously exploring MDP formulations for planning and decision making from a variety of perspectives. MDPs are more general than previous formulations used in artificial intelligence in that they permit more general kinds of goals and uncertainty.

The theory of MDPs is treated by, for example, Bertsekas (2005), White (1969), Whittle (1982, 1983), and Puterman (1994). A particularly compact treatment of the finite case is given by Ross (1983). MDPs are also studied under the heading of stochastic optimal control, where *adaptive* optimal control methods are most closely related to reinforcement learning (e.g., Kumar, 1985; Kumar and Varaiya, 1986).

The theory of MDPs evolved from efforts to understand the problem of making sequences of decisions under uncertainty, where each decision can depend on the previous decisions and their outcomes. It is sometimes called the theory of multistage decision processes, or sequential decision processes, and has roots in the statistical literature on sequential sampling beginning with the papers by Thompson (1933, 1934) and Robbins (1952) that we cited in Chapter 2 in connection with bandit problems (which are prototypical MDPs if formulated as multiple-situation problems).

The earliest instance of which we are aware in which reinforcement learning was discussed using the MDP formalism is Andreae's (1969b) description of a unified view of learning machines. Witten and Corbin (1973) experimented with a reinforcement learning system later analyzed by Witten (1977, 1976a) using the MDP formalism. Although he did not explicitly mention MDPs, Werbos (1977) suggested approximate solution methods for stochastic optimal control problems that are related to modern reinforcement learning methods (see also Werbos, 1982, 1987, 1988, 1989, 1992). Although Werbos's ideas were not widely recognized at the time, they were prescient in emphasizing the importance of approximately solving optimal control problems in a variety of domains, including artificial intelligence. The most influential integration of reinforcement learning and MDPs is due to Watkins (1989).

3.1 Our characterization of the dynamics of an MDP in terms of $p(s', r \mid s, a)$ is slightly unusual. It is more common in the MDP literature to describe the dynamics in terms of the state transition probabilities $p(s' \mid s, a)$ and expected next rewards $r(s, a)$. In reinforcement learning, however, we more often have to refer to individual actual or sample rewards (rather than just their expected values). Our notation also makes it plainer that S_t and R_t are in general jointly determined, and thus must have the same time index. In teaching reinforcement learning, we have found our notation to be more straightforward conceptually and easier to understand.

For a good intuitive discussion of the system-theoretic concept of state, see Minsky (1967).

The bioreactor example is based on the work of Ungar (1990) and Miller and Williams (1992). The recycling robot example was inspired by the can-collecting robot built by Jonathan Connell (1989). Kober and Peters (2012) present a collection of robotics applications of reinforcement learning.

3.2 The reward hypothesis was suggested by Michael Littman (personal communication).

3.3–4 The terminology of *episodic* and *continuing* tasks is different from that usually used in the MDP literature. In that literature it is common to distinguish three types of tasks: (1) finite-horizon tasks, in which interaction terminates after a particular *fixed* number of time steps; (2) indefinite-horizon tasks, in which interaction can last arbitrarily long but must eventually terminate; and (3) infinite-horizon tasks, in which interaction does not terminate. Our episodic and continuing tasks are similar to indefinite-horizon and infinite-horizon tasks, respectively, but we prefer to emphasize the difference in the nature of the interaction. This difference seems more fundamental than the difference in the objective functions emphasized by the usual terms. Often episodic tasks use an indefinite-horizon objective function and continuing tasks an infinite-horizon objective function, but we see this as a common coincidence rather than a fundamental difference.

The pole-balancing example is from Michie and Chambers (1968) and Barto, Sutton, and Anderson (1983).

3.5–6 Assigning value on the basis of what is good or bad in the long run has ancient roots. In control theory, mapping states to numerical values representing the long-term consequences of control decisions is a key part of optimal control theory, which was developed in the 1950s by extending nineteenth century state-function theories of classical mechanics (see, e.g., Schultz and Melsa, 1967). In describing how a computer could be programmed to play chess, Shannon (1950) suggested using an evaluation function that took into account the long-term advantages and disadvantages of chess positions.

Watkins's (1989) Q-learning algorithm for estimating q_* (Chapter 6) made action-value functions an important part of reinforcement learning, and consequently

these functions are often called "Q-functions." But the idea of an action-value function is much older than this. Shannon (1950) suggested that a function $h(P, M)$ could be used by a chess-playing program to decide whether a move M in position P is worth exploring. Michie's (1961, 1963) MENACE system and Michie and Chambers's (1968) BOXES system can be understood as estimating action-value functions. In classical physics, Hamilton's principal function is an action-value function; Newtonian dynamics are greedy with respect to this function (e.g., Goldstein, 1957). Action-value functions also played a central role in Denardo's (1967) theoretical treatment of dynamic programming in terms of contraction mappings.

The Bellman optimality equation (for v_*) was popularized by Richard Bellman (1957a), who called it the "basic functional equation." The counterpart of the Bellman optimality equation for continuous time and state problems is known as the Hamilton–Jacobi–Bellman equation (or often just the Hamilton–Jacobi equation), indicating its roots in classical physics (e.g., Schultz and Melsa, 1967).

The golf example was suggested by Chris Watkins.

Chapter 4

Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. DP provides an essential foundation for the understanding of the methods presented in the rest of this book. In fact, all of these methods can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

We usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets, $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$, are finite, and that its dynamics are given by a set of probabilities $p(s', r \,|\, s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $r \in \mathcal{R}$, and $s' \in \mathcal{S}^+$ ($\mathcal{S}^+$ is $\mathcal{S}$ plus a terminal state if the problem is episodic). Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods. The methods we explore in Chapter 9 are applicable to continuous problems and are a significant extension of that approach.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. In this chapter we show how DP can be used to compute the value functions defined in Chapter 3. As discussed there, we can easily obtain optimal policies once we have found the optimal value functions, v_* or q_*, which satisfy the Bellman optimality equations:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma v_*(s')\Big], \text{ or} \qquad (4.1) \\ q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\ &= \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big], \qquad (4.2) \end{aligned}$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $s' \in \mathcal{S}^+$. As we shall see, DP algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.

4.1 Policy Evaluation (Prediction)

First we consider how to compute the state-value function v_π for an arbitrary policy π. This is called *policy evaluation* in the DP literature. We also refer to it as the *prediction problem*. Recall from Chapter 3 that, for all $s \in \mathcal{S}$,

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && \text{(from (3.9))} \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (4.3) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma v_\pi(s')\Big], && (4.4) \end{aligned}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π, and the expectations are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π.

If the environment's dynamics are completely known, then (4.4) is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns (the $v_\pi(s)$, $s \in \mathcal{S}$). In principle, its solution is a straightforward, if tedious, computation. For our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions $v_0, v_1, v_2, \ldots$, each mapping $\mathcal{S}^+$ to $\mathbb{R}$ (the real numbers). The initial approximation, v_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π (4.4) as an update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma v_k(s')\Big], \qquad (4.5) \end{aligned}$$

for all $s \in \mathcal{S}$. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called *iterative policy evaluation*.

To produce each successive approximation, v_{k+1} from v_k, iterative policy evaluation applies the same operation to each state s: it replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. We call this kind of operation an *expected update*. Each iteration of iterative policy evaluation updates the value of every state once to produce the new approximate value function

v_{k+1}. There are several different kinds of expected updates, depending on whether a state (as here) or a state–action pair is being updated, and depending on the precise way the estimated values of the successor states are combined. All the updates done in DP algorithms are called *expected* updates because they are based on an expectation over all possible next states rather than on a sample next state. The nature of an update can be expressed in an equation, as above, or in a backup diagram like those introduced in Chapter 3. For example, the backup diagram corresponding to the expected update used in iterative policy evaluation is shown on page 59.

To write a sequential computer program to implement iterative policy evaluation as given by (4.5) you would have to use two arrays, one for the old values, $v_k(s)$, and one for the new values, $v_{k+1}(s)$. With two arrays, the new values can be computed one by one from the old values without the old values being changed. Of course it is easier to use one array and update the values "in place," that is, with each new value immediately overwriting the old one. Then, depending on the order in which the states are updated, sometimes new values are used instead of old ones on the right-hand side of (4.5). This in-place algorithm also converges to v_π; in fact, it usually converges faster than the two-array version, as you might expect, because it uses new data as soon as they are available. We think of the updates as being done in a *sweep* through the state space. For the in-place algorithm, the order in which states have their values updated during the sweep has a significant influence on the rate of convergence. We usually have the in-place version in mind when we think of DP algorithms.

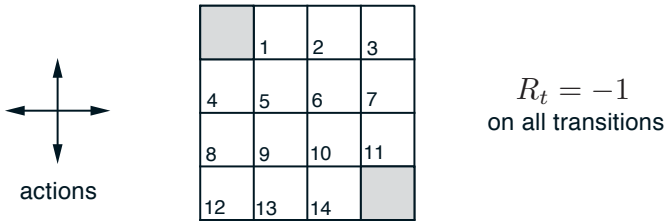
A complete in-place version of iterative policy evaluation is shown in pseudocode in the box below. Note how it handles termination. Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this. The pseudocode tests the quantity $\max_{s\in\mathcal{S}} |v_{k+1}(s) - v_k(s)|$ after each sweep and stops when it is sufficiently small.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Example 4.1 Consider the 4×4 gridworld shown below.



The nonterminal states are $\mathcal{S} = \{1, 2, \ldots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. Thus, for instance, $p(6, -1|5, \texttt{right}) = 1$, $p(7, -1|7, \texttt{right}) = 1$, and $p(10, r|5, \texttt{right}) = 0$ for all $r \in \mathcal{R}$. This is an undiscounted, episodic task. The reward is -1 on all transitions until the terminal state is reached. The terminal state is shaded in the figure (although it is shown in two places, it is formally one state). The expected reward function is thus $r(s, a, s') = -1$ for all states s, s' and actions a. Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 4.1 shows the sequence of value functions $\{v_k\}$ computed by iterative policy evaluation. The final estimate is in fact v_π, which in this case gives for each state the negation of the expected number of steps from that state until termination. ■

Exercise 4.1 In Example 4.1, if π is the equiprobable random policy, what is $q_\pi(11, \texttt{down})$? What is $q_\pi(7, \texttt{down})$? □

Exercise 4.2 In Example 4.1, suppose a new state 15 is added to the gridworld just below state 13, and its actions, `left`, `up`, `right`, and `down`, take the agent to states 12, 13, 14, and 15, respectively. Assume that the transitions *from* the original states are unchanged. What, then, is $v_\pi(15)$ for the equiprobable random policy? Now suppose the dynamics of state 13 are also changed, such that action `down` from state 13 takes the agent to the new state 15. What is $v_\pi(15)$ for the equiprobable random policy in this case? □

Exercise 4.3 What are the equations analogous to (4.3), (4.4), and (4.5) for the action-value function q_π and its successive approximation by a sequence of functions $q_0, q_1, q_2, \ldots$? □

4.2 Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function v_π for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. We know how good it is to follow the current policy from s—that is $v_\pi(s)$—but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting a in s and thereafter

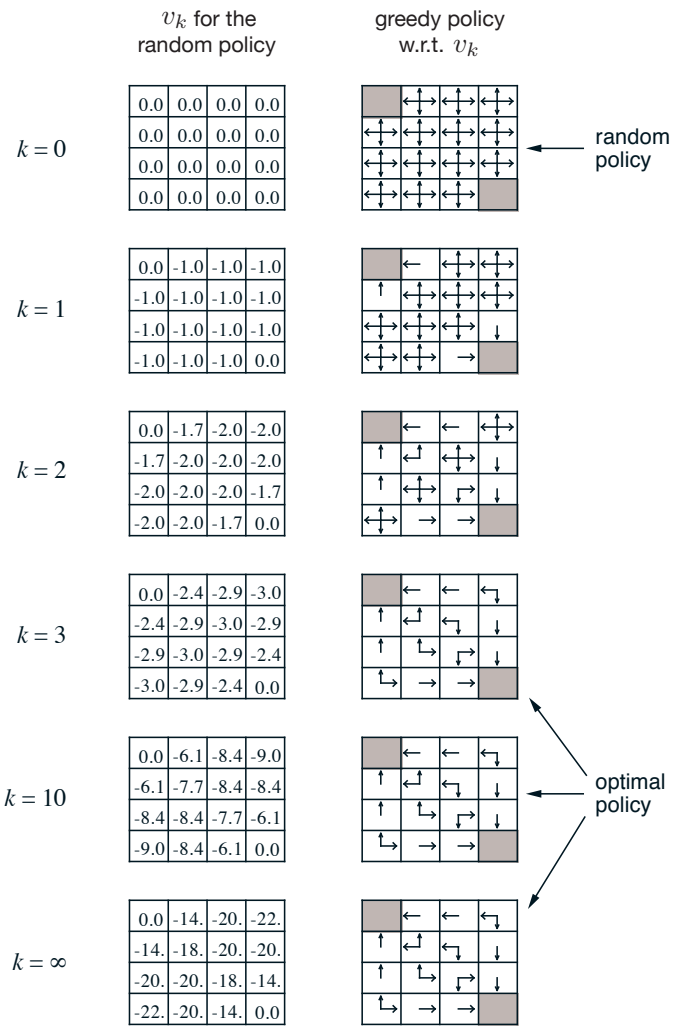


Figure 4.1: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equally likely). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum, and the numbers shown are rounded to two significant digits). The last policy is guaranteed only to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal.

following the existing policy, π. The value of this way of behaving is

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big]. \end{aligned} \tag{4.6}$$

The key criterion is whether this is greater than or less than $v_\pi(s)$. If it is greater—that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time—then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.

That this is true is a special case of a general result called the *policy improvement theorem*. Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s). \tag{4.7}$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s). \tag{4.8}$$

Moreover, if there is strict inequality of (4.7) at any state, then there must be strict inequality of (4.8) at that state. This result applies in particular to the two policies that we considered in the previous paragraph, an original deterministic policy, π, and a changed policy, π', that is identical to π except that $\pi'(s) = a \neq \pi(s)$. Obviously, (4.7) holds at all states other than s. Thus, if $q_\pi(s,a) > v_\pi(s)$, then the changed policy is indeed better than π.

The idea behind the proof of the policy improvement theorem is easy to understand. Starting from (4.7), we keep expanding the q_π side with (4.6) and reapplying (4.7) until we get $v_{\pi'}(s)$:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)] && \text{(by (4.6))} \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] && \text{(by (4.7))} \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}, A_{t+1} = \pi'(S_{t+1})] \mid S_t = s] \\ &= \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \;\big|\; S_t = s\big] \\ &\leq \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) \;\big|\; S_t = s\big] \\ &\;\;\vdots \\ &\leq \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \;\big|\; S_t = s\big] \\ &= v_{\pi'}(s). \end{aligned}$$

So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state to a particular action. It is a natural extension

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

$$\begin{aligned}\pi'(s) &\doteq \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t{=}s, A_t{=}a] \\ &= \arg\max_a \sum_{s',r} p(s',r\,|\,s,a)\Big[r + \gamma v_\pi(s')\Big],\end{aligned} \tag{4.9}$$

where $\arg\max_a$ denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). The greedy policy takes the action that looks best in the short term—after one step of lookahead—according to v_π. By construction, the greedy policy meets the conditions of the policy improvement theorem (4.7), so we know that it is as good as, or better than, the original policy. The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*.

Suppose the new greedy policy, π', is as good as, but not better than, the old policy π. Then $v_\pi = v_{\pi'}$, and from (4.9) it follows that for all $s \in \mathcal{S}$:

$$\begin{aligned}v_{\pi'}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t{=}s, A_t{=}a] \\ &= \max_a \sum_{s',r} p(s',r\,|\,s,a)\Big[r + \gamma v_{\pi'}(s')\Big].\end{aligned}$$

But this is the same as the Bellman optimality equation (4.1), and therefore, $v_{\pi'}$ must be v_*, and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

So far in this section we have considered the special case of deterministic policies. In the general case, a stochastic policy π specifies probabilities, $\pi(a|s)$, for taking each action, a, in each state, s. We will not go through the details, but in fact all the ideas of this section extend easily to stochastic policies. In particular, the policy improvement theorem carries through as stated for the stochastic case. In addition, if there are ties in policy improvement steps such as (4.9)—that is, if there are several actions at which the maximum is achieved—then in the stochastic case we need not select a single action from among them. Instead, each maximizing action can be given a portion of the probability of being selected in the new greedy policy. Any apportioning scheme is allowed as long as all submaximal actions are given zero probability.

The last row of Figure 4.1 shows an example of policy improvement for stochastic policies. Here the original policy, π, is the equiprobable random policy, and the new policy, π', is greedy with respect to v_π. The value function v_π is shown in the bottom-left diagram and the set of possible π' is shown in the bottom-right diagram. The states with multiple arrows in the π' diagram are those in which several actions achieve the maximum in (4.9); any apportionment of probability among these actions is permitted. The value function of any such policy, $v_{\pi'}(s)$, can be seen by inspection to be either -1, -2, or -3 at all states, $s \in \mathcal{S}$, whereas $v_\pi(s)$ is at most -14. Thus, $v_{\pi'}(s) \geq v_\pi(s)$, for all

$s \in \mathcal{S}$, illustrating policy improvement. Although in this case the new policy π' happens to be optimal, in general only an improvement is guaranteed.

4.3 Policy Iteration

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

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

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

This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given in the box below. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s',r\,|\,s,\pi(s))\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r\,|\,s,a)\big[r + \gamma V(s')\big]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Example 4.2: Jack's Car Rental Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is $\frac{\lambda^n}{n!}e^{-\lambda}$, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight. Figure 4.2 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars.

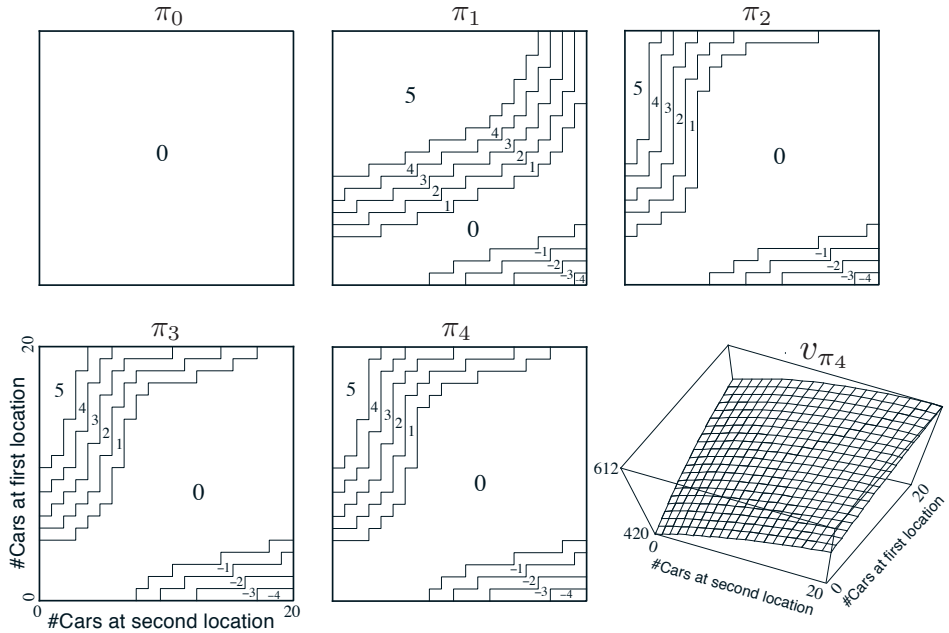


Figure 4.2: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal. ■

Policy iteration often converges in surprisingly few iterations, as the example of Jack's car rental illustrates, and as is also illustrated by the example in Figure 4.1. The bottom-left diagram of Figure 4.1 shows the value function for the equiprobable random policy, and the bottom-right diagram shows a greedy policy for this value function. The policy improvement theorem assures us that these policies are better than the original random policy. In this case, however, these policies are not just better, but optimal, proceeding to the terminal states in the minimum number of steps. In this example, policy iteration would find the optimal policy after just one iteration.

Exercise 4.4 The policy iteration algorithm on page 80 has a subtle bug in that it may never terminate if the policy continually switches between two or more policies that are equally good. This is ok for pedagogy, but not for actual use. Modify the pseudocode so that convergence is guaranteed. □

Exercise 4.5 How would policy iteration be defined for action values? Give a complete algorithm for computing q_*, analogous to that on page 80 for computing v_*. Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book. □

Exercise 4.6 Suppose you are restricted to considering only policies that are ε-*soft*, meaning that the probability of selecting each action in each state, s, is at least $\varepsilon/|\mathcal{A}(s)|$. Describe qualitatively the changes that would be required in each of the steps 3, 2, and 1, in that order, of the policy iteration algorithm for v_* on page 80. □

Exercise 4.7 (programming) Write a program for policy iteration and re-solve Jack's car rental problem with the following changes. One of Jack's employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs \$2, as do all cars moved in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of \$4 must be incurred to use a second parking lot (independent of how many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming. To check your program, first replicate the results given for the original problem. □

4.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to v_π occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? The example in Figure 4.1 certainly suggests that it may be possible to truncate policy evaluation. In that example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special

case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big], \end{aligned} \tag{4.10}$$

for all $s \in \mathcal{S}$. For arbitrary v_0, the sequence $\{v_k\}$ can be shown to converge to v_* under the same conditions that guarantee the existence of v_*.

Another way of understanding value iteration is by reference to the Bellman optimality equation (4.1). Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration update is identical to the policy evaluation update (4.5) except that it requires the maximum to be taken over all actions. Another way of seeing this close relationship is to compare the backup diagrams for these algorithms on page 59 (policy evaluation) and on the left of Figure 3.4 (value iteration). These two are the natural backup operations for computing v_π and v_*.

Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_*. In practice, we stop once the value function changes by only a small amount in a sweep. The box below shows a complete algorithm with this kind of termination condition.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates. Because the max operation in (4.10) is the only difference between

these updates, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs.

Example 4.3: Gambler's Problem A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of \$100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital, $s \in \{1, 2, \ldots, 99\}$ and the actions are stakes, $a \in \{0, 1, \ldots, \min(s, 100 - s)\}$. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is +1. The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let p_h denote the probability of the coin coming up heads. If p_h is known, then the entire problem is known and it can be solved, for instance, by value iteration. Figure 4.3 shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of $p_h = 0.4$. This policy is optimal, but not unique. In fact, there is a whole family of optimal policies, all corresponding to ties for the argmax action selection with respect to the optimal value function. Can you guess what the entire family looks like?

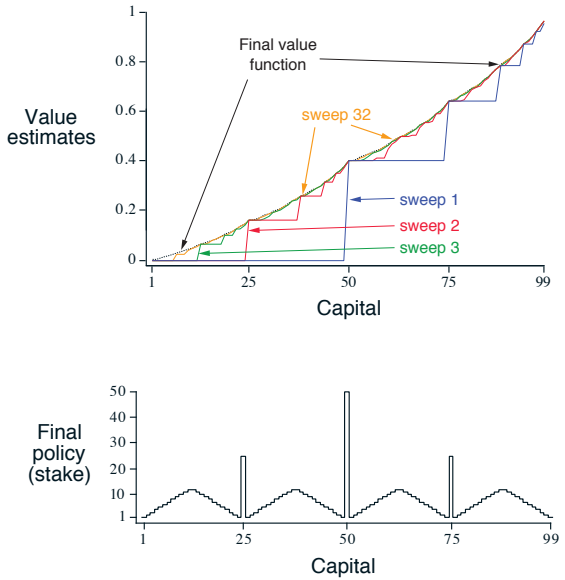


Figure 4.3: The solution to the gambler's problem for $p_h = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy.

■

Exercise 4.8 Why does the optimal policy for the gambler's problem have such a curious form? In particular, for capital of 50 it bets it all on one flip, but for capital of 51 it does not. Why is this a good policy? □

Exercise 4.9 (programming) Implement value iteration for the gambler's problem and solve it for $p_h = 0.25$ and $p_h = 0.55$. In programming, you may find it convenient to introduce two dummy states corresponding to termination with capital of 0 and 100, giving them values of 0 and 1 respectively. Show your results graphically, as in Figure 4.3.

Are your results stable as $\theta \to 0$? □

Exercise 4.10 What is the analog of the value iteration update (4.10) for action values, $q_{k+1}(s,a)$? □

4.5 Asynchronous Dynamic Programming

A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive. For example, the game of backgammon has over 10^{20} states. Even if we could perform the value iteration update on a million states per second, it would take over a thousand years to complete a single sweep.

Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms update the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be updated several times before the values of others are updated once. To converge correctly, however, an asynchronous algorithm must continue to update the values of all the states: it can't ignore any state after some point in the computation. Asynchronous DP algorithms allow great flexibility in selecting states to update.

For example, one version of asynchronous value iteration updates the value, in place, of only one state, s_k, on each step, k, using the value iteration update (4.10). If $0 \leq \gamma < 1$, asymptotic convergence to v_* is guaranteed given only that all states occur in the sequence $\{s_k\}$ an infinite number of times (the sequence could even be stochastic). (In the undiscounted episodic case, it is possible that there are some orderings of updates that do not result in convergence, but it is relatively easy to avoid these.) Similarly, it is possible to intermix policy evaluation and value iteration updates to produce a kind of asynchronous truncated policy iteration. Although the details of this and other more unusual DP algorithms are beyond the scope of this book, it is clear that a few different updates form building blocks that can be used flexibly in a wide variety of sweepless DP algorithms.

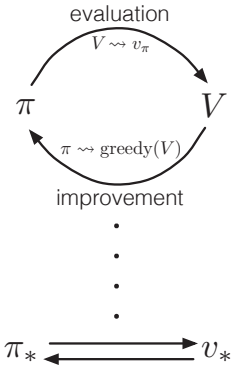
Of course, avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy. We can try to take advantage of this flexibility by selecting the states to which we apply updates so as to improve the algorithm's rate of progress. We can try to order the updates to let value information propagate from state to state in an efficient way. Some states may not need their values updated as often as others. We might even try to skip updating some states entirely if they are not relevant to optimal behavior. Some ideas for doing this are discussed in Chapter 8.

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

4.6 Generalized Policy Iteration

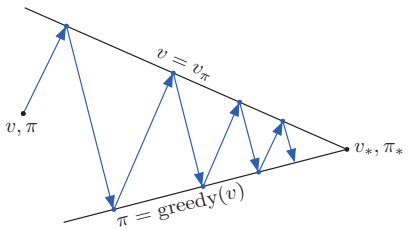
Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes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

We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy, as suggested by the diagram to the right. If both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal. The value function stabilizes only when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function.



Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation (4.1) holds, and thus that the policy and the value function are optimal.

The evaluation and improvement processes in GPI can be viewed as both competing and cooperating. They compete in the sense that they pull in opposing directions. Making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes that policy no longer to be greedy. In the long run, however, these two processes interact to find a single joint solution: the optimal value function and an optimal policy.



One might also think of the interaction between the evaluation and improvement processes in GPI in terms of two constraints or goals—for example, as two lines in two-dimensional space as suggested by the diagram to the right. Although the real geometry is much more complicated than this, the diagram suggests what happens in the real case. Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals. The goals interact because the two lines are not orthogonal. Driving directly toward one goal causes some movement away from the other goal. Inevitably, however, the joint process is brought closer to the overall goal of optimality. The arrows in this diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely. In GPI one could also take smaller, incomplete steps toward each goal. In either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly.

4.7 Efficiency of Dynamic Programming

DP may not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient. If we ignore a few technical details, then the (worst case) time DP methods take to find an optimal policy is polynomial in the number of states and actions. If n and k denote the number of states and actions, this means that a DP method takes a number of computational operations that is less than some polynomial function of n and k. A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is k^n. In this sense, DP is exponentially faster than any direct search in policy space could be, because direct search would have to exhaustively examine each policy to provide the same guarantee. Linear programming methods can also be used to solve MDPs, and in some cases their worst-case convergence guarantees are better than those of DP methods. But linear programming methods become impractical at a much smaller number of states than do DP methods (by a factor of about 100). For the largest problems, only DP methods are feasible.

DP is sometimes thought to be of limited applicability because of the *curse of dimensionality*, the fact that the number of states often grows exponentially with the number of state variables. Large state sets do create difficulties, but these are inherent difficulties of the problem, not of DP as a solution method. In fact, DP is comparatively better suited to handling large state spaces than competing methods such as direct search and linear programming.

In practice, DP methods can be used with today's computers to solve MDPs with millions of states. Both policy iteration and value iteration are widely used, and it is not clear which, if either, is better in general. In practice, these methods usually converge much faster than their theoretical worst-case run times, particularly if they are started with good initial value functions or policies.

On problems with large state spaces, *asynchronous* DP methods are often preferred. To complete even one sweep of a synchronous method requires computation and memory for every state. For some problems, even this much memory and computation is impractical, yet the problem is still potentially solvable because relatively few states occur along optimal solution trajectories. Asynchronous methods and other variations of GPI can be applied in such cases and may find good or optimal policies much faster than synchronous methods can.

4.8 Summary

In this chapter we have become familiar with the basic ideas and algorithms of dynamic programming as they relate to solving finite MDPs. *Policy evaluation* refers to the (typically) iterative computation of the value functions for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Putting these two computations together, we obtain *policy iteration* and *value iteration*, the two most popular DP methods. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing an *expected update* operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Expected updates are closely related to Bellman equations: they are little more than these equations turned into assignment statements. When the updates no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation. Just as there are four primary value functions (v_π, v_*, q_π, and q_*), there are four corresponding Bellman equations and four corresponding expected updates. An intuitive view of the operation of DP updates is given by their *backup diagrams*.

Insight into DP methods and, in fact, into almost all reinforcement learning methods, can be gained by viewing them as *generalized policy iteration* (GPI). GPI is the general idea of two interacting processes revolving around an approximate policy and an approximate value function. One process takes the policy as given and performs some form of policy evaluation, changing the value function to be more like the true value function for the policy. The other process takes the value function as given and performs some form of policy improvement, changing the policy to make it better, assuming that the value function is its value function. Although each process changes the basis for the other, overall they work together to find a joint solution: a policy and value function that are unchanged by either process and, consequently, are optimal. In some cases, GPI can be proved to converge, most notably for the classical DP methods that we have presented in this chapter. In other cases convergence has not been proved, but still the idea of GPI improves our understanding of the methods.

It is not necessary to perform DP methods in complete sweeps through the state set. *Asynchronous DP* methods are in-place iterative methods that update states in an arbitrary order, perhaps stochastically determined and using out-of-date information.

Many of these methods can be viewed as fine-grained forms of GPI.

Finally, we note one last special property of DP methods. All of them update estimates of the values of states based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea *bootstrapping*. Many reinforcement learning methods perform bootstrapping, even those that do not require, as DP requires, a complete and accurate model of the environment. In the next chapter we explore reinforcement learning methods that do not require a model and do not bootstrap. In the chapter after that we explore methods that do not require a model but do bootstrap. These key features and properties are separable, yet can be mixed in interesting combinations.

Bibliographical and Historical Remarks

The term "dynamic programming" is due to Bellman (1957a), who showed how these methods could be applied to a wide range of problems. Extensive treatments of DP can be found in many texts, including Bertsekas (2005, 2012), Bertsekas and Tsitsiklis (1996), Dreyfus and Law (1977), Ross (1983), White (1969), and Whittle (1982, 1983). Our interest in DP is restricted to its use in solving MDPs, but DP also applies to other types of problems. Kumar and Kanal (1988) provide a more general look at DP.

To the best of our knowledge, the first connection between DP and reinforcement learning was made by Minsky (1961) in commenting on Samuel's checkers player. In a footnote, Minsky mentioned that it is possible to apply DP to problems in which Samuel's backing-up process can be handled in closed analytic form. This remark may have misled artificial intelligence researchers into believing that DP was restricted to analytically tractable problems and therefore largely irrelevant to artificial intelligence. Andreae (1969b) mentioned DP in the context of reinforcement learning, specifically policy iteration, although he did not make specific connections between DP and learning algorithms. Werbos (1977) suggested an approach to approximating DP called "heuristic dynamic programming" that emphasizes gradient-descent methods for continuous-state problems (Werbos, 1982, 1987, 1988, 1989, 1992). These methods are closely related to the reinforcement learning algorithms that we discuss in this book. Watkins (1989) was explicit in connecting reinforcement learning to DP, characterizing a class of reinforcement learning methods as "incremental dynamic programming."

4.1–4 These sections describe well-established DP algorithms that are covered in any of the general DP references cited above. The policy improvement theorem and the policy iteration algorithm are due to Bellman (1957a) and Howard (1960). Our presentation was influenced by the local view of policy improvement taken by Watkins (1989). Our discussion of value iteration as a form of truncated policy iteration is based on the approach of Puterman and Shin (1978), who presented a class of algorithms called *modified policy iteration*, which includes policy iteration and value iteration as special cases. An analysis showing how value iteration can be made to find an optimal policy in finite time is given by Bertsekas (1987).

Iterative policy evaluation is an example of a classical successive approximation algorithm for solving a system of linear equations. The version of the algorithm that uses two arrays, one holding the old values while the other is updated, is often called a *Jacobi-style* algorithm, after Jacobi's classical use of this method. It is also sometimes called a *synchronous* algorithm because the effect is as if all the values are updated at the same time. The second array is needed to simulate this parallel computation sequentially. The in-place version of the algorithm is often called a *Gauss–Seidel-style* algorithm after the classical Gauss–Seidel algorithm for solving systems of linear equations. In addition to iterative policy evaluation, other DP algorithms can be implemented in these different versions. Bertsekas and Tsitsiklis (1989) provide excellent coverage of these variations and their performance differences.

4.5 Asynchronous DP algorithms are due to Bertsekas (1982, 1983), who also called them distributed DP algorithms. The original motivation for asynchronous DP was its implementation on a multiprocessor system with communication delays between processors and no global synchronizing clock. These algorithms are extensively discussed by Bertsekas and Tsitsiklis (1989). Jacobi-style and Gauss–Seidel-style DP algorithms are special cases of the asynchronous version. Williams and Baird (1990) presented DP algorithms that are asynchronous at a finer grain than the ones we have discussed: the update operations themselves are broken into steps that can be performed asynchronously.

4.7 This section, written with the help of Michael Littman, is based on Littman, Dean, and Kaelbling (1995). The phrase "curse of dimensionality" is due to Bellman (1957a).

Foundational work on the linear programming approach to reinforcement learning was done by Daniela de Farias (de Farias, 2002; de Farias and Van Roy, 2003).

Chapter 5

Monte Carlo Methods

In this chapter we consider our first learning methods for estimating value functions and discovering optimal policies. Unlike the previous chapter, here we do not assume complete knowledge of the environment. Monte Carlo methods require only *experience*—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Learning from *actual* experience is striking because it requires no prior knowledge of the environment's dynamics, yet can still attain optimal behavior. Learning from *simulated* experience is also powerful. Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP). In surprisingly many cases it is easy to generate experience sampled according to the desired probability distributions, but infeasible to obtain the distributions in explicit form.

Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, here we define Monte Carlo methods only for episodic tasks. That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. Only on the completion of an episode are value estimates and policies changed. Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense. The term "Monte Carlo" is often used more broadly for any estimation method whose operation involves a significant random component. Here we use it specifically for methods based on averaging complete returns (as opposed to methods that learn from partial returns, considered in the next chapter).

Monte Carlo methods sample and average *returns* for each state–action pair much like the bandit methods we explored in Chapter 2 sample and average *rewards* for each action. The main difference is that now there are multiple states, each acting like a different bandit problem (like an associative-search or contextual bandit) and the different bandit problems are interrelated. That is, the return after taking an action in one state depends on the actions taken in later states in the same episode. Because all the action selections are undergoing learning, the problem becomes nonstationary from the point of view of the earlier state.

To handle the nonstationarity, we adapt the idea of general policy iteration (GPI) developed in Chapter 4 for DP. Whereas there we *computed* value functions from knowledge of the MDP, here we *learn* value functions from sample returns with the MDP. The value functions and corresponding policies still interact to attain optimality in essentially the same way (GPI). As in the DP chapter, first we consider the prediction problem (the computation of v_π and q_π for a fixed arbitrary policy π) then policy improvement, and, finally, the control problem and its solution by GPI. Each of these ideas taken from DP is extended to the Monte Carlo case in which only sample experience is available.

5.1 Monte Carlo Prediction

We begin by considering Monte Carlo methods for learning the state-value function for a given policy. Recall that the value of a state is the expected return—expected cumulative future discounted reward—starting from that state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

In particular, suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a *visit* to s. Of course, s may be visited multiple times in the same episode; let us call the first time it is visited in an episode the *first visit* to s. The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the *every-visit MC method* averages the returns following all visits to s. These two Monte Carlo (MC) methods are very similar but have slightly different theoretical properties. First-visit MC has been most widely studied, dating back to the 1940s, and is the one we focus on in this chapter. Every-visit MC extends more naturally to function approximation and eligibility traces, as discussed in Chapters 9 and 12. First-visit MC is shown in procedural form in the box. Every-visit MC would be the same except without the check for S_t having occurred earlier in the episode.

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

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average($Returns(S_t)$)

Both first-visit MC and every-visit MC converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to infinity. This is easy to see for the case of first-visit MC. In this case each return is an independent, identically distributed estimate of $v_\pi(s)$ with finite variance. By the law of large numbers the sequence of averages of these estimates converges to their expected value. Each average is itself an unbiased estimate, and the standard deviation of its error falls as $1/\sqrt{n}$, where n is the number of returns averaged. Every-visit MC is less straightforward, but its estimates also converge quadratically to $v_\pi(s)$ (Singh and Sutton, 1996).

The use of Monte Carlo methods is best illustrated through an example.

Example 5.1: Blackjack The object of the popular casino card game of *blackjack* is to obtain cards the sum of whose numerical values is as great as possible without exceeding 21. All face cards count as 10, and an ace can count either as 1 or as 11. We consider the version in which each player competes independently against the dealer. The game begins with two cards dealt to both dealer and player. One of the dealer's cards is face up and the other is face down. If the player has 21 immediately (an ace and a 10-card), it is called a *natural*. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, then he can request additional cards, one by one (*hits*), until he either stops (*sticks*) or exceeds 21 (*goes bust*). If he goes bust, he loses; if he sticks, then it becomes the dealer's turn. The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise, the outcome—win, lose, or draw—is determined by whose final sum is closer to 21.

Playing blackjack is naturally formulated as an episodic finite MDP. Each game of blackjack is an episode. Rewards of +1, −1, and 0 are given for winning, losing, and drawing, respectively. All rewards within a game are zero, and we do not discount ($\gamma = 1$); therefore these terminal rewards are also the returns. The player's actions are to hit or to stick. The states depend on the player's cards and the dealer's showing card. We assume that cards are dealt from an infinite deck (i.e., with replacement) so that there is no advantage to keeping track of the cards already dealt. If the player holds an ace that he could count as 11 without going bust, then the ace is said to be *usable*. In this case it is always counted as 11 because counting it as 1 would make the sum 11 or less, in which case there is no decision to be made because, obviously, the player should always hit. Thus, the player makes decisions on the basis of three variables: his current sum (12–21), the dealer's one showing card (ace–10), and whether or not he holds a usable ace. This makes for a total of 200 states.

Consider the policy that sticks if the player's sum is 20 or 21, and otherwise hits. To find the state-value function for this policy by a Monte Carlo approach, one simulates many blackjack games using the policy and averages the returns following each state. In this way, we obtained the estimates of the state-value function shown in Figure 5.1. The estimates for states with a usable ace are less certain and less regular because these states are less common. In any event, after 500,000 games the value function is very well approximated.

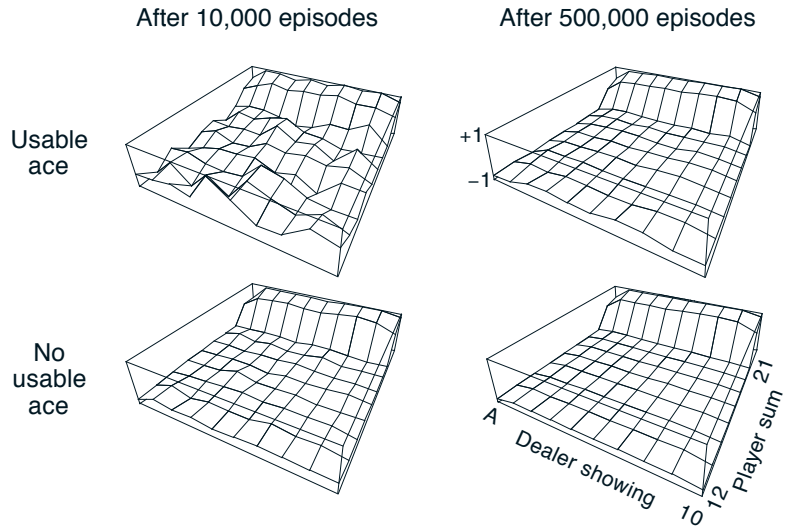


Figure 5.1: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation. ■

Exercise 5.1 Consider the diagrams on the right in Figure 5.1. Why does the estimated value function jump up for the last two rows in the rear? Why does it drop off for the whole last row on the left? Why are the frontmost values higher in the upper diagrams than in the lower? □

Exercise 5.2 Suppose every-visit MC was used instead of first-visit MC on the blackjack task. Would you expect the results to be very different? Why or why not? □

Although we have complete knowledge of the environment in the blackjack task, it would not be easy to apply DP methods to compute the value function. DP methods require the distribution of next events—in particular, they require the environments dynamics as given by the four-argument function p—and it is not easy to determine this for blackjack. For example, suppose the player's sum is 14 and he chooses to stick. What is his probability of terminating with a reward of +1 as a function of the dealer's showing card? All of the probabilities must be computed *before* DP can be applied, and such computations are often complex and error-prone. In contrast, generating the sample games required by Monte Carlo methods is easy. This is the case surprisingly often; the ability of Monte Carlo methods to work with sample episodes alone can be a significant advantage even when one has complete knowledge of the environment's dynamics.

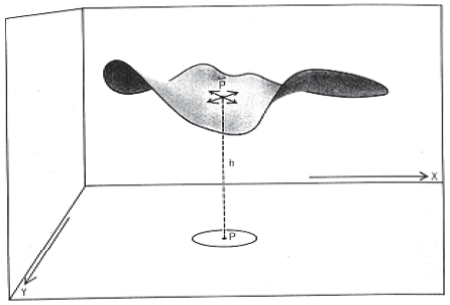
Can we generalize the idea of backup diagrams to Monte Carlo algorithms? The general idea of a backup diagram is to show at the top the root node to be updated and to show below all the transitions and leaf nodes whose rewards and estimated values contribute to the update. For Monte Carlo estimation of v_π, the root is a state node, and below it is the entire trajectory of transitions along a particular single episode, ending

at the terminal state, as shown to the right. Whereas the DP diagram (page 59) shows all possible transitions, the Monte Carlo diagram shows only those sampled on the one episode. Whereas the DP diagram includes only one-step transitions, the Monte Carlo diagram goes all the way to the end of the episode. These differences in the diagrams accurately reflect the fundamental differences between the algorithms.

An important fact about Monte Carlo methods is that the estimates for each state are independent. The estimate for one state does not build upon the estimate of any other state, as is the case in DP. In other words, Monte Carlo methods do not *bootstrap* as we defined it in the previous chapter.

In particular, note that the computational expense of estimating the value of a single state is independent of the number of states. This can make Monte Carlo methods particularly attractive when one requires the value of only one or a subset of states. One can generate many sample episodes starting from the states of interest, averaging returns from only these states, ignoring all others. This is a third advantage Monte Carlo methods can have over DP methods (after the ability to learn from actual experience and from simulated experience).

Example 5.2: Soap Bubble Suppose a wire frame forming a closed loop is dunked in soapy water to form a soap surface or bubble conforming at its edges to the wire frame. If the geometry of the wire frame is irregular but known, how can you compute the shape of the surface? The shape has the property that the total force on each point exerted by neighboring points is zero (or else the shape would change). This means that the surface's height at any point is the average of its heights at points in a small circle around that point. In addition, the surface must meet at its boundaries with the wire frame. The usual approach to problems of this kind is to put a grid over the area covered by the surface and solve for its height at the grid points by an iterative computation. Grid points at the boundary are forced to the wire frame, and all others are adjusted toward the average of the heights of their four nearest neighbors. This process then iterates, much like DP's iterative policy evaluation, and ultimately converges to a close approximation to the desired surface.



A bubble on a wire loop.

From Hersh and Griego (1969). Reproduced with permission. ©1969 Scientific American, a division of Nature America, Inc. All rights reserved.

This is similar to the kind of problem for which Monte Carlo methods were originally designed. Instead of the iterative computation described above, imagine standing on the surface and taking a random walk, stepping randomly from grid point to neighboring grid point, with equal probability, until you reach the boundary. It turns out that the expected value of the height at the boundary is a close approximation to the height of the desired surface at the starting point (in fact, it is exactly the value computed by the iterative method described above). Thus, one can closely approximate the height of the surface at a point by simply averaging the boundary heights of many walks started at the point. If one is interested in only the value at one point, or any fixed small set of points, then this Monte Carlo method can be far more efficient than the iterative method based on local consistency. ■

5.2 Monte Carlo Estimation of Action Values

If a model is not available, then it is particularly useful to estimate *action* values (the values of state–action pairs) rather than *state* values. With a model, state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state, as we did in the chapter on DP. Without a model, however, state values alone are not sufficient. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Thus, one of our primary goals for Monte Carlo methods is to estimate q_*. To achieve this, we first consider the policy evaluation problem for action values.

The policy evaluation problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π. The Monte Carlo methods for this are essentially the same as just presented for state values, except now we talk about visits to a state–action pair rather than to a state. A state–action pair s, a is said to be visited in an episode if ever the state s is visited and action a is taken in it. The every-visit MC method estimates the value of a state–action pair as the average of the returns that have followed all the visits to it. The first-visit MC method averages the returns following the first time in each episode that the state was visited and the action was selected. These methods converge quadratically, as before, to the true expected values as the number of visits to each state–action pair approaches infinity.

The only complication is that many state–action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience. This is a serious problem because the purpose of learning action values is to help in choosing among the actions available in each state. To compare alternatives we need to estimate the value of *all* the actions from each state, not just the one we currently favor.

This is the general problem of *maintaining exploration*, as discussed in the context of the k-armed bandit problem in Chapter 2. For policy evaluation to work for action values, we must assure continual exploration. One way to do this is by specifying that the episodes *start in a state–action pair*, and that every pair has a nonzero probability of being selected as the start. This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. We call this the assumption of *exploring starts*.

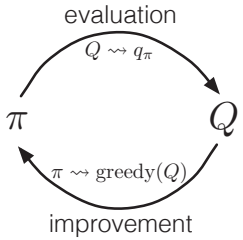
The assumption of exploring starts is sometimes useful, but of course it cannot be relied upon in general, particularly when learning directly from actual interaction with an environment. In that case the starting conditions are unlikely to be so helpful. The most common alternative approach to assuring that all state–action pairs are encountered is

to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state. We discuss two important variants of this approach in later sections. For now, we retain the assumption of exploring starts and complete the presentation of a full Monte Carlo control method.

Exercise 5.3 What is the backup diagram for Monte Carlo estimation of q_π? □

5.3 Monte Carlo Control

We are now ready to consider how Monte Carlo estimation can be used in control, that is, to approximate optimal policies. The overall idea is to proceed according to the same pattern as in the DP chapter, that is, according to the idea of generalized policy iteration (GPI). In GPI one maintains both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function, as suggested by the diagram to the right. These two kinds of changes work against each other to some extent, as each creates a moving target for the other, but together they cause both policy and value function to approach optimality.



To begin, let us consider a Monte Carlo version of classical policy iteration. In this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

where $\xrightarrow{\text{E}}$ denotes a complete policy evaluation and $\xrightarrow{\text{I}}$ denotes a complete policy improvement. Policy evaluation is done exactly as described in the preceding section. Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. For the moment, let us assume that we do indeed observe an infinite number of episodes and that, in addition, the episodes are generated with exploring starts. Under these assumptions, the Monte Carlo methods will compute each q_{π_k} exactly, for arbitrary π_k.

Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an *action*-value function, and therefore no model is needed to construct the greedy policy. For any action-value function q, the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$, deterministically chooses an action with maximal action-value:

$$\pi(s) \doteq \arg\max_a q(s,a). \tag{5.1}$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}. The policy improvement theorem (Section 4.2) then applies to π_k

and π_{k+1} because, for all $s \in \mathcal{S}$,

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s,a)) \\ &= \max_a q_{\pi_k}(s,a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

As we discussed in the previous chapter, the theorem assures us that each π_{k+1} is uniformly better than π_k, or just as good as π_k, in which case they are both optimal policies. This in turn assures us that the overall process converges to the optimal policy and optimal value function. In this way Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.

We made two unlikely assumptions above in order to easily obtain this guarantee of convergence for the Monte Carlo method. One was that the episodes have exploring starts, and the other was that policy evaluation could be done with an infinite number of episodes. To obtain a practical algorithm we will have to remove both assumptions. We postpone consideration of the first assumption until later in this chapter.

For now we focus on the assumption that policy evaluation operates on an infinite number of episodes. This assumption is relatively easy to remove. In fact, the same issue arises even in classical DP methods such as iterative policy evaluation, which also converge only asymptotically to the true value function. In both DP and Monte Carlo cases there are two ways to solve the problem. One is to hold firm to the idea of approximating q_{π_k} in each policy evaluation. Measurements and assumptions are made to obtain bounds on the magnitude and probability of error in the estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficiently small. This approach can probably be made completely satisfactory in the sense of guaranteeing correct convergence up to some level of approximation. However, it is also likely to require far too many episodes to be useful in practice on any but the smallest problems.

There is a second approach to avoiding the infinite number of episodes nominally required for policy evaluation, in which we give up trying to complete policy evaluation before returning to policy improvement. On each evaluation step we move the value function *toward* q_{π_k}, but we do not expect to actually get close except over many steps. We used this idea when we first introduced the idea of GPI in Section 4.6. One extreme form of the idea is value iteration, in which only one iteration of iterative policy evaluation is performed between each step of policy improvement. The in-place version of value iteration is even more extreme; there we alternate between improvement and evaluation steps for single states.

For Monte Carlo policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. A complete simple algorithm along these lines, which we call *Monte Carlo ES*, for Monte Carlo with Exploring Starts, is given in pseudocode in the box on the next page.

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Exercise 5.4 The pseudocode for Monte Carlo ES is inefficient because, for each state–action pair, it maintains a list of all returns and repeatedly calculates their mean. It would be more efficient to use techniques similar to those explained in Section 2.4 to maintain just the mean and a count (for each state–action pair) and update them incrementally. Describe how the pseudocode would be altered to achieve this. □

In Monte Carlo ES, all the returns for each state–action pair are accumulated and averaged, irrespective of what policy was in force when they were observed. It is easy to see that Monte Carlo ES cannot converge to any suboptimal policy. If it did, then the value function would eventually converge to the value function for that policy, and that in turn would cause the policy to change. Stability is achieved only when both the policy and the value function are optimal. Convergence to this optimal fixed point seems inevitable as the changes to the action-value function decrease over time, but has not yet been formally proved. In our opinion, this is one of the most fundamental open theoretical questions in reinforcement learning (for a partial solution, see Tsitsiklis, 2002).

Example 5.3: Solving Blackjack It is straightforward to apply Monte Carlo ES to blackjack. Because the episodes are all simulated games, it is easy to arrange for exploring starts that include all possibilities. In this case one simply picks the dealer's cards, the player's sum, and whether or not the player has a usable ace, all at random with equal probability. As the initial policy we use the policy evaluated in the previous blackjack example, that which sticks only on 20 or 21. The initial action-value function can be zero for all state–action pairs. Figure 5.2 shows the optimal policy for blackjack found by Monte Carlo ES. This policy is the same as the "basic" strategy of Thorp (1966) with the sole exception of the leftmost notch in the policy for a usable ace, which is not present in Thorp's strategy. We are uncertain of the reason for this discrepancy, but confident that what is shown here is indeed the optimal policy for the version of blackjack we have described.

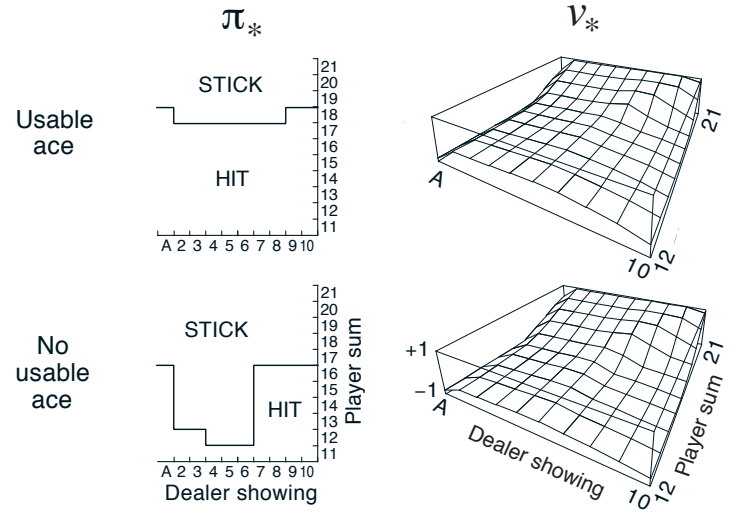


Figure 5.2: The optimal policy and state-value function for blackjack, found by Monte Carlo ES. The state-value function shown was computed from the action-value function found by Monte Carlo ES. ■

5.4 Monte Carlo Control without Exploring Starts

How can we avoid the unlikely assumption of exploring starts? The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches to ensuring this, resulting in what we call *on-policy* methods and *off-policy* methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. The Monte Carlo ES method developed above is an example of an on-policy method. In this section we show how an on-policy Monte Carlo control method can be designed that does not use the unrealistic assumption of exploring starts. Off-policy methods are considered in the next section.

In on-policy control methods the policy is generally *soft*, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$, but gradually shifted closer and closer to a deterministic optimal policy. Many of the methods discussed in Chapter 2 provide mechanisms for this. The on-policy method we present in this section uses *ε-greedy* policies, meaning that most of the time they choose an action that has maximal estimated action value, but with probability ε they instead select an action at random. That is, all nongreedy actions are given the minimal probability of selection, $\frac{\varepsilon}{|\mathcal{A}(s)|}$, and the remaining bulk of the probability, $1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$, is given to the greedy action. The ε-greedy policies are examples of *ε-soft* policies, defined as policies for which $\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $\varepsilon > 0$. Among ε-soft policies, ε-greedy policies are in some sense those

that are closest to greedy.

The overall idea of on-policy Monte Carlo control is still that of GPI. As in Monte Carlo ES, we use first-visit MC methods to estimate the action-value function for the current policy. Without the assumption of exploring starts, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of nongreedy actions. Fortunately, GPI does not require that the policy be taken all the way to a greedy policy, only that it be moved *toward* a greedy policy. In our on-policy method we will move it only to an ε-greedy policy. For any ε-soft policy, π, any ε-greedy policy with respect to q_π is guaranteed to be better than or equal to π. The complete algorithm is given in the box below.

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

That any ε-greedy policy with respect to q_π is an improvement over any ε-soft policy π is assured by the policy improvement theorem. Let π' be the ε-greedy policy. The conditions of the policy improvement theorem apply because for any $s \in \mathcal{S}$:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \max_a q_\pi(s,a) \qquad (5.2) \\ &\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum_a \frac{\pi(a|s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1-\varepsilon} q_\pi(s,a) \end{aligned}$$

(the sum is a weighted average with nonnegative weights summing to 1, and as such it must be less than or equal to the largest number averaged)

$$\begin{aligned} &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) - \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a|s) q_\pi(s,a) \\ &= v_\pi(s). \end{aligned}$$

Thus, by the policy improvement theorem, $\pi' \geq \pi$ (i.e., $v_{\pi'}(s) \geq v_\pi(s)$, for all $s \in \mathcal{S}$). We now prove that equality can hold only when both π' and π are optimal among the ε-soft policies, that is, when they are better than or equal to all other ε-soft policies.

Consider a new environment that is just like the original environment, except with the requirement that policies be ε-soft "moved inside" the environment. The new environment has the same action and state set as the original and behaves as follows. If in state s and taking action a, then with probability $1-\varepsilon$ the new environment behaves exactly like the old environment. With probability ε it repicks the action at random, with equal probabilities, and then behaves like the old environment with the new, random action. The best one can do in this new environment with general policies is the same as the best one could do in the original environment with ε-soft policies. Let $\tilde{v}_*$ and $\tilde{q}_*$ denote the optimal value functions for the new environment. Then a policy π is optimal among ε-soft policies if and only if $v_\pi = \tilde{v}_*$. From the definition of $\tilde{v}_*$ we know that it is the unique solution to

$$\begin{aligned} \tilde{v}_*(s) &= (1-\varepsilon) \max_a \tilde{q}_*(s,a) + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \tilde{q}_*(s,a) \\ &= (1-\varepsilon) \max_a \sum_{s',r} p(s', r \,|\, s, a) \Big[r + \gamma \tilde{v}_*(s') \Big] \\ &\quad + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s',r} p(s', r \,|\, s, a) \Big[r + \gamma \tilde{v}_*(s') \Big]. \end{aligned}$$

When equality holds and the ε-soft policy π is no longer improved, then we also know, from (5.2), that

$$\begin{aligned} v_\pi(s) &= (1-\varepsilon) \max_a q_\pi(s,a) + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) \\ &= (1-\varepsilon) \max_a \sum_{s',r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big] \\ &\quad + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s',r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big]. \end{aligned}$$

However, this equation is the same as the previous one, except for the substitution of v_π for $\tilde{v}_*$. Because $\tilde{v}_*$ is the unique solution, it must be that $v_\pi = \tilde{v}_*$.

In essence, we have shown in the last few pages that policy iteration works for ε-soft policies. Using the natural notion of greedy policy for ε-soft policies, one is assured of improvement on every step, except when the best policy has been found among the ε-soft policies. This analysis is independent of how the action-value functions are determined at each stage, but it does assume that they are computed exactly. This brings us to

roughly the same point as in the previous section. Now we only achieve the best policy among the ε-soft policies, but on the other hand, we have eliminated the assumption of exploring starts.

5.5 Off-policy Prediction via Importance Sampling

All learning control methods face a dilemma: They seek to learn action values conditional on subsequent *optimal* behavior, but they need to behave non-optimally in order to explore all actions (to *find* the optimal actions). How can they learn about the optimal policy while behaving according to an exploratory policy? The on-policy approach in the preceding section is actually a compromise—it learns action values not for the optimal policy, but for a near-optimal policy that still explores. A more straightforward approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. The policy being learned about is called the *target policy*, and the policy used to generate behavior is called the *behavior policy*. In this case we say that learning is from data "off" the target policy, and the overall process is termed *off-policy learning*.

Throughout the rest of this book we consider both on-policy and off-policy methods. On-policy methods are generally simpler and are considered first. Off-policy methods require additional concepts and notation, and because the data is due to a different policy, off-policy methods are often of greater variance and are slower to converge. On the other hand, off-policy methods are more powerful and general. They include on-policy methods as the special case in which the target and behavior policies are the same. Off-policy methods also have a variety of additional uses in applications. For example, they can often be applied to learn from data generated by a conventional non-learning controller, or from a human expert. Off-policy learning is also seen by some as key to learning multi-step predictive models of the world's dynamics (see Section 17.2; Sutton, 2009; Sutton et al., 2011).

In this section we begin the study of off-policy methods by considering the *prediction* problem, in which both target and behavior policies are fixed. That is, suppose we wish to estimate v_π or q_π, but all we have are episodes following another policy b, where $b \neq \pi$. In this case, π is the target policy, b is the behavior policy, and both policies are considered fixed and given.

In order to use episodes from b to estimate values for π, we require that every action taken under π is also taken, at least occasionally, under b. That is, we require that $\pi(a|s) > 0$ implies $b(a|s) > 0$. This is called the assumption of *coverage*. It follows from coverage that b must be stochastic in states where it is not identical to π. The target policy π, on the other hand, may be deterministic, and, in fact, this is a case of particular interest in control applications. In control, the target policy is typically the deterministic greedy policy with respect to the current estimate of the action-value function. This policy becomes a deterministic optimal policy while the behavior policy remains stochastic and more exploratory, for example, an ε-greedy policy. In this section, however, we consider the prediction problem, in which π is unchanging and given.

Almost all off-policy methods utilize *importance sampling*, a general technique for

estimating expected values under one distribution given samples from another. We apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the *importance-sampling ratio*. Given a starting state S_t, the probability of the subsequent state–action trajectory, $A_t, S_{t+1}, A_{t+1}, \dots, S_T$, occurring under any policy π is

$$\begin{aligned}
&\Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T \mid S_t, A_{t:T-1} \sim \pi\} \\
&\quad = \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\
&\quad = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k),
\end{aligned}$$

where p here is the state-transition probability function defined by (3.4). Thus, the relative probability of the trajectory under the target and behavior policies (the importance-sampling ratio) is

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{5.3}$$

Although the trajectory probabilities depend on the MDP's transition probabilities, which are generally unknown, they appear identically in both the numerator and denominator, and thus cancel. The importance sampling ratio ends up depending only on the two policies and the sequence, not on the MDP.

Recall that we wish to estimate the expected returns (values) under the target policy, but all we have are returns G_t due to the behavior policy. These returns have the wrong expectation $\mathbb{E}[G_t|S_t = s] = v_b(s)$ and so cannot be averaged to obtain v_π. This is where importance sampling comes in. The ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value:

$$\mathbb{E}[\rho_{t:T-1}G_t \mid S_t = s] = v_\pi(s). \tag{5.4}$$

Now we are ready to give a Monte Carlo algorithm that averages returns from a batch of observed episodes following policy b to estimate $v_\pi(s)$. It is convenient here to number time steps in a way that increases across episode boundaries. That is, if the first episode of the batch ends in a terminal state at time 100, then the next episode begins at time $t = 101$. This enables us to use time-step numbers to refer to particular steps in particular episodes. In particular, we can define the set of all time steps in which state s is visited, denoted $\mathcal{T}(s)$. This is for an every-visit method; for a first-visit method, $\mathcal{T}(s)$ would only include time steps that were first visits to s within their episodes. Also, let $T(t)$ denote the first time of termination following time t, and G_t denote the return after t up through $T(t)$. Then $\{G_t\}_{t\in\mathcal{T}(s)}$ are the returns that pertain to state s, and $\{\rho_{t:T(t)-1}\}_{t\in\mathcal{T}(s)}$ are the corresponding importance-sampling ratios. To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}G_t}{|\mathcal{T}(s)|}. \tag{5.5}$$

When importance sampling is done as a simple average in this way it is called *ordinary importance sampling*.

An important alternative is *weighted importance sampling*, which uses a *weighted* average, defined as

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}}, \tag{5.6}$$

or zero if the denominator is zero. To understand these two varieties of importance sampling, consider the estimates of their first-visit methods after observing a single return from state s. In the weighted-average estimate, the ratio $\rho_{t:T(t)-1}$ for the single return cancels in the numerator and denominator, so that the estimate is equal to the observed return independent of the ratio (assuming the ratio is nonzero). Given that this return was the only one observed, this is a reasonable estimate, but its expectation is $v_b(s)$ rather than $v_\pi(s)$, and in this statistical sense it is biased. In contrast, the first-visit version of the ordinary importance-sampling estimator (5.5) is always $v_\pi(s)$ in expectation (it is unbiased), but it can be extreme. Suppose the ratio were ten, indicating that the trajectory observed is ten times as likely under the target policy as under the behavior policy. In this case the ordinary importance-sampling estimate would be *ten times* the observed return. That is, it would be quite far from the observed return even though the episode's trajectory is considered very representative of the target policy.

Formally, the difference between the first-visit methods of the two kinds of importance sampling is expressed in their biases and variances. Ordinary importance sampling is unbiased whereas weighted importance sampling is biased (though the bias converges asymptotically to zero). On the other hand, the variance of ordinary importance sampling is in general unbounded because the variance of the ratios can be unbounded, whereas in the weighted estimator the largest weight on any single return is one. In fact, assuming bounded returns, the variance of the weighted importance-sampling estimator converges to zero even if the variance of the ratios themselves is infinite (Precup, Sutton, and Dasgupta 2001). In practice, the weighted estimator usually has dramatically lower variance and is strongly preferred. Nevertheless, we will not totally abandon ordinary importance sampling as it is easier to extend to the approximate methods using function approximation that we explore in the second part of this book.

The every-visit methods for ordinary and weighed importance sampling are both biased, though, again, the bias falls asymptotically to zero as the number of samples increases. In practice, every-visit methods are often preferred because they remove the need to keep track of which states have been visited and because they are much easier to extend to approximations. A complete every-visit MC algorithm for off-policy policy evaluation using weighted importance sampling is given in the next section on page 110.

Exercise 5.5 Consider an MDP with a single nonterminal state and a single action that transitions back to the nonterminal state with probability p and transitions to the terminal state with probability $1-p$. Let the reward be $+1$ on all transitions, and let $\gamma=1$. Suppose you observe one episode that lasts 10 steps, with a return of 10. What are the first-visit and every-visit estimators of the value of the nonterminal state? □

Example 5.4: Off-policy Estimation of a Blackjack State Value We applied both ordinary and weighted importance-sampling methods to estimate the value of a single blackjack state (Example 5.1) from off-policy data. Recall that one of the advantages of Monte Carlo methods is that they can be used to evaluate a single state without forming estimates for any other states. In this example, we evaluated the state in which the dealer is showing a deuce, the sum of the player's cards is 13, and the player has a usable ace (that is, the player holds an ace and a deuce, or equivalently three aces). The data was generated by starting in this state then choosing to hit or stick at random with equal probability (the behavior policy). The target policy was to stick only on a sum of 20 or 21, as in Example 5.1. The value of this state under the target policy is approximately -0.27726 (this was determined by separately generating one-hundred million episodes using the target policy and averaging their returns). Both off-policy methods closely approximated this value after 1000 off-policy episodes using the random policy. To make sure they did this reliably, we performed 100 independent runs, each starting from estimates of zero and learning for 10,000 episodes. Figure 5.3 shows the resultant learning curves—the squared error of the estimates of each method as a function of number of episodes, averaged over the 100 runs. The error approaches zero for both algorithms, but the weighted importance-sampling method has much lower error at the beginning, as is typical in practice.

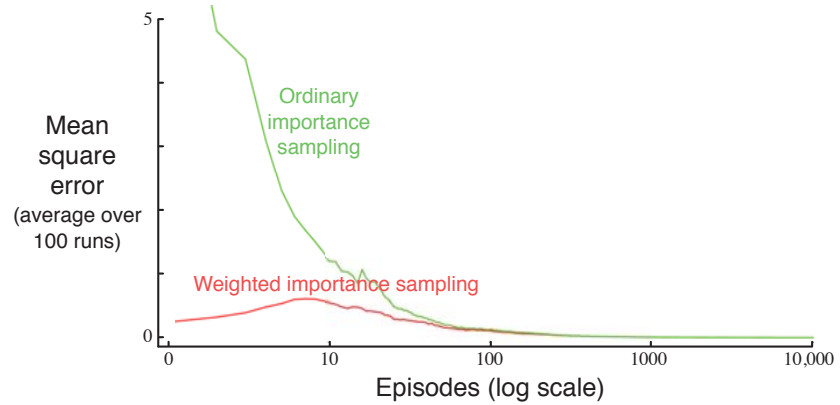


Figure 5.3: Weighted importance sampling produces lower error estimates of the value of a single blackjack state from off-policy episodes. ■

Example 5.5: Infinite Variance The estimates of ordinary importance sampling will typically have infinite variance, and thus unsatisfactory convergence properties, whenever the scaled returns have infinite variance—and this can easily happen in off-policy learning when trajectories contain loops. A simple example is shown inset in Figure 5.4. There is only one nonterminal state s and two actions, right and left. The right action causes a deterministic transition to termination, whereas the left action transitions, with probability 0.9, back to s or, with probability 0.1, on to termination. The rewards are $+1$ on the latter transition and otherwise zero. Consider the target policy that always selects left. All episodes under this policy consist of some number (possibly zero) of transitions back

to s followed by termination with a reward and return of +1. Thus the value of s under the target policy is 1 ($\gamma = 1$). Suppose we are estimating this value from off-policy data using the behavior policy that selects right and left with equal probability.

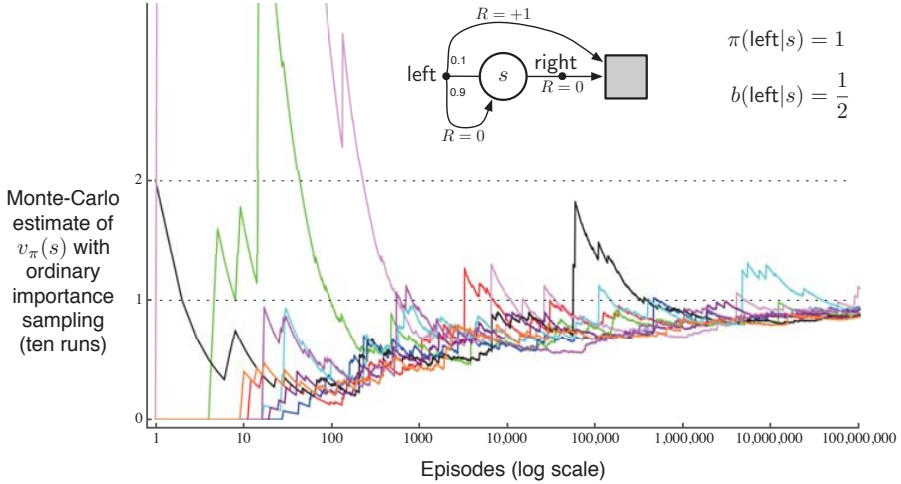


Figure 5.4: Ordinary importance sampling produces surprisingly unstable estimates on the one-state MDP shown inset (Example 5.5). The correct estimate here is 1 ($\gamma = 1$), and, even though this is the expected value of a sample return (after importance sampling), the variance of the samples is infinite, and the estimates do not converge to this value. These results are for off-policy first-visit MC.

The lower part of Figure 5.4 shows ten independent runs of the first-visit MC algorithm using ordinary importance sampling. Even after millions of episodes, the estimates fail to converge to the correct value of 1. In contrast, the weighted importance-sampling algorithm would give an estimate of exactly 1 forever after the first episode that ended with the left action. All returns not equal to 1 (that is, ending with the right action) would be inconsistent with the target policy and thus would have a $\rho_{t:T(t)-1}$ of zero and contribute neither to the numerator nor denominator of (5.6). The weighted importance-sampling algorithm produces a weighted average of only the returns consistent with the target policy, and all of these would be exactly 1.

We can verify that the variance of the importance-sampling-scaled returns is infinite in this example by a simple calculation. The variance of any random variable X is the expected value of the deviation from its mean $\bar{X}$, which can be written

$$\mathrm{Var}[X] \doteq \mathbb{E}\Big[\big(X - \bar{X}\big)^2\Big] = \mathbb{E}\big[X^2 - 2X\bar{X} + \bar{X}^2\big] = \mathbb{E}\big[X^2\big] - \bar{X}^2.$$

Thus, if the mean is finite, as it is in our case, the variance is infinite if and only if the

expectation of the square of the random variable is infinite. Thus, we need only show that the expected square of the importance-sampling-scaled return is infinite:

$$\mathbb{E}_b\left[\left(\prod_{t=0}^{T-1}\frac{\pi(A_t|S_t)}{b(A_t|S_t)}G_0\right)^2\right].$$

To compute this expectation, we break it down into cases based on episode length and termination. First note that, for any episode ending with the right action, the importance sampling ratio is zero, because the target policy would never take this action; these episodes thus contribute nothing to the expectation (the quantity in parenthesis will be zero) and can be ignored. We need only consider episodes that involve some number (possibly zero) of left actions that transition back to the nonterminal state, followed by a left action transitioning to termination. All of these episodes have a return of 1, so the G_0 factor can be ignored. To get the expected square we need only consider each length of episode, multiplying the probability of the episode's occurrence by the square of its importance-sampling ratio, and add these up:

$$\begin{aligned}
&= \frac{1}{2}\cdot 0.1\left(\frac{1}{0.5}\right)^2 && \text{(the length 1 episode)}\\
&\quad + \frac{1}{2}\cdot 0.9\cdot\frac{1}{2}\cdot 0.1\left(\frac{1}{0.5}\frac{1}{0.5}\right)^2 && \text{(the length 2 episode)}\\
&\quad + \frac{1}{2}\cdot 0.9\cdot\frac{1}{2}\cdot 0.9\cdot\frac{1}{2}\cdot 0.1\left(\frac{1}{0.5}\frac{1}{0.5}\frac{1}{0.5}\right)^2 && \text{(the length 3 episode)}\\
&\quad + \cdots\\
&= 0.1\sum_{k=0}^{\infty} 0.9^k\cdot 2^k\cdot 2 = \ 0.2\sum_{k=0}^{\infty} 1.8^k = \ \infty. && \blacksquare
\end{aligned}$$

Exercise 5.6 What is the equation analogous to (5.6) for *action* values $Q(s,a)$ instead of state values $V(s)$, again given returns generated using b? □

Exercise 5.7 In learning curves such as those shown in Figure 5.3 error generally decreases with training, as indeed happened for the ordinary importance-sampling method. But for the weighted importance-sampling method error first increased and then decreased. Why do you think this happened? □

Exercise 5.8 The results with Example 5.5 and shown in Figure 5.4 used a first-visit MC method. Suppose that instead an every-visit MC method was used on the same problem. Would the variance of the estimator still be infinite? Why or why not? □

5.6 Incremental Implementation

Monte Carlo prediction methods can be implemented incrementally, on an episode-by-episode basis, using extensions of the techniques described in Chapter 2 (Section 2.4). Whereas in Chapter 2 we averaged *rewards*, in Monte Carlo methods we average *returns*. In all other respects exactly the same methods as used in Chapter 2 can be used for *on-policy* Monte Carlo methods. For *off-policy* Monte Carlo methods, we need to separately consider those that use *ordinary* importance sampling and those that use *weighted* importance sampling.

In ordinary importance sampling, the returns are scaled by the importance sampling ratio $\rho_{t:T(t)-1}$ (5.3), then simply averaged, as in (5.5). For these methods we can again use the incremental methods of Chapter 2, but using the scaled returns in place of the rewards of that chapter. This leaves the case of off-policy methods using *weighted* importance sampling. Here we have to form a weighted average of the returns, and a slightly different incremental algorithm is required.

Suppose we have a sequence of returns $G_1, G_2, \ldots, G_{n-1}$, all starting in the same state and each with a corresponding random weight W_i (e.g., $W_i = \rho_{t_i:T(t_i)-1}$). We wish to form the estimate

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \qquad n \geq 2, \tag{5.7}$$

and keep it up-to-date as we obtain a single additional return G_n. In addition to keeping track of V_n, we must maintain for each state the cumulative sum C_n of the weights given to the first n returns. The update rule for V_n is

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\Big[G_n - V_n\Big], \qquad n \geq 1, \tag{5.8}$$

and

$$C_{n+1} \doteq C_n + W_{n+1},$$

where $C_0 \doteq 0$ (and V_1 is arbitrary and thus need not be specified). The box on the next page contains a complete episode-by-episode incremental algorithm for Monte Carlo policy evaluation. The algorithm is nominally for the off-policy case, using weighted importance sampling, but applies as well to the on-policy case just by choosing the target and behavior policies as the same (in which case ($\pi = b$), W is always 1). The approximation Q converges to q_π (for all encountered state–action pairs) while actions are selected according to a potentially different policy, b.

Exercise 5.9 Modify the algorithm for first-visit MC policy evaluation (Section 5.1) to use the incremental implementation for sample averages described in Section 2.4. □

Exercise 5.10 Derive the weighted-average update rule (5.8) from (5.7). Follow the pattern of the derivation of the unweighted rule (2.3). □

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \in \mathbb{R}$ (arbitrarily)
  $C(s,a) \leftarrow 0$

Loop forever (for each episode):
  $b \leftarrow$ any policy with coverage of π
  Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}\left[G - Q(S_t, A_t)\right]$
    $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
    If $W = 0$ then exit For loop

5.7 Off-policy Monte Carlo Control

We are now ready to present an example of the second class of learning control methods we consider in this book: off-policy methods. Recall that the distinguishing feature of on-policy methods is that they estimate the value of a policy while using it for control. In off-policy methods these two functions are separated. The policy used to generate behavior, called the *behavior* policy, may in fact be unrelated to the policy that is evaluated and improved, called the *target* policy. An advantage of this separation is that the target policy may be deterministic (e.g., greedy), while the behavior policy can continue to sample all possible actions.

Off-policy Monte Carlo control methods use one of the techniques presented in the preceding two sections. They follow the behavior policy while learning about and improving the target policy. These techniques require that the behavior policy has a nonzero probability of selecting all actions that might be selected by the target policy (coverage). To explore all possibilities, we require that the behavior policy be soft (i.e., that it select all actions in all states with nonzero probability).

The box on the next page shows an off-policy Monte Carlo control method, based on GPI and weighted importance sampling, for estimating π_* and q_*. The target policy $\pi \approx \pi_*$ is the greedy policy with respect to Q, which is an estimate of q_π. The behavior policy b can be anything, but in order to assure convergence of π to the optimal policy, an infinite number of returns must be obtained for each pair of state and action. This can be assured by choosing b to be ε-soft. The policy π converges to optimal at all encountered states even though actions are selected according to a different soft policy b, which may change between or even within episodes.

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \in \mathbb{R}$ (arbitrarily)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

A potential problem is that this method learns only from the tails of episodes, when all of the remaining actions in the episode are greedy. If nongreedy actions are common, then learning will be slow, particularly for states appearing in the early portions of long episodes. Potentially, this could greatly slow learning. There has been insufficient experience with off-policy Monte Carlo methods to assess how serious this problem is. If it is serious, the most important way to address it is probably by incorporating temporal-difference learning, the algorithmic idea developed in the next chapter. Alternatively, if γ is less than 1, then the idea developed in the next section may also help significantly.

Exercise 5.11 In the boxed algorithm for off-policy MC control, you may have been expecting the W update to have involved the importance-sampling ratio $\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$, but instead it involves $\frac{1}{b(A_t|S_t)}$. Why is this nevertheless correct? □

Exercise 5.12: Racetrack (programming) Consider driving a race car around a turn like those shown in Figure 5.5. You want to go as fast as possible, but not so fast as to run off the track. In our simplified racetrack, the car is at one of a discrete set of grid positions, the cells in the diagram. The velocity is also discrete, a number of grid cells moved horizontally and vertically per time step. The actions are increments to the velocity components. Each may be changed by +1, −1, or 0 in each step, for a total of nine (3×3) actions. Both velocity components are restricted to be nonnegative and less than 5, and they cannot both be zero except at the starting line. Each episode begins in one of the randomly selected start states with both velocity components zero and ends when the car crosses the finish line. The rewards are −1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random position on the starting line, both velocity components are reduced to zero, and the

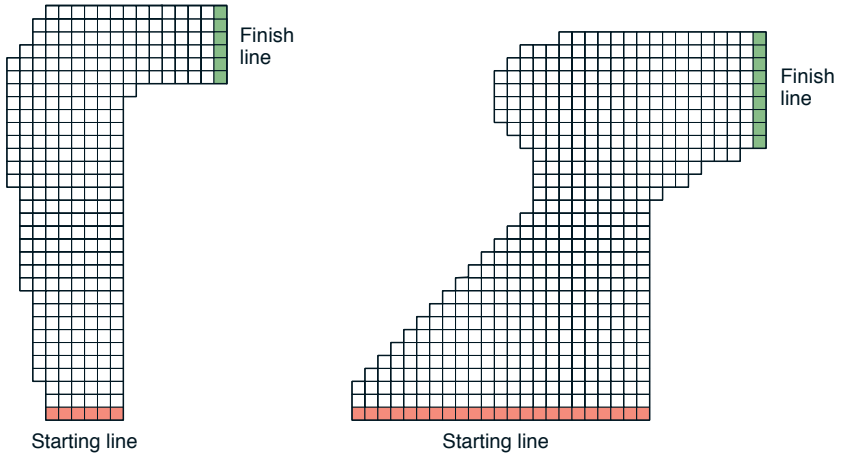


Figure 5.5: A couple of right turns for the racetrack task.

episode continues. Before updating the car's location at each time step, check to see if the projected path of the car intersects the track boundary. If it intersects the finish line, the episode ends; if it intersects anywhere else, the car is considered to have hit the track boundary and is sent back to the starting line. To make the task more challenging, with probability 0.1 at each time step the velocity increments are both zero, independently of the intended increments. Apply a Monte Carlo control method to this task to compute the optimal policy from each starting state. Exhibit several trajectories following the optimal policy (but turn the noise off for these trajectories). □

5.8 *Discounting-aware Importance Sampling

The off-policy methods that we have considered so far are based on forming importance-sampling weights for returns considered as unitary wholes, without taking into account the returns' internal structures as sums of discounted rewards. We now briefly consider cutting-edge research ideas for using this structure to significantly reduce the variance of off-policy estimators.

For example, consider the case where episodes are long and γ is significantly less than 1. For concreteness, say that episodes last 100 steps and that $\gamma = 0$. The return from time 0 will then be just $G_0 = R_1$, but its importance sampling ratio will be a product of 100 factors, $\frac{\pi(A_0|S_0)}{b(A_0|S_0)} \frac{\pi(A_1|S_1)}{b(A_1|S_1)} \cdots \frac{\pi(A_{99}|S_{99})}{b(A_{99}|S_{99})}$. In ordinary importance sampling, the return will be scaled by the entire product, but it is really only necessary to scale by the first factor, by $\frac{\pi(A_0|S_0)}{b(A_0|S_0)}$. The other 99 factors $\frac{\pi(A_1|S_1)}{b(A_1|S_1)} \cdots \frac{\pi(A_{99}|S_{99})}{b(A_{99}|S_{99})}$ are irrelevant because after the first reward the return has already been determined. These later factors are all independent of the return and of expected value 1; they do not change the expected update, but they add enormously to its variance. In some cases they could even make the

variance infinite. Let us now consider an idea for avoiding this large extraneous variance.

The essence of the idea is to think of discounting as determining a probability of termination or, equivalently, a *degree* of partial termination. For any $\gamma \in [0,1)$, we can think of the return G_0 as partly terminating in one step, to the degree $1-\gamma$, producing a return of just the first reward, R_1, and as partly terminating after two steps, to the degree $(1-\gamma)\gamma$, producing a return of $R_1 + R_2$, and so on. The latter degree corresponds to terminating on the second step, $1-\gamma$, and not having already terminated on the first step, γ. The degree of termination on the third step is thus $(1-\gamma)\gamma^2$, with the γ^2 reflecting that termination did not occur on either of the first two steps. The partial returns here are called *flat partial returns*:

$$\bar{G}_{t:h} \doteq R_{t+1} + R_{t+2} + \cdots + R_h, \qquad 0 \le t < h \le T,$$

where "flat" denotes the absence of discounting, and "partial" denotes that these returns do not extend all the way to termination but instead stop at h, called the *horizon* (and T is the time of termination of the episode). The conventional full return G_t can be viewed as a sum of flat partial returns as suggested above as follows:

$$\begin{aligned}
G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \\
&= (1-\gamma) R_{t+1} \\
&\quad + (1-\gamma)\gamma \left(R_{t+1} + R_{t+2}\right) \\
&\quad + (1-\gamma)\gamma^2 \left(R_{t+1} + R_{t+2} + R_{t+3}\right) \\
&\quad \vdots \\
&\quad + (1-\gamma)\gamma^{T-t-2} \left(R_{t+1} + R_{t+2} + \cdots + R_{T-1}\right) \\
&\quad + \gamma^{T-t-1} \left(R_{t+1} + R_{t+2} + \cdots + R_T\right) \\
&= (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} \quad + \quad \gamma^{T-t-1} \bar{G}_{t:T}.
\end{aligned}$$

Now we need to scale the flat partial returns by an importance sampling ratio that is similarly truncated. As $\bar{G}_{t:h}$ only involves rewards up to a horizon h, we only need the ratio of the probabilities up to h. We define an ordinary importance-sampling estimator, analogous to (5.5), as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} \;+\; \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{|\mathcal{T}(s)|}, \tag{5.9}$$

and a weighted importance-sampling estimator, analogous to (5.6), as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} \;+\; \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \;+\; \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \right)}. \tag{5.10}$$

We call these two estimators *discounting-aware* importance sampling estimators. They take into account the discount rate but have no effect (are the same as the off-policy estimators from Section 5.5) if $\gamma = 1$.

5.9 *Per-decision Importance Sampling

There is one more way in which the structure of the return as a sum of rewards can be taken into account in off-policy importance sampling, a way that may be able to reduce variance even in the absence of discounting (that is, even if $\gamma = 1$). In the off-policy estimators (5.5) and (5.6), each term of the sum in the numerator is itself a sum:

$$\begin{aligned}
\rho_{t:T-1} G_t &= \rho_{t:T-1} \left(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T \right) \\
&= \rho_{t:T-1} R_{t+1} + \gamma \rho_{t:T-1} R_{t+2} + \cdots + \gamma^{T-t-1} \rho_{t:T-1} R_T.
\end{aligned} \tag{5.11}$$

The off-policy estimators rely on the expected values of these terms, which can be written in a simpler way. Note that each sub-term of (5.11) is a product of a random reward and a random importance-sampling ratio. For example, the first sub-term can be written, using (5.3), as

$$\rho_{t:T-1} R_{t+1} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})} \frac{\pi(A_{t+2}|S_{t+2})}{b(A_{t+2}|S_{t+2})} \cdots \frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})} R_{t+1}. \tag{5.12}$$

Of all these factors, one might suspect that only the first and the last (the reward) are related; all the others are for events that occurred after the reward. Moreover, the expected value of all these other factors is one:

$$\mathbb{E}\left[\frac{\pi(A_k|S_k)}{b(A_k|S_k)} \right] \doteq \sum_a b(a|S_k) \frac{\pi(a|S_k)}{b(a|S_k)} = \sum_a \pi(a|S_k) = 1. \tag{5.13}$$

With a few more steps, one can show that, as suspected, all of these other factors have no effect in expectation, in other words, that

$$\mathbb{E}[\rho_{t:T-1} R_{t+1}] = \mathbb{E}[\rho_{t:t} R_{t+1}]. \tag{5.14}$$

If we repeat this process for the kth sub-term of (5.11), we get

$$\mathbb{E}[\rho_{t:T-1} R_{t+k}] = \mathbb{E}[\rho_{t:t+k-1} R_{t+k}].$$

It follows then that the expectation of our original term (5.11) can be written

$$\mathbb{E}[\rho_{t:T-1} G_t] = \mathbb{E}\left[\tilde{G}_t\right],$$

where

$$\tilde{G}_t = \rho_{t:t} R_{t+1} + \gamma \rho_{t:t+1} R_{t+2} + \gamma^2 \rho_{t:t+2} R_{t+3} + \cdots + \gamma^{T-t-1} \rho_{t:T-1} R_T.$$

We call this idea *per-decision* importance sampling. It follows immediately that there is an alternate importance-sampling estimator, with the same unbiased expectation (in the first-visit case) as the ordinary-importance-sampling estimator (5.5), using $\tilde{G}_t$:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \tilde{G}_t}{|\mathcal{T}(s)|}, \tag{5.15}$$

which we might expect to sometimes be of lower variance.

Is there a per-decision version of *weighted* importance sampling? This is less clear. So far, all the estimators that have been proposed for this that we know of are not consistent (that is, they do not converge to the true value with infinite data).

**Exercise 5.13* Show the steps to derive (5.14) from (5.12). □

**Exercise 5.14* Modify the algorithm for off-policy Monte Carlo control (page 111) to use the idea of the truncated weighted-average estimator (5.10). Note that you will first need to convert this equation to action values. □

5.10 Summary

The Monte Carlo methods presented in this chapter learn value functions and optimal policies from experience in the form of *sample episodes*. This gives them at least three kinds of advantages over DP methods. First, they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics. Second, they can be used with simulation or *sample models*. For surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods. Third, it is easy and efficient to *focus* Monte Carlo methods on a small subset of the states. A region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set (we explore this further in Chapter 8).

A fourth advantage of Monte Carlo methods, which we discuss later in the book, is that they may be less harmed by violations of the Markov property. This is because they do not update their value estimates on the basis of the value estimates of successor states. In other words, it is because they do not bootstrap.

In designing Monte Carlo control methods we have followed the overall schema of *generalized policy iteration* (GPI) introduced in Chapter 4. GPI involves interacting processes of policy evaluation and policy improvement. Monte Carlo methods provide an alternative policy evaluation process. Rather than use a model to compute the value of each state, they simply average many returns that start in the state. Because a state's value is the expected return, this average can become a good approximation to the value. In control methods we are particularly interested in approximating action-value functions, because these can be used to improve the policy without requiring a model of the environment's transition dynamics. Monte Carlo methods intermix policy evaluation and policy improvement steps on an episode-by-episode basis, and can be incrementally implemented on an episode-by-episode basis.

Maintaining *sufficient exploration* is an issue in Monte Carlo control methods. It is not enough just to select the actions currently estimated to be best, because then no returns will be obtained for alternative actions, and it may never be learned that they are actually better. One approach is to ignore this problem by assuming that episodes begin with state–action pairs randomly selected to cover all possibilities. Such *exploring starts* can sometimes be arranged in applications with simulated episodes, but are unlikely in learning from real experience. In *on-policy* methods, the agent commits to always exploring and tries to find the best policy that still explores. In *off-policy* methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

Off-policy prediction refers to learning the value function of a *target policy* from data generated by a different *behavior policy*. Such learning methods are based on some form of *importance sampling*, that is, on weighting returns by the ratio of the probabilities of taking the observed actions under the two policies, thereby transforming their expectations from the behavior policy to the target policy. *Ordinary importance sampling* uses a simple average of the weighted returns, whereas *weighted importance sampling* uses a weighted average. Ordinary importance sampling produces unbiased estimates, but has larger, possibly infinite, variance, whereas weighted importance sampling always has finite variance and is preferred in practice. Despite their conceptual simplicity, off-policy Monte Carlo methods for both prediction and control remain unsettled and are a subject of ongoing research.

The Monte Carlo methods treated in this chapter differ from the DP methods treated in the previous chapter in two major ways. First, they operate on sample experience, and thus can be used for direct learning without a model. Second, they do not bootstrap. That is, they do not update their value estimates on the basis of other value estimates. These two differences are not tightly linked, and can be separated. In the next chapter we consider methods that learn from experience, like Monte Carlo methods, but also bootstrap, like DP methods.

Bibliographical and Historical Remarks

The term "Monte Carlo" dates from the 1940s, when physicists at Los Alamos devised games of chance that they could study to help understand complex physical phenomena relating to the atom bomb. Coverage of Monte Carlo methods in this sense can be found in several textbooks (e.g., Kalos and Whitlock, 1986; Rubinstein, 1981).

5.1–2 Singh and Sutton (1996) distinguished between every-visit and first-visit MC methods and proved results relating these methods to reinforcement learning algorithms. The blackjack example is based on an example used by Widrow, Gupta, and Maitra (1973). The soap bubble example is a classical Dirichlet problem whose Monte Carlo solution was first proposed by Kakutani (1945; see Hersh and Griego, 1969; Doyle and Snell, 1984).

Barto and Duff (1994) discussed policy evaluation in the context of classical Monte Carlo algorithms for solving systems of linear equations. They used the analysis of Curtiss (1954) to point out the computational advantages of Monte Carlo policy evaluation for large problems.

5.3–4 Monte Carlo ES was introduced in the 1998 edition of this book. That may have been the first explicit connection between Monte Carlo estimation and control methods based on policy iteration. An early use of Monte Carlo methods to estimate action values in a reinforcement learning context was by Michie and

Chambers (1968). In pole balancing (page 56), they used averages of episode durations to assess the worth (expected balancing "life") of each possible action in each state, and then used these assessments to control action selections. Their method is similar in spirit to Monte Carlo ES with every-visit MC estimates. Narendra and Wheeler (1986) studied a Monte Carlo method for ergodic finite Markov chains that used the return accumulated between successive visits to the same state as a reward for adjusting a learning automaton's action probabilities.

5.5 Efficient off-policy learning has become recognized as an important challenge that arises in several fields. For example, it is closely related to the idea of "interventions" and "counterfactuals" in probabalistic graphical (Bayesian) models (e.g., Pearl, 1995; Balke and Pearl, 1994). Off-policy methods using importance sampling have a long history and yet still are not well understood. Weighted importance sampling, which is also sometimes called normalized importance sampling (e.g., Koller and Friedman, 2009), is discussed by Rubinstein (1981), Hesterberg (1988), Shelton (2001), and Liu (2001) among others.

The target policy in off-policy learning is sometimes referred to in the literature as the "estimation" policy, as it was in the first edition of this book.

5.7 The racetrack exercise is adapted from Barto, Bradtke, and Singh (1995), and from Gardner (1973).

5.8 Our treatment of the idea of discounting-aware importance sampling is based on the analysis of Sutton, Mahmood, Precup, and van Hasselt (2014). It has been worked out most fully to date by Mahmood (2017; Mahmood, van Hasselt, and Sutton, 2014).

5.9 Per-decision importance sampling was introduced by Precup, Sutton, and Singh (2000). They also combined off-policy learning with temporal-difference learning, eligibility traces, and approximation methods, introducing subtle issues that we consider in later chapters.

Chapter 6

Temporal-Difference Learning

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be *temporal-difference* (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The relationship between TD, DP, and Monte Carlo methods is a recurring theme in the theory of reinforcement learning; this chapter is the beginning of our exploration of it. Before we are done, we will see that these ideas and methods blend into each other and can be combined in many ways. In particular, in Chapter 7 we introduce n-step algorithms, which provide a bridge from TD to Monte Carlo methods, and in Chapter 12 we introduce the TD(λ) algorithm, which seamlessly unifies them.

As usual, we start by focusing on the policy evaluation or *prediction* problem, the problem of estimating the value function v_π for a given policy π. For the *control* problem (finding an optimal policy), DP, TD, and Monte Carlo methods all use some variation of generalized policy iteration (GPI). The differences in the methods are primarily differences in their approaches to the prediction problem.

6.1 TD Prediction

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate V of v_π for the nonterminal states S_t occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big], \tag{6.1}$$

where G_t is the actual return following time t, and α is a constant step-size parameter (c.f., Equation 2.4). Let us call this method *constant-α MC*. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known), TD methods need to wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big] \tag{6.2}$$

immediately on transition to S_{t+1} and receiving R_{t+1}. In effect, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called *TD(0)*, or *one-step TD*, because it is a special case of the TD(λ) and n-step TD methods developed in Chapter 12 and Chapter 7. The box below specifies TD(0) completely in procedural form.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Because TD(0) bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP. We know from Chapter 3 that

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] && (6.3)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && \text{(from (3.9))}\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]\,. && (6.4) \end{aligned}$$

Roughly speaking, Monte Carlo methods use an estimate of (6.3) as a target, whereas DP methods use an estimate of (6.4) as a target. The Monte Carlo target is an estimate because the expected value in (6.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (6.4) *and* it uses the current estimate V instead of the true v_π. Thus, TD methods combine the sampling of

Monte Carlo with the bootstrapping of DP. As we shall see, with care and imagination this can take us a long way toward obtaining the advantages of both Monte Carlo and DP methods.

Shown to the right is the backup diagram for tabular TD(0). The value estimate for the state node at the top of the backup diagram is updated on the basis of the one sample transition from it to the immediately following state. We refer to TD and Monte Carlo updates as *sample updates* because they involve looking ahead to a sample successor state (or state–action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state–action pair) accordingly. *Sample* updates differ from the *expected* updates of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.

TD(0)

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

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{6.5}$$

Notice that the TD error at each time is the error in the estimate *made at that time*. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. That is, δ_t is the error in $V(S_t)$, available at time $t+1$. Also note that if the array V does not change during the episode (as it does not in Monte Carlo methods), then the Monte Carlo error can be written as a sum of TD errors:

$$\begin{aligned}
G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) && \text{(from (3.9))} \\
&= \delta_t + \gamma\big(G_{t+1} - V(S_{t+1})\big) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\big(G_{t+2} - V(S_{t+2})\big) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(G_T - V(S_T)\big) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(0 - 0\big) \\
&= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. && (6.6)
\end{aligned}$$

This identity is not exact if V is updated during the episode (as it is in TD(0)), but if the step size is small then it may still hold approximately. Generalizations of this identity play an important role in the theory and algorithms of temporal-difference learning.

Exercise 6.1 If V changes during the episode, then (6.6) only holds approximately; what would the difference be between the two sides? Let V_t denote the array of state values used at time t in the TD error (6.5) and in the TD update (6.2). Redo the derivation above to determine the additional amount that must be added to the sum of TD errors in order to equal the Monte Carlo error. □

Example 6.1: Driving Home Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant. Say on this Friday you are leaving at exactly 6 o'clock, and you estimate that it will take 30 minutes to get home. As you reach your car it is 6:05, and you notice it is starting to rain. Traffic is often slower in the rain, so you reestimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn onto the side street where you live at 6:40. Three minutes later you are home. The sequence of states, times, and predictions is thus as follows:

State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

The rewards in this example are the elapsed times on each leg of the journey.[1] We are not discounting ($\gamma = 1$), and thus the return for each state is the actual time to go from that state. The value of each state is the *expected* time to go. The second column of numbers gives the current estimated value for each state encountered.

A simple way to view the operation of Monte Carlo methods is to plot the predicted total time (the last column) over the sequence, as in Figure 6.1 (left). The red arrows show the changes in predictions recommended by the constant-α MC method (6.1), for $\alpha = 1$. These are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go). For example, when you exited the highway you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes. Equation 6.1 applies at this point and determines an increment in the estimate of time to go after exiting the highway. The error, $G_t - V(S_t)$, at this time is eight minutes. Suppose the step-size parameter, α, is 1/2. Then the predicted time to go after exiting the highway would be revised upward by four minutes as a result of this experience. This is probably too large a change in this case; the truck was probably just an unlucky break. In any event, the change can only be made offline, that is, after you have reached home. Only at this point do you know any of the actual returns.

Is it necessary to wait until the final outcome is known before learning can begin? Suppose on another day you again estimate when leaving your office that it will take 30 minutes to drive home, but then you become stuck in a massive traffic jam. Twenty-five minutes after leaving the office you are still bumper-to-bumper on the highway. You now

[1] If this were a control problem with the objective of minimizing travel time, then we would of course make the rewards the *negative* of the elapsed time. But because we are concerned here only with prediction (policy evaluation), we can keep things simple by using positive numbers.

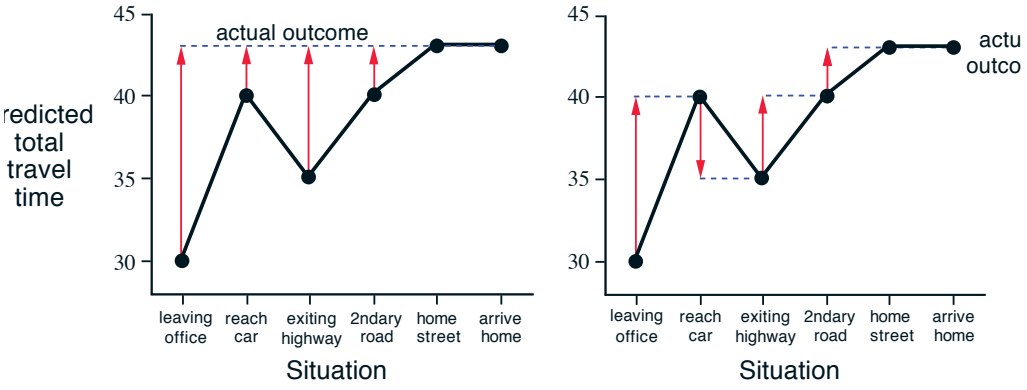


Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

estimate that it will take another 25 minutes to get home, for a total of 50 minutes. As you wait in traffic, you already know that your initial estimate of 30 minutes was too optimistic. Must you wait until you get home before increasing your estimate for the initial state? According to the Monte Carlo approach you must, because you don't yet know the true return.

According to a TD approach, on the other hand, you would learn immediately, shifting your initial estimate from 30 minutes toward 50. In fact, each estimate would be shifted toward the estimate that immediately follows it. Returning to our first day of driving, Figure 6.1 (right) shows the changes in the predictions recommended by the TD rule (6.2) (these are the changes made by the rule if $\alpha = 1$). Each error is proportional to the change over time of the prediction, that is, to the *temporal differences* in predictions.

Besides giving you something to do while waiting in traffic, there are several computational reasons why it is advantageous to learn based on your current predictions rather than waiting until termination when you know the actual return. We briefly discuss some of these in the next section. ■

Exercise 6.2 This is an exercise to help develop your intuition about why TD methods are often more efficient than Monte Carlo methods. Consider the driving home example and how it is addressed by TD and Monte Carlo methods. Can you imagine a scenario in which a TD update would be better on average than a Monte Carlo update? Give an example scenario—a description of past experience and a current state—in which you would expect the TD update to be better. Here's a hint: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original scenario? □

6.2 Advantages of TD Prediction Methods

TD methods update their estimates based in part on other estimates. They learn a guess from a guess—they *bootstrap*. Is this a good thing to do? What advantages do TD methods have over Monte Carlo and DP methods? Developing and answering such questions will take the rest of this book and more. In this section we briefly anticipate some of the answers.

Obviously, TD methods have an advantage over DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

The next most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an online, fully incremental fashion. With Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step. Surprisingly often this turns out to be a critical consideration. Some applications have very long episodes, so that delaying all learning until the end of the episode is too slow. Other applications are continuing tasks and have no episodes at all. Finally, as we noted in the previous chapter, some Monte Carlo methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken.

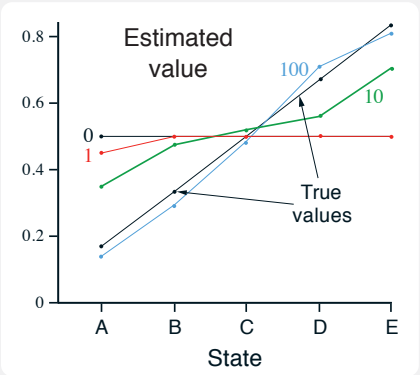
But are TD methods sound? Certainly it is convenient to learn one guess from the next, without waiting for an actual outcome, but can we still guarantee convergence to the correct answer? Happily, the answer is yes. For any fixed policy π, TD(0) has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions (2.7). Most convergence proofs apply only to the table-based case of the algorithm presented above (6.2), but some also apply to the case of general linear function approximation. These results are discussed in a more general setting in Chapter 9.

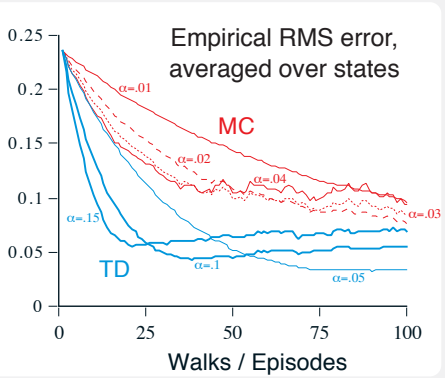
If both TD and Monte Carlo methods converge asymptotically to the correct predictions, then a natural next question is "Which gets there first?" In other words, which method learns faster? Which makes the more efficient use of limited data? At the current time this is an open question in the sense that no one has been able to prove mathematically that one method converges faster than the other. In fact, it is not even clear what is the most appropriate formal way to phrase this question! In practice, however, TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks, as illustrated in Example 6.2.

Example 6.2 Random Walk

In this example we empirically compare the prediction abilities of TD(0) and constant-α MC when applied to the following Markov reward process:

A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which there is no need to distinguish the dynamics due to the environment from those due to the agent. In this MRP, all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero. For example, a typical episode might consist of the following state-and-reward sequence: C, 0, B, 0, C, 0, D, 0, E, 1. Because this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $v_\pi(\mathsf{C}) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$.





The left graph above shows the values learned after various numbers of episodes on a single run of TD(0). The estimates after 100 episodes are about as close as they ever come to the true values—with a constant step-size parameter ($\alpha = 0.1$ in this example), the values fluctuate indefinitely in response to the outcomes of the most recent episodes. The right graph shows learning curves for the two methods for various values of α. The performance measure shown is the root mean-squared (RMS) error between the value function learned and the true value function, averaged over the five states, then averaged over 100 runs. In all cases the approximate value function was initialized to the intermediate value $V(s) = 0.5$, for all s. The TD method was consistently better than the MC method on this task.

Exercise 6.3 From the results shown in the left graph of the random walk example it appears that the first episode results in a change in only $V(\mathsf{A})$. What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed? □

Exercise 6.4 The specific results shown in the right graph of the random walk example are dependent on the value of the step-size parameter, α. Do you think the conclusions about which algorithm is better would be affected if a wider range of α values were used? Is there a different, fixed value of α at which either algorithm would have performed significantly better than shown? Why or why not? □

**Exercise 6.5* In the right graph of the random walk example, the RMS error of the TD method seems to go down and then up again, particularly at high α's. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized? □

Exercise 6.6 In Example 6.2 we stated that the true values for the random walk example are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$, for states A through E. Describe at least two different ways that these could have been computed. Which would you guess we actually used? Why? □

6.3 Optimality of TD(0)

Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function, V, the increments specified by (6.1) or (6.2) are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges. We call this *batch updating* because updates are made only after processing each complete *batch* of training data.

Under batch updating, TD(0) converges deterministically to a single answer independent of the step-size parameter, α, as long as α is chosen to be sufficiently small. The constant-α MC method also converges deterministically under the same conditions, but to a different answer. Understanding these two answers will help us understand the difference between the two methods. Under normal updating the methods do not move all the way to their respective batch answers, but in some sense they take steps in these directions. Before trying to understand the two answers in general, for all possible tasks, we first look at a few examples.

Example 6.3: Random walk under batch updating Batch-updating versions of TD(0) and constant-α MC were applied as follows to the random walk prediction example (Example 6.2). After each new episode, all episodes seen so far were treated as a batch. They were repeatedly presented to the algorithm, either TD(0) or constant-α MC, with α sufficiently small that the value function converged. The resulting value function was then compared with v_π, and the average root mean-squared error across the five states (and across 100 independent repetitions of the whole experiment) was plotted to obtain

the learning curves shown in Figure 6.2. Note that the batch TD method was consistently better than the batch Monte Carlo method.

Under batch training, constant-α MC converges to values, $V(s)$, that are sample averages of the actual returns experienced after visiting each state s. These are optimal estimates in the sense that they minimize the mean-squared error from the actual returns in the training set. In this sense it is surprising that the batch TD method was able to perform better according to the root mean-squared error measure shown in the figure to the right. How is it that batch TD was able to perform better than this optimal method? The answer is that the Monte Carlo method is optimal only in a limited way, and that TD is optimal in a way that is more relevant to predicting returns. ■

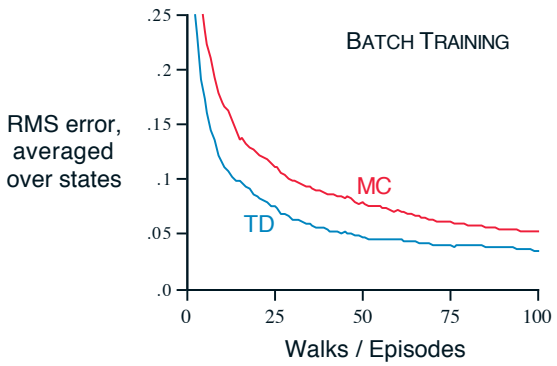


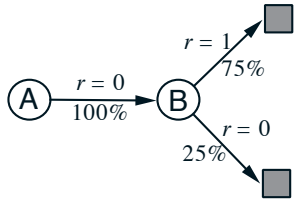
Figure 6.2: Performance of TD(0) and constant-α MC under batch training on the random walk task.

Example 6.4: You are the Predictor Place yourself now in the role of the predictor of returns for an unknown Markov reward process. Suppose you observe the following eight episodes:

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

This means that the first episode started in state A, transitioned to B with a reward of 0, and then terminated from B with a reward of 0. The other seven episodes were even shorter, starting from B and terminating immediately. Given this batch of data, what would you say are the optimal predictions, the best values for the estimates $V(\mathsf{A})$ and $V(\mathsf{B})$? Everyone would probably agree that the optimal value for $V(\mathsf{B})$ is $\frac{3}{4}$, because six out of the eight times in state B the process terminated immediately with a return of 1, and the other two times in B the process terminated immediately with a return of 0.

But what is the optimal value for the estimate $V(\mathsf{A})$ given this data? Here there are two reasonable answers. One is to observe that 100% of the times the process was in state A it traversed immediately to B (with a reward of 0); and because we have already decided that B has value $\frac{3}{4}$, therefore A must have value $\frac{3}{4}$ as well. One way of viewing this answer is that it is based on first modeling the Markov process, in this case as shown to the right, and then computing the correct estimates given the model, which indeed in this case gives $V(\mathsf{A}) = \frac{3}{4}$. This is also the answer that batch TD(0) gives.



The other reasonable answer is simply to observe that we have seen A once and the return that followed it was 0; we therefore estimate $V(\mathsf{A})$ as 0. This is the answer that batch Monte Carlo methods give. Notice that it is also the answer that gives minimum squared error on the training data. In fact, it gives zero error on the data. But still we expect the first answer to be better. If the process is Markov, we expect that the first answer will produce lower error on *future* data, even though the Monte Carlo answer is better on the existing data. ■

Example 6.4 illustrates a general difference between the estimates found by batch TD(0) and batch Monte Carlo methods. Batch Monte Carlo methods always find the estimates that minimize mean-squared error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest. In this case, the maximum-likelihood estimate is the model of the Markov process formed in the obvious way from the observed episodes: the estimated transition probability from i to j is the fraction of observed transitions from i that went to j, and the associated expected reward is the average of the rewards observed on those transitions. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch TD(0) converges to the certainty-equivalence estimate.

This helps explain why TD methods converge more quickly than Monte Carlo methods. In batch form, TD(0) is faster than Monte Carlo methods because it computes the true certainty-equivalence estimate. This explains the advantage of TD(0) shown in the batch results on the random walk task (Figure 6.2). The relationship to the certainty-equivalence estimate may also explain in part the speed advantage of nonbatch TD(0) (e.g., Example 6.2, page 125, right graph). Although the nonbatch methods do not achieve either the certainty-equivalence or the minimum squared-error estimates, they can be understood as moving roughly in these directions. Nonbatch TD(0) may be faster than constant-α MC because it is moving toward a better estimate, even though it is not getting all the way there. At the current time nothing more definite can be said about the relative efficiency of online TD and Monte Carlo methods.

Finally, it is worth noting that although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly. If $n = |\mathcal{S}|$ is the number of states, then just forming the maximum-likelihood estimate of the process may require on the order of n^2 memory, and computing the corresponding value function requires on the order of n^3 computational steps if done conventionally. In these terms it is indeed striking that TD methods can approximate the same solution using memory no more than order n and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

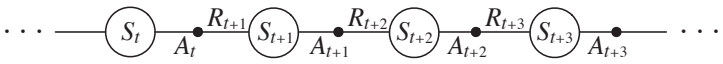
**Exercise 6.7* Design an off-policy version of the TD(0) update that can be used with

arbitrary target policy π and covering behavior policy b, using at each step t the importance sampling ratio $\rho_{t:t}$ (5.3). □

6.4 Sarsa: On-policy TD Control

We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s,a)$ for the current behavior policy π and for all states s and actions a. This can be done using essentially the same TD method described above for learning v_π. Recall that an episode consists of an alternating sequence of states and state–action pairs:



In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state–action pair to state–action pair, and learn the values of state–action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)\Big]. \tag{6.7}$$

This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1},A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm. The backup diagram for Sarsa is as shown to the right.


Sarsa

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

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q. For example, one could use ε-greedy or ε-soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state–action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ε-greedy policies by setting $\varepsilon = 1/t$).

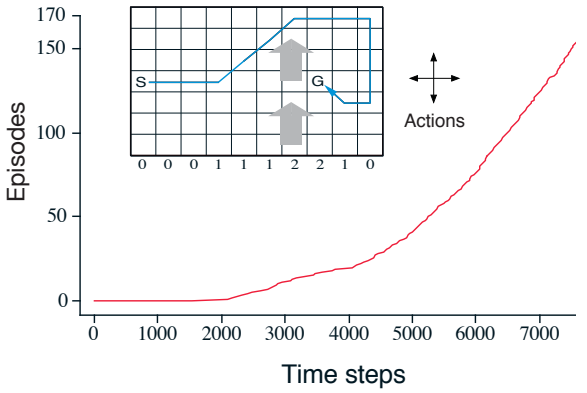
Exercise 6.8 Show that an action-value version of (6.6) holds for the action-value form of the TD error $\delta_t = R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)$, again assuming that the values don't change from step to step. □

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

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Example 6.5: Windy Gridworld Shown inset below is a standard gridworld, with start and goal states, but with one difference: there is a crosswind running upward through the middle of the grid. The actions are the standard four—`up`, `down`, `right`, and `left`—but in the middle region the resultant next states are shifted upward by a "wind," the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted upward. For example, if you are one cell to the right of the goal, then the action `left` takes you to the cell just above the goal. This is an undiscounted episodic task, with constant rewards of −1 until the goal state is reached.



The graph to the right shows the results of applying ε-greedy Sarsa to this task, with $\varepsilon = 0.1$, $\alpha = 0.5$, and the initial values $Q(s,a) = 0$ for all s, a. The increasing slope of the graph shows that the goal was reached more quickly over time. By 8000 time steps, the greedy policy was long since optimal (a trajectory from it is shown inset); continued ε-greedy exploration kept the average episode length at about 17 steps, two more than the minimum of 15. Note that Monte Carlo methods cannot easily be used here because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Online learning methods such as Sarsa do not have this problem because they quickly learn *during the episode* that such policies are poor, and switch to something else. ■

Exercise 6.9: Windy Gridworld with King's Moves (programming) Re-solve the windy gridworld assuming eight possible actions, including the diagonal moves, rather than the

usual four. How much better can you do with the extra actions? Can you do even better by including a ninth action that causes no movement at all other than that caused by the wind? □

Exercise 6.10: Stochastic Wind (programming) Re-solve the windy gridworld task with King's moves, assuming that the effect of the wind, if there is any, is stochastic, sometimes varying by 1 from the mean values given for each column. That is, a third of the time you move exactly according to these values, as in the previous exercise, but also a third of the time you move one cell above that, and another third of the time you move one cell below that. For example, if you are one cell to the right of the goal and you move `left`, then one-third of the time you move one cell above the goal, one-third of the time you move two cells above the goal, and one-third of the time you move to the goal. □

6.5 Q-learning: Off-policy TD Control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989), defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]. \tag{6.8}$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*. The Q-learning algorithm is shown below in procedural form.

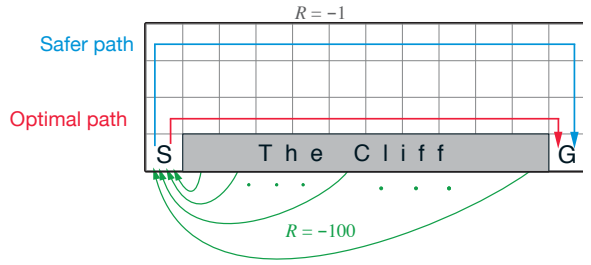
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

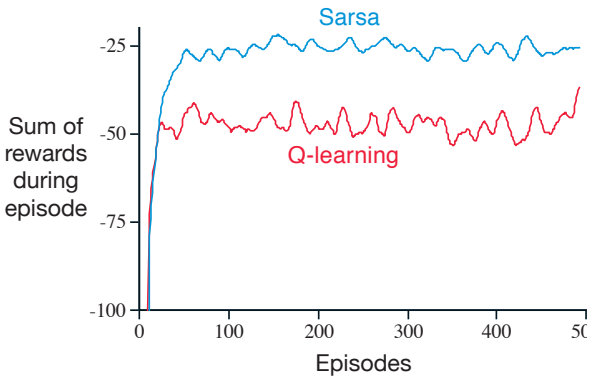
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

What is the backup diagram for Q-learning? The rule (6.8) updates a state–action pair, so the top node, the root of the update, must be a small, filled action node. The update is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these "next action" nodes with an arc across them (Figure 3.4-right). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer in Figure 6.4 on page 134.

Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown to the right. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is −1 on all transitions except those into the region marked "The Cliff." Stepping into this region incurs a reward of −100 and sends the agent instantly back to the start.



The graph to the right shows the performance of the Sarsa and Q-learning methods with ε-greedy action selection, $\varepsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε-greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its online performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy. ■



Exercise 6.11 Why is Q-learning considered an *off-policy* control method? □

Exercise 6.12 Suppose action selection is greedy. Is Q-learning then exactly the same algorithm as Sarsa? Will they make exactly the same action selections and weight updates? □

6.6 Expected Sarsa

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

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned} \tag{6.9}$$

but that otherwise follows the schema of Q-learning. Given the next state, S_{t+1}, this algorithm moves *deterministically* in the same direction as Sarsa moves *in expectation*, and accordingly it is called *Expected Sarsa*. Its backup diagram is shown on the right in Figure 6.4.

Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_{t+1}. Given the same amount of experience we might expect it to perform slightly better than Sarsa, and indeed it generally does. Figure 6.3 shows summary results on the cliff-walking task with Expected Sarsa compared to Sarsa and Q-learning. Expected Sarsa retains the significant advantage of Sarsa over Q-learning on this problem. In addition, Expected Sarsa shows a significant improvement

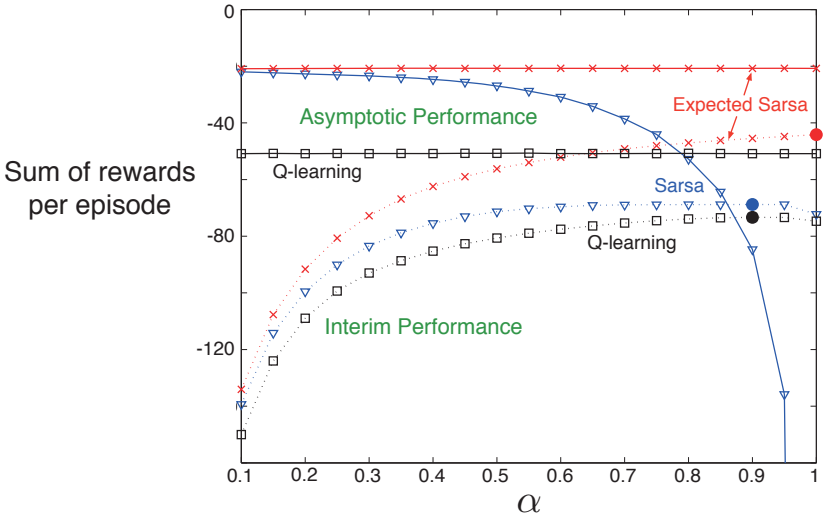


Figure 6.3: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α. All algorithms used an ε-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method. Adapted from van Seijen et al. (2009).

Figure 6.4: The backup diagrams for Q-learning and Expected Sarsa.

over Sarsa over a wide range of values for the step-size parameter α. In cliff walking the state transitions are all deterministic and all randomness comes from the policy. In such cases, Expected Sarsa can safely set $\alpha = 1$ without suffering any degradation of asymptotic performance, whereas Sarsa can only perform well in the long run at a small value of α, at which short-term performance is poor. In this and other examples there is a consistent empirical advantage of Expected Sarsa over Sarsa.

In these cliff walking results Expected Sarsa was used on-policy, but in general it might use a policy different from the target policy π to generate behavior, in which case it becomes an off-policy algorithm. For example, suppose π is the greedy policy while behavior is more exploratory; then Expected Sarsa is exactly Q-learning. In this sense Expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa. Except for the small additional computational cost, Expected Sarsa may completely dominate both of the other more-well-known TD control algorithms.

6.7 Maximization Bias and Double Learning

All the control algorithms that we have discussed so far involve maximization in the construction of their target policies. For example, in Q-learning the target policy is the greedy policy given the current action values, which is defined with a max, and in Sarsa the policy is often ε-greedy, which also involves a maximization operation. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. To see why, consider a single state s where there are many actions a whose true values, $q(s, a)$, are all zero but whose estimated values, $Q(s, a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive, a positive bias. We call this *maximization bias*.

Example 6.7: Maximization Bias Example The small MDP shown inset in Figure 6.5 provides a simple example of how maximization bias can harm the performance of TD control algorithms. The MDP has two non-terminal states A and B. Episodes always start in A with a choice between two actions, left and right. The right action transitions immediately to the terminal state with a reward and return of zero. The left action transitions to B, also with a reward of zero, from which there are many possible actions all of which cause immediate termination with a reward drawn from a normal distribution with mean −0.1 and variance 1.0. Thus, the expected return for any trajectory starting with left is −0.1, and thus taking left in state A is always a

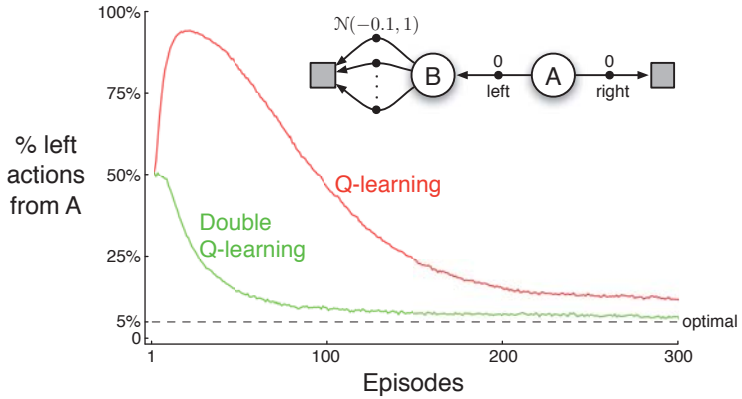


Figure 6.5: Comparison of Q-learning and Double Q-learning on a simple episodic MDP (shown inset). Q-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by ε-greedy action selection with $\varepsilon = 0.1$. In contrast, Double Q-learning is essentially unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. Any ties in ε-greedy action selection were broken randomly.

mistake. Nevertheless, our control methods may favor left because of maximization bias making B appear to have a positive value. Figure 6.5 shows that Q-learning with ε-greedy action selection initially learns to strongly favor the left action on this example. Even at asymptote, Q-learning takes the left action about 5% more often than is optimal at our parameter settings ($\varepsilon = 0.1$, $\alpha = 0.1$, and $\gamma = 1$). ■

Are there algorithms that avoid maximization bias? To start, consider a bandit case in which we have noisy estimates of the value of each of many actions, obtained as sample averages of the rewards received on all the plays with each action. As we discussed above, there will be a positive maximization bias if we use the maximum of the estimates as an estimate of the maximum of the true values. One way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divided the plays in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$, for all $a \in \mathcal{A}$. We could then use one estimate, say Q_1, to determine the maximizing action $A^* = \text{argmax}_a Q_1(a)$, and the other, Q_2, to provide the estimate of its value, $Q_2(A^*) = Q_2(\text{argmax}_a Q_1(a))$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(\text{argmax}_a Q_2(a))$. This is the idea of *double learning*. Note that although we learn two estimates, only one estimate is updated on each play; double learning doubles the memory requirements, but does not increase the amount of computation per step.

The idea of double learning extends naturally to algorithms for full MDPs. For example, the double learning algorithm analogous to Q-learning, called Double Q-learning, divides the time steps in two, perhaps by flipping a coin on each step. If the coin comes up heads,

the update is

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q_2\big(S_{t+1}, \operatorname*{argmax}_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big]. \quad (6.10)$$

If the coin comes up tails, then the same update is done with Q_1 and Q_2 switched, so that Q_2 is updated. The two approximate value functions are treated completely symmetrically. The behavior policy can use both action-value estimates. For example, an ε-greedy policy for Double Q-learning could be based on the average (or sum) of the two action-value estimates. A complete algorithm for Double Q-learning is given in the box below. This is the algorithm used to produce the results in Figure 6.5. In that example, double learning seems to eliminate the harm caused by maximization bias. Of course there are also double versions of Sarsa and Expected Sarsa.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\textit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \text{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \text{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

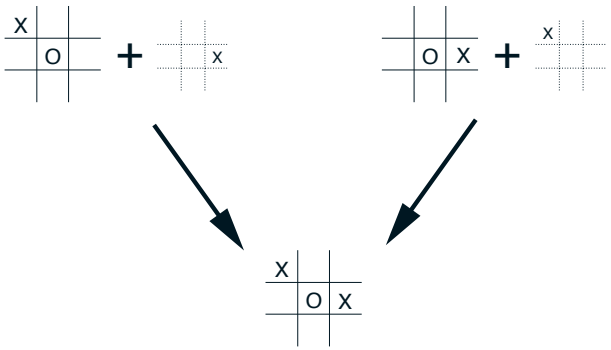
**Exercise 6.13* What are the update equations for Double Expected Sarsa with an ε-greedy target policy? □

6.8 Games, Afterstates, and Other Special Cases

In this book we try to present a uniform approach to a wide class of tasks, but of course there are always exceptional tasks that are better treated in a specialized way. For example, our general approach involves learning an *action*-value function, but in Chapter 1 we presented a TD method for learning to play tic-tac-toe that learned something much more like a *state*-value function. If we look closely at that example, it becomes apparent that the function learned there is neither an action-value function nor a state-value function in the usual sense. A conventional state-value function evaluates states in which the agent has the option of selecting an action, but the state-value function used in

tic-tac-toe evaluates board positions *after* the agent has made its move. Let us call these *afterstates*, and value functions over these, *afterstate value functions*. Afterstates are useful when we have knowledge of an initial part of the environment's dynamics but not necessarily of the full dynamics. For example, in games we typically know the immediate effects of our moves. We know for each possible chess move what the resulting position will be, but not how our opponent will reply. Afterstate value functions are a natural way to take advantage of this kind of knowledge and thereby produce a more efficient learning method.

The reason it is more efficient to design algorithms in terms of afterstates is apparent from the tic-tac-toe example. A conventional action-value function would map from positions *and* moves to an estimate of the value. But many position–move pairs produce the same resulting position, as in the example below:



In such cases the position–move pairs are different but produce the same "afterposition," and thus must have the same value. A conventional action-value function would have to separately assess both pairs, whereas an afterstate value function would immediately assess both equally. Any learning about the position–move pair on the left would immediately transfer to the pair on the right.

Afterstates arise in many tasks, not just games. For example, in queuing tasks there are actions such as assigning customers to servers, rejecting customers, or discarding information. In such cases the actions are in fact defined in terms of their immediate effects, which are completely known.

It is impossible to describe all the possible kinds of specialized problems and corresponding specialized learning algorithms. However, the principles developed in this book should apply widely. For example, afterstate methods are still aptly described in terms of generalized policy iteration, with a policy and (afterstate) value function interacting in essentially the same way. In many cases one will still face the choice between on-policy and off-policy methods for managing the need for persistent exploration.

Exercise 6.14 Describe how the task of Jack's Car Rental (Example 4.2) could be reformulated in terms of afterstates. Why, in terms of this specific task, would such a reformulation be likely to speed convergence? □

6.9 Summary

In this chapter we introduced a new kind of learning method, temporal-difference (TD) learning, and showed how it can be applied to the reinforcement learning problem. As usual, we divided the overall problem into a prediction problem and a control problem. TD methods are alternatives to Monte Carlo methods for solving the prediction problem. In both cases, the extension to the control problem is via the idea of generalized policy iteration (GPI) that we abstracted from dynamic programming. This is the idea that approximate policy and value functions should interact in such a way that they both move toward their optimal values.

One of the two processes making up GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. The other process drives the policy to improve locally (e.g., to be ε-greedy) with respect to the current value function. When the first process is based on experience, a complication arises concerning maintaining sufficient exploration. We can classify TD control methods according to whether they deal with this complication by using an on-policy or off-policy approach. Sarsa is an on-policy method, and Q-learning is an off-policy method. Expected Sarsa is also an off-policy method as we present it here. There is a third way in which TD methods can be extended to control which we did not include in this chapter, called actor–critic methods. These methods are covered in full in Chapter 13.

The methods presented in this chapter are today the most widely used reinforcement learning methods. This is probably due to their great simplicity: they can be applied online, with a minimal amount of computation, to experience generated from interaction with an environment; they can be expressed nearly completely by single equations that can be implemented with small computer programs. In the next few chapters we extend these algorithms, making them slightly more complicated and significantly more powerful. All the new algorithms will retain the essence of those introduced here: they will be able to process experience online, with relatively little computation, and they will be driven by TD errors. The special cases of TD methods introduced in the present chapter should rightly be called *one-step, tabular, model-free* TD methods. In the next two chapters we extend them to n-step forms (a link to Monte Carlo methods) and forms that include a model of the environment (a link to planning and dynamic programming). Then, in the second part of the book we extend them to various forms of function approximation rather than tables (a link to deep learning and artificial neural networks).

Finally, in this chapter we have discussed TD methods entirely within the context of reinforcement learning problems, but TD methods are actually more general than this. They are general methods for learning to make long-term predictions about dynamical systems. For example, TD methods may be relevant to predicting financial data, life spans, election outcomes, weather patterns, animal behavior, demands on power stations, or customer purchases. It was only when TD methods were analyzed as pure prediction methods, independent of their use in reinforcement learning, that their theoretical properties first came to be well understood. Even so, these other potential applications of TD learning methods have not yet been extensively explored.

Bibliographical and Historical Remarks

As we outlined in Chapter 1, the idea of TD learning has its early roots in animal learning psychology and artificial intelligence, most notably the work of Samuel (1959) and Klopf (1972). Samuel's work is described as a case study in Section 16.2. Also related to TD learning are Holland's (1975, 1976) early ideas about consistency among value predictions. These influenced one of the authors (Barto), who was a graduate student from 1970 to 1975 at the University of Michigan, where Holland was teaching. Holland's ideas led to a number of TD-related systems, including the work of Booker (1982) and the bucket brigade of Holland (1986), which is related to Sarsa as discussed below.

6.1–2 Most of the specific material from these sections is from Sutton (1988), including the TD(0) algorithm, the random walk example, and the term "temporal-difference learning." The characterization of the relationship to dynamic programming and Monte Carlo methods was influenced by Watkins (1989), Werbos (1987), and others. The use of backup diagrams was new to the first edition of this book.

Tabular TD(0) was proved to converge in the mean by Sutton (1988) and with probability 1 by Dayan (1992), based on the work of Watkins and Dayan (1992). These results were extended and strengthened by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994) by using extensions of the powerful existing theory of stochastic approximation. Other extensions and generalizations are covered in later chapters.

6.3 The optimality of the TD algorithm under batch training was established by Sutton (1988). Illuminating this result is Barnard's (1993) derivation of the TD algorithm as a combination of one step of an incremental method for learning a model of the Markov chain and one step of a method for computing predictions from the model. The term *certainty equivalence* is from the adaptive control literature (e.g., Goodwin and Sin, 1984).

6.4 The Sarsa algorithm was introduced by Rummery and Niranjan (1994). They explored it in conjunction with artificial neural networks and called it "Modified Connectionist Q-learning". The name "Sarsa" was introduced by Sutton (1996). The convergence of one-step tabular Sarsa (the form treated in this chapter) has been proved by Singh, Jaakkola, Littman, and Szepesvári (2000). The "windy gridworld" example was suggested by Tom Kalt.

Holland's (1986) bucket brigade idea evolved into an algorithm closely related to Sarsa. The original idea of the bucket brigade involved chains of rules triggering each other; it focused on passing credit back from the current rule to the rules that triggered it. Over time, the bucket brigade came to be more like TD learning in passing credit back to any temporally preceding rule, not just to the ones that triggered the current rule. The modern form of the bucket brigade, when simplified in various natural ways, is nearly identical to one-step Sarsa, as detailed by Wilson (1994).

6.5 Q-learning was introduced by Watkins (1989), whose outline of a convergence proof was made rigorous by Watkins and Dayan (1992). More general convergence results were proved by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994).

6.6 The Expected Sarsa algorithm was introduced by George John (1994), who called it "$\overline{\text{Q}}$-learning" and stressed its advantages over Q-learning as an off-policy algorithm. John's work was not known to us when we presented Expected Sarsa in the first edition of this book as an exercise, or to van Seijen, van Hasselt, Whiteson, and Weiring (2009) when they established Expected Sarsa's convergence properties and conditions under which it will outperform regular Sarsa and Q-learning. Our Figure 6.3 is adapted from their results. Van Seijen et al. defined "Expected Sarsa" to be an on-policy method exclusively (as we did in the first edition), whereas now we use this name for the general algorithm in which the target and behavior policies may differ. The general off-policy view of Expected Sarsa was noted by van Hasselt (2011), who called it "General Q-learning."

6.7 Maximization bias and double learning were introduced and extensively investigated by van Hasselt (2010, 2011). The example MDP in Figure 6.5 was adapted from that in his Figure 4.1 (van Hasselt, 2011).

6.8 The notion of an afterstate is the same as that of a "post-decision state" (Van Roy, Bertsekas, Lee, and Tsitsiklis, 1997; Powell, 2011).

Chapter 7

n-step Bootstrapping

In this chapter we unify the Monte Carlo (MC) methods and the one-step temporal-difference (TD) methods presented in the previous two chapters. Neither MC methods nor one-step TD methods are always the best. In this chapter we present *n-step TD methods* that generalize both methods so that one can shift from one to the other smoothly as needed to meet the demands of a particular task. n-step methods span a spectrum with MC methods at one end and one-step TD methods at the other. The best methods are often intermediate between the two extremes.

Another way of looking at the benefits of n-step methods is that they free you from the tyranny of the time step. With one-step TD methods the same time step determines how often the action can be changed and the time interval over which bootstrapping is done. In many applications one wants to be able to update the action very fast to take into account anything that has changed, but bootstrapping works best if it is over a length of time in which a significant and recognizable state change has occurred. With one-step TD methods, these time intervals are the same, and so a compromise must be made. n-step methods enable bootstrapping to occur over multiple steps, freeing us from the tyranny of the single time step.

The idea of n-step methods is usually used as an introduction to the algorithmic idea of *eligibility traces* (Chapter 12), which enable bootstrapping over multiple time intervals simultaneously. Here we instead consider the n-step bootstrapping idea on its own, postponing the treatment of eligibility-trace mechanisms until later. This allows us to separate the issues better, dealing with as many of them as possible in the simpler n-step setting.

As usual, we first consider the prediction problem and then the control problem. That is, we first consider how n-step methods can help in predicting returns as a function of state for a fixed policy (i.e., in estimating v_π). Then we extend the ideas to action values and control methods.

7.1 n-step TD Prediction

What is the space of methods lying between Monte Carlo and TD methods? Consider estimating v_π from sample episodes generated using π. Monte Carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode. The update of one-step TD methods, on the other hand, is based on just the one next reward, bootstrapping from the value of the state one step later as a proxy for the remaining rewards. One kind of intermediate method, then, would perform an update based on an intermediate number of rewards: more than one, but less than all of them until termination. For example, a two-step update would be based on the first two rewards and the estimated value of the state two steps later. Similarly, we could have three-step updates, four-step updates, and so on. Figure 7.1 shows the backup diagrams of the spectrum of *n-step updates* for v_π, with the one-step TD update on the left and the up-until-termination Monte Carlo update on the right.

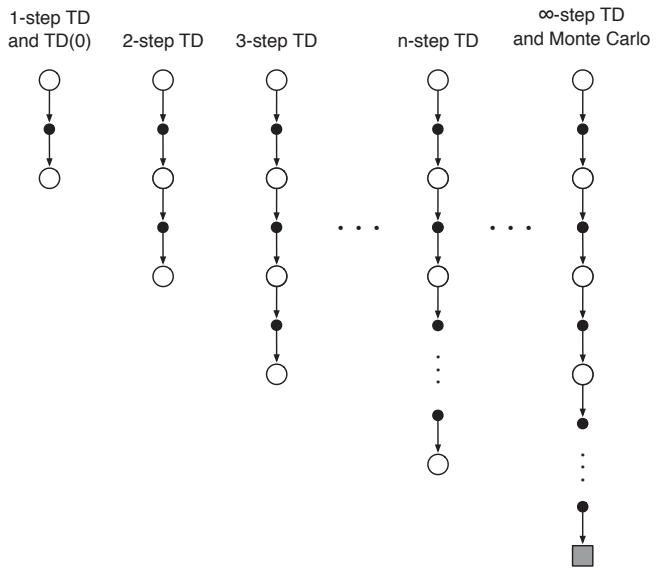


Figure 7.1: The backup diagrams of n-step methods. These methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

The methods that use n-step updates are still TD methods because they still change an earlier estimate based on how it differs from a later estimate. Now the later estimate is not one step later, but n steps later. Methods in which the temporal difference extends over n steps are called *n-step TD methods*. The TD methods introduced in the previous chapter all used one-step updates, which is why we called them one-step TD methods.

More formally, consider the update of the estimated value of state S_t as a result of the state–reward sequence, $S_t, R_{t+1}, S_{t+1}, R_{t+2}, \ldots, R_T, S_T$ (omitting the actions). We know that in Monte Carlo updates the estimate of $v_\pi(S_t)$ is updated in the direction of the

complete return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T,$$

where T is the last time step of the episode. Let us call this quantity the *target* of the update. Whereas in Monte Carlo updates the target is the return, in one-step updates the target is the first reward plus the discounted estimated value of the next state, which we call the *one-step return*:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1}),$$

where $V_t : \mathcal{S} \to \mathbb{R}$ here is the estimate at time t of v_π. The subscripts on $G_{t:t+1}$ indicate that it is a truncated return for time t using rewards up until time $t+1$, with the discounted estimate $\gamma V_t(S_{t+1})$ taking the place of the other terms $\gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$ of the full return, as discussed in the previous chapter. Our point now is that this idea makes just as much sense after two steps as it does after one. The target for a two-step update is the *two-step return*:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2}),$$

where now $\gamma^2 V_{t+1}(S_{t+2})$ corrects for the absence of the terms $\gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots + \gamma^{T-t-1} R_T$. Similarly, the target for an arbitrary n-step update is the *n-step return*:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \tag{7.1}$$

for all n, t such that $n \geq 1$ and $0 \leq t < T - n$. All n-step returns can be considered approximations to the full return, truncated after n steps and then corrected for the remaining missing terms by $V_{t+n-1}(S_{t+n})$. If $t + n \geq T$ (if the n-step return extends to or beyond termination), then all the missing terms are taken as zero, and the n-step return defined to be equal to the ordinary full return ($G_{t:t+n} \doteq G_t$ if $t + n \geq T$).

Note that n-step returns for $n > 1$ involve future rewards and states that are not available at the time of transition from t to $t + 1$. No real algorithm can use the n-step return until after it has seen R_{t+n} and computed V_{t+n-1}. The first time these are available is $t + n$. The natural state-value learning algorithm for using n-step returns is thus

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\big[G_{t:t+n} - V_{t+n-1}(S_t)\big], \qquad 0 \leq t < T, \tag{7.2}$$

while the values of all other states remain unchanged: $V_{t+n}(s) = V_{t+n-1}(s)$, for all $s \neq S_t$. We call this algorithm *n-step TD*. Note that no changes at all are made during the first $n - 1$ steps of each episode. To make up for that, an equal number of additional updates are made at the end of the episode, after termination and before starting the next episode. Complete pseudocode is given in the box on the next page.

Exercise 7.1 In Chapter 6 we noted that the Monte Carlo error can be written as the sum of TD errors (6.6) if the value estimates don't change from step to step. Show that the n-step error used in (7.2) can also be written as a sum TD errors (again if the value estimates don't change) generalizing the earlier result. □

Exercise 7.2 (programming) With an n-step method, the value estimates *do* change from step to step, so an algorithm that used the sum of TD errors (see previous exercise) in

n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ ($G_{\tau:\tau+n}$)
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
  Until $\tau = T - 1$

place of the error in (7.2) would actually be a slightly different algorithm. Would it be a better algorithm or a worse one? Devise and program a small experiment to answer this question empirically. □

The n-step return uses the value function V_{t+n-1} to correct for the missing rewards beyond R_{t+n}. An important property of n-step returns is that their expectation is guaranteed to be a better estimate of v_π than V_{t+n-1} is, in a worst-state sense. That is, the worst error of the expected n-step return is guaranteed to be less than or equal to γ^n times the worst error under V_{t+n-1}:

$$\max_s \Big|\mathbb{E}_\pi[G_{t:t+n} | S_t = s] - v_\pi(s)\Big| \;\leq\; \gamma^n \max_s \Big|V_{t+n-1}(s) - v_\pi(s)\Big|, \tag{7.3}$$

for all $n \geq 1$. This is called the *error reduction property* of n-step returns. Because of the error reduction property, one can show formally that all n-step TD methods converge to the correct predictions under appropriate technical conditions. The n-step TD methods thus form a family of sound methods, with one-step TD methods and Monte Carlo methods as extreme members.

Example 7.1: *n*-step TD Methods on the Random Walk Consider using n-step TD methods on the 5-state random walk task described in Example 6.2 (page 125). Suppose the first episode progressed directly from the center state, C, to the right, through D and E, and then terminated on the right with a return of 1. Recall that the estimated values of all the states started at an intermediate value, $V(s) = 0.5$. As a result of this experience, a one-step method would change only the estimate for the last state,

$V(\mathsf{E})$, which would be incremented toward 1, the observed return. A two-step method, on the other hand, would increment the values of the two states preceding termination: $V(\mathsf{D})$ and $V(\mathsf{E})$ both would be incremented toward 1. A three-step method, or any n-step method for $n > 2$, would increment the values of all three of the visited states toward 1, all by the same amount.

Which value of n is better? Figure 7.2 shows the results of a simple empirical test for a larger random walk process, with 19 states instead of 5 (and with a -1 outcome on the left, all values initialized to 0), which we use as a running example in this chapter. Results are shown for n-step TD methods with a range of values for n and α. The performance measure for each parameter setting, shown on the vertical axis, is the square-root of the average squared error between the predictions at the end of the episode for the 19 states and their true values, then averaged over the first 10 episodes and 100 repetitions of the whole experiment (the same sets of walks were used for all parameter settings). Note that methods with an intermediate value of n worked best. This illustrates how the generalization of TD and Monte Carlo methods to n-step methods can potentially perform better than either of the two extreme methods.

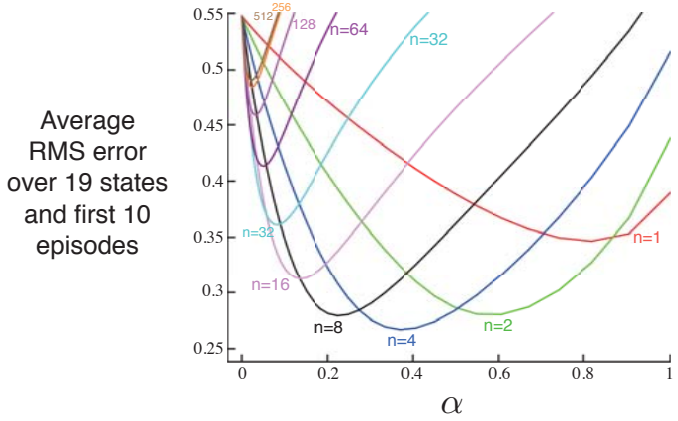


Figure 7.2: Performance of n-step TD methods as a function of α, for various values of n, on a 19-state random walk task (Example 7.1). ■

Exercise 7.3 Why do you think a larger random walk task (19 states instead of 5) was used in the examples of this chapter? Would a smaller walk have shifted the advantage to a different value of n? How about the change in left-side outcome from 0 to -1 made in the larger walk? Do you think that made any difference in the best value of n? □

7.2 n-step Sarsa

How can n-step methods be used not just for prediction, but for control? In this section we show how n-step methods can be combined with Sarsa in a straightforward way to

produce an on-policy TD control method. The n-step version of Sarsa we call n-step Sarsa, and the original version presented in the previous chapter we henceforth call *one-step Sarsa*, or *Sarsa(0)*.

The main idea is to simply switch states for actions (state–action pairs) and then use an ε-greedy policy. The backup diagrams for n-step Sarsa (shown in Figure 7.3), like those of n-step TD (Figure 7.1), are strings of alternating states and actions, except that the Sarsa ones all start and end with an action rather a state. We redefine n-step returns (update targets) in terms of estimated action values:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \quad n \geq 1, 0 \leq t < T-n, \tag{7.4}$$

with $G_{t:t+n} \doteq G_t$ if $t + n \geq T$. The natural algorithm is then

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right], \qquad 0 \leq t < T, \tag{7.5}$$

while the values of all other states remain unchanged: $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$. This is the algorithm we call *n-step Sarsa*. Pseudocode is shown in the box on the next page, and an example of why it can speed up learning compared to one-step methods is given in Figure 7.4.

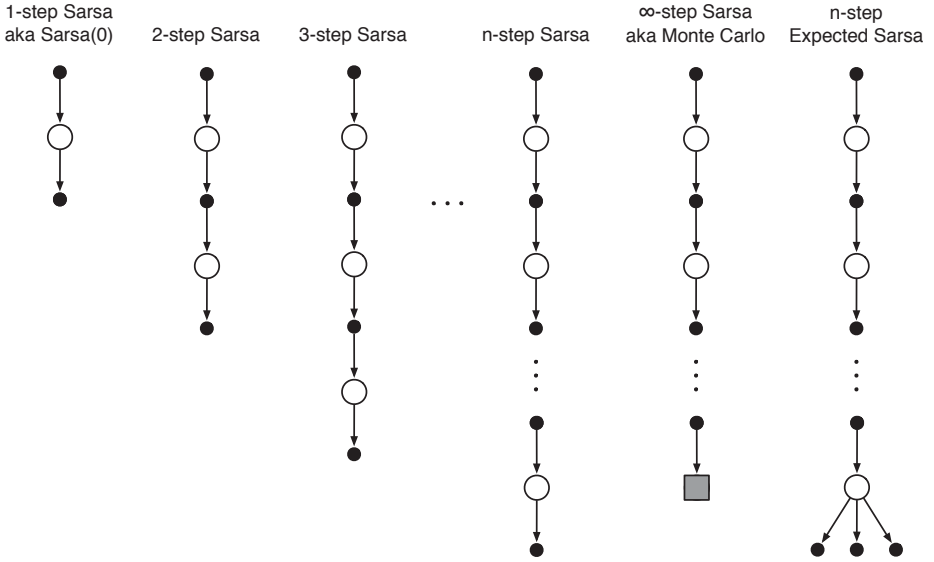


Figure 7.3: The backup diagrams for the spectrum of n-step methods for state–action values. They range from the one-step update of Sarsa(0) to the up-until-termination update of the Monte Carlo method. In between are the n-step updates, based on n steps of real rewards and the estimated value of the nth next state–action pair, all appropriately discounted. On the far right is the backup diagram for n-step Expected Sarsa.

n-step Sarsa for estimating $Q \approx q_*$ or q_π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t+1$
      else:
        Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\left[G - Q(S_\tau, A_\tau)\right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T - 1$

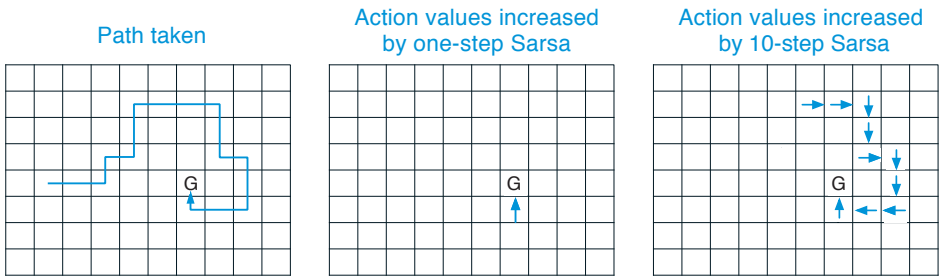


Figure 7.4: Gridworld example of the speedup of policy learning due to the use of n-step methods. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the G. In this example the values were all initially 0, and all rewards were zero except for a positive reward at G. The arrows in the other two panels show which action values were strengthened as a result of this path by one-step and n-step Sarsa methods. The one-step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the n-step method strengthens the last n actions of the sequence, so that much more is learned from the one episode.

Exercise 7.4 Prove that the n-step return of Sarsa (7.4) can be written exactly in terms of a novel TD error, as

$$G_{t:t+n} = Q_{t-1}(S_t, A_t) + \sum_{k=t}^{\min(t+n,T)-1} \gamma^{k-t} \left[R_{k+1} + \gamma Q_k(S_{k+1}, A_{k+1}) - Q_{k-1}(S_k, A_k)\right]. \tag{7.6}$$

□

What about Expected Sarsa? The backup diagram for the n-step version of Expected Sarsa is shown on the far right in Figure 7.3. It consists of a linear string of sample actions and states, just as in n-step Sarsa, except that its last element is a branch over all action possibilities weighted, as always, by their probability under π. This algorithm can be described by the same equation as n-step Sarsa (above) except with the n-step return redefined as

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}), \qquad t+n < T, \tag{7.7}$$

(with $G_{t:t+n} \doteq G_t$ for $t+n \geq T$) where $\bar{V}_t(s)$ is the *expected approximate value* of state s, using the estimated action values at time t, under the target policy:

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) Q_t(s,a), \qquad \text{for all } s \in \mathcal{S}. \tag{7.8}$$

Expected approximate values are used in developing many of the action-value methods in the rest of this book. If s is terminal, then its expected approximate value is defined to be 0.

7.3 n-step Off-policy Learning

Recall that off-policy learning is learning the value function for one policy, π, while following another policy, b. Often, π is the greedy policy for the current action-value-function estimate, and b is a more exploratory policy, perhaps ε-greedy. In order to use the data from b we must take into account the difference between the two policies, using their relative probability of taking the actions that were taken (see Section 5.5). In n-step methods, returns are constructed over n steps, so we are interested in the relative probability of just those n actions. For example, to make a simple off-policy version of n-step TD, the update for time t (actually made at time $t+n$) can simply be weighted by $\rho_{t:t+n-1}$:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1} \left[G_{t:t+n} - V_{t+n-1}(S_t)\right], \qquad 0 \leq t < T, \tag{7.9}$$

where $\rho_{t:t+n-1}$, called the *importance sampling ratio*, is the relative probability under the two policies of taking the n actions from A_t to A_{t+n-1} (cf. Eq. 5.3):

$$\rho_{t:h} \doteq \prod_{k=t}^{\min(h,T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{7.10}$$

For example, if any one of the actions would never be taken by π (i.e., $\pi(A_k|S_k) = 0$) then the n-step return should be given zero weight and be totally ignored. On the other hand, if by chance an action is taken that π would take with much greater probability than b does, then this will increase the weight that would otherwise be given to the return. This makes sense because that action is characteristic of π (and therefore we want to learn about it) but is selected only rarely by b and thus rarely appears in the data. To make up for this we have to over-weight it when it does occur. Note that if the two policies are actually the same (the on-policy case) then the importance sampling ratio is always 1. Thus our new update (7.9) generalizes and can completely replace our earlier n-step TD update. Similarly, our previous n-step Sarsa update can be completely replaced by a simple off-policy form:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n} \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right], \qquad (7.11)$$

for $0 \leq t < T$. Note that the importance sampling ratio here starts and ends one step later than for n-step TD (7.9). This is because here we are updating a state–action pair. We do not have to care how likely we were to select the action; now that we have selected it we want to learn fully from what happens, with importance sampling only for subsequent actions. Pseudocode for the full algorithm is shown in the box below.

Off-policy n-step Sarsa for estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t + 1$
      else:
        Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n-1, T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$ $(\rho_{\tau+1:t+n-1})$
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho \left[G - Q(S_\tau, A_\tau) \right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

The off-policy version of n-step Expected Sarsa would use the same update as above for n-step Sarsa except that the importance sampling ratio would have one less factor in it. That is, the above equation would use $\rho_{t+1:t+n-1}$ instead of $\rho_{t+1:t+n}$, and of course it would use the Expected Sarsa version of the n-step return (7.7). This is because in Expected Sarsa all possible actions are taken into account in the last state; the one actually taken has no effect and does not have to be corrected for.

7.4 *Per-decision Methods with Control Variates

The multi-step off-policy methods presented in the previous section are simple and conceptually clear, but are probably not the most efficient. A more sophisticated approach would use per-decision importance sampling ideas such as were introduced in Section 5.9. To understand this approach, first note that the ordinary n-step return (7.1), like all returns, can be written recursively. For the n steps ending at horizon h, the n-step return can be written

$$G_{t:h} = R_{t+1} + \gamma G_{t+1:h}, \qquad t < h < T, \qquad (7.12)$$

where $G_{h:h} \doteq V_{h-1}(S_h)$. (Recall that this return is used at time h, previously denoted $t + n$.) Now consider the effect of following a behavior policy b that is not the same as the target policy π. All of the resulting experience, including the first reward R_{t+1} and the next state S_{t+1} must be weighted by the importance sampling ratio for time t, $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$. One might be tempted to simply weight the righthand side of the above equation, but one can do better. Suppose the action at time t would never be selected by π, so that ρ_t is zero. Then a simple weighting would result in the n-step return being zero, which could result in high variance when it was used as a target. Instead, in this more sophisticated approach, one uses an alternate, *off-policy* definition of the n-step return ending at horizon h, as

$$G_{t:h} \doteq \rho_t \left(R_{t+1} + \gamma G_{t+1:h} \right) + (1 - \rho_t) V_{h-1}(S_t), \qquad t < h < T, \qquad (7.13)$$

where again $G_{h:h} \doteq V_{h-1}(S_h)$. In this approach, if ρ_t is zero, then instead of the target being zero and causing the estimate to shrink, the target is the same as the estimate and causes no change. The importance sampling ratio being zero means we should ignore the sample, so leaving the estimate unchanged seems appropriate. The second, additional term in (7.13) is called a *control variate* (for obscure reasons). Notice that the control variate does not change the expected update; the importance sampling ratio has expected value one (Section 5.9) and is uncorrelated with the estimate, so the expected value of the control variate is zero. Also note that the off-policy definition (7.13) is a strict generalization of the earlier on-policy definition of the n-step return (7.1), as the two are identical in the on-policy case, in which ρ_t is always 1.

For a conventional n-step method, the learning rule to use in conjunction with (7.13) is the n-step TD update (7.2), which has no explicit importance sampling ratios other than those embedded in the return.

Exercise 7.5 Write the pseudocode for the off-policy state-value prediction algorithm described above. □

For action values, the off-policy definition of the n-step return is a little different because the first action does not play a role in the importance sampling. That first action is the one being learned; it does not matter if it was unlikely or even impossible under the target policy—it has been taken and now full unit weight must be given to the reward and state that follows it. Importance sampling will apply only to the actions that follow it.

First note that for action values the n-step *on-policy* return ending at horizon h, expectation form (7.7), can be written recursively just as in (7.12), except that for action values the recursion ends with $G_{h:h} \doteq \bar{V}_{h-1}(S_h)$ as in (7.8). An off-policy form with control variates is

$$\begin{aligned} G_{t:h} &\doteq R_{t+1} + \gamma\Big(\rho_{t+1}G_{t+1:h} + \bar{V}_{h-1}(S_{t+1}) - \rho_{t+1}Q_{h-1}(S_{t+1}, A_{t+1})\Big), \\ &= R_{t+1} + \gamma\rho_{t+1}\Big(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})\Big) + \gamma\bar{V}_{h-1}(S_{t+1}), \quad t < h \leq T. \end{aligned} \tag{7.14}$$

If $h < T$, then the recursion ends with $G_{h:h} \doteq Q_{h-1}(S_h, A_h)$, whereas, if $h \geq T$, the recursion ends with and $G_{T-1:h} \doteq R_T$. The resultant prediction algorithm (after combining with (7.5)) is analogous to Expected Sarsa.

Exercise 7.6 Prove that the control variate in the above equations does not change the expected value of the return. □

**Exercise 7.7* Write the pseudocode for the off-policy action-value prediction algorithm described immediately above. Pay particular attention to the termination conditions for the recursion upon hitting the horizon or the end of episode. □

Exercise 7.8 Show that the general (off-policy) version of the n-step return (7.13) can still be written exactly and compactly as the sum of state-based TD errors (6.5) if the approximate state value function does not change. □

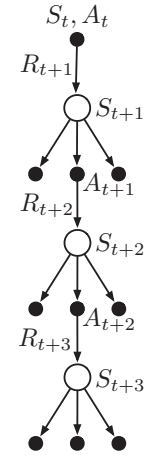
Exercise 7.9 Repeat the above exercise for the action version of the off-policy n-step return (7.14) and the Expected Sarsa TD error (the quantity in brackets in Equation 6.9). □

Exercise 7.10 (programming) Devise a small off-policy prediction problem and use it to show that the off-policy learning algorithm using (7.13) and (7.2) is more data efficient than the simpler algorithm using (7.1) and (7.9). □

The importance sampling that we have used in this section, the previous section, and in Chapter 5, enables sound off-policy learning, but also results in high variance updates, forcing the use of a small step-size parameter and thereby causing learning to be slow. It is probably inevitable that off-policy training is slower than on-policy training—after all, the data is less relevant to what is being learned. However, it is probably also true that these methods can be improved on. The control variates are one way of reducing the variance. Another is to rapidly adapt the step sizes to the observed variance, as in the Autostep method (Mahmood, Sutton, Degris and Pilarski, 2012). Yet another promising approach is the invariant updates of Karampatziakis and Langford (2010) as extended to TD by Tian (in preparation). The usage technique of Mahmood (2017; Mahmood and Sutton, 2015) may also be part of the solution. In the next section we consider an off-policy learning method that does not use importance sampling.

7.5 Off-policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm

Is off-policy learning possible without importance sampling? Q-learning and Expected Sarsa from Chapter 6 do this for the one-step case, but is there a corresponding multi-step algorithm? In this section we present just such an n-step method, called the *tree-backup algorithm*.



the 3-step tree-backup update

The idea of the algorithm is suggested by the 3-step tree-backup backup diagram shown to the right. Down the central spine and labeled in the diagram are three sample states and rewards, and two sample actions. These are the random variables representing the events occurring after the initial state–action pair S_t, A_t. Hanging off to the sides of each state are the actions that were *not* selected. (For the last state, all the actions are considered to have not (yet) been selected.) Because we have no sample data for the unselected actions, we bootstrap and use the estimates of their values in forming the target for the update. This slightly extends the idea of a backup diagram. So far we have always updated the estimated value of the node at the top of the diagram toward a target combining the rewards along the way (appropriately discounted) and the estimated values of the nodes at the bottom. In the tree-backup update, the target includes all these things *plus* the estimated values of the dangling action nodes hanging off the sides, at all levels. This is why it is called a *tree-backup* update; it is an update from the entire tree of estimated action values.

More precisely, the update is from the estimated action values of the *leaf nodes* of the tree. The action nodes in the interior, corresponding to the actual actions taken, do not participate. Each leaf node contributes to the target with a weight proportional to its probability of occurring under the target policy π. Thus each first-level action a contributes with a weight of $\pi(a|S_{t+1})$, except that the action actually taken, A_{t+1}, does not contribute at all. Its probability, $\pi(A_{t+1}|S_{t+1})$, is used to weight all the second-level action values. Thus, each non-selected second-level action a' contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(a'|S_{t+2})$. Each third-level action contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(A_{t+2}|S_{t+2})\pi(a''|S_{t+3})$, and so on. It is as if each arrow to an action node in the diagram is weighted by the action's probability of being selected under the target policy and, if there is a tree below the action, then that weight applies to all the leaf nodes in the tree.

We can think of the 3-step tree-backup update as consisting of 6 half-steps, alternating between sample half-steps from an action to a subsequent state, and expected half-steps considering from that state all possible actions with their probabilities of occuring under the policy.

Now let us develop the detailed equations for the n-step tree-backup algorithm. The one-step return (target) is the same as that of Expected Sarsa,

$$G_{t:t+1} \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a), \tag{7.15}$$

for $t < T-1$, and the two-step tree-backup return is

$$\begin{aligned} G_{t:t+2} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) \\ &\qquad + \gamma \pi(A_{t+1}|S_{t+1}) \Big(R_{t+2} + \gamma \sum_a \pi(a|S_{t+2}) Q_{t+1}(S_{t+2}, a) \Big) \\ &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+2}, \end{aligned}$$

for $t < T-2$. The latter form suggests the general recursive definition of the tree-backup n-step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) \; + \; \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}, \tag{7.16}$$

for $t < T-1, n \geq 2$, with the $n = 1$ case handled by (7.15) except for $G_{T-1:t+n} \doteq R_T$. This target is then used with the usual action-value update rule from n-step Sarsa:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right],$$

for $0 \leq t < T$, while the values of all other state–action pairs remain unchanged: $Q_{t+n}(s,a) = Q_{t+n-1}(s,a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$. Pseudocode for this algorithm is shown in the box on the next page.

Exercise 7.11 Show that if the approximate action values are unchanging, then the tree-backup return (7.16) can be written as a sum of expectation-based TD errors:

$$G_{t:t+n} = Q(S_t, A_t) + \sum_{k=t}^{\min(t+n-1, T-1)} \delta_k \prod_{i=t+1}^{k} \gamma \pi(A_i|S_i),$$

where $\delta_t \doteq R_{t+1} + \gamma \bar{V}_t(S_{t+1}) - Q(S_t, A_t)$ and $\bar{V}_t$ is given by (7.8). □

n-step Tree Backup for estimating $Q \approx q_*$ or q_π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose an action A_0 arbitrarily as a function of S_0; Store A_0
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$:
      Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
      If S_{t+1} is terminal:
        $T \leftarrow t+1$
      else:
        Choose an action A_{t+1} arbitrarily as a function of S_{t+1}; Store A_{t+1}
    $\tau \leftarrow t+1-n$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      If $t+1 \geq T$:
        $G \leftarrow R_T$
      else
        $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
      Loop for $k = \min(t, T-1)$ down through $\tau + 1$:
        $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k|S_k) G$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \left[G - Q(S_\tau, A_\tau) \right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T-1$

7.6 *A Unifying Algorithm: n-step $Q(\sigma)$

So far in this chapter we have considered three different kinds of action-value algorithms, corresponding to the first three backup diagrams shown in Figure 7.5. n-step Sarsa has all sample transitions, the tree-backup algorithm has all state-to-action transitions fully branched without sampling, and n-step Expected Sarsa has all sample transitions except for the last state-to-action one, which is fully branched with an expected value. To what extent can these algorithms be unified?

One idea for unification is suggested by the fourth backup diagram in Figure 7.5. This is the idea that one might decide on a step-by-step basis whether one wanted to take the action as a sample, as in Sarsa, or consider the expectation over all actions instead, as in the tree-backup update. Then, if one chose always to sample, one would obtain Sarsa, whereas if one chose never to sample, one would get the tree-backup algorithm. Expected Sarsa would be the case where one chose to sample for all steps except for the last one.

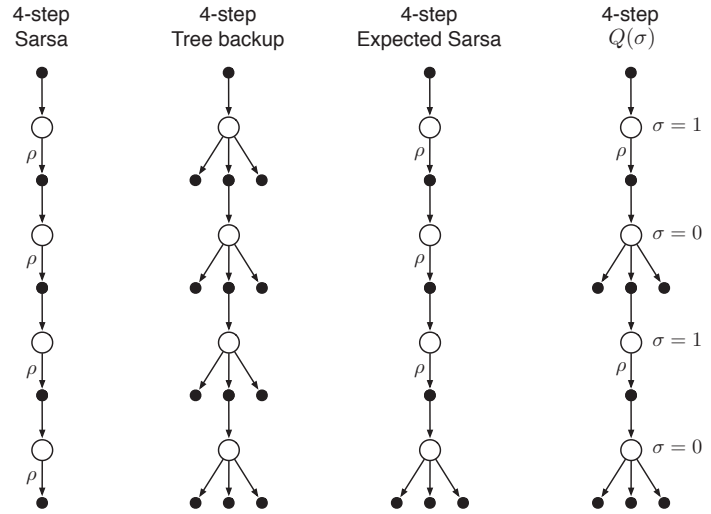


Figure 7.5: The backup diagrams of the three kinds of n-step action-value updates considered so far in this chapter (4-step case) plus the backup diagram of a fourth kind of update that unifies them all. The 'ρ's indicate half transitions on which importance sampling is required in the off-policy case. The fourth kind of update unifies all the others by choosing on a state-by-state basis whether to sample ($\sigma_t = 1$) or not ($\sigma_t = 0$).

And of course there would be many other possibilities, as suggested by the last diagram in the figure. To increase the possibilities even further we can consider a continuous variation between sampling and expectation. Let $\sigma_t \in [0, 1]$ denote the degree of sampling on step t, with $\sigma = 1$ denoting full sampling and $\sigma = 0$ denoting a pure expectation with no sampling. The random variable σ_t might be set as a function of the state, action, or state–action pair at time t. We call this proposed new algorithm n-step $Q(\sigma)$.

Now let us develop the equations of n-step $Q(\sigma)$. First we write the tree-backup n-step return (7.16) in terms of the horizon $h = t + n$ and then in terms of the expected approximate value $\bar{V}$ (7.8):

$$\begin{aligned} G_{t:h} &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{h-1}(S_{t+1}, a) \;+\; \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:h} \\ &= R_{t+1} + \gamma \bar{V}_{h-1}(S_{t+1}) - \gamma \pi(A_{t+1}|S_{t+1}) Q_{h-1}(S_{t+1}, A_{t+1}) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:h} \\ &= R_{t+1} + \gamma \pi(A_{t+1}|S_{t+1}) \Big(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1}) \Big) + \gamma \bar{V}_{h-1}(S_{t+1}), \end{aligned}$$

after which it is exactly like the n-step return for Sarsa with control variates (7.14) except with the action probability $\pi(A_{t+1}|S_{t+1})$ substituted for the importance-sampling ratio ρ_{t+1}. For $Q(\sigma)$, we slide linearly between these two cases:

$$\begin{aligned} G_{t:h} \doteq R_{t+1} &+ \gamma \Big(\sigma_{t+1} \rho_{t+1} + (1 - \sigma_{t+1}) \pi(A_{t+1}|S_{t+1}) \Big) \Big(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1}) \Big) \\ &+ \gamma \bar{V}_{h-1}(S_{t+1}), \end{aligned} \tag{7.17}$$

for $t < h \leq T$. The recursion ends with $G_{h:h} \doteq Q_{h-1}(S_h, A_h)$ if $h < T$, or with $G_{T-1:T} \doteq R_T$ if $h = T$. Then we use the general (off-policy) update for n-step Sarsa (7.11). A complete algorithm is given in the box.

Off-policy n-step $Q(\sigma)$ for estimating $Q \approx q_*$ or q_π

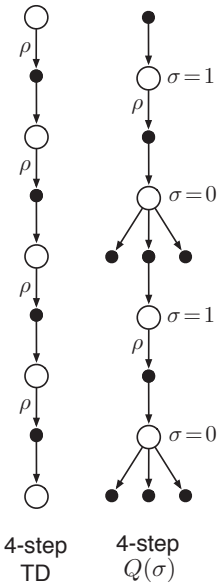
Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$:
      Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
      If S_{t+1} is terminal:
        $T \leftarrow t + 1$
      else:
        Choose and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
        Select and store σ_{t+1}
        Store $\frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$ as ρ_{t+1}
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow 0$:
      Loop for $k = \min(t + 1, T)$ down through $\tau + 1$:
        if $k = T$:
          $G \leftarrow R_T$
        else:
          $\bar{V} \leftarrow \sum_a \pi(a|S_k) Q(S_k, a)$
          $G \leftarrow R_k + \gamma\big(\sigma_k \rho_k + (1 - \sigma_k)\pi(A_k|S_k)\big)\big(G - Q(S_k, A_k)\big) + \gamma \bar{V}$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \left[G - Q(S_\tau, A_\tau) \right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

7.7 Summary

In this chapter we have developed a range of temporal-difference learning methods that lie in between the one-step TD methods of the previous chapter and the Monte Carlo methods of the chapter before. Methods that involve an intermediate amount of bootstrapping are important because they will typically perform better than either extreme.

Our focus in this chapter has been on n-step methods, which look ahead to the next n rewards, states, and actions. The two 4-step backup diagrams to the right together summarize most of the methods introduced. The state-value update shown is for n-step TD with importance sampling, and the action-value update is for n-step $Q(\sigma)$, which generalizes Expected Sarsa and Q-learning. All n-step methods involve a delay of n time steps before updating, as only then are all the required future events known. A further drawback is that they involve more computation per time step than previous methods. Compared to one-step methods, n-step methods also require more memory to record the states, actions, rewards, and sometimes other variables over the last n time steps. Eventually, in Chapter 12, we will see how multi-step TD methods can be implemented with minimal memory and computational complexity using eligibility traces, but there will always be some additional computation beyond one-step methods. Such costs can be well worth paying to escape the tyranny of the single time step.



Although n-step methods are more complex than those using eligibility traces, they have the great benefit of being conceptually clear. We have sought to take advantage of this by developing two approaches to off-policy learning in the n-step case. One, based on importance sampling is conceptually simple but can be of high variance. If the target and behavior policies are very different it probably needs some new algorithmic ideas before it can be efficient and practical. The other, based on tree-backup updates, is the natural extension of Q-learning to the multi-step case with stochastic target policies. It involves no importance sampling but, again if the target and behavior policies are substantially different, the bootstrapping may span only a few steps even if n is large.

Bibliographical and Historical Remarks

The notion of n-step returns is due to Watkins (1989), who also first discussed their error reduction property. n-step algorithms were explored in the first edition of this book, in which they were treated as of conceptual interest, but not feasible in practice. The work of Cichosz (1995) and particularly van Seijen (2016) showed that they are actually completely practical algorithms. Given this, and their conceptual clarity and simplicity, we have chosen to highlight them here in the second edition. In particular, we now postpone all discussion of the backward view and of eligibility traces until Chapter 12.

7.1–2 The results in the random walk examples were made for this text based on work of Sutton (1988) and Singh and Sutton (1996). The use of backup diagrams to describe these and other algorithms in this chapter is new.

7.3–5 The developments in these sections are based on the work of Precup, Sutton, and Singh (2000), Precup, Sutton, and Dasgupta (2001), and Sutton, Mahmood, Precup, and van Hasselt (2014).

The tree-backup algorithm is due to Precup, Sutton, and Singh (2000), but the presentation of it here is new.

7.6 The $Q(\sigma)$ algorithm is new to this text, but closely related algorithms have been explored further by De Asis, Hernandez-Garcia, Holland, and Sutton (2017).

Chapter 8

Planning and Learning with Tabular Methods

In this chapter we develop a unified view of reinforcement learning methods that require a model of the environment, such as dynamic programming and heuristic search, and methods that can be used without a model, such as Monte Carlo and temporal-difference methods. These are respectively called *model-based* and *model-free* reinforcement learning methods. Model-based methods rely on *planning* as their primary component, while model-free methods primarily rely on *learning*. Although there are real differences between these two kinds of methods, there are also great similarities. In particular, the heart of both kinds of methods is the computation of value functions. Moreover, all the methods are based on looking ahead to future events, computing a backed-up value, and then using it as an update target for an approximate value function. Earlier in this book we presented Monte Carlo and temporal-difference methods as distinct alternatives, then showed how they can be unified by n-step methods. Our goal in this chapter is a similar integration of model-based and model-free methods. Having established these as distinct in earlier chapters, we now explore the extent to which they can be intermixed.

8.1 Models and Planning

By a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call *distribution models*. Other models produce just one of the possibilities, sampled according to the probabilities; these we call *sample models*. For example, consider modeling the sum of a dozen dice. A distribution model would produce all possible sums and their probabilities of occurring, whereas a sample model would produce an individual

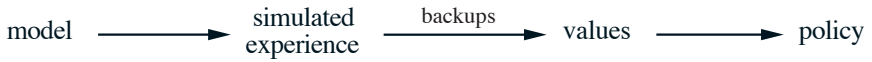
sum drawn according to this probability distribution. The kind of model assumed in dynamic programming—estimates of the MDP's dynamics, $p(s', r \mid s, a)$—is a distribution model. The kind of model used in the blackjack example in Chapter 5 is a sample model. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in many applications it is much easier to obtain sample models than distribution models. The dozen dice are a simple example of this. It would be easy to write a computer program to simulate the dice rolls and return the sum, but harder and more error-prone to figure out all the possible sums and their probabilities.

Models can be used to mimic or simulate experience. Given a starting state and action, a sample model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, we say the model is used to *simulate* the environment and produce *simulated experience*.

The word *planning* is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:

In artificial intelligence, there are two distinct approaches to planning according to our definition. *State-space planning*, which includes the approach we take in this book, is viewed primarily as a search through the state space for an optimal policy or an optimal path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In what we call *plan-space planning*, planning is instead a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. Plan-space planning includes evolutionary methods and "partial-order planning," a common kind of planning in artificial intelligence in which the ordering of steps is not completely determined at all stages of planning. Plan-space methods are difficult to apply efficiently to the stochastic sequential decision problems that are the focus in reinforcement learning, and we do not consider them further (but see, e.g., Russell and Norvig, 2010).

The unified view we present in this chapter is that all state-space planning methods share a common structure, a structure that is also present in the learning methods presented in this book. It takes the rest of the chapter to develop this view, but there are two basic ideas: (1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute value functions by updates or backup operations applied to simulated experience. This common structure can be diagrammed as follows:



Dynamic programming methods clearly fit this structure: they make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value (update target) and update the

state's estimated value. In this chapter we argue that various other state-space planning methods also fit this structure, with individual methods differing only in the kinds of updates they do, the order in which they do them, and in how long the backed-up information is retained.

Viewing planning methods in this way emphasizes their relationship to the learning methods that we have described in this book. The heart of both learning and planning methods is the estimation of value functions by backing-up update operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment. Of course this difference leads to a number of other differences, for example, in how performance is assessed and in how flexibly experience can be generated. But the common structure means that many ideas and algorithms can be transferred between planning and learning. In particular, in many cases a learning algorithm can be substituted for the key update step of a planning method. Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. The box below shows a simple example of a planning method based on one-step tabular Q-learning and on random samples from a sample model. This method, which we call *random-sample one-step tabular Q-planning*, converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment (each state–action pair must be selected an infinite number of times in Step 1, and α must decrease appropriately over time).

Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain
   a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

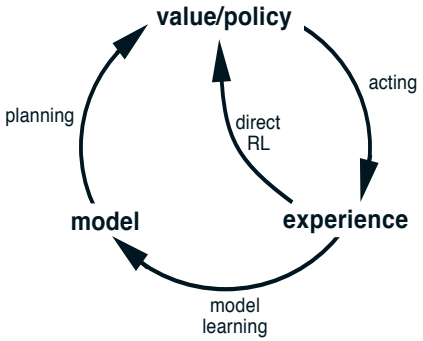
In addition to the unified view of planning and learning methods, a second theme in this chapter is the benefits of planning in small, incremental steps. This enables planning to be interrupted or redirected at any time with little wasted computation, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. Planning in very small steps may be the most efficient approach even on pure planning problems if the problem is too large to be solved exactly.

8.2 Dyna: Integrated Planning, Acting, and Learning

When planning is done online, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected

in the near future. If decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an online planning agent. Each function appears in Dyna-Q in a simple, almost trivial, form. In subsequent sections we elaborate some of the alternate ways of achieving each function and the trade-offs between them. For now, we seek merely to illustrate the ideas and stimulate your intuition.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of reinforcement learning methods we have discussed in previous chapters. The former we call *model-learning*, and the latter we call *direct reinforcement learning* (direct RL). The possible relationships between experience, model, values, and policy are summarized in the diagram to the right. Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value functions and policies either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.



Both direct and indirect methods have advantages and disadvantages. Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affected by biases in the design of the model. Some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning. Related debates in psychology and artificial intelligence concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision making (see Chapter 14 for discussion of some of these issues from the perspective of psychology). Our view is that the contrast between the alternatives in all these debates has been exaggerated, that more insight can be gained by recognizing the similarities between these two sides than by opposing them. For example, in this book we have emphasized the deep similarities between dynamic programming and temporal-difference methods, even though one was designed for planning and the other for model-free learning.

Dyna-Q includes all of the processes shown in the diagram above—planning, acting, model-learning, and direct RL—all occurring continually. The planning method is the random-sample one-step tabular Q-planning method on page 161. The direct RL method is one-step tabular Q-learning. The model-learning method is also table-based and assumes the environment is deterministic. After each transition $S_t, A_t \to R_{t+1}, S_{t+1}$, the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow. Thus, if the model is queried with a state–action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction.

During planning, the Q-planning algorithm randomly samples only from state–action pairs that have previously been experienced (in Step 1), so the model is never queried with a pair about which it has no information.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 8.1. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term *search control* to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the "final common path" for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

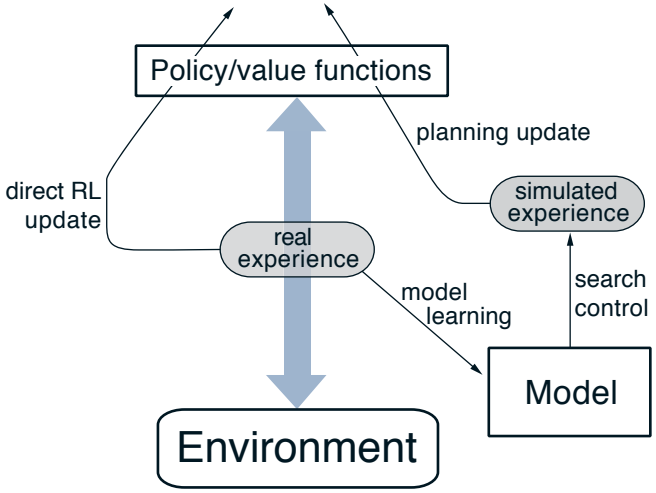


Figure 8.1: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

Conceptually, planning, acting, model-learning, and direct RL occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, however, we fully specify the order in which they occur within a time step. In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devoted to the planning process, which is inherently computation-intensive. Let us assume that there is time in each step, after acting, model-learning, and direct RL, to complete

n iterations (Steps 1–3) of the Q-planning algorithm. In the pseudocode algorithm for Dyna-Q in the box below, $Model(s,a)$ denotes the contents of the (predicted next state and reward) for state–action pair (s,a). Direct reinforcement learning, model-learning, and planning are implemented by steps (d), (e), and (f), respectively. If (e) and (f) were omitted, the remaining algorithm would be one-step tabular Q-learning.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Example 8.1: Dyna Maze Consider the simple maze shown inset in Figure 8.2. In each of the 47 states there are four actions, `up`, `down`, `right`, and `left`, which take the agent deterministically to the corresponding neighboring states, except when movement is blocked by an obstacle or the edge of the maze, in which case the agent remains where it is. Reward is zero on all transitions, except those into the goal state, on which it is +1. After reaching the goal state (G), the agent returns to the start state (S) to begin a new episode. This is a discounted, episodic task with $\gamma = 0.95$.

The main part of Figure 8.2 shows average learning curves from an experiment in which Dyna-Q agents were applied to the maze task. The initial action values were zero, the step-size parameter was $\alpha = 0.1$, and the exploration parameter was $\varepsilon = 0.1$. When selecting greedily among actions, ties were broken randomly. The agents varied in the number of planning steps, n, they performed per real step. For each n, the curves show the number of steps taken by the agent to reach the goal in each episode, averaged over 30 repetitions of the experiment. In each repetition, the initial seed for the random number generator was held constant across algorithms. Because of this, the first episode was exactly the same (about 1700 steps) for all values of n, and its data are not shown in the figure. After the first episode, performance improved for all values of n, but much more rapidly for larger values. Recall that the $n = 0$ agent is a nonplanning agent, using only direct reinforcement learning (one-step tabular Q-learning). This was by far the slowest agent on this problem, despite the fact that the parameter values (α and ε) were optimized for it. The nonplanning agent took about 25 episodes to reach (ε-)optimal performance, whereas the $n = 5$ agent took about five episodes, and the $n = 50$ agent took only three episodes.

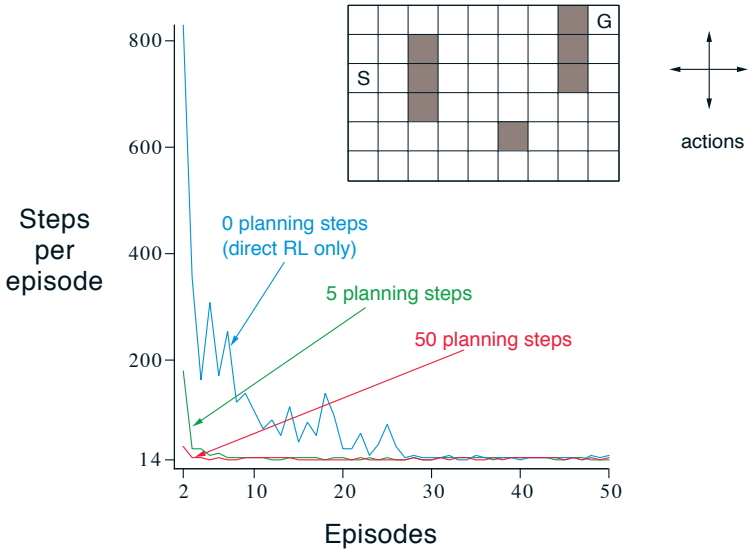


Figure 8.2: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from S to G as quickly as possible.

Figure 8.3 shows why the planning agents found the solution so much faster than the nonplanning agent. Shown are the policies found by the $n = 0$ and $n = 50$ agents halfway through the second episode. Without planning ($n = 0$), each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far. With planning, again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the end of the episode will reach almost back to the start state. This policy is built by the planning process while the agent is still wandering near the start state. By the end of the third episode a complete optimal policy will have been found and perfect performance attained.

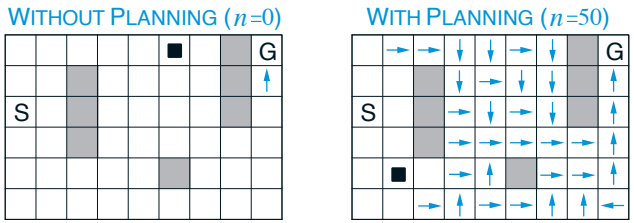


Figure 8.3: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent. ■

In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning. Because planning proceeds incrementally, it is trivial to intermix planning and acting. Both proceed as fast as they can. The agent is always reactive and always deliberative, responding instantly to the latest sensory information and yet always planning in the background. Also ongoing in the background is the model-learning process. As new information is gained, the model is updated to better match reality. As the model changes, the ongoing planning process will gradually compute a different way of behaving to match the new model.

Exercise 8.1 The nonplanning method looks particularly poor in Figure 8.3 because it is a one-step method; a method using multi-step bootstrapping would do better. Do you think one of the multi-step bootstrapping methods from Chapter 7 could do as well as the Dyna method? Explain why or why not. □

8.3 When the Model Is Wrong

In the maze example presented in the previous section, the changes in the model were relatively modest. The model started out empty, and was then filled only with exactly correct information. In general, we cannot expect to be so fortunate. Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, or because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. When the model is incorrect, the planning process is likely to compute a suboptimal policy.

In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they do not exist.

Example 8.2: Blocking Maze A maze example illustrating this relatively minor kind of modeling error and recovery from it is shown in Figure 8.4. Initially, there is a short path from start to goal, to the right of the barrier, as shown in the upper left of the figure. After 1000 time steps, the short path is "blocked," and a longer path is opened up along the left-hand side of the barrier, as shown in upper right of the figure. The graph shows average cumulative reward for a Dyna-Q agent and an enhanced Dyna-Q+ agent to be described shortly. The first part of the graph shows that both Dyna agents found the short path within 1000 steps. When the environment changed, the graphs become flat, indicating a period during which the agents obtained no reward because they were wandering around behind the barrier. After a while, however, they were able to find the new opening and the new optimal behavior.

Greater difficulties arise when the environment changes to become *better* than it was before, and yet the formerly correct policy does not reveal the improvement. In these cases the modeling error may not be detected for a long time, if ever.

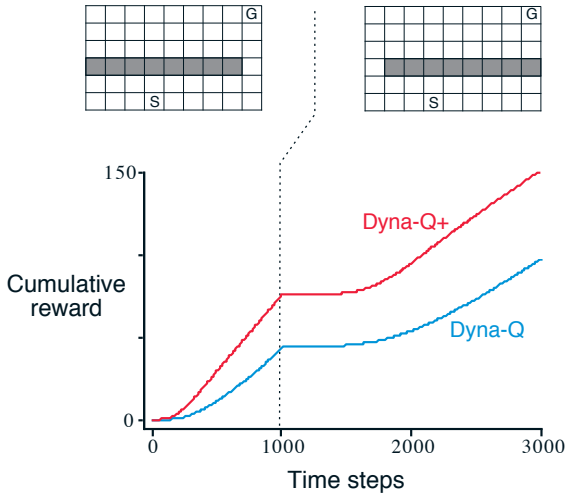


Figure 8.4: Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration. ■

Example 8.3: Shortcut Maze
The problem caused by this kind of environmental change is illustrated by the maze example shown in Figure 8.5. Initially, the optimal path is to go around the left side of the barrier (upper left). After 3000 steps, however, a shorter path is opened up along the right side, without disturbing the longer path (upper right). The graph shows that the regular Dyna-Q agent never switched to the shortcut. In fact, it never realized that it existed. Its model said that there was no shortcut, so the more it planned, the less likely it was to step to the right and discover it. Even with an ε-greedy policy, it is very unlikely that an agent will take so many exploratory actions as to discover the shortcut.

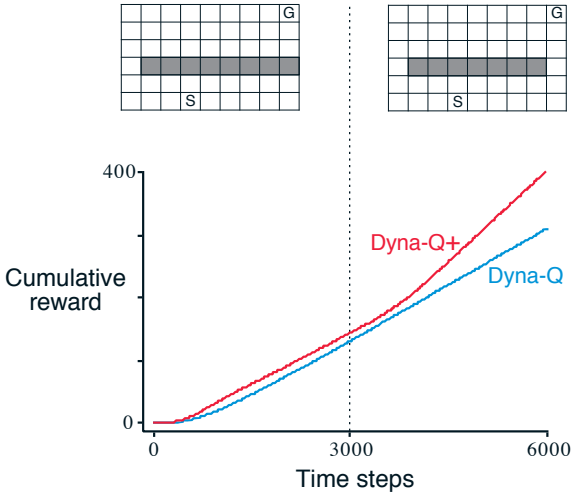


Figure 8.5: Average performance of Dyna agents on a shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest. ■

The general problem here is another version of the conflict between exploration and exploitation. In a planning context, exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model.

We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded. As in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent that did solve the shortcut maze uses one such heuristic. This agent keeps track for each state–action pair of how many time steps have elapsed since the pair was last tried in a real interaction with the environment. The more time that has elapsed, the greater (we might presume) the chance that the dynamics of this pair has changed and that the model of it is incorrect. To encourage behavior that tests long-untried actions, a special "bonus reward" is given on simulated experiences involving these actions. In particular, if the modeled reward for a transition is r, and the transition has not been tried in τ time steps, then planning updates are done as if that transition produced a reward of $r + \kappa\sqrt{\tau}$, for some small κ. This encourages the agent to keep testing all accessible state transitions and even to find long sequences of actions in order to carry out such tests.[1] Of course all this testing has its cost, but in many cases, as in the shortcut maze, this kind of computational curiosity is well worth the extra exploration.

Exercise 8.2 Why did the Dyna agent with exploration bonus, Dyna-Q+, perform better in the first phase as well as in the second phase of the blocking and shortcut experiments? □

Exercise 8.3 Careful inspection of Figure 8.5 reveals that the difference between Dyna-Q+ and Dyna-Q narrowed slightly over the first part of the experiment. What is the reason for this? □

Exercise 8.4 (programming) The exploration bonus described above actually changes the estimated values of states and actions. Is this necessary? Suppose the bonus $\kappa\sqrt{\tau}$ was used not in updates, but solely in action selection. That is, suppose the action selected was always that for which $Q(S_t, a) + \kappa\sqrt{\tau(S_t, a)}$ was maximal. Carry out a gridworld experiment that tests and illustrates the strengths and weaknesses of this alternate approach. □

Exercise 8.5 How might the tabular Dyna-Q algorithm shown on page 164 be modified to handle stochastic environments? How might this modification perform poorly on changing environments such as considered in this section? How could the algorithm be modified to handle stochastic environments *and* changing environments? □

8.4 Prioritized Sweeping

In the Dyna agents presented in the preceding sections, simulated transitions are started in state–action pairs selected uniformly at random from all previously experienced pairs. But a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and updates are focused on particular state–action pairs. For example, consider

[1] The Dyna-Q+ agent was changed in two other ways as well. First, actions that had never been tried before from a state were allowed to be considered in the planning step (f) of the Tabular Dyna-Q algorithm in the box above. Second, the initial model for such actions was that they would lead back to the same state with a reward of zero.

what happens during the second episode of the first maze task (Figure 8.3). At the beginning of the second episode, only the state–action pair leading directly into the goal has a positive value; the values of all other pairs are still zero. This means that it is pointless to perform updates along almost all transitions, because they take the agent from one zero-valued state to another, and thus the updates would have no effect. Only an update along a transition into the state just prior to the goal, or from it, will change any values. If simulated transitions are generated uniformly, then many wasteful updates will be made before stumbling onto one of these useful ones. As planning progresses, the region of useful updates grows, but planning is still far less efficient than it would be if focused where it would do the most good. In the much larger problems that are our real objective, the number of states is so large that an unfocused search would be extremely inefficient.

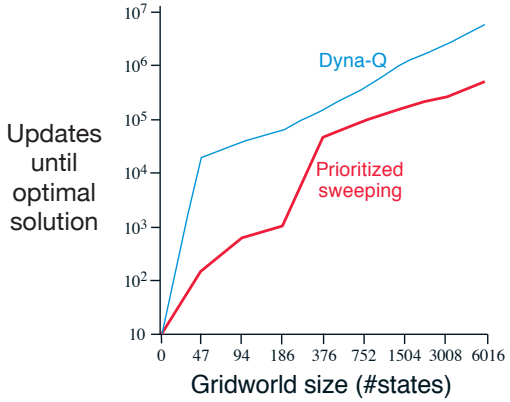
This example suggests that search might be usefully focused by working *backward* from goal states. Of course, we do not really want to use any methods specific to the idea of "goal state." We want methods that work for general reward functions. Goal states are just a special case, convenient for stimulating intuition. In general, we want to work back not just from goal states but from any state whose value has changed. Suppose that the values are initially correct given the model, as they were in the maze example prior to discovering the goal. Suppose now that the agent discovers a change in the environment and changes its estimated value of one state, either up or down. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step updates are those of actions that lead directly into the one state whose value has been changed. If the values of these actions are updated, then the values of the predecessor states may change in turn. If so, then actions leading into them need to be updated, and then *their* predecessor states may have changed. In this way one can work backward from arbitrary states that have changed in value, either performing useful updates or terminating the propagation. This general idea might be termed *backward focusing* of planning computations.

As the frontier of useful updates propagates backward, it often grows rapidly, producing many state–action pairs that could usefully be updated. But not all of these will be equally useful. The values of some states may have changed a lot, whereas others may have changed little. The predecessor pairs of those that have changed a lot are more likely to also change a lot. In a stochastic environment, variations in estimated transition probabilities also contribute to variations in the sizes of changes and in the urgency with which pairs need to be updated. It is natural to prioritize the updates according to a measure of their urgency, and perform them in order of priority. This is the idea behind *prioritized sweeping*. A queue is maintained of every state–action pair whose estimated value would change nontrivially if updated , prioritized by the size of the change. When the top pair in the queue is updated, the effect on each of its predecessor pairs is computed. If the effect is greater than some small threshold, then the pair is inserted in the queue with the new priority (if there is a previous entry of the pair in the queue, then insertion results in only the higher priority entry remaining in the queue). In this way the effects of changes are efficiently propagated backward until quiescence. The full algorithm for the case of deterministic environments is given in the box on the next page.

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow policy(S,Q)$
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Model(S,A) \leftarrow R, S'$
- (e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$.
- (f) if $P > \theta$, then insert S,A into $PQueue$ with priority P
- (g) Loop repeat n times, while $PQueue$ is not empty:
  - $S,A \leftarrow first(PQueue)$
  - $R,S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
  - Loop for all $\bar{S},\bar{A}$ predicted to lead to S:
    - $\bar{R} \leftarrow$ predicted reward for $\bar{S},\bar{A},S$
    - $P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.
    - if $P > \theta$ then insert $\bar{S},\bar{A}$ into $PQueue$ with priority P

Example 8.4: Prioritized Sweeping on Mazes Prioritized sweeping has been found to dramatically increase the speed at which optimal solutions are found in maze tasks, often by a factor of 5 to 10. A typical example is shown to the right. These data are for a sequence of maze tasks of exactly the same structure as the one shown in Figure 8.2, except that they vary in the grid resolution. Prioritized sweeping maintained a decisive advantage over unprioritized Dyna-Q. Both systems made at most $n = 5$ updates per environmental interaction. Adapted from Peng and Williams (1993).

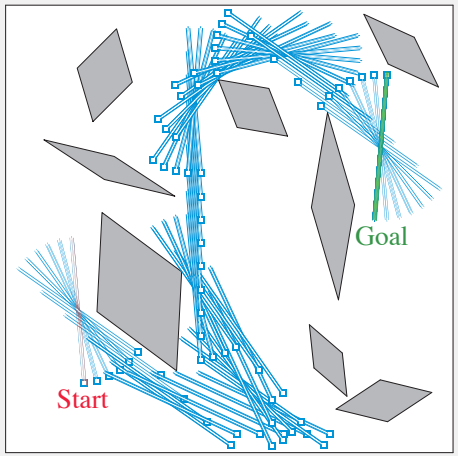


Extensions of prioritized sweeping to stochastic environments are straightforward. The model is maintained by keeping counts of the number of times each state–action pair has been experienced and of what the next states were. It is natural then to update each pair not with a sample update, as we have been using so far, but with an expected update, taking into account all possible next states and their probabilities of occurring.

Prioritized sweeping is just one way of distributing computations to improve planning efficiency, and probably not the best way. One of prioritized sweeping's limitations is that it uses *expected* updates, which in stochastic environments may waste lots of computation on low-probability transitions. As we show in the following section, sample updates

Example 8.5 Prioritized Sweeping for Rod Maneuvering

The objective in this task is to maneuver a rod around some awkwardly placed obstacles within a limited rectangular work space to a goal position in the fewest number of steps. The rod can be translated along its long axis or perpendicular to that axis, or it can be rotated in either direction around its center. The distance of each movement is approximately 1/20 of the work space, and the rotation increment is 10 degrees. Translations are deterministic and quantized to one of 20×20 positions. To the right is shown the obstacles and the shortest solution from start to goal, found by prioritized sweeping. This problem is deterministic, but has four actions and 14,400 potential states (some of these are unreachable because of the obstacles). This problem is probably too large to be solved with unprioritized methods. Figure reprinted from Moore and Atkeson (1993).



can in many cases get closer to the true value function with less computation despite the variance introduced by sampling. Sample updates can win because they break the overall backing-up computation into smaller pieces—those corresponding to individual transitions—which then enables it to be focused more narrowly on the pieces that will have the largest impact. This idea was taken to what may be its logical limit in the "small backups" introduced by van Seijen and Sutton (2013). These are updates along a single transition, like a sample update, but based on the probability of the transition without sampling, as in an expected update. By selecting the order in which small updates are done it is possible to greatly improve planning efficiency beyond that possible with prioritized sweeping.

We have suggested in this chapter that all kinds of state-space planning can be viewed as sequences of value updates, varying only in the type of update, expected or sample, large or small, and in the order in which the updates are done. In this section we have emphasized backward focusing, but this is just one strategy. For example, another would be to focus on states according to how easily they can be reached from the states that are visited frequently under the current policy, which might be called *forward focusing*. Peng and Williams (1993) and Barto, Bradtke and Singh (1995) have explored versions of forward focusing, and the methods introduced in the next few sections take it to an extreme form.

8.5 Expected vs. Sample Updates

The examples in the previous sections give some idea of the range of possibilities for combining methods of learning and planning. In the rest of this chapter, we analyze some of the component ideas involved, starting with the relative advantages of expected and sample updates.

Much of this book has been about different kinds of value-function updates, and we have considered a great many varieties. Focusing for the moment on one-step updates, they vary primarily along three binary dimensions. The first two dimensions are whether they update state values or action values and whether they estimate the value for the optimal policy or for an arbitrary given policy. These two dimensions give rise to four classes of updates for approximating the four value functions, q_*, v_*, q_π, and v_π. The other binary dimension is whether the updates are *expected* updates, considering all possible events that might happen, or *sample* updates, considering a single sample of what might happen. These three binary dimensions give rise to eight cases, seven of which correspond to specific algorithms, as shown in the figure to the right. (The eighth case does not seem to correspond to any useful update.) Any of these one-step updates can be used in planning methods. The Dyna-Q agents discussed earlier use q_* sample updates, but they could just as well use q_* expected updates, or either expected or sample q_π updates. The Dyna-AC system uses v_π sample updates together with a learning policy structure (as in Chapter 13). For stochastic problems, prioritized sweeping is always done using one of the expected updates.

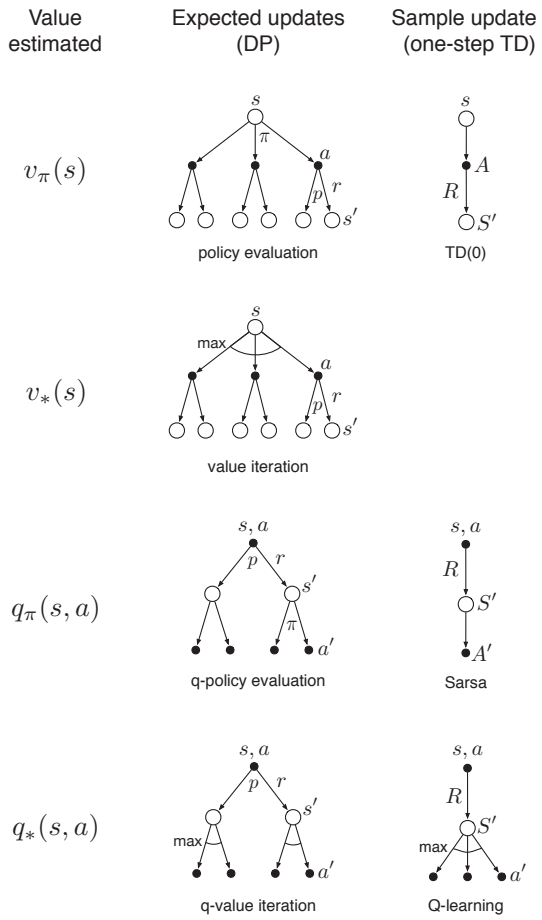


Figure 8.6: Backup diagrams for all the one-step updates considered in this book.

When we introduced one-step sample updates in Chapter 6, we presented them as substitutes for expected updates. In the absence of a distribution model, expected updates are not possible, but sample updates can be done using sample transitions from the environment or a sample model. Implicit in that point of view is that expected updates, if possible, are preferable to sample updates. But are they? Expected

updates certainly yield a better estimate because they are uncorrupted by sampling error, but they also require more computation, and computation is often the limiting resource in planning. To properly assess the relative merits of expected and sample updates for planning we must control for their different computational requirements.

For concreteness, consider the expected and sample updates for approximating q_*, and the special case of discrete states and actions, a table-lookup representation of the approximate value function, Q, and a model in the form of estimated dynamics, $\hat{p}(s', r \,|\, s, a)$. The expected update for a state–action pair, s, a, is:

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s', r \,|\, s, a) \Big[r + \gamma \max_{a'} Q(s', a') \Big]. \tag{8.1}$$

The corresponding sample update for s, a, given a sample next state and reward, S' and R (from the model), is the Q-learning-like update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[R + \gamma \max_{a'} Q(S', a') - Q(s,a) \Big], \tag{8.2}$$

where α is the usual positive step-size parameter.

The difference between these expected and sample updates is significant to the extent that the environment is stochastic, specifically, to the extent that, given a state and action, many possible next states may occur with various probabilities. If only one next state is possible, then the expected and sample updates given above are identical (taking $\alpha = 1$). If there are many possible next states, then there may be significant differences. In favor of the expected update is that it is an exact computation, resulting in a new $Q(s, a)$ whose correctness is limited only by the correctness of the $Q(s', a')$ at successor states. The sample update is in addition affected by sampling error. On the other hand, the sample update is cheaper computationally because it considers only one next state, not all possible next states. In practice, the computation required by update operations is usually dominated by the number of state–action pairs at which Q is evaluated. For a particular starting pair, s, a, let b be the *branching factor* (i.e., the number of possible next states, s', for which $\hat{p}(s' \,|\, s, a) > 0$). Then an expected update of this pair requires roughly b times as much computation as a sample update.

If there is enough time to complete an expected update, then the resulting estimate is generally better than that of b sample updates because of the absence of sampling error. But if there is insufficient time to complete an expected update, then sample updates are always preferable because they at least make some improvement in the value estimate with fewer than b updates. In a large problem with many state–action pairs, we are often in the latter situation. With so many state–action pairs, expected updates of all of them would take a very long time. Before that we may be much better off with a few sample updates at many state–action pairs than with expected updates at a few pairs. Given a unit of computational effort, is it better devoted to a few expected updates or to b times as many sample updates?

Figure 8.7 shows the results of an analysis that suggests an answer to this question. It shows the estimation error as a function of computation time for expected and sample updates for a variety of branching factors, b. The case considered is that in which all

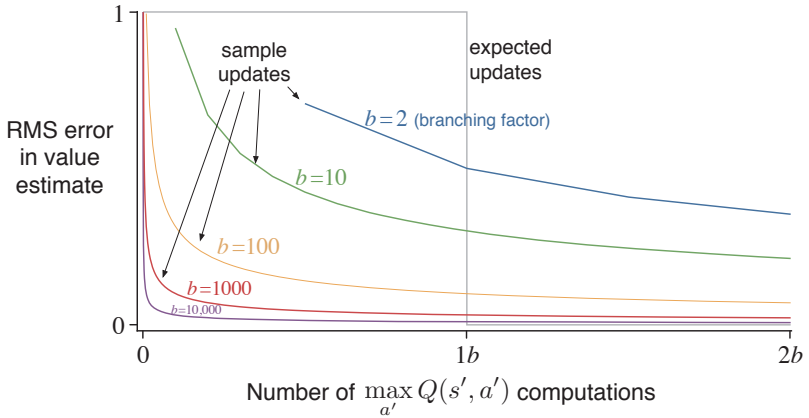


Figure 8.7: Comparison of efficiency of expected and sample updates.

b successor states are equally likely and in which the error in the initial estimate is 1. The values at the next states are assumed correct, so the expected update reduces the error to zero upon its completion. In this case, sample updates reduce the error according to $\sqrt{\frac{b-1}{bt}}$ where t is the number of sample updates that have been performed (assuming sample averages, i.e., $\alpha = 1/t$). The key observation is that for moderately large b the error falls dramatically with a tiny fraction of b updates. For these cases, many state–action pairs could have their values improved dramatically, to within a few percent of the effect of an expected update, in the same time that a single state–action pair could undergo an expected update.

The advantage of sample updates shown in Figure 8.7 is probably an underestimate of the real effect. In a real problem, the values of the successor states would be estimates that are themselves updated. By causing estimates to be more accurate sooner, sample updates will have a second advantage in that the values backed up from the successor states will be more accurate. These results suggest that sample updates are likely to be superior to expected updates on problems with large stochastic branching factors and too many states to be solved exactly.

Exercise 8.6 The analysis above assumed that all of the b possible next states were equally likely to occur. Suppose instead that the distribution was highly skewed, that some of the b states were much more likely to occur than most. Would this strengthen or weaken the case for sample updates over expected updates? Support your answer. □

8.6 Trajectory Sampling

In this section we compare two ways of distributing updates. The classical approach, from dynamic programming, is to perform sweeps through the entire state (or state–action) space, updating each state (or state–action pair) once per sweep. This is problematic

on large tasks because there may not be time to complete even one sweep. In many tasks the vast majority of the states are irrelevant because they are visited only under very poor policies or with very low probability. Exhaustive sweeps implicitly devote equal time to all parts of the state space rather than focusing where it is needed. As we discussed in Chapter 4, exhaustive sweeps and the equal treatment of all states that they imply are not necessary properties of dynamic programming. In principle, updates can be distributed any way one likes (to assure convergence, all states or state–action pairs must be visited in the limit an infinite number of times; although an exception to this is discussed in Section 8.7 below), but in practice exhaustive sweeps are often used.

The second approach is to sample from the state or state–action space according to some distribution. One could sample uniformly, as in the Dyna-Q agent, but this would suffer from some of the same problems as exhaustive sweeps. More appealing is to distribute updates according to the on-policy distribution, that is, according to the distribution observed when following the current policy. One advantage of this distribution is that it is easily generated; one simply interacts with the model, following the current policy. In an episodic task, one starts in a start state (or according to the starting-state distribution) and simulates until the terminal state. In a continuing task, one starts anywhere and just keeps simulating. In either case, sample state transitions and rewards are given by the model, and sample actions are given by the current policy. In other words, one simulates explicit individual trajectories and performs updates at the state or state–action pairs encountered along the way. We call this way of generating experience and updates *trajectory sampling*.

It is hard to imagine any efficient way of distributing updates according to the on-policy distribution other than by trajectory sampling. If one had an explicit representation of the on-policy distribution, then one could sweep through all states, weighting the update of each according to the on-policy distribution, but this leaves us again with all the computational costs of exhaustive sweeps. Possibly one could sample and update individual state–action pairs from the distribution, but even if this could be done efficiently, what benefit would this provide over simulating trajectories? Even knowing the on-policy distribution in an explicit form is unlikely. The distribution changes whenever the policy changes, and computing the distribution requires computation comparable to a complete policy evaluation. Consideration of such other possibilities makes trajectory sampling seem both efficient and elegant.

Is the on-policy distribution of updates a good one? Intuitively it seems like a good choice, at least better than the uniform distribution. For example, if you are learning to play chess, you study positions that might arise in real games, not random positions of chess pieces. The latter may be valid states, but to be able to accurately value them is a different skill from evaluating positions in real games. We will also see in Part II that the on-policy distribution has significant advantages when function approximation is used. Whether or not function approximation is used, one might expect on-policy focusing to significantly improve the speed of planning.

Focusing on the on-policy distribution could be beneficial because it causes vast, uninteresting parts of the space to be ignored, or it could be detrimental because it causes the same old parts of the space to be updated over and over. We conducted a small

experiment to assess the effect empirically. To isolate the effect of the update distribution, we used entirely one-step expected tabular updates, as defined by (8.1). In the *uniform* case, we cycled through all state–action pairs, updating each in place, and in the *on-policy* case we simulated episodes, all starting in the same state, updating each state–action pair that occurred under the current ε-greedy policy ($\varepsilon = 0.1$). The tasks were undiscounted episodic tasks, generated randomly as follows. From each of the $|\mathcal{S}|$ states, two actions were possible, each of which resulted in one of b next states, all equally likely, with a different random selection of b states for each state–action pair. The branching factor, b, was the same for all state–action pairs. In addition, on all transitions there was a 0.1 probability of transition to the terminal state, ending the episode. The expected reward on each transition was selected from a Gaussian distribution with mean 0 and variance 1.

At any point in the planning process one can stop and exhaustively compute $v_{\tilde{\pi}}(s_0)$, the true value of the start state under the greedy policy, $\tilde{\pi}$, given the current action-value function Q, as an indication of how well the agent would do on a new episode on which it acted greedily (all the while assuming the model is correct).

The upper part of the figure to the right shows results averaged over 200 sample tasks with 1000 states and branching factors of 1, 3, and 10. The quality of the policies found is plotted as a function of the number of expected updates completed. In all cases, sampling according to the on-policy distribution resulted in faster planning initially and retarded planning in the long run. The effect was stronger, and the initial period of faster planning was longer, at smaller branching factors. In other experiments, we found that these effects also became stronger as the number of states increased. For example, the lower part of the figure shows results for a branching factor of 1 for tasks with 10,000 states. In this case the advantage of on-policy focusing is large and long-lasting.

All of these results make sense. In the short term, sampling according to the on-policy distribution helps by focusing on states that are near descendants of

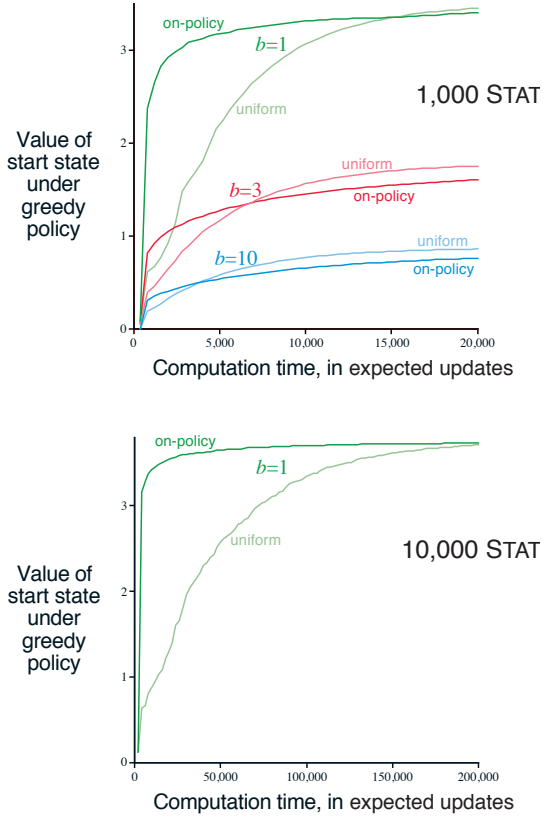


Figure 8.8: Relative efficiency of updates distributed uniformly across the state space versus focused on simulated on-policy trajectories, each starting in the same state. Results are for randomly generated tasks of two sizes and various branching factors, b.

the start state. If there are many states and a small branching factor, this effect will be large and long-lasting. In the long run, focusing on the on-policy distribution may hurt because the commonly occurring states all already have their correct values. Sampling them is useless, whereas sampling other states may actually perform some useful work. This presumably is why the exhaustive, unfocused approach does better in the long run, at least for small problems. These results are not conclusive because they are only for problems generated in a particular, random way, but they do suggest that sampling according to the on-policy distribution can be a great advantage for large problems, in particular for problems in which a small subset of the state–action space is visited under the on-policy distribution.

Exercise 8.7 Some of the graphs in Figure 8.8 seem to be scalloped in their early portions, particularly the upper graph for $b = 1$ and the uniform distribution. Why do you think this is? What aspects of the data shown support your hypothesis? □

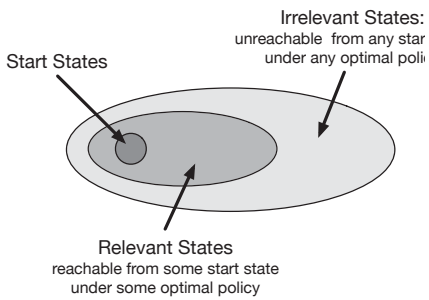
Exercise 8.8 (programming) Replicate the experiment whose results are shown in the lower part of Figure 8.8, then try the same experiment but with $b = 3$. Discuss the meaning of your results. □

8.7 Real-time Dynamic Programming

Real-time dynamic programming, or RTDP, is an on-policy trajectory-sampling version of the value-iteration algorithm of dynamic programming (DP). Because it is closely related to conventional sweep-based policy iteration, RTDP illustrates in a particularly clear way some of the advantages that on-policy trajectory sampling can provide. RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates as defined by (4.10). It is basically the algorithm that produced the on-policy results shown in Figure 8.8.

The close connection between RTDP and conventional DP makes it possible to derive some theoretical results by adapting existing theory. RTDP is an example of an *asynchronous* DP algorithm as described in Section 4.5. Asynchronous DP algorithms are not organized in terms of systematic sweeps of the state set; they update state values in any order whatsoever, using whatever values of other states happen to be available. In RTDP, the update order is dictated by the order states are visited in real or simulated trajectories.

If trajectories can start only from a designated set of start states, and if you are interested in the prediction problem for a given policy, then on-policy trajectory sampling allows the algorithm to completely skip states that cannot be reached by the given policy from any of the start states: such states are *irrelevant* to the prediction problem. For a control problem, where the goal is to find an optimal policy instead of evaluating a given policy, there might well be states that cannot be



reached by any optimal policy from any of the start states, and there is no need to specify optimal actions for these irrelevant states. What is needed is an *optimal partial policy*, meaning a policy that is optimal for the relevant states but can specify arbitrary actions, or even be undefined, for the irrelevant states.

But *finding* such an optimal partial policy with an on-policy trajectory-sampling control method, such as Sarsa (Section 6.4), in general requires visiting all state–action pairs—even those that will turn out to be irrelevant—an infinite number of times. This can be done, for example, by using exploring starts (Section 5.3). This is true for RTDP as well: for episodic tasks with exploring starts, RTDP is an asynchronous value-iteration algorithm that converges to optimal polices for discounted finite MDPs (and for the undiscounted case under certain conditions). Unlike the situation for a prediction problem, it is generally not possible to stop updating any state or state–action pair if convergence to an optimal policy is important.

The most interesting result for RTDP is that for certain types of problems satisfying reasonable conditions, RTDP is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all. Indeed, in some problems, only a small fraction of the states need to be visited. This can be a great advantage for problems with very large state sets, where even a single sweep may not be feasible.

The tasks for which this result holds are undiscounted episodic tasks for MDPs with absorbing goal states that generate zero rewards, as described in Section 3.4. At every step of a real or simulated trajectory, RTDP selects a greedy action (breaking ties randomly) and applies the expected value-iteration update operation to the current state. It can also update the values of an arbitrary collection of other states at each step; for example, it can update the values of states visited in a limited-horizon look-ahead search from the current state.

For these problems, with each episode beginning in a state randomly chosen from the set of start states and ending at a goal state, RTDP converges with probability one to a policy that is optimal for all the relevant states provided: 1) the initial value of every goal state is zero, 2) there exists at least one policy that guarantees that a goal state will be reached with probability one from any start state, 3) all rewards for transitions from non-goal states are strictly negative, and 4) all the initial values are equal to, or greater than, their optimal values (which can be satisfied by simply setting the initial values of all states to zero). This result was proved by Barto, Bradtke, and Singh (1995) by combining results for asynchronous DP with results about a heuristic search algorithm known as *learning real-time A** due to Korf (1990).

Tasks having these properties are examples of *stochastic optimal path problems*, which are usually stated in terms of cost minimization instead of as reward maximization as we do here. Maximizing the negative returns in our version is equivalent to minimizing the costs of paths from a start state to a goal state. Examples of this kind of task are minimum-time control tasks, where each time step required to reach a goal produces a reward of -1, or problems like the Golf example in Section 3.5, whose objective is to hit the hole with the fewest strokes.

Example 8.6: RTDP on the Racetrack The racetrack problem of Exercise 5.12 (page 111) is a stochastic optimal path problem. Comparing RTDP and the conventional DP value iteration algorithm on an example racetrack problem illustrates some of the advantages of on-policy trajectory sampling.

Recall from the exercise that an agent has to learn how to drive a car around a turn like those shown in Figure 5.5 and cross the finish line as quickly as possible while staying on the track. Start states are all the zero-speed states on the starting line; the goal states are all the states that can be reached in one time step by crossing the finish line from inside the track. Unlike Exercise 5.12, here there is no limit on the car's speed, so the state set is potentially infinite. However, the set of states that can be reached from the set of start states via any policy is finite and can be considered to be the state set of the problem. Each episode begins in a randomly selected start state and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random start state, and the episode continues.

A racetrack similar to the small racetrack on the left of Figure 5.5 has 9,115 states reachable from start states by any policy, only 599 of which are relevant, meaning that they are reachable from some start state via some optimal policy. (The number of relevant states was estimated by counting the states visited while executing optimal actions for 10^7 episodes.)

The table below compares solving this task by conventional DP and by RTDP. These results are averages over 25 runs, each begun with a different random number seed. Conventional DP in this case is value iteration using exhaustive sweeps of the state set, with values updated one state at a time in place, meaning that the update for each state uses the most recent values of the other states (This is the Gauss-Seidel version of value iteration, which was found to be approximately twice as fast as the Jacobi version on this problem. See Section 4.8.) No special attention was paid to the ordering of the updates; other orderings could have produced faster convergence. Initial values were all zero for each run of both methods. DP was judged to have converged when the maximum change in a state value over a sweep was less than 10^{-4}, and RTDP was judged to have converged when the average time to cross the finish line over 20 episodes appeared to stabilize at an asymptotic number of steps. This version of RTDP updated only the value of the current state on each step.

	DP	RTDP
Average computation to convergence	28 sweeps	4000 episodes
Average number of updates to convergence	252,784	127,600
Average number of updates per episode	—	31.9
% of states updated $\leq$ 100 times	—	98.45
% of states updated $\leq$ 10 times	—	80.51
% of states updated 0 times	—	3.18

Both methods produced policies averaging between 14 and 15 steps to cross the finish line, but RTDP required only roughly half of the updates that DP did. This is the result of RTDP's on-policy trajectory sampling. Whereas the value of every state was updated in each sweep of DP, RTDP focused updates on fewer states. In an average run, RTDP updated the values of 98.45% of the states no more than 100 times and 80.51% of the states no more than 10 times; the values of about 290 states were not updated at all in an average run. ■

Another advantage of RTDP is that as the value function approaches the optimal value function v_*, the policy used by the agent to generate trajectories approaches an optimal policy because it is always greedy with respect to the current value function. This is in contrast to the situation in conventional value iteration. In practice, value iteration terminates when the value function changes by only a small amount in a sweep, which is how we terminated it to obtain the results in the table above. At this point, the value function closely approximates v_*, and a greedy policy is close to an optimal policy. However, it is possible that policies that are greedy with respect to the latest value function were optimal, or nearly so, long before value iteration terminates. (Recall from Chapter 4 that optimal policies can be greedy with respect to many different value functions, not just v_*.) Checking for the emergence of an optimal policy before value iteration converges is not a part of the conventional DP algorithm and requires considerable additional computation.

In the racetrack example, by running many test episodes after each DP sweep, with actions selected greedily according to the result of that sweep, it was possible to estimate the earliest point in the DP computation at which the approximated optimal evaluation function was good enough so that the corresponding greedy policy was nearly optimal. For this racetrack, a close-to-optimal policy emerged after 15 sweeps of value iteration, or after 136,725 value-iteration updates. This is considerably less than the 252,784 updates DP needed to converge to v_*, but sill more than the 127,600 updates RTDP required.

Although these simulations are certainly not definitive comparisons of the RTDP with conventional sweep-based value iteration, they illustrate some of advantages of on-policy trajectory sampling. Whereas conventional value iteration continued to update the value of all the states, RTDP strongly focused on subsets of the states that were relevant to the problem's objective. This focus became increasingly narrow as learning continued. Because the convergence theorem for RTDP applies to the simulations, we know that RTDP eventually would have focused only on relevant states, i.e., on states making up optimal paths. RTDP achieved nearly optimal control with about 50% of the computation required by sweep-based value iteration.

8.8 Planning at Decision Time

Planning can be used in at least two ways. The one we have considered so far in this chapter, typified by dynamic programming and Dyna, is to use planning to gradually improve a policy or value function on the basis of simulated experience obtained from a model (either a sample or a distribution model). Selecting actions is then a matter of comparing the current state's action values obtained from a table in the tabular case we have thus far considered, or by evaluating a mathematical expression in the approximate methods we consider in Part II below. Well before an action is selected for any current state S_t, planning has played a part in improving the table entries, or the

mathematical expression, needed to select the action for many states, including S_t. Used this way, planning is not focussed on the current state. We call planning used in this way *background planning*.

The other way to use planning is to begin and complete it *after* encountering each new state S_t, as a computation whose output is the selection of a single action A_t; on the next step planning begins anew with S_{t+1} to produce A_{t+1}, and so on. The simplest, and almost degenerate, example of this use of planning is when only state values are available, and an action is selected by comparing the values of model-predicted next states for each action (or by comparing the values of afterstates as in the tic-tac-toe example in Chapter 1). More generally, planning used in this way can look much deeper than one-step-ahead and evaluate action choices leading to many different predicted state and reward trajectories. Unlike the first use of planning, here planning focuses on a particular state. We call this *decision-time planning*.

These two ways of thinking about planning—using simulated experience to gradually improve a policy or value function, or using simulated experience to select an action for the current state—can blend together in natural and interesting ways, but they have tended to be studied separately, and that is a good way to first understand them. Let us now take a closer look at decision-time planning.

Even when planning is only done at decision time, we can still view it, as we did in Section 8.1, as proceeding from simulated experience to updates and values, and ultimately to a policy. It is just that now the values and policy are specific to the current state and the action choices available there, so much so that the values and policy created by the planning process are typically discarded after being used to select the current action. In many applications this is not a great loss because there are very many states and we are unlikely to return to the same state for a long time. In general, one may want to do a mix of both: focus planning on the current state *and* store the results of planning so as to be that much farther along should one return to the same state later. Decision-time planning is most useful in applications in which fast responses are not required. In chess playing programs, for example, one may be permitted seconds or minutes of computation for each move, and strong programs may plan dozens of moves ahead within this time. On the other hand, if low latency action selection is the priority, then one is generally better off doing planning in the background to compute a policy that can then be rapidly applied to each newly encountered state.

8.9 Heuristic Search

The classical state-space planning methods in artificial intelligence are decision-time planning methods collectively known as *heuristic search*. In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the expected updates with maxes (those for v_* and q_*) discussed throughout this book. The backing up stops at the state–action nodes for the current state. Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all

backed-up values are discarded.

In conventional heuristic search no effort is made to save the backed-up values by changing the approximate value function. In fact, the value function is generally designed by people and never changed as a result of search. However, it is natural to consider allowing the value function to be improved over time, using either the backed-up values computed during heuristic search or any of the other methods presented throughout this book. In a sense we have taken this approach all along. Our greedy, ε-greedy, and UCB (Section 2.7) action-selection methods are not unlike heuristic search, albeit on a smaller scale. For example, to compute the greedy action given a model and a state-value function, we must look ahead from each possible action to each possible next state, take into account the rewards and estimated values, and then pick the best action. Just as in conventional heuristic search, this process computes backed-up values of the possible actions, but does not attempt to save them. Thus, heuristic search can be viewed as an extension of the idea of a greedy policy beyond a single step.

The point of searching deeper than one step is to obtain better action selections. If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies.[2] Certainly, if the search is all the way to the end of the episode, then the effect of the imperfect value function is eliminated, and the action determined in this way must be optimal. If the search is of sufficient depth k such that γ^k is very small, then the actions will be correspondingly near optimal. On the other hand, the deeper the search, the more computation is required, usually resulting in a slower response time. A good example is provided by Tesauro's grandmaster-level backgammon player, TD-Gammon (Section 16.1). This system used TD learning to learn an afterstate value function through many games of self-play, using a form of heuristic search to make its moves. As a model, TD-Gammon used a priori knowledge of the probabilities of dice rolls and the assumption that the opponent always selected the actions that TD-Gammon rated as best for it. Tesauro found that the deeper the heuristic search, the better the moves made by TD-Gammon, but the longer it took to make each move. Backgammon has a large branching factor, yet moves must be made within a few seconds. It was only feasible to search ahead selectively a few steps, but even so the search resulted in significantly better action selections.

We should not overlook the most obvious way in which heuristic search focuses updates: on the current state. Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follow the current state. You may spend more of your life playing chess than checkers, but when you play checkers, it pays to think about checkers and about your particular checkers position, your likely next moves, and successor positions. No matter how you select actions, it is these states and actions that are of highest priority for updates and where you most urgently want your approximate value function to be accurate. Not only should your computation be preferentially devoted to imminent events, but so should your limited memory resources. In chess, for example, there are far too many possible positions to store distinct value estimates for each of them, but chess programs based on heuristic search can easily store distinct estimates for the millions of positions they encounter

[2] There are interesting exceptions to this (see, e.g., Pearl, 1984).

looking ahead from a single position. This great focusing of memory and computational resources on the current decision is presumably the reason why heuristic search can be so effective.

The distribution of updates can be altered in similar ways to focus on the current state and its likely successors. As a limiting case we might use exactly the methods of heuristic search to construct a search tree, and then perform the individual, one-step updates from bottom up, as suggested by Figure 8.9. If the updates are ordered in this way and a tabular representation is used, then exactly the same overall update would be achieved as in depth-first heuristic search. Any state-space search can be viewed in this way as the piecing together of a large number of individual one-step updates. Thus, the performance improvement observed with deeper searches is not due to the use of multistep updates as such. Instead, it is due to the focus and concentration of updates on states and actions immediately downstream from the current state. By devoting a large amount of computation specifically relevant to the candidate actions, decision-time planning can produce better decisions than can be produced by relying on unfocused updates.

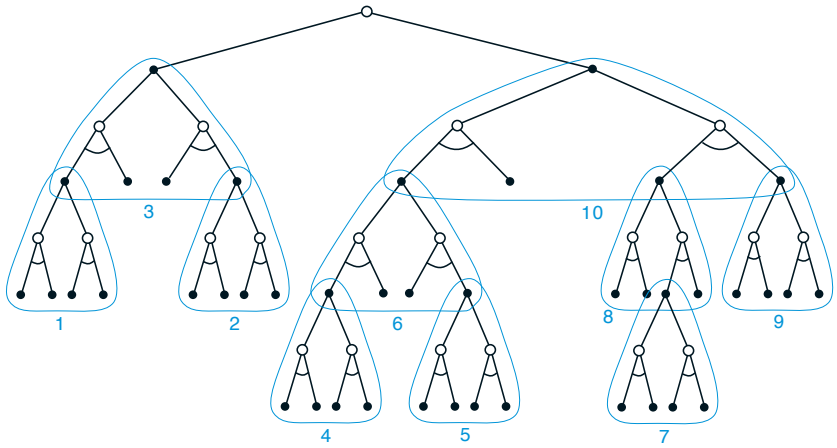


Figure 8.9: Heuristic search can be implemented as a sequence of one-step updates (shown here outlined in blue) backing up values from the leaf nodes toward the root. The ordering shown here is for a selective depth-first search.

8.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy. When the action-value estimates are considered to be accurate enough, the action (or one of the

actions) having the highest estimated value is executed, after which the process is carried out anew from the resulting next state. As explained by Tesauro and Galperin (1997), who experimented with rollout algorithms for playing backgammon, the term "rollout" comes from estimating the value of a backgammon position by playing out, i.e., "rolling out," the position many times to the game's end with randomly generated sequences of dice rolls, where the moves of both players are made by some fixed policy.

Unlike the Monte Carlo control algorithms described in Chapter 5, the goal of a rollout algorithm is not to estimate a complete optimal action-value function, q_*, or a complete action-value function, q_π, for a given policy π. Instead, they produce Monte Carlo estimates of action values only for each current state and for a given policy usually called the *rollout policy*. As decision-time planning algorithms, rollout algorithms make immediate use of these action-value estimates, then discard them. This makes rollout algorithms relatively simple to implement because there is no need to sample outcomes for every state-action pair, and there is no need to approximate a function over either the state space or the state-action space.

What then do rollout algorithms accomplish? The policy improvement theorem described in Section 4.2 tells us that given any two policies π and π' that are identical except that $\pi'(s) = a \neq \pi(s)$ for some state s, if $q_\pi(s, a) \geq v_\pi(s)$, then policy π' is as good as, or better, than π. Moreover, if the inequality is strict, then π' is in fact better than π. This applies to rollout algorithms where s is the current state and π is the rollout policy. Averaging the returns of the simulated trajectories produces estimates of $q_\pi(s, a')$ for each action $a' \in \mathcal{A}(s)$. Then the policy that selects an action in s that maximizes these estimates and thereafter follows π is a good candidate for a policy that improves over π. The result is like one step of the policy-iteration algorithm of dynamic programming discussed in Section 4.3 (though it is more like one step of *asynchronous* value iteration described in Section 4.5 because it changes the action for just the current state).

In other words, the aim of a rollout algorithm is to improve upon the rollout policy; not to find an optimal policy. Experience has shown that rollout algorithms can be surprisingly effective. For example, Tesauro and Galperin (1997) were surprised by the dramatic improvements in backgammon playing ability produced by the rollout method. In some applications, a rollout algorithm can produce good performance even if the rollout policy is completely random. But the performance of the improved policy depends on properties of the rollout policy and the ranking of actions produced by the Monte Carlo value estimates. Intuition suggests that the better the rollout policy and the more accurate the value estimates, the better the policy produced by a rollout algorithm is likely be (but see Gelly and Silver, 2007).

This involves important tradeoffs because better rollout policies typically mean that more time is needed to simulate enough trajectories to obtain good value estimates. As decision-time planning methods, rollout algorithms usually have to meet strict time constraints. The computation time needed by a rollout algorithm depends on the number of actions that have to be evaluated for each decision, the number of time steps in the simulated trajectories needed to obtain useful sample returns, the time it takes the rollout policy to make decisions, and the number of simulated trajectories needed to obtain good Monte Carlo action-value estimates.

Balancing these factors is important in any application of rollout methods, though there are several ways to ease the challenge. Because the Monte Carlo trials are independent of one another, it is possible to run many trials in parallel on separate processors. Another approach is to truncate the simulated trajectories short of complete episodes, correcting the truncated returns by means of a stored evaluation function (which brings into play all that we have said about truncated returns and updates in the preceding chapters). It is also possible, as Tesauro and Galperin (1997) suggest, to monitor the Monte Carlo simulations and prune away candidate actions that are unlikely to turn out to be the best, or whose values are close enough to that of the current best that choosing them instead would make no real difference (though Tesauro and Galperin point out that this would complicate a parallel implementation).

We do not ordinarily think of rollout algorithms as *learning* algorithms because they do not maintain long-term memories of values or policies. However, these algorithms take advantage of some of the features of reinforcement learning that we have emphasized in this book. As instances of Monte Carlo control, they estimate action values by averaging the returns of a collection of sample trajectories, in this case trajectories of simulated interactions with a sample model of the environment. In this way they are like reinforcement learning algorithms in avoiding the exhaustive sweeps of dynamic programming by trajectory sampling, and in avoiding the need for distribution models by relying on sample, instead of expected, updates. Finally, rollout algorithms take advantage of the policy improvement property by acting greedily with respect to the estimated action values.

8.11 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning. At its base, MCTS is a rollout algorithm as described above, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo simulations in order to successively direct simulations toward more highly-rewarding trajectories. MCTS is largely responsible for the improvement in computer Go from a weak amateur level in 2005 to a grandmaster level (6 dan or more) in 2015. Many variations of the basic algorithm have been developed, including a variant that we discuss in Section 16.6 that was critical for the stunning 2016 victories of the program AlphaGo over an 18-time world champion Go player. MCTS has proved to be effective in a wide variety of competitive settings, including general game playing (e.g., see Finnsson and Björnsson, 2008; Genesereth and Thielscher, 2014), but it is not limited to games; it can be effective for single-agent sequential decision problems if there is an environment model simple enough for fast multistep simulation.

MCTS is executed after encountering each new state to select the agent's action for that state; it is executed again to select the action for the next state, and so on. As in a rollout algorithm, each execution is an iterative process that simulates many trajectories starting from the current state and running to a terminal state (or until discounting makes any further reward negligible as a contribution to the return). The core idea of MCTS is to successively focus multiple simulations starting at the current state by

extending the initial portions of trajectories that have received high evaluations from earlier simulations. MCTS does not have to retain approximate value functions or policies from one action selection to the next, though in many implementations it retains selected action values likely to be useful for its next execution.

For the most part, the actions in the simulated trajectories are generated using a simple policy, usually called a rollout policy as it is for simpler rollout algorithms. When both the rollout policy and the model do not require a lot of computation, many simulated trajectories can be generated in a short period of time. As in any tabular Monte Carlo method, the value of a state–action pair is estimated as the average of the (simulated) returns from that pair. Monte Carlo value estimates are maintained only for the subset of state–action pairs that are most likely to be reached in a few steps, which form a tree rooted at the current state, as illustrated in Figure 8.10. MCTS incrementally extends the tree by adding nodes representing states that look promising based on the results of the simulated trajectories. Any simulated trajectory will pass through the tree and then exit it at some leaf node. Outside the tree and at the leaf nodes the rollout policy is used for action selections, but at the states inside the tree something better is possible. For these states we have value estimates for of at least some of the actions, so we can pick among them using an informed policy, called the *tree policy*, that balances exploration

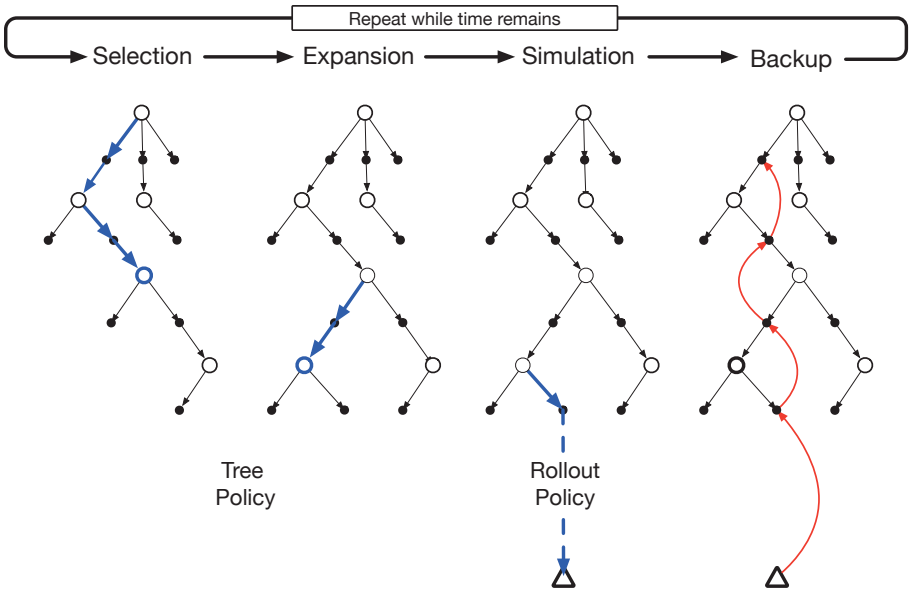


Figure 8.10: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**, as explained in the text and illustrated by the bold arrows in the trees. Adapted from Chaslot, Bakkes, Szita, and Spronck (2008).

and exploitation. For example, the tree policy could select actions using an ε-greedy or UCB selection rule (Chapter 2).

In more detail, each iteration of a basic version of MCTS consists of the following four steps as illustrated in Figure 8.10:

1. **Selection.** Starting at the root node, a *tree policy* based on the action values attached to the edges of the tree traverses the tree to select a leaf node.

2. **Expansion.** On some iterations (depending on details of the application), the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.

3. **Simulation.** From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.

4. **Backup.** The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree. Figure 8.10 illustrates this by showing a backup from the terminal state of the simulated trajectory directly to the state–action node in the tree where the rollout policy began (though in general, the entire return over the simulated trajectory is backed up to this state–action node).

MCTS continues executing these four steps, starting each time at the tree's root node, until no more time is left, or some other computational resource is exhausted. Then, finally, an action from the root node (which still represents the current state of the environment) is selected according to some mechanism that depends on the accumulated statistics in the tree; for example, it may be an action having the largest action value of all the actions available from the root state, or perhaps the action with the largest visit count to avoid selecting outliers. This is the action MCTS actually selects. After the environment transitions to a new state, MCTS is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS; all the remaining nodes are discarded, along with the action values associated with them.

MCTS was first proposed to select moves in programs playing two-person competitive games, such as Go. For game playing, each simulated episode is one complete play of the game in which both players select actions by the tree and rollout policies. Section 16.6 describes an extension of MCTS used in the AlphaGo program that combines the Monte Carlo evaluations of MCTS with action values learned by a deep artificial neural network via self-play reinforcement learning.

Relating MCTS to the reinforcement learning principles we describe in this book provides some insight into how it achieves such impressive results. At its base, MCTS is a decision-time planning algorithm based on Monte Carlo control applied to simulations

that start from the root state; that is, it is a kind of rollout algorithm as described in the previous section. It therefore benefits from online, incremental, sample-based value estimation and policy improvement. Beyond this, it saves action-value estimates attached to the tree edges and updates them using reinforcement learning's sample updates. This has the effect of focusing the Monte Carlo trials on trajectories whose initial segments are common to high-return trajectories previously simulated. Further, by incrementally expanding the tree, MCTS effectively grows a lookup table to store a partial action-value function, with memory allocated to the estimated values of state–action pairs visited in the initial segments of high-yielding sample trajectories. MCTS thus avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration.

The striking success of decision-time planning by MCTS has deeply influenced artificial intelligence, and many researchers are studying modifications and extensions of the basic procedure for use in both games and single-agent applications.

8.12 Summary of the Chapter

Planning requires a model of the environment. A *distribution model* consists of the probabilities of next states and rewards for possible actions; a sample model produces single transitions and rewards generated according to these probabilities. Dynamic programming requires a distribution model because it uses *expected updates*, which involve computing expectations over all the possible next states and rewards. A *sample model*, on the other hand, is what is needed to simulate interacting with the environment during which *sample updates*, like those used by many reinforcement learning algorithms, can be used. Sample models are generally much easier to obtain than distribution models.

We have presented a perspective emphasizing the surprisingly close relationships between planning optimal behavior and learning optimal behavior. Both involve estimating the same value functions, and in both cases it is natural to update the estimates incrementally, in a long series of small backing-up operations. This makes it straightforward to integrate learning and planning processes simply by allowing both to update the same estimated value function. In addition, any of the learning methods can be converted into planning methods simply by applying them to simulated (model-generated) experience rather than to real experience. In this case learning and planning become even more similar; they are possibly identical algorithms operating on two different sources of experience.

It is straightforward to integrate incremental planning methods with acting and model-learning. Planning, acting, and model-learning interact in a circular fashion (as in the diagram on page 162), each producing what the other needs to improve; no other interaction among them is either required or prohibited. The most natural approach is for all processes to proceed asynchronously and in parallel. If the processes must share computational resources, then the division can be handled almost arbitrarily—by whatever organization is most convenient and efficient for the task at hand.

In this chapter we have touched upon a number of dimensions of variation among state-space planning methods. One dimension is the variation in the size of updates. The

smaller the updates, the more incremental the planning methods can be. Among the smallest updates are one-step sample updates, as in Dyna. Another important dimension is the distribution of updates, that is, of the focus of search. Prioritized sweeping focuses backward on the predecessors of states whose values have recently changed. On-policy trajectory sampling focuses on states or state–action pairs that the agent is likely to encounter when controlling its environment. This can allow computation to skip over parts of the state space that are irrelevant to the prediction or control problem. Real-time dynamic programming, an on-policy trajectory sampling version of value iteration, illustrates some of the advantages this strategy has over conventional sweep-based policy iteration.

Planning can also focus forward from pertinent states, such as states actually encountered during an agent-environment interaction. The most important form of this is when planning is done at decision time, that is, as part of the action-selection process. Classical heuristic search as studied in artificial intelligence is an example of this. Other examples are rollout algorithms and Monte Carlo Tree Search that benefit from online, incremental, sample-based value estimation and policy improvement.

8.13 Summary of Part I: Dimensions

This chapter concludes Part I of this book. In it we have tried to present reinforcement learning not as a collection of individual methods, but as a coherent set of ideas cutting across methods. Each idea can be viewed as a dimension along which methods vary. The set of such dimensions spans a large space of possible methods. By exploring this space at the level of dimensions we hope to obtain the broadest and most lasting understanding. In this section we use the concept of dimensions in method space to recapitulate the view of reinforcement learning developed so far in this book.

All of the methods we have explored so far in this book have three key ideas in common: first, they all seek to estimate value functions; second, they all operate by backing up values along actual or possible state trajectories; and third, they all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other. These three ideas are central to the subjects covered in this book. We suggest that value functions, backing up value updates, and GPI are powerful organizing principles potentially relevant to any model of intelligence, whether artificial or natural.

Two of the most important dimensions along which the methods vary are shown in Figure 8.11. These dimensions have to do with the kind of update used to improve the value function. The horizontal dimension is whether they are sample updates (based on a sample trajectory) or expected updates (based on a distribution of possible trajectories). Expected updates require a distribution model, whereas sample updates need only a sample model, or can be done from actual experience with no model at all (another dimension of variation). The vertical dimension of Figure 8.11 corresponds to the depth of updates, that is, to the degree of bootstrapping. At three of the four corners of the space are the three primary methods for estimating values: dynamic programming, TD, and Monte Carlo. Along the left edge of the space are the sample-update methods,

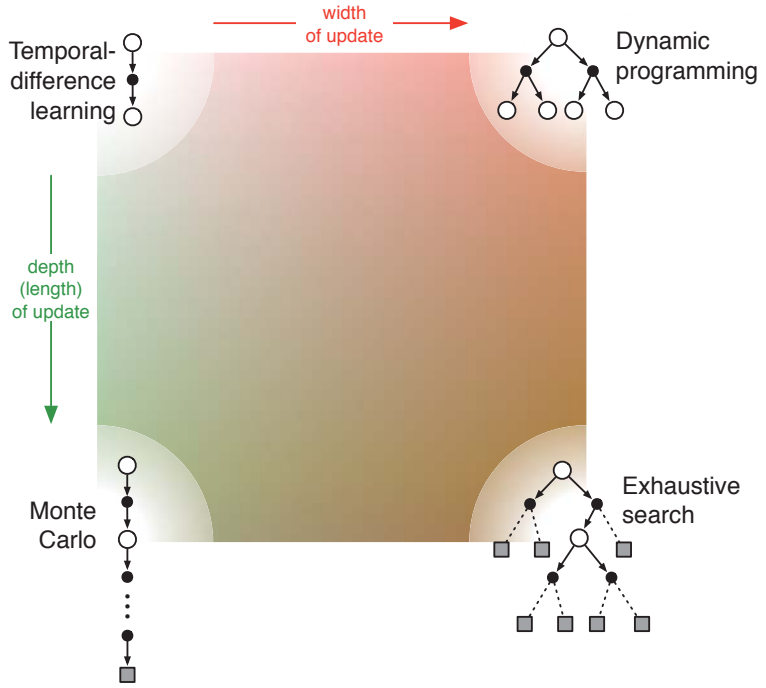


Figure 8.11: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

ranging from one-step TD updates to full-return Monte Carlo updates. Between these is a spectrum including methods based on n-step updates (and in Chapter 12 we will extend this to mixtures of n-step updates such as the λ-updates implemented by eligibility traces).

Dynamic programming methods are shown in the extreme upper-right corner of the space because they involve one-step expected updates. The lower-right corner is the extreme case of expected updates so deep that they run all the way to terminal states (or, in a continuing task, until discounting has reduced the contribution of any further rewards to a negligible level). This is the case of exhaustive search. Intermediate methods along this dimension include heuristic search and related methods that search and update up to a limited depth, perhaps selectively. There are also methods that are intermediate along the horizontal dimension. These include methods that mix expected and sample updates, as well as the possibility of methods that mix samples and distributions within a single update. The interior of the square is filled in to represent the space of all such intermediate methods.

A third dimension that we have emphasized in this book is the binary distinction between on-policy and off-policy methods. In the former case, the agent learns the value function for the policy it is currently following, whereas in the latter case it learns the

value function for the policy for a different policy, often the one that the agent currently thinks is best. The policy generating behavior is typically different from what is currently thought best because of the need to explore. This third dimension might be visualized as perpendicular to the plane of the page in Figure 8.11.

In addition to the three dimensions just discussed, we have identified a number of others throughout the book:

Definition of return Is the task episodic or continuing, discounted or undiscounted?

Action values vs. state values vs. afterstate values What kind of values should be estimated? If only state values are estimated, then either a model or a separate policy (as in actor–critic methods) is required for action selection.

Action selection/exploration How are actions selected to ensure a suitable trade-off between exploration and exploitation? We have considered only the simplest ways to do this: ε-greedy, optimistic initialization of values, soft-max, and upper confidence bound.

Synchronous vs. asynchronous Are the updates for all states performed simultaneously or one by one in some order?

Real vs. simulated Should one update based on real experience or simulated experience? If both, how much of each?

Location of updates What states or state–action pairs should be updated? Model-free methods can choose only among the states and state–action pairs actually encountered, but model-based methods can choose arbitrarily. There are many possibilities here.

Timing of updates Should updates be done as part of selecting actions, or only afterward?

Memory for updates How long should updated values be retained? Should they be retained permanently, or only while computing an action selection, as in heuristic search?

Of course, these dimensions are neither exhaustive nor mutually exclusive. Individual algorithms differ in many other ways as well, and many algorithms lie in several places along several dimensions. For example, Dyna methods use both real and simulated experience to affect the same value function. It is also perfectly sensible to maintain multiple value functions computed in different ways or over different state and action representations. These dimensions do, however, constitute a coherent set of ideas for describing and exploring a wide space of possible methods.

The most important dimension not mentioned here, and not covered in Part I of this book, is that of function approximation. Function approximation can be viewed as an orthogonal spectrum of possibilities ranging from tabular methods at one extreme through state aggregation, a variety of linear methods, and then a diverse set of nonlinear methods. This dimension is explored in Part II.

Bibliographical and Historical Remarks

8.1 The overall view of planning and learning presented here has developed gradually over a number of years, in part by the authors (Sutton, 1990, 1991a, 1991b; Barto, Bradtke, and Singh, 1991, 1995; Sutton and Pinette, 1985; Sutton and Barto, 1981b); it has been strongly influenced by Agre and Chapman (1990; Agre 1988), Bertsekas and Tsitsiklis (1989), Singh (1993), and others. The authors were also strongly influenced by psychological studies of latent learning (Tolman, 1932) and by psychological views of the nature of thought (e.g., Galanter and Gerstenhaber, 1956; Craik, 1943; Campbell, 1960; Dennett, 1978). In Part III of the book, Section 14.6 relates model-based and model-free methods to psychological theories of learning and behavior, and Section 15.11 discusses ideas about how the brain might implement these types of methods.

8.2 The terms *direct* and *indirect*, which we use to describe different kinds of reinforcement learning, are from the adaptive control literature (e.g., Goodwin and Sin, 1984), where they are used to make the same kind of distinction. The term *system identification* is used in adaptive control for what we call *model-learning* (e.g., Goodwin and Sin, 1984; Ljung and Söderstrom, 1983; Young, 1984). The Dyna architecture is due to Sutton (1990), and the results in this and the next section are based on results reported there. Barto and Singh (1990) consider some of the issues in comparing direct and indirect reinforcement learning methods. Early work extending Dyna to linear function approximation (Chapter 9) was done by Sutton, Szepesvári, Geramifard, and Bowling (2008) and by Parr, Li, Taylor, Painter-Wakefield, and Littman (2008).

8.3 There have been several works with model-based reinforcement learning that take the idea of exploration bonuses and optimistic initialization to its logical extreme, in which all incompletely explored choices are assumed maximally rewarding and optimal paths are computed to test them. The E^3 algorithm of Kearns and Singh (2002) and the R-max algorithm of Brafman and Tennenholtz (2003) are guaranteed to find a near-optimal solution in time polynomial in the number of states and actions. This is usually too slow for practical algorithms but is probably the best that can be done in the worst case.

8.4 Prioritized sweeping was developed simultaneously and independently by Moore and Atkeson (1993) and Peng and Williams (1993). The results in the box on page 170 are due to Peng and Williams (1993). The results in the box on page 171 are due to Moore and Atkeson. Key subsequent work in this area includes that by McMahan and Gordon (2005) and by van Seijen and Sutton (2013).

8.5 This section was strongly influenced by the experiments of Singh (1993).

8.6–7 Trajectory sampling has implicitly been a part of reinforcement learning from the outset, but it was most explicitly emphasized by Barto, Bradtke, and Singh (1995) in their introduction of RTDP. They recognized that Korf's (1990) *learning*

*real-time A** (LRTA*) algorithm is an asynchronous DP algorithm that applies to stochastic problems as well as the deterministic problems on which Korf focused. Beyond LRTA*, RTDP includes the option of updating the values of many states in the time intervals between the execution of actions. Barto et al. (1995) proved the convergence result described here by combining Korf's (1990) convergence proof for LRTA* with the result of Bertsekas (1982) (also Bertsekas and Tsitsiklis, 1989) ensuring convergence of asynchronous DP for stochastic shortest path problems in the undiscounted case. Combining model-learning with RTDP is called *Adaptive* RTDP, also presented by Barto et al. (1995) and discussed by Barto (2011).

8.9 For further reading on heuristic search, the reader is encouraged to consult texts and surveys such as those by Russell and Norvig (2009) and Korf (1988). Peng and Williams (1993) explored a forward focusing of updates much as is suggested in this section.

8.10 Abramson's (1990) expected-outcome model is a rollout algorithm applied to two-person games in which the play of both simulated players is random. He argued that even with random play, it is a "powerful heuristic" that is "precise, accurate, easily estimable, efficiently calculable, and domain-independent." Tesauro and Galperin (1997) demonstrated the effectiveness of rollout algorithms for improving the play of backgammon programs, adopting the term "rollout" from its use in evaluating backgammon positions by playing out positions with different randomly generating sequences of dice rolls. Bertsekas, Tsitsiklis, and Wu (1997) examine rollout algorithms applied to combinatorial optimization problems, and Bertsekas (2013) surveys their use in discrete deterministic optimization problems, remarking that they are "often surprisingly effective."

8.11 The central ideas of MCTS were introduced by Coulom (2006) and by Kocsis and Szepesvári (2006). They built upon previous research with Monte Carlo planning algorithms as reviewed by these authors. Browne, Powley, Whitehouse, Lucas, Cowling, Rohlfshagen, Tavener, Perez, Samothrakis, and Colton (2012) is an excellent survey of MCTS methods and their applications. David Silver contributed to the ideas and presentation in this section.

Part II: Approximate Solution Methods

In the second part of the book we extend the tabular methods presented in the first part to apply to problems with arbitrarily large state spaces. In many of the tasks to which we would like to apply reinforcement learning the state space is combinatorial and enormous; the number of possible camera images, for example, is much larger than the number of atoms in the universe. In such cases we cannot expect to find an optimal policy or the optimal value function even in the limit of infinite time and data; our goal instead is to find a good approximate solution using limited computational resources. In this part of the book we explore such approximate solution methods.

The problem with large state spaces is not just the memory needed for large tables, but the time and data needed to fill them accurately. In many of our target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one. In other words, the key issue is that of *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

Fortunately, generalization from examples has already been extensively studied, and we do not need to invent totally new methods for use in reinforcement learning. To some extent we need only combine reinforcement learning methods with existing generalization methods. The kind of generalization we require is often called *function approximation* because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function. Function approximation is an instance of *supervised learning*, the primary topic studied in machine learning, artificial neural networks, pattern recognition, and statistical curve fitting. In theory, any of the methods studied in these fields can be used in the role of function approximator within reinforcement learning algorithms, although in practice some fit more easily into this role than others.

Reinforcement learning with function approximation involves a number of new issues that do not normally arise in conventional supervised learning, such as nonstationarity, bootstrapping, and delayed targets. We introduce these and other issues successively over the five chapters of this part. Initially we restrict attention to on-policy training, treating in Chapter 9 the prediction case, in which the policy is given and only its value function is approximated, and then in Chapter 10 the control case, in which an approximation to the optimal policy is found. The challenging problem of off-policy learning with function approximation is treated in Chapter 11. In each of these three chapters we will have to return to first principles and re-examine the objectives of the learning to take into account function approximation. Chapter 12 introduces and analyzes the algorithmic mechanism of *eligibility traces*, which dramatically improves the computational properties of multi-step reinforcement learning methods in many cases. The final chapter of this part explores a different approach to control, *policy-gradient methods*, which approximate the optimal policy directly and need never form an approximate value function (although they may be much more efficient if they do approximate a value function as well the policy).

Chapter 9

On-policy Prediction with Approximation

In this chapter, we begin our study of function approximation in reinforcement learning by considering its use in estimating the state-value function from on-policy data, that is, in approximating v_π from experience generated using a known policy π. The novelty in this chapter is that the approximate value function is represented not as a table but as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$. We will write $\hat{v}(s,\mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$. For example, $\hat{v}$ might be a linear function in features of the state, with $\mathbf{w}$ the vector of feature weights. More generally, $\hat{v}$ might be the function computed by a multi-layer artificial neural network, with $\mathbf{w}$ the vector of connection weights in all the layers. By adjusting the weights, any of a wide range of different functions can be implemented by the network. Or $\hat{v}$ might be the function computed by a decision tree, where $\mathbf{w}$ is all the numbers defining the split points and leaf values of the tree. Typically, the number of weights (the dimensionality of $\mathbf{w}$) is much less than the number of states ($d \ll |\mathcal{S}|$), and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such *generalization* makes the learning potentially more powerful but also potentially more difficult to manage and understand.

Perhaps surprisingly, extending reinforcement learning to function approximation also makes it applicable to partially observable problems, in which the full state is not available to the agent. If the parameterized function form for $\hat{v}$ does not allow the estimated value to depend on certain aspects of the state, then it is just as if those aspects are unobservable. In fact, all the theoretical results for methods using function approximation presented in this part of the book apply equally well to cases of partial observability. What function approximation can't do, however, is augment the state representation with memories of past observations. Some such possible further extensions are discussed briefly in Section 17.3.

9.1 Value-function Approximation

All of the prediction methods covered in this book have been described as updates to an estimated value function that shift its value at particular states toward a "backed-up value," or *update target*, for that state. Let us refer to an individual update by the notation $s \mapsto u$, where s is the state updated and u is the update target that s's estimated value is shifted toward. For example, the Monte Carlo update for value prediction is $S_t \mapsto G_t$, the TD(0) update is $S_t \mapsto R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t)$, and the n-step TD update is $S_t \mapsto G_{t:t+n}$. In the DP (dynamic programming) policy-evaluation update, $s \mapsto \mathbb{E}_\pi[R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) \mid S_t = s]$, an arbitrary state s is updated, whereas in the other cases the state encountered in actual experience, S_t, is updated.

It is natural to interpret each update as specifying an example of the desired input–output behavior of the value function. In a sense, the update $s \mapsto u$ means that the estimated value for state s should be more like the update target u. Up to now, the actual update has been trivial: the table entry for s's estimated value has simply been shifted a fraction of the way toward u, and the estimated values of all other states were left unchanged. Now we permit arbitrarily complex and sophisticated methods to implement the update, and updating at s generalizes so that the estimated values of many other states are changed as well. Machine learning methods that learn to mimic input–output examples in this way are called *supervised learning* methods, and when the outputs are numbers, like u, the process is often called *function approximation*. Function approximation methods expect to receive examples of the desired input–output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \mapsto g$ of each update as a training example. We then interpret the approximate function they produce as an estimated value function.

Viewing each update as a conventional training example in this way enables us to use any of a wide range of existing function approximation methods for value prediction. In principle, we can use any method for supervised learning from examples, including artificial neural networks, decision trees, and various kinds of multivariate regression. However, not all function approximation methods are equally well suited for use in reinforcement learning. The most sophisticated artificial neural network and statistical methods all assume a static training set over which multiple passes are made. In reinforcement learning, however, it is important that learning be able to occur online, while the agent interacts with its environment or with a model of its environment. To do this requires methods that are able to learn efficiently from incrementally acquired data. In addition, reinforcement learning generally requires function approximation methods able to handle nonstationary target functions (target functions that change over time). For example, in control methods based on GPI (generalized policy iteration) we often seek to learn q_π while π changes. Even if the policy remains the same, the target values of training examples are nonstationary if they are generated by bootstrapping methods (DP and TD learning). Methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning.

9.2 The Prediction Objective ($\overline{\text{VE}}$)

Up to now we have not specified an explicit objective for prediction. In the tabular case a continuous measure of prediction quality was not necessary because the learned value function could come to equal the true value function exactly. Moreover, the learned values at each state were decoupled—an update at one state affected no other. But with genuine approximation, an update at one state affects many others, and it is not possible to get the values of all states exactly correct. By assumption we have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate. We are obligated then to say which states we care most about. We must specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s. By the error in a state s we mean the square of the difference between the approximate value $\hat{v}(s,\mathbf{w})$ and the true value $v_\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, the *Mean Squared Value Error*, denoted $\overline{\text{VE}}$:

$$\overline{\text{VE}}(\mathbf{w}) \doteq \sum_{s\in\mathcal{S}} \mu(s)\Big[v_\pi(s) - \hat{v}(s,\mathbf{w})\Big]^2. \tag{9.1}$$

The square root of this measure, the root $\overline{\text{VE}}$, gives a rough measure of how much the approximate values differ from the true values and is often used in plots. Often $\mu(s)$ is chosen to be the fraction of time spent in s. Under on-policy training this is called the *on-policy distribution*; we focus entirely on this case in this chapter. In continuing tasks, the on-policy distribution is the stationary distribution under π.

The on-policy distribution in episodic tasks

In an episodic task, the on-policy distribution is a little different in that it depends on how the initial states of episodes are chosen. Let $h(s)$ denote the probability that an episode begins in each state s, and let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode. Time is spent in a state s if episodes start in s, or if transitions are made into s from a preceding state $\bar{s}$ in which time is spent:

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s\,|\,\bar{s}, a), \quad \text{for all } s \in \mathcal{S}. \tag{9.2}$$

This system of equations can be solved for the expected number of visits $\eta(s)$. The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \text{for all } s \in \mathcal{S}. \tag{9.3}$$

This is the natural choice without discounting. If there is discounting ($\gamma < 1$) it should be treated as a form of termination, which can be done simply by including a factor of γ in the second term of (9.2).

The two cases, continuing and episodic, behave similarly, but with approximation they must be treated separately in formal analyses, as we will see repeatedly in this part of the book. This completes the specification of the learning objective.

But it is not completely clear that the $\overline{\text{VE}}$ is the right performance objective for reinforcement learning. Remember that our ultimate purpose—the reason we are learning a value function—is to find a better policy. The best value function for this purpose is not necessarily the best for minimizing $\overline{\text{VE}}$. Nevertheless, it is not yet clear what a more useful alternative goal for value prediction might be. For now, we will focus on $\overline{\text{VE}}$.

An ideal goal in terms of $\overline{\text{VE}}$ would be to find a *global optimum*, a weight vector $\mathbf{w}^*$ for which $\overline{\text{VE}}(\mathbf{w}^*) \leq \overline{\text{VE}}(\mathbf{w})$ for all possible $\mathbf{w}$. Reaching this goal is sometimes possible for simple function approximators such as linear ones, but is rarely possible for complex function approximators such as artificial neural networks and decision trees. Short of this, complex function approximators may seek to converge instead to a *local optimum*, a weight vector $\mathbf{w}^*$ for which $\overline{\text{VE}}(\mathbf{w}^*) \leq \overline{\text{VE}}(\mathbf{w})$ for all $\mathbf{w}$ in some neighborhood of $\mathbf{w}^*$. Although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear function approximators, and often it is enough. Still, for many cases of interest in reinforcement learning there is no guarantee of convergence to an optimum, or even to within a bounded distance of an optimum. Some methods may in fact diverge, with their $\overline{\text{VE}}$ approaching infinity in the limit.

In the last two sections we outlined a framework for combining a wide range of reinforcement learning methods for value prediction with a wide range of function approximation methods, using the updates of the former to generate training examples for the latter. We also described a $\overline{\text{VE}}$ performance measure which these methods may aspire to minimize. The range of possible function approximation methods is far too large to cover all, and anyway too little is known about most of them to make a reliable evaluation or recommendation. Of necessity, we consider only a few possibilities. In the rest of this chapter we focus on function approximation methods based on gradient principles, and on linear gradient-descent methods in particular. We focus on these methods in part because we consider them to be particularly promising and because they reveal key theoretical issues, but also because they are simple and our space is limited.

9.3 Stochastic-gradient and Semi-gradient Methods

We now develop in detail one class of learning methods for function approximation in value prediction, those based on stochastic gradient descent (SGD). SGD methods are among the most widely used of all function approximation methods and are particularly well suited to online reinforcement learning.

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w} \doteq (w_1, w_2, \ldots, w_d)^\top$,[1] and the approximate value function $\hat{v}(s,\mathbf{w})$ is a differentiable function of $\mathbf{w}$ for all $s \in \mathcal{S}$. We will be updating $\mathbf{w}$ at each of a series of discrete time steps, $t = 0, 1, 2, 3, \ldots$, so we will need a notation $\mathbf{w}_t$ for the

[1] The $^\top$ denotes transpose, needed here to turn the horizontal row vector in the text into a vertical column vector; in this book vectors are generally taken to be column vectors unless explicitly written out horizontally or transposed.

weight vector at each step. For now, let us assume that, on each step, we observe a new example $S_t \mapsto v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy. These states might be successive states from an interaction with the environment, but for now we do not assume so. Even though we are given the exact, correct values, $v_\pi(S_t)$ for each S_t, there is still a difficult problem because our function approximator has limited resources and thus limited resolution. In particular, there is generally no $\mathbf{w}$ that gets all the states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in examples.

We assume that states appear in examples with the same distribution, μ, over which we are trying to minimize the $\overline{\text{VE}}$ as given by (9.1). A good strategy in this case is to try to minimize error on the observed examples. *Stochastic gradient-descent* (SGD) methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]^2 && (9.4)\\ &= \mathbf{w}_t + \alpha\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t), && (9.5)\end{aligned}$$

where α is a positive step-size parameter, and $\nabla f(\mathbf{w})$, for any scalar expression $f(\mathbf{w})$ that is a function of a vector (here $\mathbf{w}$), denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right)^\top. \tag{9.6}$$

This derivative vector is the *gradient* of f with respect to $\mathbf{w}$. SGD methods are "gradient descent" methods because the overall step in $\mathbf{w}_t$ is proportional to the negative gradient of the example's squared error (9.4). This is the direction in which the error falls most rapidly. Gradient descent methods are called "stochastic" when the update is done, as here, on only a single example, which might have been selected stochastically. Over many examples, making small steps, the overall effect is to minimize an average performance measure such as the $\overline{\text{VE}}$.

It may not be immediately apparent why SGD takes only a small step in the direction of the gradient. Could we not move all the way in this direction and completely eliminate the error on the example? In many cases this could be done, but usually it is not desirable. Remember that we do not seek or expect to find a value function that has zero error for all states, but only an approximation that balances the errors in different states. If we completely corrected each example in one step, then we would not find such a balance. In fact, the convergence results for SGD methods assume that α decreases over time. If it decreases in such a way as to satisfy the standard stochastic approximation conditions (2.7), then the SGD method (9.5) is guaranteed to converge to a local optimum.

We turn now to the case in which the target output, here denoted $U_t \in \mathbb{R}$, of the tth training example, $S_t \mapsto U_t$, is not the true value, $v_\pi(S_t)$, but some, possibly random, approximation to it. For example, U_t might be a noise-corrupted version of $v_\pi(S_t)$, or it might be one of the bootstrapping targets using $\hat{v}$ mentioned in the previous section. In

these cases we cannot perform the exact update (9.5) because $v_\pi(S_t)$ is unknown, but we can approximate it by substituting U_t in place of $v_\pi(S_t)$. This yields the following general SGD method for state-value prediction:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t). \tag{9.7}$$

If U_t is an *unbiased* estimate, that is, if $\mathbb{E}[U_t|S_t{=}s] = v_\pi(S_t)$, for each t, then $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (2.7) for decreasing α.

For example, suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy π. Because the true value of a state is the expected value of the return following it, the Monte Carlo target $U_t \doteq G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$. With this choice, the general SGD method (9.7) converges to a locally optimal approximation to $v_\pi(S_t)$. Thus, the gradient-descent version of Monte Carlo state-value prediction is guaranteed to find a locally optimal solution. Pseudocode for a complete algorithm is shown in the box below.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t in (9.7). Bootstrapping targets such as n-step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r}\pi(a|S_t)p(s',r|S_t,a)[r+\gamma\hat{v}(s',\mathbf{w}_t)]$ all depend on the current value of the weight vector $\mathbf{w}_t$, which implies that they will be biased and that they will not produce a true gradient-descent method. One way to look at this is that the key step from (9.4) to (9.5) relies on the target being independent of $\mathbf{w}_t$. This step would not be valid if a bootstrapping estimate were used in place of $v_\pi(S_t)$. Bootstrapping methods are not in fact instances of true gradient descent (Barnard, 1993). They take into account the effect of changing the weight vector $\mathbf{w}_t$ on the estimate, but ignore its effect on the target. They include only a part of the gradient and, accordingly, we call them *semi-gradient methods*.

Although semi-gradient (bootstrapping) methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the linear case discussed in the next section. Moreover, they offer important advantages that make them often clearly preferred. One reason for this is that they typically enable significantly faster learning, as we have seen in Chapters 6 and 7. Another is that they enable learning to

be continual and online, without waiting for the end of an episode. This enables them to be used on continuing problems and provides computational advantages. A prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w})$ as its target. Complete pseudocode for this method is given in the box below.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

State aggregation is a simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector $\mathbf{w}$) for each group. The value of a state is estimated as its group's component, and when the state is updated, that component alone is updated. State aggregation is a special case of SGD (9.7) in which the gradient, $\nabla\hat{v}(S_t,\mathbf{w}_t)$, is 1 for S_t's group's component and 0 for the other components.

Example 9.1: State Aggregation on the 1000-state Random Walk Consider a 1000-state version of the random walk task (Examples 6.2 and 7.1 on pages 125 and 144). The states are numbered from 1 to 1000, left to right, and all episodes begin near the center, in state 500. State transitions are from the current state to one of the 100 neighboring states to its left, or to one of the 100 neighboring states to its right, all with equal probability. Of course, if the current state is near an edge, then there may be fewer than 100 neighbors on that side of it. In this case, all the probability that would have gone into those missing neighbors goes into the probability of terminating on that side (thus, state 1 has a 0.5 chance of terminating on the left, and state 950 has a 0.25 chance of terminating on the right). As usual, termination on the left produces a reward of -1, and termination on the right produces a reward of $+1$. All other transitions have a reward of zero. We use this task as a running example throughout this section.

Figure 9.1 shows the true value function v_π for this task. It is nearly a straight line, but curving slightly toward the horizontal for the last 100 states at each end. Also shown is the final approximate value function learned by the gradient Monte-Carlo algorithm with state aggregation after 100,000 episodes with a step size of $\alpha = 2 \times 10^{-5}$. For the state aggregation, the 1000 states were partitioned into 10 groups of 100 states each (i.e., states 1–100 were one group, states 101–200 were another, and so on). The staircase effect

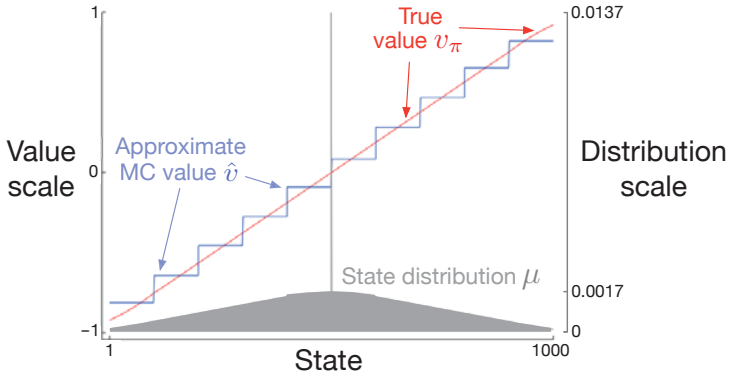


Figure 9.1: Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm (page 202).

shown in the figure is typical of state aggregation; within each group, the approximate value is constant, and it changes abruptly from one group to the next. These approximate values are close to the global minimum of the $\overline{\text{VE}}$ (9.1).

Some of the details of the approximate values are best appreciated by reference to the state distribution μ for this task, shown in the lower portion of the figure with a right-side scale. State 500, in the center, is the first state of every episode, but is rarely visited again. On average, about 1.37% of the time steps are spent in the start state. The states reachable in one step from the start state are the second most visited, with about 0.17% of the time steps being spent in each of them. From there μ falls off almost linearly, reaching about 0.0147% at the extreme states 1 and 1000. The most visible effect of the distribution is on the leftmost groups, whose values are clearly shifted higher than the unweighted average of the true values of states within the group, and on the rightmost groups, whose values are clearly shifted lower. This is due to the states in these areas having the greatest asymmetry in their weightings by μ. For example, in the leftmost group, state 100 is weighted more than 3 times more strongly than state 1. Thus the estimate for the group is biased toward the true value of state 100, which is higher than the true value of state 1. ■

9.4 Linear Methods

One of the most important special cases of function approximation is that in which the approximate function, $\hat{v}(\cdot,\mathbf{w})$, is a linear function of the weight vector, $\mathbf{w}$. Corresponding to every state s, there is a real-valued vector $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \ldots, x_d(s))^\top$, with the same number of components as $\mathbf{w}$. Linear methods approximate state-value function by

the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$:

$$\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s). \tag{9.8}$$

In this case the approximate value function is said to be *linear in the weights*, or simply *linear*.

The vector $\mathbf{x}(s)$ is called a *feature vector* representing state s. Each component $x_i(s)$ of $\mathbf{x}(s)$ is the value of a function $x_i : \mathcal{S} \to \mathbb{R}$. We think of a *feature* as the entirety of one of these functions, and we call its value for a state s a *feature of* s. For linear methods, features are *basis functions* because they form a linear basis for the set of approximate functions. Constructing d-dimensional feature vectors to represent states is the same as selecting a set of d basis functions. Features may be defined in many different ways; we cover a few possibilities in the next sections.

It is natural to use SGD updates with linear function approximation. The gradient of the approximate value function with respect to $\mathbf{w}$ in this case is

$$\nabla \hat{v}(s,\mathbf{w}) = \mathbf{x}(s).$$

Thus, in the linear case the general SGD update (9.7) reduces to a particularly simple form:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{v}(S_t,\mathbf{w}_t)\Big]\mathbf{x}(S_t).$$

Because it is so simple, the linear SGD case is one of the most favorable for mathematical analysis. Almost all useful convergence results for learning systems of all kinds are for linear (or simpler) function approximation methods.

In particular, in the linear case there is only one optimum (or, in degenerate cases, one set of equally good optima), and thus any method that is guaranteed to converge to or near a local optimum is automatically guaranteed to converge to or near the global optimum. For example, the gradient Monte Carlo algorithm presented in the previous section converges to the global optimum of the $\overline{\text{VE}}$ under linear function approximation if α is reduced over time according to the usual conditions.

The semi-gradient TD(0) algorithm presented in the previous section also converges under linear function approximation, but this does not follow from general results on SGD; a separate theorem is necessary. The weight vector converged to is also not the global optimum, but rather a point near the local optimum. It is useful to consider this important case in more detail, specifically for the continuing case. The update at each time t is

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\Big(R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t\Big)\mathbf{x}_t \\ &= \mathbf{w}_t + \alpha\Big(R_{t+1}\mathbf{x}_t - \mathbf{x}_t\big(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\big)^\top \mathbf{w}_t\Big),\end{aligned} \tag{9.9}$$

where here we have used the notational shorthand $\mathbf{x}_t = \mathbf{x}(S_t)$. Once the system has reached steady state, for any given $\mathbf{w}_t$, the expected next weight vector can be written

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t), \tag{9.10}$$

where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \quad \text{and} \quad \mathbf{A} \doteq \mathbb{E}\Big[\mathbf{x}_t\big(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\big)^\top\Big] \in \mathbb{R}^d \times \mathbb{R}^d \tag{9.11}$$

From (9.10) it is clear that, if the system converges, it must converge to the weight vector $\mathbf{w}_{\text{TD}}$ at which

$$\begin{aligned} & \mathbf{b} - \mathbf{A}\mathbf{w}_{\text{TD}} = \mathbf{0} \\ \Rightarrow \quad & \mathbf{b} = \mathbf{A}\mathbf{w}_{\text{TD}} \\ \Rightarrow \quad & \mathbf{w}_{\text{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}. \end{aligned} \tag{9.12}$$

This quantity is called the *TD fixed point*. In fact linear semi-gradient TD(0) converges to this point. Some of the theory proving its convergence, and the existence of the inverse above, is given in the box.

Proof of Convergence of Linear TD(0)

What properties assure convergence of the linear TD(0) algorithm (9.9)? Some insight can be gained by rewriting (9.10) as

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = (\mathbf{I} - \alpha\mathbf{A})\mathbf{w}_t + \alpha\mathbf{b}. \tag{9.13}$$

Note that the matrix $\mathbf{A}$ multiplies the weight vector $\mathbf{w}_t$ and not $\mathbf{b}$; only $\mathbf{A}$ is important to convergence. To develop intuition, consider the special case in which $\mathbf{A}$ is a diagonal matrix. If any of the diagonal elements are negative, then the corresponding diagonal element of $\mathbf{I} - \alpha\mathbf{A}$ will be greater than one, and the corresponding component of $\mathbf{w}_t$ will be amplified, which will lead to divergence if continued. On the other hand, if the diagonal elements of $\mathbf{A}$ are all positive, then α can be chosen smaller than one over the largest of them, such that $\mathbf{I} - \alpha\mathbf{A}$ is diagonal with all diagonal elements between 0 and 1. In this case the first term of the update tends to shrink $\mathbf{w}_t$, and stability is assured. In general, $\mathbf{w}_t$ will be reduced toward zero whenever $\mathbf{A}$ is *positive definite*, meaning $y^\top \mathbf{A} y > 0$ for any real vector $y \neq 0$. Positive definiteness also ensures that the inverse $\mathbf{A}^{-1}$ exists.

For linear TD(0), in the continuing case with $\gamma < 1$, the $\mathbf{A}$ matrix (9.11) can be written

$$\begin{aligned}\mathbf{A} &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{r,s'} p(r,s'|s,a)\mathbf{x}(s)\big(\mathbf{x}(s) - \gamma\mathbf{x}(s')\big)^\top \\ &= \sum_s \mu(s) \sum_{s'} p(s'|s)\mathbf{x}(s)\big(\mathbf{x}(s) - \gamma\mathbf{x}(s')\big)^\top \\ &= \sum_s \mu(s)\mathbf{x}(s)\Big(\mathbf{x}(s) - \gamma\sum_{s'} p(s'|s)\mathbf{x}(s')\Big)^\top \\ &= \mathbf{X}^\top \mathbf{D}(\mathbf{I} - \gamma\mathbf{P})\mathbf{X},\end{aligned}$$

where $\mu(s)$ is the stationary distribution under π, $p(s'|s)$ is the probability of

transition from s to s' under policy π, $\mathbf{P}$ is the $|\mathcal{S}| \times |\mathcal{S}|$ matrix of these probabilities, $\mathbf{D}$ is the $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with the $\mu(s)$ on its diagonal, and $\mathbf{X}$ is the $|\mathcal{S}| \times d$ matrix with $\mathbf{x}(s)$ as its rows. From here it is clear that the inner matrix $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$ is key to determining the positive definiteness of $\mathbf{A}$.

For a key matrix of this form, positive definiteness is assured if all of its columns sum to a nonnegative number. This was shown by Sutton (1988, p. 27) based on two previously established theorems. One theorem says that any matrix $\mathbf{M}$ is positive definite if and only if the symmetric matrix $\mathbf{S} = \mathbf{M} + \mathbf{M}^\top$ is positive definite (Sutton 1988, appendix). The second theorem says that any symmetric real matrix $\mathbf{S}$ is positive definite if all of its diagonal entries are positive and greater than the sum of the absolute values of the corresponding off-diagonal entries (Varga 1962, p. 23). For our key matrix, $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$, the diagonal entries are positive and the off-diagonal entries are negative, so all we have to show is that each row sum plus the corresponding column sum is positive. The row sums are all positive because $\mathbf{P}$ is a stochastic matrix and $\gamma < 1$. Thus it only remains to show that the column sums are nonnegative. Note that the row vector of the column sums of any matrix $\mathbf{M}$ can be written as $\mathbf{1}^\top\mathbf{M}$, where $\mathbf{1}$ is the column vector with all components equal to 1. Let $\boldsymbol{\mu}$ denote the $|\mathcal{S}|$-vector of the $\mu(s)$, where $\boldsymbol{\mu} = \mathbf{P}^\top\boldsymbol{\mu}$ by virtue of μ being the stationary distribution. The column sums of our key matrix, then, are:

$$\begin{aligned}
\mathbf{1}^\top\mathbf{D}(\mathbf{I} - \gamma\mathbf{P}) &= \boldsymbol{\mu}^\top(\mathbf{I} - \gamma\mathbf{P}) \\
&= \boldsymbol{\mu}^\top - \gamma\boldsymbol{\mu}^\top\mathbf{P} \\
&= \boldsymbol{\mu}^\top - \gamma\boldsymbol{\mu}^\top \qquad \text{(because } \boldsymbol{\mu} \text{ is the stationary distribution)} \\
&= (1-\gamma)\boldsymbol{\mu}^\top,
\end{aligned}$$

all components of which are positive. Thus, the key matrix and its $\mathbf{A}$ matrix are positive definite, and on-policy TD(0) is stable. (Additional conditions and a schedule for reducing α over time are needed to prove convergence with probability one.)

At the TD fixed point, it has also been proven (in the continuing case) that the $\overline{\text{VE}}$ is within a bounded expansion of the lowest possible error:

$$\overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \;\leq\; \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \tag{9.14}$$

That is, the asymptotic error of the TD method is no more than $\frac{1}{1-\gamma}$ times the smallest possible error, that attained in the limit by the Monte Carlo method. Because γ is often near one, this expansion factor can be quite large, so there is substantial potential loss in asymptotic performance with the TD method. On the other hand, recall that the TD methods are often of vastly reduced variance compared to Monte Carlo methods, and thus faster, as we saw in Chapters 6 and 7. Which method will be best depends on the nature of the approximation and problem, and on how long learning continues.

A bound analogous to (9.14) applies to other on-policy bootstrapping methods as well. For example, linear semi-gradient DP (Eq. 9.7 with $U_t \doteq \sum_a \pi(a|S_t) \sum_{s',r} p(s', r|S_t, a)[r + \gamma\hat{v}(s', \mathbf{w}_t)]$) with updates according to the on-policy distribution will also converge to the TD fixed point. One-step semi-gradient *action-value* methods, such as semi-gradient Sarsa(0) covered in the next chapter converge to an analogous fixed point and an analogous bound. For episodic tasks, there is a slightly different but related bound (see Bertsekas and Tsitsiklis, 1996). There are also a few technical conditions on the rewards, features, and decrease in the step-size parameter, which we have omitted here. The full details can be found in the original paper (Tsitsiklis and Van Roy, 1997).

Critical to the these convergence results is that states are updated according to the on-policy distribution. For other update distributions, bootstrapping methods using function approximation may actually diverge to infinity. Examples of this and a discussion of possible solution methods are given in Chapter 11.

Example 9.2: Bootstrapping on the 1000-state Random Walk State aggregation is a special case of linear function approximation, so let's return to the 1000-state random walk to illustrate some of the observations made in this chapter. The left panel of Figure 9.2 shows the final value function learned by the semi-gradient TD(0) algorithm (page 203) using the same state aggregation as in Example 9.1. We see that the near-asymptotic TD approximation is indeed farther from the true values than the Monte Carlo approximation shown in Figure 9.1.

Nevertheless, TD methods retain large potential advantages in learning rate, and generalize Monte Carlo methods, as we investigated fully with n-step TD methods in Chapter 7. The right panel of Figure 9.2 shows results with an n-step semi-gradient TD method using state aggregation on the 1000-state random walk that are strikingly similar to those we obtained earlier with tabular methods and the 19-state random walk (Figure 7.2). To obtain such quantitatively similar results we switched the state aggregation to 20 groups of 50 states each. The 20 groups were then quantitatively close

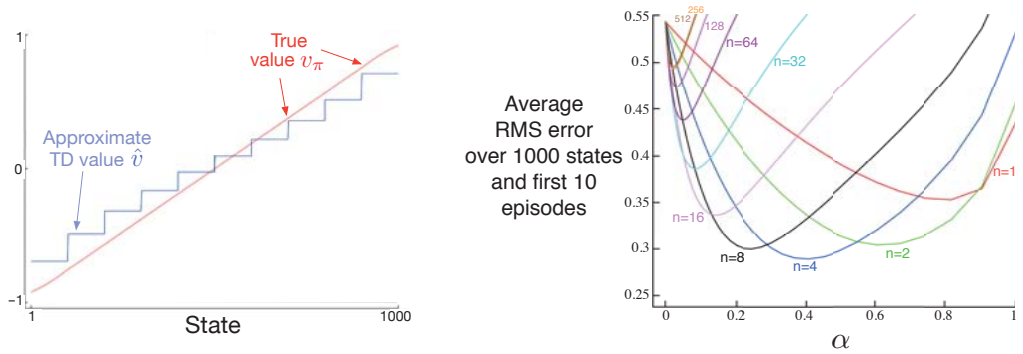


Figure 9.2: Bootstrapping with state aggregation on the 1000-state random walk task. *Left*: Asymptotic values of semi-gradient TD are worse than the asymptotic Monte Carlo values in Figure 9.1. *Right*: Performance of n-step methods with state-aggregation are strikingly similar to those with tabular representations (cf. Figure 7.2). These data are averages over 100 runs.

to the 19 states of the tabular problem. In particular, recall that state transitions were up to 100 states to the left or right. A typical transition would then be of 50 states to the right or left, which is quantitatively analogous to the single-state state transitions of the 19-state tabular system. To complete the match, we use here the same performance measure—an unweighted average of the RMS error over all states and over the first 10 episodes—rather than a $\overline{\text{VE}}$ objective as is otherwise more appropriate when using function approximation. ■

The semi-gradient n-step TD algorithm used in the example above is the natural extension of the tabular n-step TD algorithm presented in Chapter 7 to semi-gradient function approximation. Pseudocode is given in the box below.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer n
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | | Take an action according to $\pi(\cdot|S_t)$
  | | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | | If S_{t+1} is terminal, then $T \leftarrow t+1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
  | | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[G - \hat{v}(S_\tau,\mathbf{w})]\,\nabla\hat{v}(S_\tau,\mathbf{w})$
  Until $\tau = T - 1$

The key equation of this algorithm, analogous to (7.2), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\,[G_{t:t+n} - \hat{v}(S_t,\mathbf{w}_{t+n-1})]\,\nabla\hat{v}(S_t,\mathbf{w}_{t+n-1}), \qquad 0 \leq t < T, \tag{9.15}$$

where the n-step return is generalized from (7.1) to

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n},\mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T-n. \tag{9.16}$$

Exercise 9.1 Show that tabular methods such as presented in Part I of this book are a special case of linear function approximation. What would the feature vectors be? □

9.5 Feature Construction for Linear Methods

Linear methods are interesting because of their convergence guarantees, but also because in practice they can be very efficient in terms of both data and computation. Whether or not this is so depends critically on how the states are represented in terms of features, which we investigate in this large section. Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems. Intuitively, the features should correspond to the aspects of the state space along which generalization may be appropriate. If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function. If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, and so on.

A limitation of the linear form is that it cannot take into account any interactions between features, such as the presence of feature i being good only in the absence of feature j. For example, in the pole-balancing task (Example 3.4) high angular velocity can be either good or bad depending on the angle. If the angle is high, then high angular velocity means an imminent danger of falling—a bad state—whereas if the angle is low, then high angular velocity means the pole is righting itself—a good state. A linear value function could not represent this if its features coded separately for the angle and the angular velocity. It needs instead, or in addition, features for combinations of these two underlying state dimensions. In the following subsections we consider a variety of general ways of doing this.

9.5.1 Polynomials

The states of many problems are initially expressed as numbers, such as positions and velocities in the pole-balancing task (Example 3.4), the number of cars in each lot in the Jack's car rental problem (Example 4.2), or the gambler's capital in the gambler problem (Example 4.3). In these types of problems, function approximation for reinforcement learning has much in common with the familiar tasks of interpolation and regression. Various families of features commonly used for interpolation and regression can also be used in reinforcement learning. Polynomials make up one of the simplest families of features used for interpolation and regression. While the basic polynomial features we discuss here do not work as well as other types of features in reinforcement learning, they serve as a good introduction because they are simple and familiar.

As an example, suppose a reinforcement learning problem has states with two numerical dimensions. For a single representative state s, let its two numbers be $s_1 \in \mathbb{R}$ and $s_2 \in \mathbb{R}$. You might choose to represent s simply by its two state dimensions, so that $\mathbf{x}(s) = (s_1, s_2)^\top$, but then you would not be able to take into account any interactions between these dimensions. In addition, if both s_1 and s_2 were zero, then the approximate value would have to also be zero. Both limitations can be overcome by instead representing s by the four-dimensional feature vector $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2)^\top$. The initial 1 feature allows the representation of affine functions in the original state numbers, and the final product feature, $s_1 s_2$, enables interactions to be taken into account. Or you might choose to use higher-dimensional feature vectors like $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$ to

take more complex interactions into account. Such feature vectors enable approximations as arbitrary quadratic functions of the state numbers—even though the approximation is still linear in the weights that have to be learned. Generalizing this example from two to k numbers, we can represent highly-complex interactions among a problem's state dimensions:

> Suppose each state s corresponds to k numbers, s_1, s_2, ..., s_k, with each $s_i \in \mathbb{R}$. For this k-dimensional state space, each order-n polynomial-basis feature x_i can be written as
>
> $$x_i(s) = \Pi_{j=1}^{k} s_j^{c_{i,j}}, \tag{9.17}$$
>
> where each $c_{i,j}$ is an integer in the set $\{0, 1, \ldots, n\}$ for an integer $n \geq 0$. These features make up the order-n polynomial basis for dimension k, which contains $(n+1)^k$ different features.

Higher-order polynomial bases allow for more accurate approximations of more complicated functions. But because the number of features in an order-n polynomial basis grows exponentially with the dimension k of the natural state space (if $n > 0$), it is generally necessary to select a subset of them for function approximation. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods developed for polynomial regression can be adapted to deal with the incremental and nonstationary nature of reinforcement learning.

Exercise 9.2 Why does (9.17) define $(n+1)^k$ distinct features for dimension k? □

Exercise 9.3 What n and $c_{i,j}$ produce the feature vectors $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$? □

9.5.2 Fourier Basis

Another linear function approximation method is based on the time-honored Fourier series, which expresses periodic functions as weighted sums of sine and cosine basis functions (features) of different frequencies. (A function f is periodic if $f(x) = f(x+\tau)$ for all x and some period τ.) The Fourier series and the more general Fourier transform are widely used in applied sciences in part because if a function to be approximated is known, then the basis function weights are given by simple formulae and, further, with enough basis functions essentially any function can be approximated as accurately as desired. In reinforcement learning, where the functions to be approximated are unknown, Fourier basis functions are of interest because they are easy to use and can perform well in a range of reinforcement learning problems.

First consider the one-dimensional case. The usual Fourier series representation of a function of one dimension having period τ represents the function as a linear combination of sine and cosine functions that are each periodic with periods that evenly divide τ (in other words, whose frequencies are integer multiples of a fundamental frequency $1/\tau$). But if you are interested in approximating an aperiodic function defined over a bounded interval, then you can use these Fourier basis features with τ set to the length the interval.

The function of interest is then just one period of the periodic linear combination of the sine and cosine features.

Furthermore, if you set τ to twice the length of the interval of interest and restrict attention to the approximation over the half interval $[0, \tau/2]$, then you can use just the cosine features. This is possible because you can represent any *even* function, that is, any function that is symmetric about the origin, with just the cosine basis. So any function over the half-period $[0, \tau/2]$ can be approximated as closely as desired with enough cosine features. (Saying "any function" is not exactly correct because the function has to be mathematically well-behaved, but we skip this technicality here.) Alternatively, it is possible to use just sine features, linear combinations of which are always *odd* functions, that is functions that are anti-symmetric about the origin. But it is generally better to keep just the cosine features because "half-even" functions tend to be easier to approximate than "half-odd" functions because the latter are often discontinuous at the origin. Of course, this does not rule out using both sine and cosine features to approximate over the interval $[0, \tau/2]$, which might be advantageous in some circumstances.

Following this logic and letting $\tau = 2$ so that the features are defined over the half-τ interval $[0, 1]$, the one-dimensional order-n Fourier cosine basis consists of the $n+1$ features

$$x_i(s) = \cos(i\pi s), \quad s \in [0, 1],$$

for $i = 0, \ldots, n$. Figure 9.3 shows one-dimensional Fourier cosine features x_i, for $i = 1, 2, 3, 4$; x_0 is a constant function.

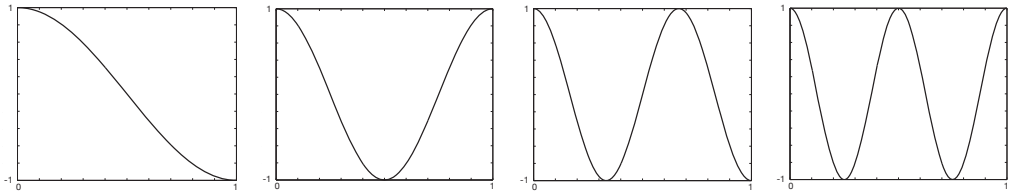


Figure 9.3: One-dimensional Fourier cosine-basis features x_i, $i = 1, 2, 3, 4$, for approximating functions over the interval $[0, 1]$. After Konidaris et al. (2011).

This same reasoning applies to the Fourier cosine series approximation in the multi-dimensional case as described in the box below.

> Suppose each state s corresponds to a vector of k numbers, $\mathbf{s} = (s_1, s_2, ..., s_k)^\top$, with each $s_i \in [0, 1]$. The ith feature in the order-n Fourier cosine basis can then be written
>
> $$x_i(s) = \cos\left(\pi \mathbf{s}^\top \mathbf{c}^i\right), \tag{9.18}$$
>
> where $\mathbf{c}^i = (c_1^i, \ldots, c_k^i)^\top$, with $c_j^i \in \{0, \ldots, n\}$ for $j = 1, \ldots, k$ and $i = 0, \ldots, (n+1)^k$. This defines a feature for each of the $(n+1)^k$ possible integer vectors $\mathbf{c}^i$. The inner

product $\mathbf{s}^\top \mathbf{c}^i$ has the effect of assigning an integer in $\{0, \ldots, n\}$ to each dimension of $\mathbf{s}$. As in the one-dimensional case, this integer determines the feature's frequency along that dimension. The features can of course be shifted and scaled to suit the bounded state space of a particular application.

As an example, consider the $k = 2$ case in which $\mathbf{s} = (s_1, s_2)^\top$, where each $\mathbf{c}^i = (c_1^i, c_2^i)^\top$. Figure 9.4 shows a selection of six Fourier cosine features, each labeled by the vector $\mathbf{c}^i$ that defines it (s_1 is the horizontal axis and $\mathbf{c}^i$ is shown as a row vector with the index i omitted). Any zero in $\mathbf{c}$ means the feature is constant along that state dimension. So if $\mathbf{c} = (0, 0)^\top$, the feature is constant over both dimensions; if $\mathbf{c} = (c_1, 0)^\top$ the feature is constant over the second dimension and varies over the first with frequency depending on c_1; and similarly, for $\mathbf{c} = (0, c_2)^\top$. When $\mathbf{c} = (c_1, c_2)^\top$ with neither $c_j = 0$, the feature varies along both dimensions and represents an interaction between the two state variables. The values of c_1 and c_2 determine the frequency along each dimension, and their ratio gives the direction of the interaction.

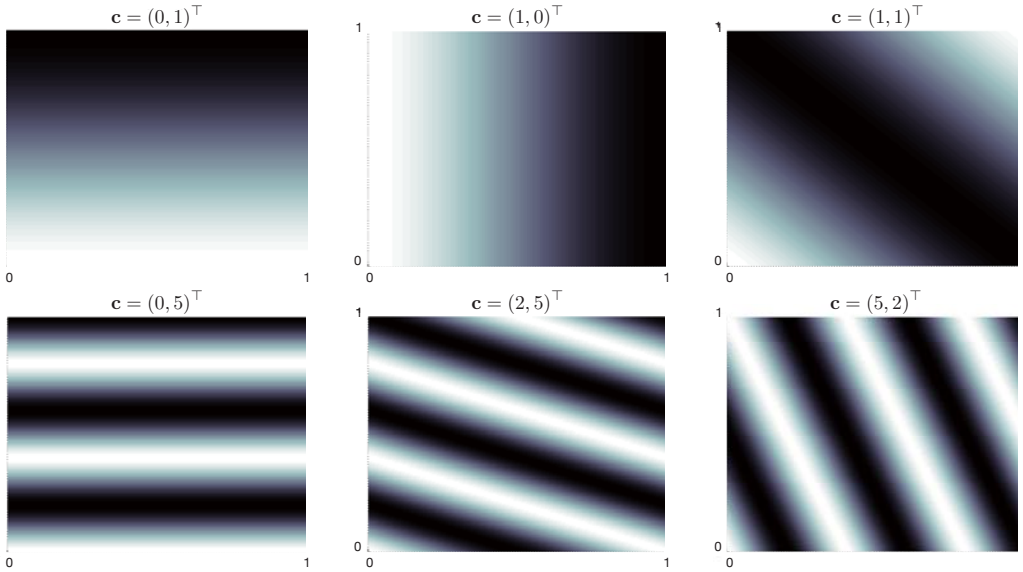


Figure 9.4: A selection of six two-dimensional Fourier cosine features, each labeled by the vector $\mathbf{c}^i$ that defines it (s_1 is the horizontal axis, and $\mathbf{c}^i$ is shown with the index i omitted). After Konidaris et al. (2011).

When using Fourier cosine features with a learning algorithm such as (9.7), semi-gradient TD(0), or semi-gradient Sarsa, it may be helpful to use a different step-size parameter for each feature. If α is the basic step-size parameter, then Konidaris, Osentoski, and Thomas (2011) suggest setting the step-size parameter for feature x_i to $\alpha_i = \alpha/\sqrt{(c_1^i)^2 + \cdots + (c_k^i)^2}$ (except when each $c_j^i = 0$, in which case $\alpha_i = \alpha$).

Fourier cosine features with Sarsa can produce good performance compared to several

other collections of basis functions, including polynomial and radial basis functions. Not surprisingly, however, Fourier features have trouble with discontinuities because it is difficult to avoid "ringing" around points of discontinuity unless very high frequency basis functions are included.

The number of features in the order-n Fourier basis grows exponentially with the dimension of the state space, but if that dimension is small enough (e.g., $k \leq 5$), then one can select n so that all of the order-n Fourier features can be used. This makes the selection of features more-or-less automatic. For high dimension state spaces, however, it is necessary to select a subset of these features. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods can be adapted to deal with the incremental and nonstationary nature of reinforcement learning. An advantage of Fourier basis features in this regard is that it is easy to select features by setting the $\mathbf{c}^i$ vectors to account for suspected interactions among the state variables and by limiting the values in the $\mathbf{c}^j$ vectors so that the approximation can filter out high frequency components considered to be noise. On the other hand, because Fourier features are non-zero over the entire state space (with the few zeros excepted), they represent global properties of states, which can make it difficult to find good ways to represent local properties.

Figure 9.5 shows learning curves comparing the Fourier and polynomial bases on the 1000-state random walk example. In general, we do not recommend using polynomials for online learning.[2]

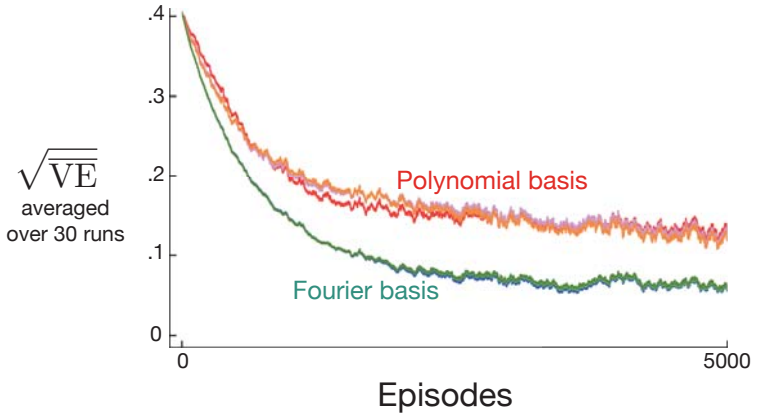


Figure 9.5: Fourier basis vs polynomials on the 1000-state random walk. Shown are learning curves for the gradient Monte Carlo method with Fourier and polynomial bases of order 5, 10, and 20. The step-size parameters were roughly optimized for each case: $\alpha = 0.0001$ for the polynomial basis and $\alpha = 0.00005$ for the Fourier basis. The performance measure (y-axis) is the root mean squared value error (9.1).

[2]There are families of polynomials more complicated than those we have discussed, for example, different families of orthogonal polynomials, and these might work better, but at present there is little experience with them in reinforcement learning.

9.5.3 Coarse Coding

Consider a task in which the natural representation of the state set is a continuous two-dimensional space. One kind of representation for this case is made up of features corresponding to *circles* in state space, as shown to the right. If the state is inside a circle, then the corresponding feature has the value 1 and is said to be *present*; otherwise the feature is 0 and is said to be *absent*. This kind of 1–0-valued feature is called a *binary feature*. Given a state, which binary features are present indicate within which circles the state lies, and thus coarsely code for its location. Representing a state with features that overlap in this way (although they need not be circles or binary) is known as *coarse coding*.

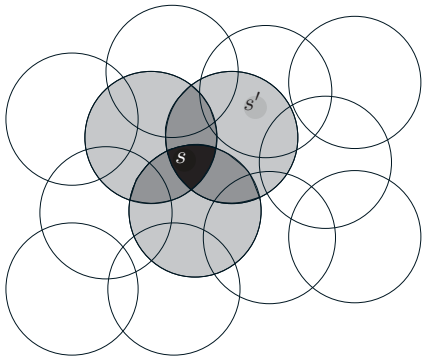


Figure 9.6: Coarse coding. Generalization from state s to state s' depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

Assuming linear gradient-descent function approximation, consider the effect of the size and density of the circles. Corresponding to each circle is a single weight (a component of $\mathbf{w}$) that is affected by learning. If we train at one state, a point in the space, then the weights of all circles intersecting that state will be affected. Thus, by (9.8), the approximate value function will be affected at all states within the union of the circles, with a greater effect the more circles a point has "in common" with the state, as shown in Figure 9.6. If the circles are small, then the generalization will be over a short distance, as in Figure 9.7 (left), whereas if they are large, it will be over a large distance, as in Figure 9.7 (middle). Moreover,

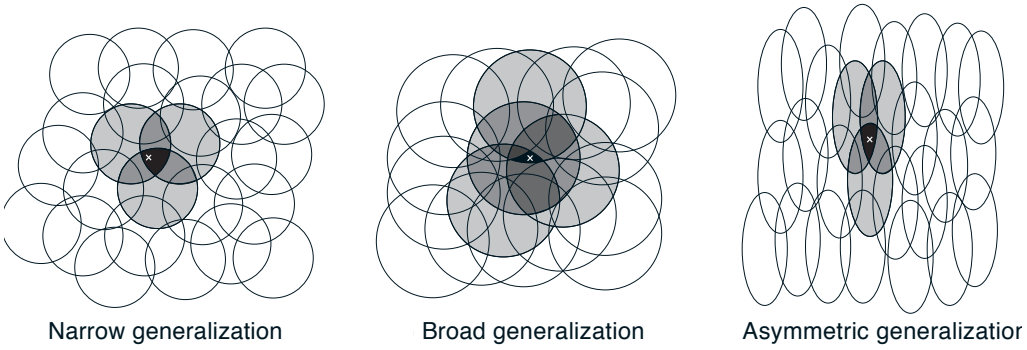


Figure 9.7: Generalization in linear function approximation methods is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.

the shape of the features will determine the nature of the generalization. For example, if they are not strictly circular, but are elongated in one direction, then generalization will be similarly affected, as in Figure 9.7 (right).

Features with large receptive fields give broad generalization, but might also seem to limit the learned function to a coarse approximation, unable to make discriminations much finer than the width of the receptive fields. Happily, this is not the case. Initial generalization from one point to another is indeed controlled by the size and shape of the receptive fields, but acuity, the finest discrimination ultimately possible, is controlled more by the total number of features.

Example 9.3: Coarseness of Coarse Coding This example illustrates the effect on learning of the size of the receptive fields in coarse coding. Linear function approximation based on coarse coding and (9.7) was used to learn a one-dimensional square-wave function (shown at the top of Figure 9.8). The values of this function were used as the targets, U_t. With just one dimension, the receptive fields were intervals rather than circles. Learning was repeated with three different sizes of the intervals: narrow, medium, and broad, as shown at the bottom of the figure. All three cases had the same density of features, about 50 over the extent of the function being learned. Training examples were generated uniformly at random over this extent. The step-size parameter was $\alpha = \frac{0.2}{n}$, where n is the number of features that were present at one time. Figure 9.8 shows the functions learned in all three cases over the course of learning. Note that the width of the features had a strong effect early in learning. With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of each trained point were changed, causing the function learned to be more bumpy. However, the final function learned was affected only slightly by the width of the features. Receptive field shape tends to have a strong effect on generalization but little effect on asymptotic solution quality.

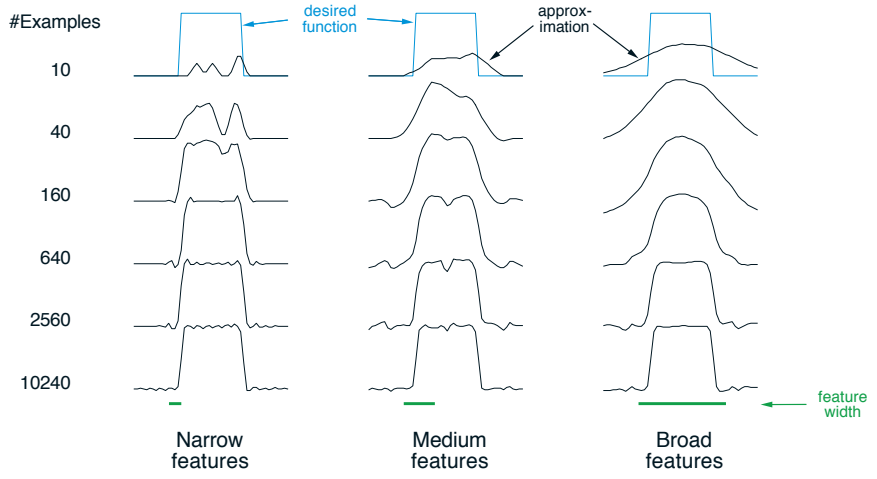


Figure 9.8: Example of feature width's strong effect on initial generalization (first row) and weak effect on asymptotic accuracy (last row). ■

9.5.4 Tile Coding

Tile coding is a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient. It may be the most practical feature representation for modern sequential digital computers.

In tile coding the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. For example, the simplest tiling of a two-dimensional state space is a uniform grid such as that shown on the left side of Figure 9.9. The tiles or receptive field here are squares rather than the circles in Figure 9.6. If just this single tiling were used, then the state indicated by the white spot would be represented by the single feature whose tile it falls within; generalization would be complete to all states within the same tile and nonexistent to states outside it. With just one tiling, we would not have coarse coding but just a case of state aggregation.

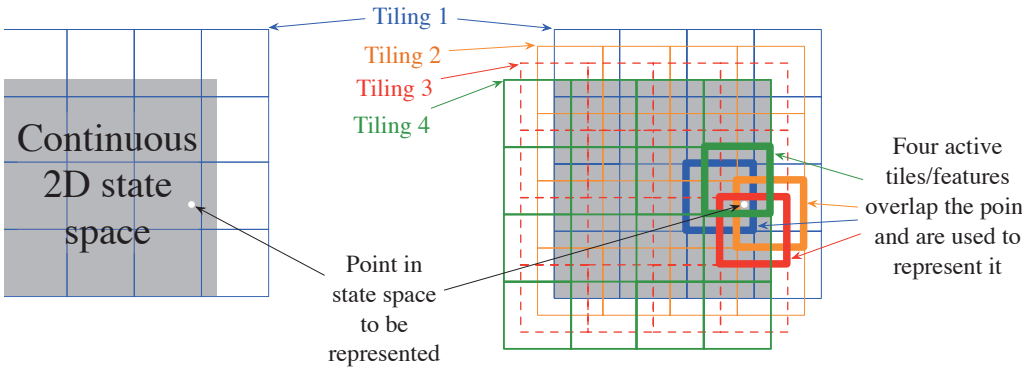


Figure 9.9: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

To get the strengths of coarse coding requires overlapping receptive fields, and by definition the tiles of a partition do not overlap. To get true coarse coding with tile coding, multiple tilings are used, each offset by a fraction of a tile width. A simple case with four tilings is shown on the right side of Figure 9.9. Every state, such as that indicated by the white spot, falls in exactly one tile in each of the four tilings. These four tiles correspond to four features that become active when the state occurs. Specifically, the feature vector $\mathbf{x}(s)$ has one component for each tile in each tiling. In this example there are $4 \times 4 \times 4 = 64$ components, all of which will be 0 except for the four corresponding to the tiles that s falls within. Figure 9.10 shows the advantage of multiple offset tilings (coarse coding) over a single tiling on the 1000-state random walk example.

An immediate practical advantage of tile coding is that, because it works with partitions, the overall number of features that are active at one time is the same for any state. Exactly one feature is present in each tiling, so the total number of features present is always the same as the number of tilings. This allows the step-size parameter, α, to be set in an easy, intuitive way. For example, choosing $\alpha = \frac{1}{n}$, where n is the number

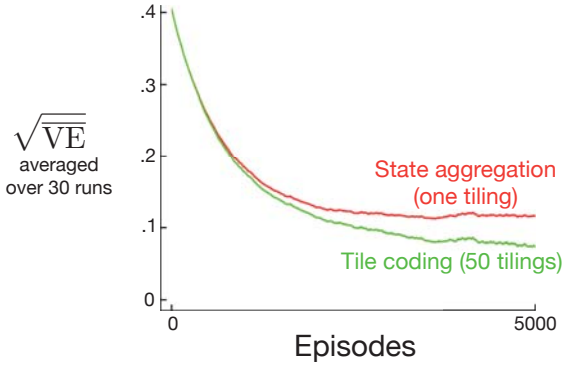


Figure 9.10: Why we use coarse coding. Shown are learning curves on the 1000-state random walk example for the gradient Monte Carlo algorithm with a single tiling and with multiple tilings. The space of 1000 states was treated as a single continuous dimension, covered with tiles each 200 states wide. The multiple tilings were offset from each other by 4 states. The step-size parameter was set so that the initial learning rate in the two cases was the same, $\alpha = 0.0001$ for the single tiling and $\alpha = 0.0001/50$ for the 50 tilings.

of tilings, results in exact one-trial learning. If the example $s \mapsto v$ is trained on, then whatever the prior estimate, $\hat{v}(s,\mathbf{w}_t)$, the new estimate will be $\hat{v}(s,\mathbf{w}_{t+1}) = v$. Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. For example, one might choose $\alpha = \frac{1}{10n}$, in which case the estimate for the trained state would move one-tenth of the way to the target in one update, and neighboring states will be moved less, proportional to the number of tiles they have in common.

Tile coding also gains computational advantages from its use of binary feature vectors. Because each component is either 0 or 1, the weighted sum making up the approximate value function (9.8) is almost trivial to compute. Rather than performing d multiplications and additions, one simply computes the indices of the $n \ll d$ active features and then adds up the n corresponding components of the weight vector.

Generalization occurs to states other than the one trained if those states fall within any of the same tiles, proportional to the number of tiles in common. Even the choice of how to offset the tilings from each other affects generalization. If they are offset uniformly in each dimension, as they were in Figure 9.9, then different states can generalize in qualitatively different ways, as shown in the upper half of Figure 9.11. Each of the eight subfigures show the pattern of generalization from a trained state to nearby points. In this example there are eight tilings, thus 64 subregions within a tile that generalize distinctly, but all according to one of these eight patterns. Note how uniform offsets result in a strong effect along the diagonal in many patterns. These artifacts can be avoided if the tilings are offset asymmetrically, as shown in the lower half of the figure. These lower generalization patterns are better because they are all well centered on the trained state with no obvious asymmetries.

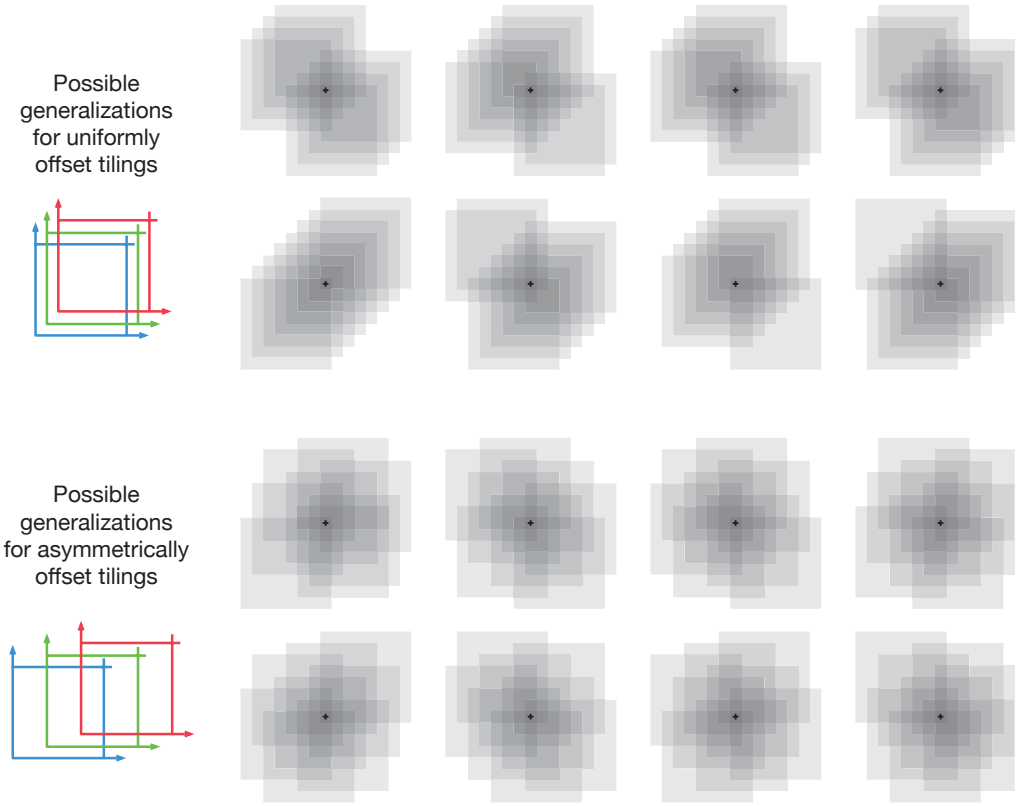


Figure 9.11: Why tile asymmetrical offsets are preferred in tile coding. Shown is the strength of generalization from a trained state, indicated by the small black plus, to nearby states, for the case of eight tilings. If the tilings are uniformly offset (above), then there are diagonal artifacts and substantial variations in the generalization, whereas with asymmetrically offset tilings the generalization is more spherical and homogeneous.

Tilings in all cases are offset from each other by a fraction of a tile width in each dimension. If w denotes the tile width and n the number of tilings, then $\frac{w}{n}$ is a fundamental unit. Within small squares $\frac{w}{n}$ on a side, all states activate the same tiles, have the same feature representation, and the same approximated value. If a state is moved by $\frac{w}{n}$ in any cartesian direction, the feature representation changes by one component/tile. Uniformly offset tilings are offset from each other by exactly this unit distance. For a two-dimensional space, we say that each tiling is offset by the displacement vector $(1, 1)$, meaning that it is offset from the previous tiling by $\frac{w}{n}$ times this vector. In these terms, the asymmetrically offset tilings shown in the lower part of Figure 9.11 are offset by a displacement vector of $(1, 3)$.

Extensive studies have been made of the effect of different displacement vectors on the generalization of tile coding (Parks and Militzer, 1991; An, 1991; An, Miller and Parks,

1991; Miller, An, Glanz and Carter, 1990), assessing their homegeneity and tendency toward diagonal artifacts like those seen for the $(1, 1)$ displacement vectors. Based on this work, Miller and Glanz (1996) recommend using displacement vectors consisting of the first odd integers. In particular, for a continuous space of dimension k, a good choice is to use the first odd integers $(1, 3, 5, 7, \ldots, 2k-1)$, with n (the number of tilings) set to an integer power of 2 greater than or equal to $4k$. This is what we have done to produce the tilings in the lower half of Figure 9.11, in which $k = 2$, $n = 2^3 \geq 4k$, and the displacement vector is $(1, 3)$. In a three-dimensional case, the first four tilings would be offset in total from a base position by $(0, 0, 0)$, $(1, 3, 5)$, $(2, 6, 10)$, and $(3, 9, 15)$. Open-source software that can efficiently make tilings like this for any k is readily available.

In choosing a tiling strategy, one has to pick the number of the tilings and the shape of the tiles. The number of tilings, along with the size of the tiles, determines the resolution or fineness of the asymptotic approximation, as in general coarse coding and illustrated in Figure 9.8. The shape of the tiles will determine the nature of generalization as in Figure 9.7. Square tiles will generalize roughly equally in each dimension as indicated in Figure 9.11 (lower). Tiles that are elongated along one dimension, such as the stripe tilings in Figure 9.12 (middle), will promote generalization along that dimension. The tilings in Figure 9.12 (middle) are also denser and thinner on the left, promoting discrimination along the horizonal dimension at lower values along that dimension. The diagonal stripe tiling in Figure 9.12 (right) will promote generalization along one diagonal. In higher dimensions, axis-aligned stripes correspond to ignoring some of the dimensions in some of the tilings, that is, to hyperplanar slices. Irregular tilings such as shown in Figure 9.12 (left) are also possible, though rare in practice and beyond the standard software.

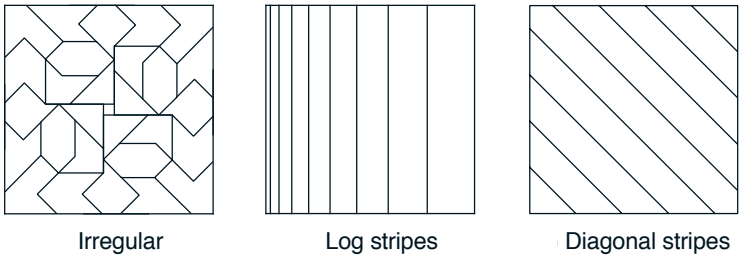
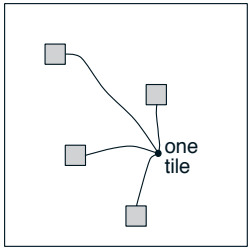


Figure 9.12: Tilings need not be grids. They can be arbitrarily shaped and non-uniform, while still in many cases being computationally efficient to compute.

In practice, it is often desirable to use different shaped tiles in different tilings. For example, one might use some vertical stripe tilings and some horizontal stripe tilings. This would encourage generalization along either dimension. However, with stripe tilings alone it is not possible to learn that a particular conjunction of horizontal and vertical coordinates has a distinctive value (whatever is learned for it will bleed into states with the same horizontal and vertical coordinates). For this one needs the conjunctive rectangular tiles such as originally shown in Figure 9.9. With multiple tilings—some horizontal, some vertical, and some conjunctive—one can get everything: a preference for generalizing along each dimension, yet the ability to learn specific values for conjunctions (see Sutton,

1996 for examples). The choice of tilings determines generalization, and until this choice can be effectively automated, it is important that tile coding enables the choice to be made flexibly and in a way that makes sense to people.

Another useful trick for reducing memory requirements is *hashing*—a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles. Hashing produces tiles consisting of noncontiguous, disjoint regions randomly spread throughout the state space, but that still form an exhaustive partition. For example, one tile might consist of the four subtiles shown to the right. Through hashing, memory requirements are often reduced by large factors with little loss of performance. This is possible because high resolution is needed in only a small fraction of the state space. Hashing frees us from the curse of dimensionality in the sense that memory requirements need not be exponential in the number of dimensions, but need merely match the real demands of the task. Open-source implementations of tile coding commonly include efficient hashing.



Exercise 9.4 Suppose we believe that one of two state dimensions is more likely to have an effect on the value function than is the other, that generalization should be primarily across this dimension rather than along it. What kind of tilings could be used to take advantage of this prior knowledge? □

9.5.5 Radial Basis Functions

Radial basis functions (RBFs) are the natural generalization of coarse coding to continuous-valued features. Rather than each feature being either 0 or 1, it can be anything in the interval $[0, 1]$, reflecting various *degrees* to which the feature is present. A typical RBF feature, x_i, has a Gaussian (bell-shaped) response $x_i(s)$ dependent only on the distance between the state, s, and the feature's prototypical or center state, c_i, and relative to the feature's width, σ_i:

$$x_i(s) \doteq \exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right).$$

The norm or distance metric of course can be chosen in whatever way seems most appropriate to the states and task at hand. The figure below shows a one-dimensional example with a Euclidean distance metric.

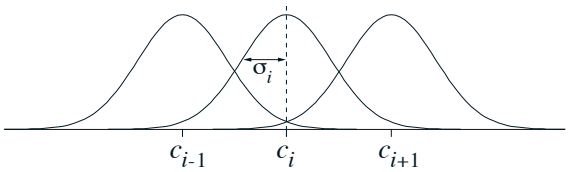


Figure 9.13: One-dimensional radial basis functions.

The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. Although this is appealing, in most cases it has no practical significance. Nevertheless, extensive studies have been made of graded response functions such as RBFs in the context of tile coding (An, 1991; Miller et al., 1991; An et al., 1991; Lane, Handelman and Gelfand, 1992). All of these methods require substantial additional computational complexity (over tile coding) and often reduce performance when there are more than two state dimensions. In high dimensions the edges of tiles are much more important, and it has proven difficult to obtain well controlled graded tile activations near the edges.

An *RBF network* is a linear function approximator using RBFs for its features. Learning is defined by equations (9.7) and (9.8), exactly as in other linear function approximators. In addition, some learning methods for RBF networks change the centers and widths of the features as well, bringing them into the realm of nonlinear function approximators. Nonlinear methods may be able to fit target functions much more precisely. The downside to RBF networks, and to nonlinear RBF networks especially, is greater computational complexity and, often, more manual tuning before learning is robust and efficient.

9.6 Selecting Step-Size Parameters Manually

Most SGD methods require the designer to select an appropriate step-size parameter α. Ideally this selection would be automated, and in some cases it has been, but for most cases it is still common practice to set it manually. To do this, and to better understand the algorithms, it is useful to develop some intuitive sense of the role of the step-size parameter. Can we say in general how it should be set?

Theoretical considerations are unfortunately of little help. The theory of stochastic approximation gives us conditions (2.7) on a slowly decreasing step-size sequence that are sufficient to guarantee convergence, but these tend to result in learning that is too slow. The classical choice $\alpha_t = 1/t$, which produces sample averages in tabular MC methods, is not appropriate for TD methods, for nonstationary problems, or for any method using function approximation. For linear methods, there are recursive least-squares methods that set an optimal *matrix* step size, and these methods can be extended to temporal-difference learning as in the LSTD method described in Section 9.8, but these require $O(d^2)$ step-size parameters, or d times more parameters than we are learning. For this reason we rule them out for use on large problems where function approximation is most needed.

To get some intuitive feel for how to set the step-size parameter manually, it is best to go back momentarily to the tabular case. There we can understand that a step size of $\alpha = 1$ will result in a complete elimination of the sample error after one target (see (2.4) with a step size of one). As discussed on page 201, we usually want to learn slower than this. In the tabular case, a step size of $\alpha = \frac{1}{10}$ would take about 10 experiences to converge approximately to their mean target, and if we wanted to learn in 100 experiences we would use $\alpha = \frac{1}{100}$. In general, if $\alpha = \frac{1}{\tau}$, then the tabular estimate for a state will approach the mean of its targets, with the most recent targets having the greatest effect, after about τ experiences with the state.

With general function approximation there is not such a clear notion of *number* of experiences with a state, as each state may be similar to and dissimilar from all the others to various degrees. However, there is a similar rule that gives similar behavior in the case of linear function approximation. Suppose you wanted to learn in about τ experiences with substantially the same feature vector. A good rule of thumb for setting the step-size parameter of linear SGD methods is then

$$\alpha \doteq \big(\tau \mathbb{E}\big[\mathbf{x}^\top \mathbf{x}\big]\big)^{-1}, \tag{9.19}$$

where $\mathbf{x}$ is a random feature vector chosen from the same distribution as input vectors will be in the SGD. This method works best if the feature vectors do not vary greatly in length; ideally $\mathbf{x}^\top \mathbf{x}$ is a constant.

Exercise 9.5 Suppose you are using tile coding to transform a seven-dimensional continuous state space into binary feature vectors to estimate a state value function $\hat{v}(s,\mathbf{w}) \approx v_\pi(s)$. You believe that the dimensions do not interact strongly, so you decide to use eight tilings of each dimension separately (stripe tilings), for $7 \times 8 = 56$ tilings. In addition, in case there are some pairwise interactions between the dimensions, you also take all $\binom{7}{2} = 21$ pairs of dimensions and tile each pair conjunctively with rectangular tiles. You make two tilings for each pair of dimensions, making a grand total of $21 \times 2 + 56 = 98$ tilings. Given these feature vectors, you suspect that you still have to average out some noise, so you decide that you want learning to be gradual, taking about 10 presentations with the same feature vector before learning nears its asymptote. What step-size parameter α should you use? Why? □

9.7 Nonlinear Function Approximation: Artificial Neural Networks

Artificial neural networks (ANNs) are widely used for nonlinear function approximation. An ANN is a network of interconnected units that have some of the properties of neurons, the main components of nervous systems. ANNs have a long history, with the latest advances in training deeply-layered ANNs (deep learning) being responsible for some of the most impressive abilities of machine learning systems, including reinforcement learning systems. In Chapter 16 we describe several impressive examples of reinforcement learning systems that use ANN function approximation.

Figure 9.14 shows a generic feedforward ANN, meaning that there are no loops in the network, that is, there are no paths within the network by which a unit's output can influence its input. The network in the figure has an output layer consisting of two output units, an input layer with four input units, and two "hidden layers": layers that are neither input nor output layers. A real-valued weight is associated with each link. A weight roughly corresponds to the efficacy of a synaptic connection in a real neural network (see Section 15.1). If an ANN has at least one loop in its connections, it is a recurrent rather than a feedforward ANN. Although both feedforward and recurrent ANNs have been used in reinforcement learning, here we look only at the simpler feedforward case.

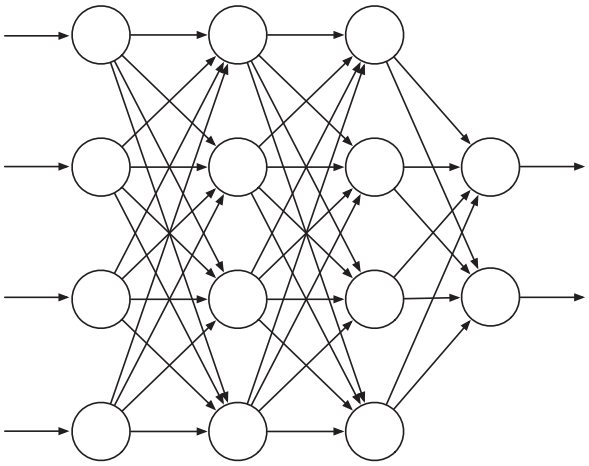


Figure 9.14: A generic feedforward ANN with four input units, two output units, and two hidden layers.

The units (the circles in Figure 9.14) are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function, called the *activation function*, to produce the unit's output, or activation. Different activation functions are used, but they are typically S-shaped, or sigmoid, functions such as the logistic function $f(x) = 1/(1 + e^{-x})$, though sometimes the rectifier nonlinearity $f(x) = \max(0, x)$ is used. A step function like $f(x) = 1$ if $x \geq \theta$, and 0 otherwise, results in a binary unit with threshold θ. The units in a network's input layer are somewhat different in having their activations set to externally-supplied values that are the inputs to the function the network is approximating.

The activation of each output unit of a feedforward ANN is a nonlinear function of the activation patterns over the network's input units. The functions are parameterized by the network's connection weights. An ANN with no hidden layers can represent only a very small fraction of the possible input-output functions. However an ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy (Cybenko, 1989). This is also true for other nonlinear activation functions that satisfy mild conditions, but nonlinearity is essential: if all the units in a multi-layer feedforward ANN have linear activation functions, the entire network is equivalent to a network with no hidden layers (because linear functions of linear functions are themselves linear).

Despite this "universal approximation" property of one-hidden-layer ANNs, both experience and theory show that approximating the complex functions needed for many artificial intelligence tasks is made easier—indeed may require—abstractions that are hierarchical compositions of many layers of lower-level abstractions, that is, abstractions produced by deep architectures such as ANNs with many hidden layers. (See Bengio, 2009, for a thorough review.) The successive layers of a deep ANN compute increasingly

abstract representations of the network's "raw" input, with each unit providing a feature contributing to a hierarchical representation of the overall input-output function of the network.

Training the hidden layers of an ANN is therefore a way to automatically create features appropriate for a given problem so that hierarchical representations can be produced without relying exclusively on hand-crafted features. This has been an enduring challenge for artificial intelligence and explains why learning algorithms for ANNs with hidden layers have received so much attention over the years. ANNs typically learn by a stochastic gradient method (Section 9.3). Each weight is adjusted in a direction aimed at improving the network's overall performance as measured by an objective function to be either minimized or maximized. In the most common supervised learning case, the objective function is the expected error, or loss, over a set of labeled training examples. In reinforcement learning, ANNs can use TD errors to learn value functions, or they can aim to maximize expected reward as in a gradient bandit (Section 2.8) or a policy-gradient algorithm (Chapter 13). In all of these cases it is necessary to estimate how a change in each connection weight would influence the network's overall performance, in other words, to estimate the partial derivative of an objective function with respect to each weight, given the current values of all the network's weights. The gradient is the vector of these partial derivatives.

The most successful way to do this for ANNs with hidden layers (provided the units have differentiable activation functions) is the backpropagation algorithm, which consists of alternating forward and backward passes through the network. Each forward pass computes the activation of each unit given the current activations of the network's input units. After each forward pass, a backward pass efficiently computes a partial derivative for each weight. (As in other stochastic gradient learning algorithms, the vector of these partial derivatives is an estimate of the true gradient.) In Section 15.10 we discuss methods for training ANNs with hidden layers that use reinforcement learning principles instead of backpropagation. These methods are less efficient than the backpropagation algorithm, but they may be closer to how real neural networks learn.

The backpropagation algorithm can produce good results for shallow networks having 1 or 2 hidden layers, but it may not work well for deeper ANNs. In fact, training a network with $k+1$ hidden layers can actually result in poorer performance than training a network with k hidden layers, even though the deeper network can represent all the functions that the shallower network can (Bengio, 2009). Explaining results like these is not easy, but several factors are important. First, the large number of weights in a typical deep ANN makes it difficult to avoid the problem of overfitting, that is, the problem of failing to generalize correctly to cases on which the network has not been trained. Second, backpropagation does not work well for deep ANNs because the partial derivatives computed by its backward passes either decay rapidly toward the input side of the network, making learning by deep layers extremely slow, or the partial derivatives grow rapidly toward the input side of the network, making learning unstable. Methods for dealing with these problems are largely responsible for many impressive recent results achieved by systems that use deep ANNs.

Overfitting is a problem for any function approximation method that adjusts functions

with many degrees of freedom on the basis of limited training data. It is less of a problem for online reinforcement learning that does not rely on limited training sets, but generalizing effectively is still an important issue. Overfitting is a problem for ANNs in general, but especially so for deep ANNs because they tend to have very large numbers of weights. Many methods have been developed for reducing overfitting. These include stopping training when performance begins to decrease on validation data different from the training data (cross validation), modifying the objective function to discourage complexity of the approximation (regularization), and introducing dependencies among the weights to reduce the number of degrees of freedom (e.g., weight sharing).

A particularly effective method for reducing overfitting by deep ANNs is the dropout method introduced by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014). During training, units are randomly removed from the network (dropped out) along with their connections. This can be thought of as training a large number of "thinned" networks. Combining the results of these thinned networks at test time is a way to improve generalization performance. The dropout method efficiently approximates this combination by multiplying each outgoing weight of a unit by the probability that that unit was retained during training. Srivastava et al. found that this method significantly improves generalization performance. It encourages individual hidden units to learn features that work well with random collections of other features. This increases the versatility of the features formed by the hidden units so that the network does not overly specialize to rarely-occurring cases.

Hinton, Osindero, and Teh (2006) took a major step toward solving the problem of training the deep layers of a deep ANN in their work with deep belief networks, layered networks closely related to the deep ANNs discussed here. In their method, the deepest layers are trained one at a time using an unsupervised learning algorithm. Without relying on the overall objective function, unsupervised learning can extract features that capture statistical regularities of the input stream. The deepest layer is trained first, then with input provided by this trained layer, the next deepest layer is trained, and so on, until the weights in all, or many, of the network's layers are set to values that now act as initial values for supervised learning. The network is then fine-tuned by backpropagation with respect to the overall objective function. Studies show that this approach generally works much better than backpropagation with weights initialized with random values. The better performance of networks trained with weights initialized this way could be due to many factors, but one idea is that this method places the network in a region of weight space from which a gradient-based algorithm can make good progress.

Batch normalization (Ioffe and Szegedy, 2015) is another technique that makes it easier to train deep ANNs. It has long been known that ANN learning is easier if the network input is normalized, for example, by adjusting each input variable to have zero mean and unit variance. Batch normalization for training deep ANNs normalizes the output of deep layers before they feed into the following layer. Ioffe and Szegedy (2015) used statistics from subsets, or "mini-batches," of training examples to normalize these between-layer signals to improve the learning rate of deep ANNs.

Another technique useful for training deep ANNs is *deep residual learning* (He, Zhang, Ren, and Sun, 2016). Sometimes it is easier to learn how a function differs from the

identity function than to learn the function itself. Then adding this difference, or residual function, to the input produces the desired function. In deep ANNs, a block of layers can be made to learn a residual function simply by adding shortcut, or skip, connections around the block. These connections add the input to the block to its output, and no additional weights are needed. He et al. (2016) evaluated this method using deep convolutional networks with skip connections around every pair of adjacent layers, finding substantial improvement over networks without the skip connections on benchmark image classification tasks. Both batch normalization and deep residual learning were used in the reinforcement learning application to the game of Go that we describe in Chapter 16.

A type of deep ANN that has proven to be very successful in applications, including impressive reinforcement learning applications (Chapter 16), is the *deep convolutional network*. This type of network is specialized for processing high-dimensional data arranged in spatial arrays, such as images. It was inspired by how early visual processing works in the brain (LeCun, Bottou, Bengio and Haffner, 1998). Because of its special architecture, a deep convolutional network can be trained by backpropagation without resorting to methods like those described above to train the deep layers.

Figure 9.15 illustrates the architecture of a deep convolutional network. This instance, from LeCun et al. (1998), was designed to recognize hand-written characters. It consists of alternating convolutional and subsampling layers, followed by several fully connected final layers. Each convolutional layer produces a number of feature maps. A feature map is a pattern of activity over an array of units, where each unit performs the same operation on data in its receptive field, which is the part of the data it "sees" from the preceding layer (or from the external input in the case of the first convolutional layer). The units of a feature map are identical to one another except that their receptive fields, which are all the same size and shape, are shifted to different locations on the arrays of incoming data. Units in the same feature map share the same weights. This means that a feature map detects the same feature no matter where it is located in the input array. In the network in Figure 9.15, for example, the first convolutional layer produces 6 feature maps, each consisting of 28×28 units. Each unit in each feature map has a 5×5 receptive field, and these receptive fields overlap (in this case by four columns and four

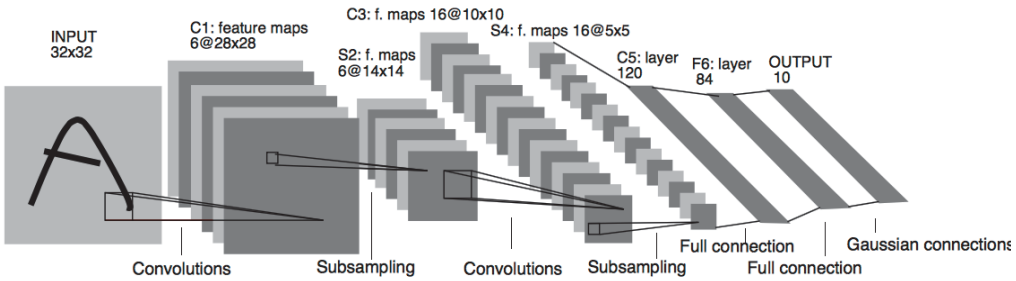


Figure 9.15: Deep Convolutional Network. Republished with permission of Proceedings of the IEEE, from Gradient-based learning applied to document recognition, LeCun, Bottou, Bengio, and Haffner, volume 86, 1998; permission conveyed through Copyright Clearance Center, Inc.

rows). Consequently, each of the 6 feature maps is specified by just 25 adjustable weights.

The subsampling layers of a deep convolutional network reduce the spatial resolution of the feature maps. Each feature map in a subsampling layer consists of units that average over a receptive field of units in the feature maps of the preceding convolutional layer. For example, each unit in each of the 6 feature maps in the first subsampling layer of the network of Figure 9.15 averages over a 2×2 non-overlapping receptive field over one of the feature maps produced by the first convolutional layer, resulting in six 14×14 feature maps. Subsampling layers reduce the network's sensitivity to the spatial locations of the features detected, that is, they help make the network's responses spatially invariant. This is useful because a feature detected at one place in an image is likely to be useful at other places as well.

Advances in the design and training of ANNs—of which we have only mentioned a few—all contribute to reinforcement learning. Although current reinforcement learning theory is mostly limited to methods using tabular or linear function approximation methods, the impressive performances of notable reinforcement learning applications owe much of their success to nonlinear function approximation by multi-layer ANNs. We discuss several of these applications in Chapter 16.

9.8 Least-Squares TD

All the methods we have discussed so far in this chapter have required computation per time step proportional to the number of parameters. With more computation, however, one can do better. In this section we present a method for linear function approximation that is arguably the best that can be done for this case.

As we established in Section 9.4 TD(0) with linear function approximation converges asymptotically (for appropriately decreasing step sizes) to the TD fixed point:

$$\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b},$$

where

$$\mathbf{A} \doteq \mathbb{E}\big[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\big] \quad \text{and} \quad \mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]\,.$$

Why, one might ask, must we compute this solution iteratively? This is wasteful of data! Could one not do better by computing estimates of $\mathbf{A}$ and $\mathbf{b}$, and then directly computing the TD fixed point? The *Least-Squares TD* algorithm, commonly known as *LSTD*, does exactly this. It forms the natural estimates

$$\widehat{\mathbf{A}}_t \doteq \sum_{k=0}^{t-1} \mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^\top + \varepsilon\mathbf{I} \quad \text{and} \quad \widehat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{t+1}\mathbf{x}_k, \tag{9.20}$$

where $\mathbf{I}$ is the identity matrix, and $\varepsilon\mathbf{I}$, for some small $\varepsilon > 0$, ensures that $\widehat{\mathbf{A}}_t$ is always invertible. It might seem that these estimates should both be divided by t, and indeed they should; as defined here, these are really estimates of t *times* $\mathbf{A}$ and t *times* $\mathbf{b}$.

However, the extra t factors cancel out when LSTD uses these estimates to estimate the TD fixed point as

$$\mathbf{w}_t \doteq \widehat{\mathbf{A}}_t^{-1}\widehat{\mathbf{b}}_t. \tag{9.21}$$

This algorithm is the most data efficient form of linear TD(0), but it is also more expensive computationally. Recall that semi-gradient TD(0) requires memory and per-step computation that is only $O(d)$.

How complex is LSTD? As it is written above the complexity seems to increase with t, but the two approximations in (9.20) could be implemented incrementally using the techniques we have covered earlier (e.g., in Chapter 2) so that they can be done in constant time per step. Even so, the update for $\widehat{\mathbf{A}}_t$ would involve an outer product (a column vector times a row vector) and thus would be a matrix update; its computational complexity would be $O(d^2)$, and of course the memory required to hold the $\widehat{\mathbf{A}}_t$ matrix would be $O(d^2)$.

A potentially greater problem is that our final computation (9.21) uses the inverse of $\widehat{\mathbf{A}}_t$, and the computational complexity of a general inverse computation is $O(d^3)$. Fortunately, an inverse of a matrix of our special form—a sum of outer products—can also be updated incrementally with only $O(d^2)$ computations, as

$$\begin{aligned}\widehat{\mathbf{A}}_t^{-1} &= \left(\widehat{\mathbf{A}}_{t-1} + \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\right)^{-1} && \text{(from (9.20))}\\ &= \widehat{\mathbf{A}}_{t-1}^{-1} - \frac{\widehat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\widehat{\mathbf{A}}_{t-1}^{-1}}{1 + (\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\widehat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_t}, && (9.22)\end{aligned}$$

for $t > 0$, with $\widehat{\mathbf{A}}_0 \doteq \varepsilon\mathbf{I}$. Although the identity (9.22), known as *the Sherman-Morrison formula*, is superficially complicated, it involves only vector-matrix and vector-vector multiplications and thus is only $O(d^2)$. Thus we can store the inverse matrix $\widehat{\mathbf{A}}_t^{-1}$, maintain it with (9.22), and then use it in (9.21), all with only $O(d^2)$ memory and per-step computation. The complete algorithm is given in the box on the next page.

Of course, $O(d^2)$ is still significantly more expensive than the $O(d)$ of semi-gradient TD. Whether the greater data efficiency of LSTD is worth this computational expense depends on how large d is, how important it is to learn quickly, and the expense of other parts of the system. The fact that LSTD requires no step-size parameter is sometimes also touted, but the advantage of this is probably overstated. LSTD does not require a step size, but it does requires ε; if ε is chosen too small the sequence of inverses can vary wildly, and if ε is chosen too large then learning is slowed. In addition, LSTD's lack of a step-size parameter means that it never forgets. This is sometimes desirable, but it is problematic if the target policy π changes as it does in reinforcement learning and GPI. In control applications, LSTD typically has to be combined with some other mechanism to induce forgeting, mooting any initial advantage of not requiring a step-size parameter.

LSTD for estimating $\hat{v} = \mathbf{w}^\top\mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal) = \mathbf{0}$
Algorithm parameter: small $\varepsilon > 0$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — A $d \times d$ matrix
$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$ — A d-dimensional vector
Loop for each episode:
  Initialize S; $\mathbf{x} \leftarrow \mathbf{x}(S)$
  Loop for each step of episode:
    Choose and take action $A \sim \pi(\cdot|S)$, observe R, S'; $\mathbf{x}' \leftarrow \mathbf{x}(S')$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top(\mathbf{x} - \gamma\mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - \big(\widehat{\mathbf{A}^{-1}}\mathbf{x}\big)\mathbf{v}^\top/\big(1 + \mathbf{v}^\top\mathbf{x}\big)$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\mathbf{w} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

9.9 Memory-based Function Approximation

So far we have discussed the *parametric* approach to approximating value functions. In this approach, a learning algorithm adjusts the parameters of a functional form intended to approximate the value function over a problem's entire state space. Each update, $s \mapsto g$, is a training example used by the learning algorithm to change the parameters with the aim of reducing the approximation error. After the update, the training example can be discarded (although it might be saved to be used again). When an approximate value of a state (which we will call the *query state*) is needed, the function is simply evaluated at that state using the latest parameters produced by the learning algorithm.

Memory-based function approximation methods are very different. They simply save training examples in memory as they arrive (or at least save a subset of the examples) without updating any parameters. Then, whenever a query state's value estimate is needed, a set of examples is retrieved from memory and used to compute a value estimate for the query state. This approach is sometimes called *lazy learning* because processing training examples is postponed until the system is queried to provide an output.

Memory-based function approximation methods are prime examples of *nonparametric* methods. Unlike parametric methods, the approximating function's form is not limited to a fixed parameterized class of functions, such as linear functions or polynomials, but is instead determined by the training examples themselves, together with some means for combining them to output estimated values for query states. As more training examples accumulate in memory, one expects nonparametric methods to produce increasingly accurate approximations of any target function.

There are many different memory-based methods depending on how the stored training examples are selected and how they are used to respond to a query. Here, we focus on *local-learning* methods that approximate a value function only locally in the neighborhood of the current query state. These methods retrieve a set of training examples from memory whose states are judged to be the most relevant to the query state, where relevance usually depends on the distance between states: the closer a training example's state is to the query state, the more relevant it is considered to be, where distance can be defined in many different ways. After the query state is given a value, the local approximation is discarded.

The simplest example of the memory-based approach is the *nearest neighbor* method, which simply finds the example in memory whose state is closest to the query state and returns that example's value as the approximate value of the query state. In other words, if the query state is s, and $s' \mapsto g$ is the example in memory in which s' is the closest state to s, then g is returned as the approximate value of s. Slightly more complicated are *weighted average* methods that retrieve a set of nearest neighbor examples and return a weighted average of their target values, where the weights generally decrease with increasing distance between their states and the query state. *Locally weighted regression* is similar, but it fits a surface to the values of a set of nearest states by means of a parametric approximation method that minimizes a weighted error measure like (9.1), where the weights depend on distances from the query state. The value returned is the evaluation of the locally-fitted surface at the query state, after which the local approximation surface is discarded.

Being nonparametric, memory-based methods have the advantage over parametric methods of not limiting approximations to pre-specified functional forms. This allows accuracy to improve as more data accumulates. Memory-based *local* approximation methods have other properties that make them well suited for reinforcement learning. Because trajectory sampling is of such importance in reinforcement learning, as discussed in Section 8.6, memory-based local methods can focus function approximation on local neighborhoods of states (or state–action pairs) visited in real or simulated trajectories. There may be no need for global approximation because many areas of the state space will never (or almost never) be reached. In addition, memory-based methods allow an agent's experience to have a relatively immediate affect on value estimates in the neighborhood of the current state, in contrast with a parametric method's need to incrementally adjust parameters of a global approximation.

Avoiding global approximation is also a way to address the curse of dimensionality. For example, for a state space with k dimensions, a tabular method storing a global approximation requires memory exponential in k. On the other hand, in storing examples for a memory-based method, each example requires memory proportional to k, and the memory required to store, say, n examples is linear in n. Nothing is exponential in k or n. Of course, the critical remaining issue is whether a memory-based method can answer queries quickly enough to be useful to an agent. A related concern is how speed degrades as the size of the memory grows. Finding nearest neighbors in a large database can take too long to be practical in many applications.

Proponents of memory-based methods have developed ways to accelerate the nearest neighbor search. Using parallel computers or special purpose hardware is one approach; another is the use of special multi-dimensional data structures to store the training data. One data structure studied for this application is the k-d tree (short for k-dimensional tree), which recursively splits a k-dimensional space into regions arranged as nodes of a binary tree. Depending on the amount of data and how it is distributed over the state space, nearest-neighbor search using k-d trees can quickly eliminate large regions of the space in the search for neighbors, making the searches feasible in some problems where naive searches would take too long.

Locally weighted regression additionally requires fast ways to do the local regression computations which have to be repeated to answer each query. Researchers have developed many ways to address these problems, including methods for forgetting entries in order to keep the size of the database within bounds. The Bibliographic and Historical Comments section at the end of this chapter points to some of the relevant literature, including a selection of papers describing applications of memory-based learning to reinforcement learning.

9.10 Kernel-based Function Approximation

Memory-based methods such as the weighted average and locally weighted regression methods described above depend on assigning weights to examples $s' \mapsto g$ in the database depending on the distance between s' and a query states s. The function that assigns these weights is called a *kernel function*, or simply a *kernel*. In the weighted average and locally weighted regressions methods, for example, a kernel function $k : \mathbb{R} \to \mathbb{R}$ assigns weights to distances between states. More generally, weights do not have to depend on distances; they can depend on some other measure of similarity between states. In this case, $k : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$, so that $k(s, s')$ is the weight given to data about s' in its influence on answering queries about s.

Viewed slightly differently, $k(s, s')$ is a measure of the strength of generalization from s' to s. Kernel functions numerically express how *relevant* knowledge about any state is to any other state. As an example, the strengths of generalization for tile coding shown in Figure 9.11 correspond to different kernel functions resulting from uniform and asymmetrical tile offsets. Although tile coding does not explicitly use a kernel function in its operation, it generalizes according to one. In fact, as we discuss more below, the strength of generalization resulting from linear parametric function approximation can always be described by a kernel function.

Kernel regression is the memory-based method that computes a kernel weighted average of the targets of *all* examples stored in memory, assigning the result to the query state. If $\mathcal{D}$ is the set of stored examples, and $g(s')$ denotes the target for state s' in a stored example, then kernel regression approximates the target function, in this case a value function depending on $\mathcal{D}$, as

$$\hat{v}(s,\mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s')g(s'). \tag{9.23}$$

The weighted average method described above is a special case in which $k(s, s')$ is non-zero only when s and s' are close to one another so that the sum need not be computed over all of $\mathcal{D}$.

A common kernel is the Gaussian radial basis function (RBF) used in RBF function approximation as described in Section 9.5.5. In the method described there, RBFs are features whose centers and widths are either fixed from the start, with centers presumably concentrated in areas where many examples are expected to fall, or are adjusted in some way during learning. Barring methods that adjust centers and widths, this is a linear parametric method whose parameters are the weights of each RBF, which are typically learned by stochastic gradient, or semi-gradient, descent. The form of the approximation is a linear combination of the pre-determined RBFs. Kernel regression with an RBF kernel differs from this in two ways. First, it is memory-based: the RBFs are centered on the states of the stored examples. Second, it is nonparametric: there are no parameters to learn; the response to a query is given by (9.23).

Of course, many issues have to be addressed for practical implementation of kernel regression, issues that are beyond the scope of our brief discussion. However, it turns out that any linear parametric regression method like those we described in Section 9.4, with states represented by feature vectors $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^\top$, can be recast as kernel regression where $k(s, s')$ is the inner product of the feature vector representations of s and s'; that is

$$k(s, s') = \mathbf{x}(s)^\top \mathbf{x}(s'). \tag{9.24}$$

Kernel regression with this kernel function produces the same approximation that a linear parametric method would if it used these feature vectors and learned with the same training data.

We skip the mathematical justification for this, which can be found in any modern machine learning text, such as Bishop (2006), and simply point out an important implication. Instead of constructing features for linear parametric function approximators, one can instead construct kernel functions directly without referring at all to feature vectors. Not all kernel functions can be expressed as inner products of feature vectors as in (9.24), but a kernel function that can be expressed like this can offer significant advantages over the equivalent parametric method. For many sets of feature vectors, (9.24) has a compact functional form that can be evaluated without any computation taking place in the d-dimensional feature space. In these cases, kernel regression is much less complex than directly using a linear parametric method with states represented by these feature vectors. This is the so-called "kernel trick" that allows effectively working in the high-dimension of an expansive feature space while actually working only with the set of stored training examples. The kernel trick is the basis of many machine learning methods, and researchers have shown how it can sometimes benefit reinforcement learning.

9.11 Looking Deeper at On-policy Learning: Interest and Emphasis

The algorithms we have considered so far in this chapter have treated all the states encountered equally, as if they were all equally important. In some cases, however, we are more interested in some states than others. In discounted episodic problems, for example, we may be more interested in accurately valuing early states in the episode than in later states where discounting may have made the rewards much less important to the value of the start state. Or, if an action-value function is being learned, it may be less important to accurately value poor actions whose value is much less than the greedy action. Function approximation resources are always limited, and if they were used in a more targeted way, then performance could be improved.

One reason we have treated all states encountered equally is that then we are updating according to the on-policy distribution, for which stronger theoretical results are available for semi-gradient methods. Recall that the on-policy distribution was defined as the distribution of states encountered in an MDP while following the target policy. Now we will generalize this concept significantly. Rather than having one on-policy distribution for the MDP, we will have many. All of them will have in common that they are a distribution of states encountered in trajectories while following the target policy, but they will vary in how the trajectories are, in a sense, initiated.

We now introduce some new concepts. First we introduce a non-negative scalar measure, a random variable I_t called *interest*, indicating the degree to which we are interested in accurately valuing the state (or state–action pair) at time t. If we don't care at all about the state, then the interest should be zero; if we fully care, it might be one, though it is formally allowed to take any non-negative value. The interest can be set in any causal way; for example, it may depend on the trajectory up to time t or the learned parameters at time t. The distribution μ in the $\overline{\text{VE}}$ (9.1) is then defined as the distribution of states encountered while following the target policy, weighted by the interest. Second, we introduce another non-negative scalar random variable, the *emphasis* M_t. This scalar multiplies the learning update and thus emphasizes or de-emphasizes the learning done at time t. The general n-step learning rule, replacing (9.15), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha M_t \left[G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})\right] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \qquad 0 \le t < T, \tag{9.25}$$

with the n-step return given by (9.16) and the emphasis determined recursively from the interest by:

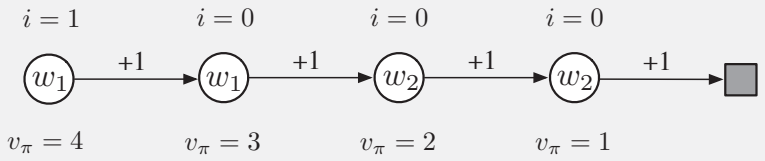
$$M_t = I_t + \gamma^n M_{t-n}, \qquad 0 \le t < T, \tag{9.26}$$

with $M_t \doteq 0$, for all $t < 0$. These equations are taken to include the Monte Carlo case, for which $G_{t:t+n} = G_t$, all the updates are made at end of the episode, $n = T - t$, and $M_t = I_t$.

Example 9.4 illustrates how interest and emphasis can result in more accurate value estimates.

Example 9.4: Interest and Emphasis

To see the potential benefits of using interest and emphasis, consider the four-state Markov reward process shown below:



Episodes start in the leftmost state, then transition one state to the right, with a reward of +1, on each step until the terminal state is reached. The true value of the first state is thus 4, of the second state 3, and so on as shown below each state. These are the true values; the estimated values can only approximate these because they are constrained by the parameterization. There are two components to the parameter vector $\mathbf{w} = (w_1, w_2)^\top$, and the parameterization is as written inside each state. The estimated values of the first two states are given by w_1 alone and thus must be the same even though their true values are different. Similarly, the estimated values of the third and fourth states are given by w_2 alone and must be the same even though their true values are different. Suppose that we are interested in accurately valuing only the leftmost state; we assign it an interest of 1 while all the other states are assigned an interest of 0, as indicated above the states.

First consider applying gradient Monte Carlo algorithms to this problem. The algorithms presented earlier in this chapter that do not take into account interest and emphasis (in (9.7) and the box on page 202) will converge (for decreasing step sizes) to the parameter vector $\mathbf{w}_\infty = (3.5, 1.5)$, which gives the first state—the only one we are interested in—a value of 3.5 (i.e., intermediate between the true values of the first and second states). The methods presented in this section that do use interest and emphasis, on the other hand, will learn the value of the first state exactly correctly; w_1 will converge to 4 while w_2 will never be updated because the emphasis is zero in all states save the leftmost.

Now consider applying two-step semi-gradient TD methods. The methods from earlier in this chapter without interest and emphasis (in (9.15) and (9.16) and the box on page 209) will again converge to $\mathbf{w}_\infty = (3.5, 1.5)$, while the methods with interest and emphasis converge to $\mathbf{w}_\infty = (4, 2)$. The latter produces the exactly correct values for the first state and for the third state (which the first state bootstraps from) while never making any updates corresponding to the second or fourth states.

9.12 Summary

Reinforcement learning systems must be capable of *generalization* if they are to be applicable to artificial intelligence or to large engineering applications. To achieve this, any of a broad range of existing methods for *supervised-learning function approximation* can be used simply by treating each update as a training example.

Perhaps the most suitable supervised learning methods are those using *parameterized function approximation*, in which the policy is parameterized by a weight vector $\mathbf{w}$. Although the weight vector has many components, the state space is much larger still, and we must settle for an approximate solution. We defined the *mean squared value error*, $\overline{\text{VE}}(\mathbf{w})$, as a measure of the error in the values $v_{\pi_\mathbf{w}}(s)$ for a weight vector $\mathbf{w}$ under the *on-policy distribution*, μ. The $\overline{\text{VE}}$ gives us a clear way to rank different value-function approximations in the on-policy case.

To find a good weight vector, the most popular methods are variations of *stochastic gradient descent* (SGD). In this chapter we have focused on the *on-policy* case with a *fixed policy*, also known as policy evaluation or prediction; a natural learning algorithm for this case is *n-step semi-gradient TD*, which includes gradient Monte Carlo and semi-gradient TD(0) algorithms as the special cases when $n = \infty$ and $n = 1$ respectively. Semi-gradient TD methods are not true gradient methods. In such bootstrapping methods (including DP), the weight vector appears in the update target, yet this is not taken into account in computing the gradient—thus they are *semi*-gradient methods. As such, they cannot rely on classical SGD results.

Nevertheless, good results can be obtained for semi-gradient methods in the special case of *linear* function approximation, in which the value estimates are sums of features times corresponding weights. The linear case is the most well understood theoretically and works well in practice when provided with appropriate features. Choosing the features is one of the most important ways of adding prior domain knowledge to reinforcement learning systems. They can be chosen as polynomials, but this case generalizes poorly in the online learning setting typically considered in reinforcement learning. Better is to choose features according the Fourier basis, or according to some form of coarse coding with sparse overlapping receptive fields. Tile coding is a form of coarse coding that is particularly computationally efficient and flexible. Radial basis functions are useful for one- or two-dimensional tasks in which a smoothly varying response is important. LSTD is the most data-efficient linear TD prediction method, but requires computation proportional to the square of the number of weights, whereas all the other methods are of complexity linear in the number of weights. Nonlinear methods include artificial neural networks trained by backpropagation and variations of SGD; these methods have become very popular in recent years under the name *deep reinforcement learning*.

Linear semi-gradient n-step TD is guaranteed to converge under standard conditions, for all n, to a $\overline{\text{VE}}$ that is within a bound of the optimal error (achieved asymptotically by Monte Carlo methods). This bound is always tighter for higher n and approaches zero as $n \to \infty$. However, in practice very high n results in very slow learning, and some degree of bootstrapping ($n < \infty$) is usually preferrable, just as we saw in comparisons of tabular n-step methods in Chapter 7 and in comparisons of tabular TD and Monte Carlo methods in Chapter 6.

Bibliographical and Historical Remarks

Generalization and function approximation have always been an integral part of reinforcement learning. Bertsekas and Tsitsiklis (1996), Bertsekas (2012), and Sugiyama et al. (2013) present the state of the art in function approximation in reinforcement learning. Some of the early work with function approximation in reinforcement learning is discussed at the end of this section.

9.3 Gradient-descent methods for minimizing mean-squared error in supervised learning are well known. Widrow and Hoff (1960) introduced the least-mean-square (LMS) algorithm, which is the prototypical incremental gradient-descent algorithm. Details of this and related algorithms are provided in many texts (e.g., Widrow and Stearns, 1985; Bishop, 1995; Duda and Hart, 1973).

Semi-gradient TD(0) was first explored by Sutton (1984, 1988), as part of the linear TD(λ) algorithm that we will treat in Chapter 12. The term "semi-gradient" to describe these bootstrapping methods is new to the second edition of this book.

The earliest use of state aggregation in reinforcement learning may have been Michie and Chambers's BOXES system (1968). The theory of state aggregation in reinforcement learning has been developed by Singh, Jaakkola, and Jordan (1995) and Tsitsiklis and Van Roy (1996). State aggregation has been used in dynamic programming from its earliest days (e.g., Bellman, 1957a).

9.4 Sutton (1988) proved convergence of linear TD(0) in the mean to the minimal $\overline{\text{VE}}$ solution for the case in which the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, are linearly independent. Convergence with probability 1 was proved by several researchers at about the same time (Peng, 1993; Dayan and Sejnowski, 1994; Tsitsiklis, 1994; Gurvits, Lin, and Hanson, 1994). In addition, Jaakkola, Jordan, and Singh (1994) proved convergence under online updating. All of these results assumed linearly independent feature vectors, which implies at least as many components to $\mathbf{w}_t$ as there are states. Convergence for the more important case of general (dependent) feature vectors was first shown by Dayan (1992). A significant generalization and strengthening of Dayan's result was proved by Tsitsiklis and Van Roy (1997). They proved the main result presented in this section, the bound on the asymptotic error of linear bootstrapping methods.

9.5 Our presentation of the range of possibilities for linear function approximation is based on that by Barto (1990).

9.5.2 Konidaris, Osentoski, and Thomas (2011) introduced the Fourier basis in a simple form suitable for reinforcement learning problems with multi-dimensional continuous state spaces and functions that do not have to be periodic.

9.5.3 The term *coarse coding* is due to Hinton (1984), and our Figure 9.6 is based on one of his figures. Waltz and Fu (1965) provide an early example of this type of function approximation in a reinforcement learning system.

9.5.4 Tile coding, including hashing, was introduced by Albus (1971, 1981). He described it in terms of his "cerebellar model articulator controller," or CMAC, as tile coding is sometimes known in the literature. The term "tile coding" was new to the first edition of this book, though the idea of describing CMAC in these terms is taken from Watkins (1989). Tile coding has been used in many reinforcement learning systems (e.g., Shewchuk and Dean, 1990; Lin and Kim, 1991; Miller, Scalera, and Kim, 1994; Sofge and White, 1992; Tham, 1994; Sutton, 1996; Watkins, 1989) as well as in other types of learning control systems (e.g., Kraft and Campagna, 1990; Kraft, Miller, and Dietz, 1992). This section draws heavily on the work of Miller and Glanz (1996). General software for tile coding is available in several languages (e.g., see `http://incompleteideas.net/tiles/tiles3.html`).

9.5.5 Function approximation using radial basis functions has received wide attention ever since being related to ANNs by Broomhead and Lowe (1988). Powell (1987) reviewed earlier uses of RBFs, and Poggio and Girosi (1989, 1990) extensively developed and applied this approach.

9.6 Automatic methods for adapting the step-size parameter include RMSprop (Tieleman and Hinton, 2012), Adam (Kingma and Ba, 2015), stochastic meta-descent methods such as Delta-Bar-Delta (Jacobs, 1988), its incremental generalization (Sutton, 1992b, c; Mahmood et al., 2012), and nonlinear generalizations (Schraudolph, 1999, 2002). Methods explicitly designed for reinforcement learning include AlphaBound (Dabney and Barto, 2012), SID and NOSID (Dabney, 2014), TIDBD (Kearney et al., in preparation) and the application of stochastic meta-descent to policy gradient learning (Schraudolph, Yu, and Aberdeen, 2006).

9.6 The introduction of the threshold logic unit as an abstract model neuron by McCulloch and Pitts (1943) was the beginning of ANNs. The history of ANNs as learning methods for classification or regression has passed through several stages: roughly, the Perceptron (Rosenblatt, 1962) and ADALINE (ADAptive LINear Element) (Widrow and Hoff, 1960) stage of learning by single-layer ANNs, the error-backpropagation stage (LeCun, 1985; Rumelhart, Hinton, and Williams, 1986) of learning by multi-layer ANNs, and the current deep-learning stage with its emphasis on representation learning (e.g., Bengio, Courville, and Vincent, 2012; Goodfellow, Bengio, and Courville, 2016). Examples of the many books on ANNs are Haykin (1994), Bishop (1995), and Ripley (2007).

ANNs as function approximation for reinforcement learning goes back to the early work of Farley and Clark (1954), who used reinforcement-like learning to modify the weights of linear threshold functions representing policies. Widrow, Gupta, and Maitra (1973) presented a neuron-like linear threshold unit implementing a learning process they called *learning with a critic* or *selective bootstrap adaptation*, a reinforcement-learning variant of the ADALINE algorithm. Werbos (1987, 1994) developed an approach to prediction and control that uses ANNs trained by error backpropation to learn policies and value functions using TD-like algorithms. Barto, Sutton, and Brouwer (1981) and Barto and Sutton (1981b) extended the

idea of an associative memory network (e.g., Kohonen, 1977; Anderson, Silverstein, Ritz, and Jones, 1977) to reinforcement learning. Barto, Anderson, and Sutton (1982) used a two-layer ANN to learn a nonlinear control policy, and emphasized the first layer's role of learning a suitable representation. Hampson (1983, 1989) was an early proponent of multilayer ANNs for learning value functions. Barto, Sutton, and Anderson (1983) presented an actor–critic algorithm in the form of an ANN learning to balance a simulated pole (see Sections 15.7 and 15.8). Barto and Anandan (1985) introduced a stochastic version of Widrow et al.'s (1973) selective bootstrap algorithm called the *associative reward-penalty* (A_{R-P}) *algorithm*. Barto (1985, 1986) and Barto and Jordan (1987) described multi-layer ANNs consisting of $\mathrm{A_{R-P}}$ units trained with a globally-broadcast reinforcement signal to learn classification rules that are not linearly separable. Barto (1985) discussed this approach to ANNs and how this type of learning rule is related to others in the literature at that time. (See Section 15.10 for additional discussion of this approach to training multi-layer ANNs.) Anderson (1986, 1987, 1989) evaluated numerous methods for training multilayer ANNs and showed that an actor–critic algorithm in which both the actor and critic were implemented by two-layer ANNs trained by error backpropagation outperformed single-layer ANNs in the pole-balancing and tower of Hanoi tasks. Williams (1988) described several ways that backpropagation and reinforcement learning can be combined for training ANNs. Gullapalli (1990) and Williams (1992) devised reinforcement learning algorithms for neuron-like units having continuous, rather than binary, outputs. Barto, Sutton, and Watkins (1990) argued that ANNs can play significant roles for approximating functions required for solving sequential decision problems. Williams (1992) related REINFORCE learning rules (Section 13.3) to the error backpropagation method for training multi-layer ANNs. Tesauro's TD-Gammon (Tesauro 1992, 1994; Section 16.1) influentially demonstrated the learning abilities of TD(λ) algorithm with function approximation by multi-layer ANNs in learning to play backgammon. The *AlphaGo*, *AlphaGo Zero*, and *AlphaZero* programs of Silver et al. (2016, 2017a, b; Section 16.6) used reinforcement learning with deep convolutional ANNs in achieving impressive results with the game of Go. Schmidhuber (2015) reviews applications of ANNs in reinforcement learning, including applications of recurrent ANNs.

9.8 LSTD is due to Bradtke and Barto (see Bradtke, 1993, 1994; Bradtke and Barto, 1996; Bradtke, Ydstie, and Barto, 1994), and was further developed by Boyan (1999, 2002), Nedić and Bertsekas (2003), and Yu (2010). The incremental update of the inverse matrix has been known at least since 1949 (Sherman and Morrison, 1949). An extension of least-squares methods to control was introduced by Lagoudakis and Parr (2003; Buşoniu, Lazaric, Ghavamzadeh, Munos, Babuška, and De Schutter, 2012).

9.9 Our discussion of memory-based function approximation is largely based on the review of locally weighted learning by Atkeson, Moore, and Schaal (1997). Atkeson (1992) discussed the use of locally weighted regression in memory-based robot learning and supplied an extensive bibliography covering the history of the idea. Stanfill and Waltz (1986) influentially argued for the importance of memory based methods in artificial intelligence, especially in light of parallel architectures then becoming available, such as the Connection Machine. Baird and Klopf (1993) introduced a novel memory-based approach and used it as the function approximation method for Q-learning applied to the pole-balancing task. Schaal and Atkeson (1994) applied locally weighted regression to a robot juggling control problem, where it was used to learn a system model. Peng (1995) used the pole-balancing task to experiment with several nearest-neighbor methods for approximating value functions, policies, and environment models. Tadepalli and Ok (1996) obtained promising results with locally-weighted linear regression to learn a value function for a simulated automatic guided vehicle task. Bottou and Vapnik (1992) demonstrated surprising efficiency of several local learning algorithms compared to non-local algorithms in some pattern recognition tasks, discussing the impact of local learning on generalization.

Bentley (1975) introduced k-d trees and reported observing average running time of O(log n) for nearest neighbor search over n records. Friedman, Bentley, and Finkel (1977) clarified the algorithm for nearest neighbor search with k-d trees. Omohundro (1987) discussed efficiency gains possible with hierarchical data structures such as k-d-trees. Moore, Schneider, and Deng (1997) introduced the use of k-d trees for efficient locally weighted regression.

9.10 The origin of kernel regression is the *method of potential functions* of Aizerman, Braverman, and Rozonoer (1964). They likened the data to point electric charges of various signs and magnitudes distributed over space. The resulting electric potential over space produced by summing the potentials of the point charges corresponded to the interpolated surface. In this analogy, the kernel function is the potential of a point charge, which falls off as the reciprocal of the distance from the charge. Connell and Utgoff (1987) applied an actor–critic method to the pole-balancing task in which the critic approximated the value function using kernel regression with an inverse-distance weighting. Predating widespread interest in kernel regression in machine learning, these authors did not use the term kernel, but referred to "Shepard's method" (Shepard, 1968). Other kernel-based approaches to reinforcement learning include those of Ormoneit and Sen (2002), Dietterich and Wang (2002), Xu, Xie, Hu, and Lu (2005), Taylor and Parr (2009), Barreto, Precup, and Pineau (2011), and Bhat, Farias, and Moallemi (2012).

9.11 For Emphatic-TD methods, see the bibliographical notes to Section 11.8.

The earliest example we know of in which function approximation methods were used for learning value functions was Samuel's checkers player (1959, 1967). Samuel followed Shannon's (1950) suggestion that a value function did not have to be exact to be a useful guide to selecting moves in a game and that it might be approximated by linear combination of features. In addition to linear function approximation, Samuel experimented with lookup tables and hierarchical lookup tables called signature tables (Griffith, 1966, 1974; Page, 1977; Biermann, Fairfield, and Beres, 1982).

At about the same time as Samuel's work, Bellman and Dreyfus (1959) proposed using function approximation methods with DP. (It is tempting to think that Bellman and Samuel had some influence on one another, but we know of no reference to the other in the work of either.) There is now a fairly extensive literature on function approximation methods and DP, such as multigrid methods and methods using splines and orthogonal polynomials (e.g., Bellman and Dreyfus, 1959; Bellman, Kalaba, and Kotkin, 1973; Daniel, 1976; Whitt, 1978; Reetz, 1977; Schweitzer and Seidmann, 1985; Chow and Tsitsiklis, 1991; Kushner and Dupuis, 1992; Rust, 1996).

Holland's (1986) classifier system used a selective feature-match technique to generalize evaluation information across state–action pairs. Each classifier matched a subset of states having specified values for a subset of features, with the remaining features having arbitrary values ("wild cards"). These subsets were then used in a conventional state-aggregation approach to function approximation. Holland's idea was to use a genetic algorithm to evolve a set of classifiers that collectively would implement a useful action-value function. Holland's ideas influenced the early research of the authors on reinforcement learning, but we focused on different approaches to function approximation. As function approximators, classifiers are limited in several ways. First, they are state-aggregation methods, with concomitant limitations in scaling and in representing smooth functions efficiently. In addition, the matching rules of classifiers can implement only aggregation boundaries that are parallel to the feature axes. Perhaps the most important limitation of conventional classifier systems is that the classifiers are learned via the genetic algorithm, an evolutionary method. As we discussed in Chapter 1, there is available during learning much more detailed information about how to learn than can be used by evolutionary methods. This perspective led us to instead adapt supervised learning methods for use in reinforcement learning, specifically gradient-descent and ANN methods. These differences between Holland's approach and ours are not surprising because Holland's ideas were developed during a period when ANNs were generally regarded as being too weak in computational power to be useful, whereas our work was at the beginning of the period that saw widespread questioning of that conventional wisdom. There remain many opportunities for combining aspects of these different approaches.

Christensen and Korf (1986) experimented with regression methods for modifying coefficients of linear value function approximations in the game of chess. Chapman and Kaelbling (1991) and Tan (1991) adapted decision-tree methods for learning value functions. Explanation-based learning methods have also been adapted for learning value functions, yielding compact representations (Yee, Saxena, Utgoff, and Barto, 1990; Dietterich and Flann, 1995).

Chapter 10

On-policy Control with Approximation

In this chapter we return to the control problem, now with parametric approximation of the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$, where $\mathbf{w} \in \mathbb{R}^d$ is a finite-dimensional weight vector. We continue to restrict attention to the on-policy case, leaving off-policy methods to Chapter 11. The present chapter features the semi-gradient Sarsa algorithm, the natural extension of semi-gradient TD(0) (last chapter) to action values and to on-policy control. In the episodic case, the extension is straightforward, but in the continuing case we have to take a few steps backward and re-examine how we have used discounting to define an optimal policy. Surprisingly, once we have genuine function approximation we have to give up discounting and switch to a new "average-reward" formulation of the control problem, with new "differential" value functions.

Starting first in the episodic case, we extend the function approximation ideas presented in the last chapter from state values to action values. Then we extend them to control following the general pattern of on-policy GPI, using ε-greedy for action selection. We show results for n-step linear Sarsa on the Mountain Car problem. Then we turn to the continuing case and repeat the development of these ideas for the average-reward case with differential values.

10.1 Episodic Semi-gradient Control

The extension of the semi-gradient prediction methods of Chapter 9 to action values is straightforward. In this case it is the approximate action-value function, $\hat{q} \approx q_\pi$, that is represented as a parameterized functional form with weight vector $\mathbf{w}$. Whereas before we considered random training examples of the form $S_t \mapsto U_t$, now we consider examples of the form $S_t, A_t \mapsto U_t$. The update target U_t can be any approximation of $q_\pi(S_t, A_t)$, including the usual backed-up values such as the full Monte Carlo return (G_t) or any of the n-step Sarsa returns (7.4). The general gradient-descent update for action-value

prediction is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t). \tag{10.1}$$

For example, the update for the one-step Sarsa method is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t). \tag{10.2}$$

We call this method *episodic semi-gradient one-step Sarsa*. For a constant policy, this method converges in the same way that TD(0) does, with the same kind of error bound (9.14).

To form control methods, we need to couple such action-value prediction methods with techniques for policy improvement and action selection. Suitable techniques applicable to continuous actions, or to actions from large discrete sets, are a topic of ongoing research with as yet no clear resolution. On the other hand, if the action set is discrete and not too large, then we can use the techniques already developed in previous chapters. That is, for each possible action a available in the current state S_t, we can compute $\hat{q}(S_t, a, \mathbf{w}_t)$ and then find the greedy action $A_t^* = \arg\max_a \hat{q}(S_t, a, \mathbf{w}_t)$. Policy improvement is then done (in the on-policy case treated in this chapter) by changing the estimation policy to a soft approximation of the greedy policy such as the ε-greedy policy. Actions are selected according to this same policy. Pseudocode for the complete algorithm is given in the box.

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Example 10.1: Mountain Car Task Consider the task of driving an underpowered car up a steep mountain road, as suggested by the diagram in the upper left of Figure 10.1. The difficulty is that gravity is stronger than the car's engine, and even at full throttle the car cannot accelerate up the steep slope. The only solution is to first move away from the goal and up the opposite slope on the left. Then, by applying full throttle the car

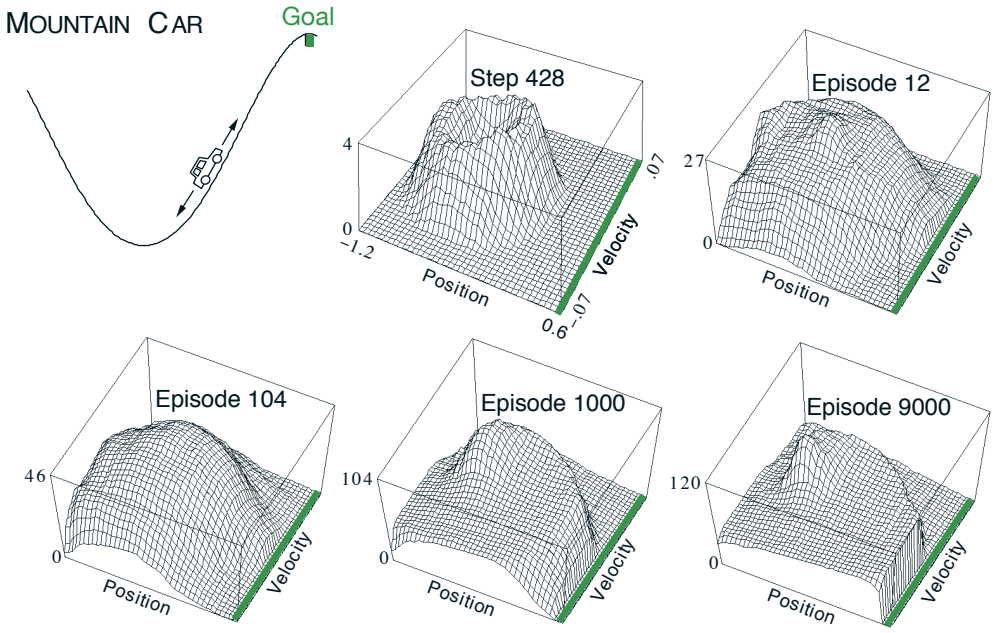


Figure 10.1: The Mountain Car task (upper left panel) and the cost-to-go function ($-\max_a \hat{q}(s, a, \mathbf{w})$) learned during one run.

can build up enough inertia to carry it up the steep slope even though it is slowing down the whole way. This is a simple example of a continuous control task where things have to get worse in a sense (farther from the goal) before they can get better. Many control methodologies have great difficulties with tasks of this kind unless explicitly aided by a human designer.

The reward in this problem is -1 on all time steps until the car moves past its goal position at the top of the mountain, which ends the episode. There are three possible actions: full throttle forward ($+1$), full throttle reverse (-1), and zero throttle (0). The car moves according to a simplified physics. Its position, x_t, and velocity, $\dot{x}_t$, are updated by

$$x_{t+1} \doteq \mathit{bound}\big[x_t + \dot{x}_{t+1}\big]$$

$$\dot{x}_{t+1} \doteq \mathit{bound}\big[\dot{x}_t + 0.001A_t - 0.0025\cos(3x_t)\big],$$

where the *bound* operation enforces $-1.2 \leq x_{t+1} \leq 0.5$ and $-0.07 \leq \dot{x}_{t+1} \leq 0.07$. In addition, when x_{t+1} reached the left bound, $\dot{x}_{t+1}$ was reset to zero. When it reached the right bound, the goal was reached and the episode was terminated. Each episode started from a random position $x_t \in [-0.6, -0.4)$ and zero velocity. To convert the two continuous state variables to binary features, we used grid-tilings as in Figure 9.9. We used 8 tilings, with each tile covering 1/8th of the bounded distance in each dimension,

and asymmetrical offsets as described in Section 9.5.4.[1] The feature vectors $\mathbf{x}(s, a)$ created by tile coding were then combined linearly with the parameter vector to approximate the action-value function:

$$\hat{q}(s, a, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s, a) = \sum_{i=1}^{d} w_i \cdot x_i(s, a), \tag{10.3}$$

for each pair of state, s, and action, a.

Figure 10.1 shows what typically happens while learning to solve this task with this form of function approximation.[2] Shown is the negative of the value function (the *cost-to-go* function) learned on a single run. The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur even though the exploration parameter, ε, was 0. This can be seen in the middle-top panel of the figure, labeled "Step 428". At this time not even one episode had been completed, but the car has oscillated back and forth in the valley, following circular trajectories in state space. All the states visited frequently are valued worse than unexplored states, because the actual rewards have been worse than what was (unrealistically) expected. This continually drives the agent away from wherever it has been, to explore new states, until a solution is found.

Figure 10.2 shows several learning curves for semi-gradient Sarsa on this problem, with various step sizes.

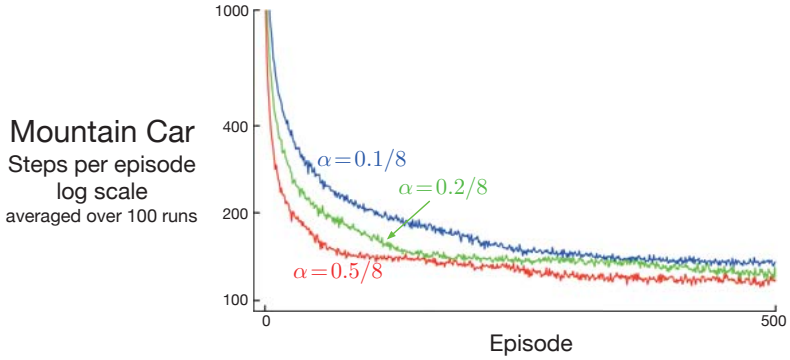


Figure 10.2: Mountain Car learning curves for the semi-gradient Sarsa method with tile-coding function approximation and ε-greedy action selection. ■

[1] In particular, we used the tile-coding software, available at `http://incompleteideas.net/tiles/tiles3.html`, with `iht=IHT(4096)` and `tiles(iht,8,[8*x/(0.5+1.2),8*xdot/(0.07+0.07)],A)` to get the indices of the ones in the feature vector for state (`x`, `xdot`) and action `A`.

[2] This data is actually from the "semi-gradient Sarsa(λ)" algorithm that we will not meet until Chapter 12, but semi-gradient Sarsa would behave similarly.

10.2 Semi-gradient n-step Sarsa

We can obtain an n-step version of episodic semi-gradient Sarsa by using an n-step return as the update target in the semi-gradient Sarsa update equation (10.1). The n-step return immediately generalizes from its tabular form (7.4) to a function approximation form:

$$G_{t:t+n} \doteq R_{t+1}+\gamma R_{t+2}+\cdots+\gamma^{n-1}R_{t+n}+\gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad t+n<T, \quad (10.4)$$

with $G_{t:t+n} \doteq G_t$ if $t+n \geq T$, as usual. The n-step update equation is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha \left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \qquad 0 \leq t < T. \quad (10.5)$$

Complete pseudocode is given in the box below.

Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer n
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or ε-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t+1$
      else:
        Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
    $\tau \leftarrow t-n+1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If $\tau+n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{q}(S_\tau, A_\tau, \mathbf{w})\right] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T-1$

As we have seen before, performance is best if an intermediate level of bootstrapping is used, corresponding to an n larger than 1. Figure 10.3 shows how this algorithm tends to learn faster and obtain a better asymptotic performance at $n=8$ than at $n=1$ on the Mountain Car task. Figure 10.4 shows the results of a more detailed study of the effect of the parameters α and n on the rate of learning on this task.

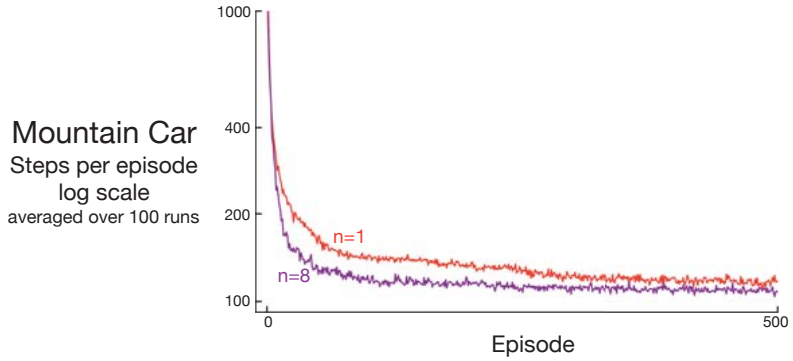


Figure 10.3: Performance of one-step vs 8-step semi-gradient Sarsa on the Mountain Car task. Good step sizes were used: $\alpha = 0.5/8$ for $n = 1$ and $\alpha = 0.3/8$ for $n = 8$.

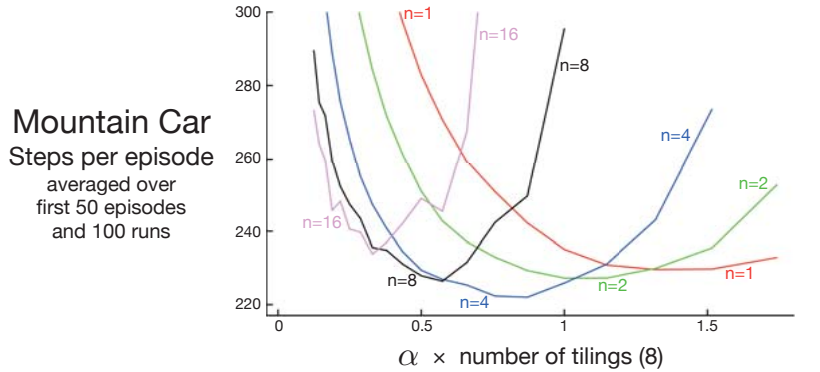


Figure 10.4: Effect of the α and n on early performance of n-step semi-gradient Sarsa and tile-coding function approximation on the Mountain Car task. As usual, an intermediate level of bootstrapping ($n = 4$) performed best. These results are for selected α values, on a log scale, and then connected by straight lines. The standard errors ranged from 0.5 (less than the line width) for $n = 1$ to about 4 for $n = 16$, so the main effects are all statistically significant.

Exercise 10.1 We have not explicitly considered or given pseudocode for any Monte Carlo methods or in this chapter. What would they be like? Why is it reasonable not to give pseudocode for them? How would they perform on the Mountain Car task? □

Exercise 10.2 Give pseudocode for semi-gradient one-step *Expected* Sarsa for control. □

Exercise 10.3 Why do the results shown in Figure 10.4 have higher standard errors at large n than at small n? □

10.3 Average Reward: A New Problem Setting for Continuing Tasks

We now introduce a third classical setting—alongside the episodic and discounted settings—for formulating the goal in Markov decision problems (MDPs). Like the discounted setting, the *average reward* setting applies to continuing problems, problems for which the interaction between agent and environment goes on and on forever without termination or start states. Unlike that setting, however, there is no discounting—the agent cares just as much about delayed rewards as it does about immediate reward. The average-reward setting is one of the major settings commonly considered in the classical theory of dynamic programming and less-commonly in reinforcement learning. As we discuss in the next section, the discounted setting is problematic with function approximation, and thus the average-reward setting is needed to replace it.

In the average-reward setting, the quality of a policy π is defined as the average rate of reward, or simply *average reward*, while following that policy, which we denote as $r(\pi)$:

$$\begin{aligned} r(\pi) &\doteq \lim_{h\to\infty} \frac{1}{h}\sum_{t=1}^{h} \mathbb{E}[R_t \mid S_0, A_{0:t-1}\sim\pi] && (10.6)\\ &= \lim_{t\to\infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1}\sim\pi], && (10.7)\\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s',r\,|\,s,a) r, \end{aligned}$$

where the expectations are conditioned on the initial state, S_0, and on the subsequent actions, $A_0, A_1, \ldots, A_{t-1}$, being taken according to π. μ_π is the steady-state distribution, $\mu_\pi(s) \doteq \lim_{t\to\infty} \Pr\{S_t{=}s \mid A_{0:t-1}\sim\pi\}$, which is assumed to exist for any π and to be independent of S_0. This assumption about the MDP is known as *ergodicity*. It means that where the MDP starts or any early decision made by the agent can have only a temporary effect; in the long run the expectation of being in a state depends only on the policy and the MDP transition probabilities. Ergodicity is sufficient to guarantee the existence of the limits in the equations above.

There are subtle distinctions that can be drawn between different kinds of optimality in the undiscounted continuing case. Nevertheless, for most practical purposes it may be adequate simply to order policies according to their average reward per time step, in other words, according to their $r(\pi)$. This quantity is essentially the average reward under π, as suggested by (10.7). In particular, we consider all policies that attain the maximal value of $r(\pi)$ to be optimal.

Note that the steady state distribution is the special distribution under which, if you select actions according to π, you remain in the same distribution. That is, for which

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s'\,|\,s,a) = \mu_\pi(s'). \tag{10.8}$$

In the average-reward setting, returns are defined in terms of differences between

rewards and the average reward:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \cdots. \tag{10.9}$$

This is known as the *differential* return, and the corresponding value functions are known as *differential* value functions. They are defined in the same way and we will use the same notation for them as we have all along: $v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s]$ and $q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$ (similarly for v_* and q_*). Differential value functions also have Bellman equations, just slightly different from those we have seen earlier. We simply remove all γs and replace all rewards by the difference between the reward and the true average reward:

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{r,s'} p(s',r\,|\,s,a)\Big[r - r(\pi) + v_\pi(s')\Big],\\ q_\pi(s,a) &= \sum_{r,s'} p(s',r\,|\,s,a)\Big[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s',a')\Big],\\ v_*(s) &= \max_a \sum_{r,s'} p(s',r\,|\,s,a)\Big[r - \max_\pi r(\pi) + v_*(s')\Big], \text{ and}\\ q_*(s,a) &= \sum_{r,s'} p(s',r\,|\,s,a)\Big[r - \max_\pi r(\pi) + \max_{a'} q_*(s',a')\Big] \end{aligned}$$

(cf. (3.14), Exercise 3.17, (3.19), and (3.20)).

There is also a differential form of the two TD errors:

$$\delta_t \doteq R_{t+1} - \bar{R}_{t+1} + \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t), \tag{10.10}$$

and

$$\delta_t \doteq R_{t+1} - \bar{R}_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{10.11}$$

where $\bar{R}_t$ is an estimate at time t of the average reward $r(\pi)$. With these alternate definitions, most of our algorithms and many theoretical results carry through to the average-reward setting without change.

For example, the average reward version of semi-gradient Sarsa is defined just as in (10.2) except with the differential version of the TD error. That is, by

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t \nabla\hat{q}(S_t, A_t, \mathbf{w}_t), \tag{10.12}$$

with δ_t given by (10.11). The pseudocode for the complete algorithm is given in the box on the next page.

Exercise 10.4 Give pseudocode for a differential version of semi-gradient Q-learning. □

Exercise 10.5 What equations are needed (beyond 10.10) to specify the differential version of TD(0)? □

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

Exercise 10.6 Consider a Markov reward process consisting of a ring of three states A, B, and C, with state transitions going deterministically around the ring. A reward of +1 is received upon arrival in A and otherwise the reward is 0. What are the differential values of the three states? □

Example 10.2: An Access-Control Queuing Task This is a decision task involving access control to a set of 10 servers. Customers of four different priorities arrive at a single queue. If given access to a server, the customers pay a reward of 1, 2, 4, or 8 to the server, depending on their priority, with higher priority customers paying more. In each time step, the customer at the head of the queue is either accepted (assigned to one of the servers) or rejected (removed from the queue, with a reward of zero). In either case, on the next time step the next customer in the queue is considered. The queue never empties, and the priorities of the customers in the queue are equally randomly distributed. Of course a customer cannot be served if there is no free server; the customer is always rejected in this case. Each busy server becomes free with probability $p = 0.06$ on each time step. Although we have just described them for definiteness, let us assume the statistics of arrivals and departures are unknown. The task is to decide on each step whether to accept or reject the next customer, on the basis of his priority and the number of free servers, so as to maximize long-term reward without discounting.

In this example we consider a tabular solution to this problem. Although there is no generalization between states, we can still consider it in the general function approximation setting as this setting generalizes the tabular setting. Thus we have a differential action-value estimate for each pair of state (number of free servers and priority of the customer at the head of the queue) and action (accept or reject). Figure 10.5 shows the solution found by differential semi-gradient Sarsa with parameters $\alpha = 0.01$, $\beta = 0.01$, and $\varepsilon = 0.1$. The initial action values and $\bar{R}$ were zero.

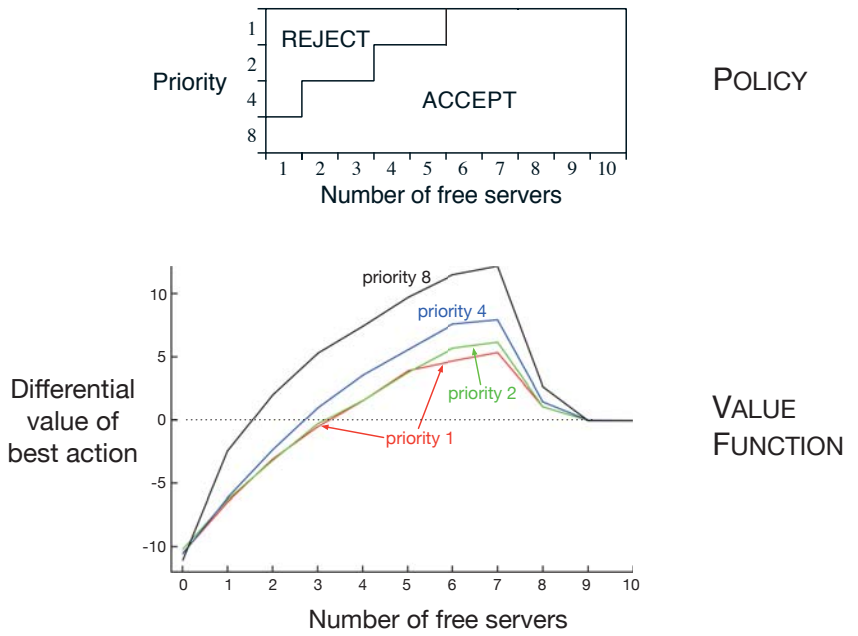


Figure 10.5: The policy and value function found by differential semi-gradient one-step Sarsa on the access-control queuing task after 2 million steps. The drop on the right of the graph is probably due to insufficient data; many of these states were never experienced. The value learned for $\bar{R}$ was about 2.31. ■

Exercise 10.7 Suppose there is an MDP that under any policy produces the deterministic sequence of rewards $+1, 0, +1, 0, +1, 0, \ldots$ going on forever. Technically, this is not allowed because it violates ergodicity; there is no stationary limiting distribution μ_π and the limit (10.7) does not exist. Nevertheless, the average reward (10.6) is well defined; What is it? Now consider two states in this MDP. From A, the reward sequence is exactly as described above, starting with a +1, whereas, from B, the reward sequence starts with a 0 and then continues with $+1, 0, +1, 0, \ldots$. The differential return (10.9) is not well defined for this case as the limit does not exist. To repair this, one could alternately define the value of a state as

$$v_\pi(s) \doteq \lim_{\gamma \to 1} \lim_{h \to \infty} \sum_{t=0}^{h} \gamma^t \Big(\mathbb{E}_\pi[R_{t+1} | S_0 = s] - r(\pi) \Big). \tag{10.13}$$

Under this definition, what are the values of states A and B? □

Exercise 10.8 The pseudocode in the box on page 251 updates $\bar{R}_{t+1}$ using δ_t as an error rather than simply $R_{t+1} - \bar{R}_{t+1}$. Both errors work, but using δ_t is better. To see why, consider the ring MRP of three states from Exercise 10.6. The estimate of the average reward should tend towards its true value of $\frac{1}{3}$. Suppose it was already there and was held

stuck there. What would the sequence of $R_t - \bar{R}_t$ errors be? What would the sequence of δ_t errors be (using (10.10))? Which error sequence would produce a more stable estimate of the average reward if the estimate were allowed to change in response to the errors? Why? □

10.4 Deprecating the Discounted Setting

The continuing, discounted problem formulation has been very useful in the tabular case, in which the returns from each state can be separately identified and averaged. But in the approximate case it is questionable whether one should ever use this problem formulation.

To see why, consider an infinite sequence of returns with no beginning or end, and no clearly identified states. The states might be represented only by feature vectors, which may do little to distinguish the states from each other. As a special case, all of the feature vectors may be the same. Thus one really has only the reward sequence (and the actions), and performance has to be assessed purely from these. How could it be done? One way is by averaging the rewards over a long interval—this is the idea of the average-reward setting. How could discounting be used? Well, for each time step we could measure the discounted return. Some returns would be small and some big, so again we would have to average them over a sufficiently large time interval. In the continuing setting there are no starts and ends, and no special time steps, so there is nothing else that could be done. However, if you do this, it turns out that the average of the discounted returns is proportional to the average reward. In fact, for policy π, the average of the discounted returns is always $r(\pi)/(1-\gamma)$, that is, it is essentially the average reward, $r(\pi)$. In particular, the *ordering* of all policies in the average discounted return setting would be exactly the same as in the average-reward setting. The discount rate γ thus has no effect on the problem formulation. It could in fact be *zero* and the ranking would be unchanged.

This surprising fact is proven in the box on the next page, but the basic idea can be seen via a symmetry argument. Each time step is exactly the same as every other. With discounting, every reward will appear exactly once in each position in some return. The tth reward will appear undiscounted in the $t-1$st return, discounted once in the $t-2$nd return, and discounted 999 times in the $t-1000$th return. The weight on the tth reward is thus $1+\gamma+\gamma^2+\gamma^3+\cdots = 1/(1-\gamma)$. Because all states are the same, they are all weighted by this, and thus the average of the returns will be this times the average reward, or $r(\pi)/(1-\gamma)$.

This example and the more general argument in the box show that if we optimized discounted value over the on-policy distribution, then the effect would be identical to optimizing *undiscounted* average reward; the actual value of γ would have no effect. This strongly suggests that discounting has no role to play in the definition of the control problem with function approximation. One can nevertheless go ahead and use discounting in solution methods. The discounting parameter γ changes from a problem parameter to a solution method parameter! However, in this case we unfortunately would not be guaranteed to optimize average reward (or the equivalent discounted value over the on-policy distribution).

The Futility of Discounting in Continuing Problems

Perhaps discounting can be saved by choosing an objective that sums discounted values over the distribution with which states occur under the policy:

$$
\begin{aligned}
J(\pi) &= \sum_s \mu_\pi(s) v_\pi^\gamma(s) && \text{(where } v_\pi^\gamma \text{ is the discounted value function)} \\
&= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \left[r + \gamma v_\pi^\gamma(s') \right] && \text{(Bellman Eq.)} \\
&= r(\pi) + \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \gamma v_\pi^\gamma(s') && \text{(from (10.7))} \\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s' \,|\, s, a) && \text{(from (3.4))} \\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \mu_\pi(s') && \text{(from (10.8))} \\
&= r(\pi) + \gamma J(\pi) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 J(\pi) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 r(\pi) + \gamma^3 r(\pi) + \cdots \\
&= \frac{1}{1-\gamma} r(\pi).
\end{aligned}
$$

The proposed discounted objective orders policies identically to the undiscounted (average reward) objective. The discount rate γ does not influence the ordering!

The root cause of the difficulties with the discounted control setting is that with function approximation we have lost the policy improvement theorem (Section 4.2). It is no longer true that if we change the policy to improve the discounted value of one state then we are guaranteed to have improved the overall policy in any useful sense. That guarantee was key to the theory of our reinforcement learning control methods. With function approximation we have lost it!

In fact, the lack of a policy improvement theorem is also a theoretical lacuna for the total-episodic and average-reward settings. Once we introduce function approximation we can no longer guarantee improvement for any setting. In Chapter 13 we introduce an alternative class of reinforcement learning algorithms based on parameterized policies, and there we have a theoretical guarantee called the "policy-gradient theorem" which plays a similar role as the policy improvement theorem. But for methods that learn action values we seem to be currently without a local improvement guarantee (possibly the approach taken by Perkins and Precup (2003) may provide a part of the answer). We do know that ε-greedification may sometimes result in an inferior policy, as policies may chatter among good policies rather than converge (Gordon, 1996a). This is an area with multiple open theoretical questions.

10.5 Differential Semi-gradient n-step Sarsa

In order to generalize to *n-step* bootstrapping, we need an n-step version of the TD error. We begin by generalizing the n-step return (7.4) to its differential form, with function approximation:

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_{t+1} + R_{t+2} - \bar{R}_{t+2} + \cdots + R_{t+n} - \bar{R}_{t+n} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \tag{10.14}$$

where $\bar{R}$ is an estimate of $r(\pi)$, $n \geq 1$, and $t + n < T$. If $t + n \geq T$, then we define $G_{t:t+n} \doteq G_t$ as usual. The n-step TD error is then

$$\delta_t \doteq G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}), \tag{10.15}$$

after which we can apply our usual semi-gradient Sarsa update (10.12). Pseudocode for the complete algorithm is given in the box.

Differential semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_\pi$ or q_*

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$, a policy π
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average-reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)
Algorithm parameters: step size $\alpha, \beta > 0$, a positive integer n
All store and access operations (S_t, A_t, and R_t) can take their index mod $n + 1$

Initialize and store S_0 and A_0
Loop for each step, $t = 0, 1, 2, \ldots$:
  Take action A_t
  Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$, or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  If $\tau \geq 0$:
    $\delta \leftarrow \sum_{i=\tau+1}^{\tau+n} (R_i - \bar{R}) + \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}) - \hat{q}(S_\tau, A_\tau, \mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \beta\delta$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S_\tau, A_\tau, \mathbf{w})$

Exercise 10.9 In the differential semi-gradient n-step Sarsa algorithm, the step-size parameter on the average reward, β, needs to be quite small so that $\bar{R}$ becomes a good long-term estimate of the average reward. Unfortunately, $\bar{R}$ will then be biased by its initial value for many steps, which may make learning inefficient. Alternatively, one could use a sample average of the observed rewards for $\bar{R}$. That would initially adapt rapidly but in the long run would also adapt slowly. As the policy slowly changed, $\bar{R}$ would also change; the potential for such long-term nonstationarity makes sample-average methods ill-suited. In fact, the step-size parameter on the average reward is a perfect place to use the unbiased constant-step-size trick from Exercise 2.7. Describe the specific changes needed to the boxed algorithm for differential semi-gradient n-step Sarsa to use this trick. □

10.6 Summary

In this chapter we have extended the ideas of parameterized function approximation and semi-gradient descent, introduced in the previous chapter, to control. The extension is immediate for the episodic case, but for the continuing case we have to introduce a whole new problem formulation based on maximizing the *average reward setting* per time step. Surprisingly, the discounted formulation cannot be carried over to control in the presence of approximations. In the approximate case most policies cannot be represented by a value function. The arbitrary policies that remain need to be ranked, and the scalar average reward $r(\pi)$ provides an effective way to do this.

The average reward formulation involves new *differential* versions of value functions, Bellman equations, and TD errors, but all of these parallel the old ones, and the conceptual changes are small. There is also a new parallel set of differential algorithms for the average-reward case.

Bibliographical and Historical Remarks

10.1 Semi-gradient Sarsa with function approximation was first explored by Rummery and Niranjan (1994). Linear semi-gradient Sarsa with ε-greedy action selection does not converge in the usual sense, but does enter a bounded region near the best solution (Gordon, 1996a, 2001). Precup and Perkins (2003) showed convergence in a differentiable action selection setting. See also Perkins and Pendrith (2002) and Melo, Meyn, and Ribeiro (2008). The mountain–car example is based on a similar task studied by Moore (1990), but the exact form used here is from Sutton (1996).

10.2 Episodic n-step semi-gradient Sarsa is based on the forward Sarsa(λ) algorithm of van Seijen (2016). The empirical results shown here are new to the second edition of this text.

10.3 The average-reward formulation has been described for dynamic programming (e.g., Puterman, 1994) and from the point of view of reinforcement learning (Mahadevan, 1996; Tadepalli and Ok, 1994; Bertsekas and Tsitiklis, 1996; Tsitsiklis and Van Roy, 1999). The algorithm described here is the on-policy analog of the "R-learning" algorithm introduced by Schwartz (1993). The name R-learning was probably meant to be the alphabetic successor to Q-learning, but we prefer to think of it as a reference to the learning of differential or *relative* values. The access-control queuing example was suggested by the work of Carlström and Nordström (1997).

10.4 The recognition of the limitations of discounting as a formulation of the reinforcement learning problem with function approximation became apparent to the authors shortly after the publication of the first edition of this text. Singh, Jaakkola, and Jordan (1994) may have been the first to observe it in print.

Chapter 11

*Off-policy Methods with Approximation

This book has treated on-policy and off-policy learning methods since Chapter 5 primarily as two alternative ways of handling the conflict between exploitation and exploration inherent in learning forms of generalized policy iteration. The two chapters preceding this have treated the *on*-policy case with function approximation, and in this chapter we treat the *off*-policy case with function approximation. The extension to function approximation turns out to be significantly different and harder for off-policy learning than it is for on-policy learning. The tabular off-policy methods developed in Chapters 6 and 7 readily extend to semi-gradient algorithms, but these algorithms do not converge as robustly as they do under on-policy training. In this chapter we explore the convergence problems, take a closer look at the theory of linear function approximation, introduce a notion of learnability, and then discuss new algorithms with stronger convergence guarantees for the off-policy case. In the end we will have improved methods, but the theoretical results will not be as strong, nor the empirical results as satisfying, as they are for on-policy learning. Along the way, we will gain a deeper understanding of approximation in reinforcement learning for on-policy learning as well as off-policy learning.

Recall that in off-policy learning we seek to learn a value function for a *target policy* π, given data due to a different *behavior policy* b. In the prediction case, both policies are static and given, and we seek to learn either state values $\hat{v} \approx v_\pi$ or action values $\hat{q} \approx q_\pi$. In the control case, action values are learned, and both policies typically change during learning—π being the greedy policy with respect to $\hat{q}$, and b being something more exploratory such as the ε-greedy policy with respect to $\hat{q}$.

The challenge of off-policy learning can be divided into two parts, one that arises in the tabular case and one that arises only with function approximation. The first part of the challenge has to do with the target of the update (not to be confused with the target policy), and the second part has to do with the distribution of the updates. The techniques related to importance sampling developed in Chapters 5 and 7 deal with the first part; these may increase variance but are needed in all successful algorithms,

tabular and approximate. The extension of these techniques to function approximation are quickly dealt with in the first section of this chapter.

Something more is needed for the second part of the challenge of off-policy learning with function approximation because the distribution of updates in the off-policy case is not according to the on-policy distribution. The on-policy distribution is important to the stability of semi-gradient methods. Two general approaches have been explored to deal with this. One is to use importance sampling methods again, this time to warp the update distribution back to the on-policy distribution, so that semi-gradient methods are guaranteed to converge (in the linear case). The other is to develop true gradient methods that do not rely on any special distribution for stability. We present methods based on both approaches. This is a cutting-edge research area, and it is not clear which of these approaches is most effective in practice.

11.1 Semi-gradient Methods

We begin by describing how the methods developed in earlier chapters for the off-policy case extend readily to function approximation as semi-gradient methods. These methods address the first part of the challenge of off-policy learning (changing the update targets) but not the second part (changing the update distribution). Accordingly, these methods may diverge in some cases, and in that sense are not sound, but still they are often successfully used. Remember that these methods *are* guaranteed stable and asymptotically unbiased for the tabular case, which corresponds to a special case of function approximation. So it may still be possible to combine them with feature selection methods in such a way that the combined system could be assured stable. In any event, these methods are simple and thus a good place to start.

In Chapter 7 we described a variety of tabular off-policy algorithms. To convert them to semi-gradient form, we simply replace the update to an array (V or Q) to an update to a weight vector ($\mathbf{w}$), using the approximate value function ($\hat{v}$ or $\hat{q}$) and its gradient. Many of these algorithms use the per-step importance sampling ratio:

$$\rho_t \doteq \rho_{t:t} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}. \tag{11.1}$$

For example, the one-step, state-value algorithm is semi-gradient off-policy TD(0), which is just like the corresponding on-policy algorithm (page 203) except for the addition of ρ_t:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\rho_t\delta_t\nabla\hat{v}(S_t,\mathbf{w}_t), \tag{11.2}$$

where δ_t is defined appropriately depending on whether the problem is episodic and discounted, or continuing and undiscounted using average reward:

$$\delta_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t), \text{ or} \tag{11.3}$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t). \tag{11.4}$$

For action values, the one-step algorithm is semi-gradient Expected Sarsa:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \text{ with} \tag{11.5}$$

$$\delta_t \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \text{ or} \quad \text{(episodic)}$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t). \quad \text{(continuing)}$$

Note that this algorithm does not use importance sampling. In the tabular case it is clear that this is appropriate because the only sample action is A_t, and in learning its value we do not have to consider any other actions. With function approximation it is less clear because we might want to weight different state–action pairs differently once they all contribute to the same overall approximation. Proper resolution of this issue awaits a more thorough understanding of the theory of function approximation in reinforcement learning.

In the multi-step generalizations of these algorithms, both the state-value and action-value algorithms involve importance sampling. For example, the n-step version of semi-gradient Expected Sarsa is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\rho_{t+1}\cdots\rho_{t+n-1}\left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) \tag{11.6}$$

with

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n\hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \text{ or} \quad \text{(episodic)}$$

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_t + \cdots + R_{t+n} - \bar{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad \text{(continuing)}$$

where here we are being slightly informal in our treatment of the ends of episodes. In the first equation, the ρ_ks for $k \geq T$ (where T is the last time step of the episode) should be taken to be 1, and $G_{t:n}$ should be taken to be G_t if $t + n \geq T$.

Recall that we also presented in Chapter 7 an off-policy algorithm that does not involve importance sampling at all: the n-step tree-backup algorithm. Here is its semi-gradient version:

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \tag{11.7}$$

$$G_{t:t+n} \doteq \hat{q}(S_t, A_t, \mathbf{w}_{t-1}) + \sum_{k=t}^{t+n-1}\delta_k \prod_{i=t+1}^{k}\gamma\pi(A_i|S_i), \tag{11.8}$$

with δ_t as defined at the top of this page for Expected Sarsa. We also defined in Chapter 7 an algorithm that unifies all action-value algorithms: n-step $Q(\sigma)$. We leave the semi-gradient form of that algorithm, and also of the n-step state-value algorithm, as exercises for the reader.

Exercise 11.1 Convert the equation of n-step off-policy TD (7.9) to semi-gradient form. Give accompanying definitions of the return for both the episodic and continuing cases. □

**Exercise 11.2* Convert the equations of n-step $Q(\sigma)$ (7.11 and 7.17) to semi-gradient form. Give definitions that cover both the episodic and continuing cases. □

11.2 Examples of Off-policy Divergence

In this section we begin to discuss the second part of the challenge of off-policy learning with function approximation—that the distribution of updates does not match the on-policy distribution. We describe some instructive counterexamples to off-policy learning—cases where semi-gradient and other simple algorithms are unstable and diverge.

To establish intuitions, it is best to consider first a very simple example. Suppose, perhaps as part of a larger MDP, there are two states whose estimated values are of the functional form w and $2w$, where the parameter vector $\mathbf{w}$ consists of only a single component w. This occurs under linear function approximation if the feature vectors for the two states are each simple numbers (single-component vectors), in this case 1 and 2. In the first state, only one action is available, and it results deterministically in a transition to the second state with a reward of 0:

where the expressions inside the two circles indicate the two state's values.

Suppose initially $w = 10$. The transition will then be from a state of estimated value 10 to a state of estimated value 20. It will look like a good transition, and w will be increased to raise the first state's estimated value. If γ is nearly 1, then the TD error will be nearly 10, and, if $\alpha = 0.1$, then w will be increased to nearly 11 in trying to reduce the TD error. However, the second state's estimated value will also be increased, to nearly 22. If the transition occurs again, then it will be from a state of estimated value ≈11 to a state of estimated value ≈22, for a TD error of ≈11—larger, not smaller, than before. It will look even more like the first state is undervalued, and its value will be increased again, this time to ≈12.1. This looks bad, and in fact with further updates w will diverge to infinity.

To see this definitively we have to look more carefully at the sequence of updates. The TD error on a transition between the two states is

$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t) = 0 + \gamma 2w_t - w_t = (2\gamma - 1)w_t,$$

and the off-policy semi-gradient TD(0) update (from (11.2)) is

$$w_{t+1} = w_t + \alpha\rho_t\delta_t\nabla\hat{v}(S_t,w_t) = w_t + \alpha\cdot 1\cdot(2\gamma-1)w_t\cdot 1 = \big(1 + \alpha(2\gamma - 1)\big)w_t.$$

Note that the importance sampling ratio, ρ_t, is 1 on this transition because there is only one action available from the first state, so its probabilities of being taken under the target and behavior policies must both be 1. In the final update above, the new parameter is the old parameter times a scalar constant, $1 + \alpha(2\gamma - 1)$. If this constant is greater than 1, then the system is unstable and w will go to positive or negative infinity depending on its initial value. Here this constant is greater than 1 whenever $\gamma > 0.5$. Note that stability does not depend on the specific step size, as long as $\alpha > 0$. Smaller or larger step sizes would affect the rate at which w goes to infinity, but not whether it goes there or not.

Key to this example is that the one transition occurs repeatedly without w being updated on other transitions. This is possible under off-policy training because the

behavior policy might select actions on those other transitions which the target policy never would. For these transitions, ρ_t would be zero and no update would be made. Under on-policy training, however, ρ_t is always one. Each time there is a transition from the w state to the $2w$ state, increasing w, there would also have to be a transition out of the $2w$ state. That transition would reduce w, unless it were to a state whose value was higher (because $\gamma < 1$) than $2w$, and then that state would have to be followed by a state of even higher value, or else again w would be reduced. Each state can support the one before only by creating a higher expectation. Eventually the piper must be paid. In the on-policy case the promise of future reward must be kept and the system is kept in check. But in the off-policy case, a promise can be made and then, after taking an action that the target policy never would, forgotten and forgiven.

This simple example communicates much of the reason why off-policy training can lead to divergence, but it is not completely convincing because it is not complete—it is just a fragment of a complete MDP. Can there really be a complete system with instability? A simple complete example of divergence is *Baird's counterexample*. Consider the episodic seven-state, two-action MDP shown in Figure 11.1. The dashed action takes the system to one of the six upper states with equal probability, whereas the solid action takes the system to the seventh state. The behavior policy b selects the dashed and solid actions with probabilities $\frac{6}{7}$ and $\frac{1}{7}$, so that the next-state distribution under it is uniform (the same for all nonterminal states), which is also the starting distribution for each episode. The target policy π always takes the solid action, and so the on-policy distribution (for π) is concentrated in the seventh state. The reward is zero on all transitions. The discount rate is $\gamma = 0.99$.

Consider estimating the state-value under the linear parameterization indicated by the expression shown in each state circle. For example, the estimated value of the leftmost state is $2w_1 + w_8$, where the subscript corresponds to the component of the

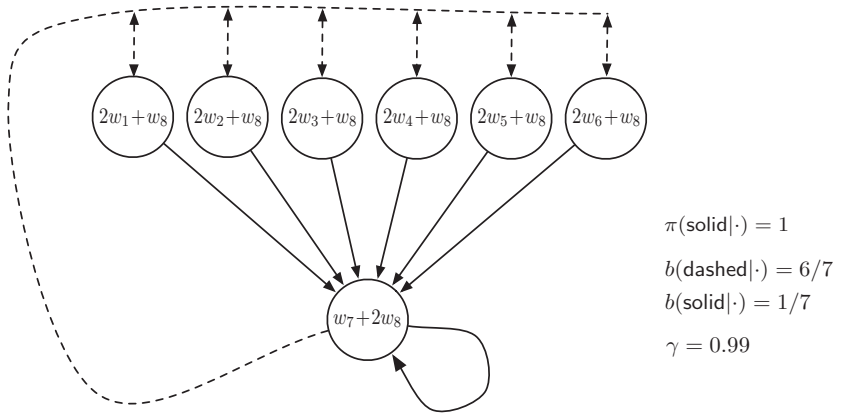


Figure 11.1: Baird's counterexample. The approximate state-value function for this Markov process is of the form shown by the linear expressions inside each state. The solid action usually results in the seventh state, and the dashed action usually results in one of the other six states, each with equal probability. The reward is always zero.

overall weight vector $\mathbf{w} \in \mathbb{R}^8$; this corresponds to a feature vector for the first state being $\mathbf{x}(1) = (2, 0, 0, 0, 0, 0, 0, 1)^\top$. The reward is zero on all transitions, so the true value function is $v_\pi(s) = 0$, for all s, which can be exactly approximated if $\mathbf{w} = \mathbf{0}$. In fact, there are many solutions, as there are more components to the weight vector (8) than there are nonterminal states (7). Moreover, the set of feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, is a linearly independent set. In all these ways this task seems a favorable case for linear function approximation.

If we apply semi-gradient TD(0) to this problem (11.2), then the weights diverge to infinity, as shown in Figure 11.2 (left). The instability occurs for any positive step size, no matter how small. In fact, it even occurs if an expected update is done as in dynamic programming (DP), as shown in Figure 11.2 (right). That is, if the weight vector, $\mathbf{w}_k$, is updated for all states at the same time in a semi-gradient way, using the DP (expectation-based) target:

$$\mathbf{w}_{k+1} \doteq \mathbf{w}_k + \frac{\alpha}{|\mathcal{S}|} \sum_s \Big(\mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_k) \mid S_t = s] - \hat{v}(s, \mathbf{w}_k) \Big) \nabla \hat{v}(s, \mathbf{w}_k). \quad (11.9)$$

In this case, there is no randomness and no asynchrony, just as in a classical DP update. The method is conventional except in its use of semi-gradient function approximation. Yet still the system is unstable.

If we alter just the distribution of DP updates in Baird's counterexample, from the uniform distribution to the on-policy distribution (which generally requires asynchronous updating), then convergence is guaranteed to a solution with error bounded by (9.14). This example is striking because the TD and DP methods used are arguably the simplest

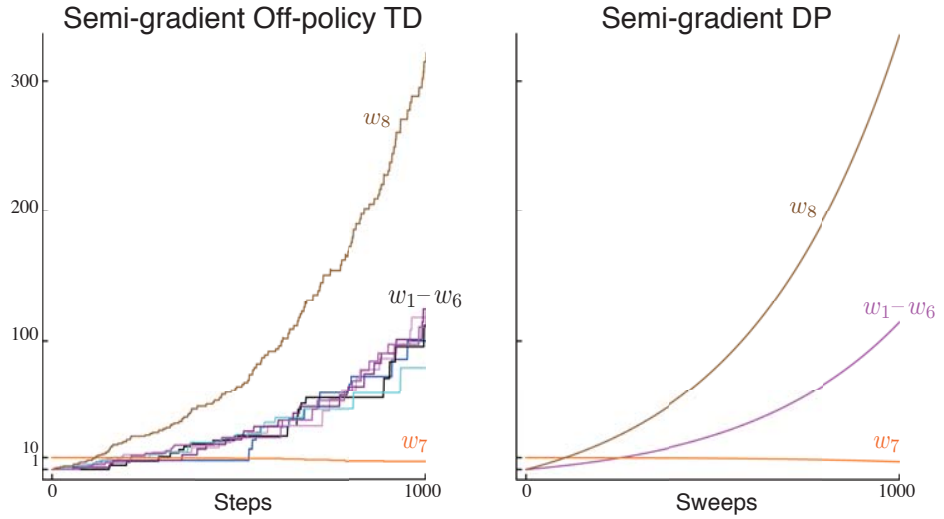


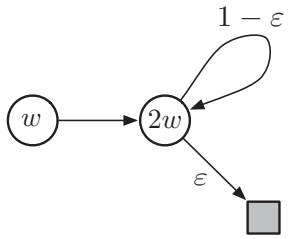
Figure 11.2: Demonstration of instability on Baird's counterexample. Shown are the evolution of the components of the parameter vector $\mathbf{w}$ of the two semi-gradient algorithms. The step size was $\alpha = 0.01$, and the initial weights were $\mathbf{w} = (1, 1, 1, 1, 1, 1, 10, 1)^\top$.

and best-understood bootstrapping methods, and the linear, semi-descent method used is arguably the simplest and best-understood kind of function approximation. The example shows that even the simplest combination of bootstrapping and function approximation can be unstable if the updates are not done according to the on-policy distribution.

There are also counterexamples similar to Baird's showing divergence for Q-learning. This is cause for concern because otherwise Q-learning has the best convergence guarantees of all control methods. Considerable effort has gone into trying to find a remedy to this problem or to obtain some weaker, but still workable, guarantee. For example, it may be possible to guarantee convergence of Q-learning as long as the behavior policy is sufficiently close to the target policy, for example, when it is the ε-greedy policy. To the best of our knowledge, Q-learning has never been found to diverge in this case, but there has been no theoretical analysis. In the rest of this section we present several other ideas that have been explored.

Suppose that instead of taking just a step toward the expected one-step return on each iteration, as in Baird's counterexample, we actually change the value function all the way to the best, least-squares approximation. Would this solve the instability problem? Of course it would if the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, formed a linearly independent set, as they do in Baird's counterexample, because then exact approximation is possible on each iteration and the method reduces to standard tabular DP. But of course the point here is to consider the case when an exact solution is *not* possible. In this case stability is not guaranteed even when forming the best approximation at each iteration, as shown in the example.

Example 11.1: Tsitsiklis and Van Roy's Counterexample This example shows that linear function approximation would not work with DP even if the least-squares solution was found at each step. The counterexample is formed by extending the w-to-$2w$ example (from earlier in this section) with a terminal state, as shown to the right. As before, the estimated value of the first state is w, and the estimated value of the second state is $2w$. The reward is zero on all transitions, so the true values are zero at both states, which is exactly representable with $w = 0$. If we set w_{k+1} at each step so as to minimize the $\overline{\text{VE}}$ between the estimated value and the expected one-step return, then we have



$$\begin{aligned} w_{k+1} &= \operatorname*{arg\,min}_{w\in\mathbb{R}} \sum_{s\in\mathcal{S}} \Big(\hat{v}(s,w) - \mathbb{E}_\pi\big[R_{t+1} + \gamma\hat{v}(S_{t+1},w_k) \;\big|\; S_t = s\big]\Big)^2 \\ &= \operatorname*{arg\,min}_{w\in\mathbb{R}} \big(w - \gamma 2w_k\big)^2 + \big(2w - (1-\varepsilon)\gamma 2w_k\big)^2 \\ &= \frac{6-4\varepsilon}{5}\gamma w_k. \end{aligned} \tag{11.10}$$

The sequence $\{w_k\}$ diverges when $\gamma > \frac{5}{6-4\varepsilon}$ and $w_0 \neq 0$. ■

Another way to try to prevent instability is to use special methods for function approximation. In particular, stability is guaranteed for function approximation methods that do not extrapolate from the observed targets. These methods, called *averagers*, include nearest neighbor methods and locally weighted regression, but not popular methods such as tile coding and artificial neural networks (ANNs).

Exercise 11.3 (programming) Apply one-step semi-gradient Q-learning to Baird's counterexample and show empirically that its weights diverge. □

11.3 The Deadly Triad

Our discussion so far can be summarized by saying that the danger of instability and divergence arises whenever we combine all of the following three elements, making up what we call *the deadly triad*:

Function approximation A powerful, scalable way of generalizing from a state space much larger than the memory and computational resources (e.g., linear function approximation or ANNs).

Bootstrapping Update targets that include existing estimates (as in dynamic programming or TD methods) rather than relying exclusively on actual rewards and complete returns (as in MC methods).

Off-policy training Training on a distribution of transitions other than that produced by the target policy. Sweeping through the state space and updating all states uniformly, as in dynamic programming, does not respect the target policy and is an example of off-policy training.

In particular, note that the danger is *not* due to control or to generalized policy iteration. Those cases are more complex to analyze, but the instability arises in the simpler prediction case whenever it includes all three elements of the deadly triad. The danger is also *not* due to learning or to uncertainties about the environment, because it occurs just as strongly in planning methods, such as dynamic programming, in which the environment is completely known.

If any two elements of the deadly triad are present, but not all three, then instability can be avoided. It is natural, then, to go through the three and see if there is any one that can be given up.

Of the three, *function approximation* most clearly cannot be given up. We need methods that scale to large problems and to great expressive power. We need at least linear function approximation with many features and parameters. State aggregation or nonparametric methods whose complexity grows with data are too weak or too expensive. Least-squares methods such as LSTD are of quadratic complexity and are therefore too expensive for large problems.

Doing without *bootstrapping* is possible, at the cost of computational and data efficiency. Perhaps most important are the losses in computational efficiency. Monte Carlo (non-bootstrapping) methods require memory to save everything that happens between making

each prediction and obtaining the final return, and all their computation is done once the final return is obtained. The cost of these computational issues is not apparent on serial von Neumann computers, but would be on specialized hardware. With bootstrapping and eligibility traces (Chapter 12), data can be dealt with when and where it is generated, then need never be used again. The savings in communication and memory made possible by bootstrapping are great.

The losses in data efficiency by giving up *bootstrapping* are also significant. We have seen this repeatedly, such as in Chapters 7 (Figure 7.2) and 9 (Figure 9.2), where some degree of bootstrapping performed much better than Monte Carlo methods on the random-walk prediction task, and in Chapter 10 where the same was seen on the Mountain-Car control task (Figure 10.4). Many other problems show much faster learning with bootstrapping (e.g., see Figure 12.14). Bootstrapping often results in faster learning because it allows learning to take advantage of the state property, the ability to recognize a state upon returning to it. On the other hand, bootstrapping can impair learning on problems where the state representation is poor and causes poor generalization (e.g., this seems to be the case on Tetris, see Şimşek, Algórta, and Kothiyal, 2016). A poor state representation can also result in bias; this is the reason for the poorer bound on the asymptotic approximation quality of bootstrapping methods (Equation 9.14). On balance, the ability to bootstrap has to be considered extremely valuable. One may sometimes choose not to use it by selecting long n-step updates (or a large bootstrapping parameter, $\lambda \approx 1$; see Chapter 12) but often bootstrapping greatly increases efficiency. It is an ability that we would very much like to keep in our toolkit.

Finally, there is *off-policy learning*; can we give that up? On-policy methods are often adequate. For model-free reinforcement learning, one can simply use Sarsa rather than Q-learning. Off-policy methods free behavior from the target policy. This could be considered an appealing convenience but not a necessity. However, off-policy learning *is essential* to other anticipated use cases, cases that we have not yet mentioned in this book but may be important to the larger goal of creating a powerful intelligent agent.

In these use cases, the agent learns not just a single value function and single policy, but large numbers of them in parallel. There is extensive psychological evidence that people and animals learn to predict many different sensory events, not just rewards. We can be surprised by unusual events, and correct our predictions about them, even if they are of neutral valence (neither good nor bad). This kind of prediction presumably underlies predictive models of the world such as are used in planning. We predict what we will see after eye movements, how long it will take to walk home, the probability of making a jump shot in basketball, and the satisfaction we will get from taking on a new project. In all these cases, the events we would like to predict depend on our acting in a certain way. To learn them all, in parallel, requires learning from the one stream of experience. There are many target policies, and thus the one behavior policy cannot equal all of them. Yet parallel learning is conceptually possible because the behavior policy may overlap in part with many of the target policies. To take full advantage of this requires off-policy learning.

11.4 Linear Value-function Geometry

To better understand the stability challenge of off-policy learning, it is helpful to think about value function approximation more abstractly and independently of how learning is done. We can imagine the space of all possible state-value functions—all functions from states to real numbers $v : \mathcal{S} \to \mathbb{R}$. Most of these value functions do not correspond to any policy. More important for our purposes is that most are not representable by the function approximator, which by design has far fewer parameters than there are states.

Given an enumeration of the state space $\mathcal{S} = \{s_1, s_2, \ldots, s_{|\mathcal{S}|}\}$, any value function v corresponds to a vector listing the value of each state in order $[v(s_1), v(s_2), \ldots, v(s_{|\mathcal{S}|})]^\top$. This vector representation of a value function has as many components as there are states. In most cases where we want to use function approximation, this would be far too many components to represent the vector explicitly. Nevertheless, the idea of this vector is conceptually useful. In the following, we treat a value function and its vector representation interchangeably.

To develop intuitions, consider the case with three states $\mathcal{S} = \{s_1, s_2, s_3\}$ and two parameters $\mathbf{w} = (w_1, w_2)^\top$. We can then view all value functions/vectors as points in a three-dimensional space. The parameters provide an alternative coordinate system over a two-dimensional subspace. Any weight vector $\mathbf{w} = (w_1, w_2)^\top$ is a point in the two-dimensional subspace and thus also a complete value function $v_\mathbf{w}$ that assigns values to all three states. With general function approximation the relationship between the full space and the subspace of representable functions could be complex, but in the case of *linear* value-function approximation the subspace is a simple plane, as suggested by Figure 11.3.

Now consider a single fixed policy π. We assume that its true value function, v_π, is too complex to be represented exactly as an approximation. Thus v_π is not in the subspace; in the figure it is depicted as being above the planar subspace of representable functions.

If v_π cannot be represented exactly, what representable value function is closest to it? This turns out to be a subtle question with multiple answers. To begin, we need a measure of the distance between two value functions. Given two value functions v_1 and v_2, we can talk about the vector difference between them, $v = v_1 - v_2$. If v is small, then the two value functions are close to each other. But how are we to measure the size of this difference vector? The conventional Euclidean norm is not appropriate because, as discussed in Section 9.2, some states are more important than others because they occur more frequently or because we are more interested in them (Section 9.11). As in Section 9.2, let us use the distribution $\mu : \mathcal{S} \to [0, 1]$ to specify the degree to which we care about different states being accurately valued (often taken to be the on-policy distribution). We can then define the distance between value functions using the norm

$$\|v\|_\mu^2 \doteq \sum_{s \in \mathcal{S}} \mu(s) v(s)^2. \tag{11.11}$$

Note that the $\overline{\text{VE}}$ from Section 9.2 can be written simply using this norm as $\overline{\text{VE}}(\mathbf{w}) = \|v_\mathbf{w} - v_\pi\|_\mu^2$. For any value function v, the operation of finding its closest value function in the subspace of representable value functions is a projection operation. We define a

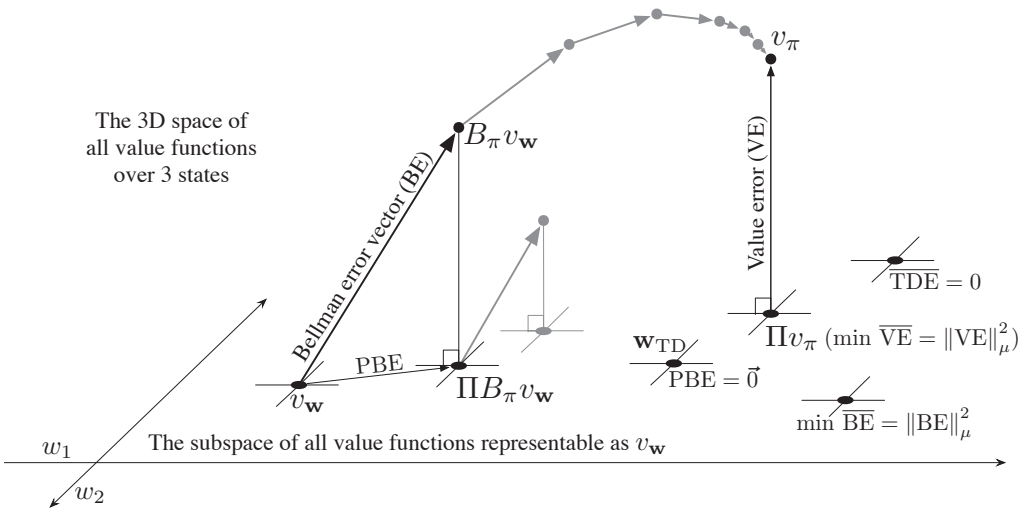


Figure 11.3: The geometry of linear value-function approximation. Shown is the three-dimensional space of all value functions over three states, while shown as a plane is the subspace of all value functions representable by a linear function approximator with parameter $\mathbf{w} = (w_1, w_2)^\top$. The true value function v_π is in the larger space and can be projected down (into the subspace, using a projection operator Π) to its best approximation in the value error (VE) sense. The best approximators in the Bellman error (BE), projected Bellman error (PBE), and temporal difference error (TDE) senses are all potentially different and are shown in the lower right. (VE, BE, and PBE are all treated as the corresponding vectors in this figure.) The Bellman operator takes a value function in the plane to one outside, which can then be projected back. If you iteratively applied the Bellman operator outside the space (shown in gray above) you would reach the true value function, as in conventional dynamic programming. If instead you kept projecting back into the subspace at each step, as in the lower step shown in gray, then the fixed point would be the point of vector-zero PBE.

projection operator Π that takes an arbitrary value function to the representable function that is closest in our norm:

$$\Pi v \doteq v_{\mathbf{w}} \quad \text{where} \quad \mathbf{w} = \operatorname*{arg\,min}_{\mathbf{w} \in \mathbb{R}^d} \|v - v_{\mathbf{w}}\|^2_\mu . \tag{11.12}$$

The representable value function that is closest to the true value function v_π is thus its projection, Πv_π, as suggested in Figure 11.3. This is the solution asymptotically found by Monte Carlo methods, albeit often very slowly. The projection operation is discussed more fully in the box on the next page.

TD methods find different solutions. To understand their rationale, recall that the Bellman equation for value function v_π is

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \left[r + \gamma v_\pi(s') \right], \qquad \text{for all } s \in \mathcal{S}. \tag{11.13}$$

The projection matrix

For a linear function approximator, the projection operation is linear, which implies that it can be represented as an $|\mathcal{S}| \times |\mathcal{S}|$ matrix:

$$\Pi \doteq \mathbf{X} \left(\mathbf{X}^\top \mathbf{D} \mathbf{X} \right)^{-1} \mathbf{X}^\top \mathbf{D}, \tag{11.14}$$

where, as in Section 9.4, $\mathbf{D}$ denotes the $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with the $\mu(s)$ on the diagonal, and $\mathbf{X}$ denotes the $|\mathcal{S}| \times d$ matrix whose rows are the feature vectors $\mathbf{x}(s)^\top$, one for each state s. If the inverse in (11.14) does not exist, then the pseudoinverse is substituted. Using these matrices, the squared norm of a vector can be written

$$\|v\|^2_\mu = v^\top \mathbf{D} v, \tag{11.15}$$

and the approximate linear value function can be written

$$v_{\mathbf{w}} = \mathbf{X}\mathbf{w}. \tag{11.16}$$

The true value function v_π is the only value function that solves (11.13) exactly. If an approximate value function $v_{\mathbf{w}}$ were substituted for v_π, the difference between the right and left sides of the modified equation could be used as a measure of how far off $v_{\mathbf{w}}$ is from v_π. We call this the *Bellman error* at state s:

$$\bar{\delta}_{\mathbf{w}}(s) \doteq \left(\sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \left[r + \gamma v_{\mathbf{w}}(s') \right] \right) - v_{\mathbf{w}}(s) \tag{11.17}$$

$$= \mathbb{E}_\pi \big[R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t) \;\big|\; S_t = s, A_t \sim \pi \big], \tag{11.18}$$

which shows clearly the relationship of the Bellman error to the TD error (11.3). The Bellman error is the expectation of the TD error.

The vector of all the Bellman errors, at all states, $\bar{\delta}_{\mathbf{w}} \in \mathbb{R}^{|\mathcal{S}|}$, is called the *Bellman error vector* (shown as BE in Figure 11.3). The overall size of this vector, in the norm, is an overall measure of the error in the value function, called the *Mean Squared Bellman Error*:

$$\overline{\text{BE}}(\mathbf{w}) = \left\| \bar{\delta}_{\mathbf{w}} \right\|^2_\mu . \tag{11.19}$$

It is not possible in general to reduce the $\overline{\text{BE}}$ to zero (at which point $v_{\mathbf{w}} = v_\pi$), but for linear function approximation there is a unique value of $\mathbf{w}$ for which the $\overline{\text{BE}}$ is minimized. This point in the representable-function subspace (labeled min $\overline{\text{BE}}$ in Figure 11.3) is different in general from that which minimizes the $\overline{\text{VE}}$ (shown as Πv_π). Methods that seek to minimize the $\overline{\text{BE}}$ are discussed in the next two sections.

The Bellman error vector is shown in Figure 11.3 as the result of applying the *Bellman operator* $B_\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ to the approximate value function. The Bellman operator is

defined by

$$(B_\pi v)(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \left[r + \gamma v(s') \right], \tag{11.20}$$

for all $s \in \mathcal{S}$ and $v : \mathcal{S} \to \mathbb{R}$. The Bellman error vector for v can be written $\bar{\delta}_{\mathbf{w}} = B_\pi v_{\mathbf{w}} - v_{\mathbf{w}}$.

If the Bellman operator is applied to a value function in the representable subspace, then, in general, it will produce a new value function that is outside the subspace, as suggested in the figure. In dynamic programming (without function approximation), this operator is applied repeatedly to the points outside the representable space, as suggested by the gray arrows in the top of Figure 11.3. Eventually that process converges to the true value function v_π, the only fixed point for the Bellman operator, the only value function for which

$$v_\pi = B_\pi v_\pi, \tag{11.21}$$

which is just another way of writing the Bellman equation for π (11.13).

With function approximation, however, the intermediate value functions lying outside the subspace cannot be represented. The gray arrows in the upper part of Figure 11.3 cannot be followed because after the first update (dark line) the value function must be projected back into something representable. The next iteration then begins within the subspace; the value function is again taken outside of the subspace by the Bellman operator and then mapped back by the projection operator, as suggested by the lower gray arrow and line. Following these arrows is a DP-like process with approximation.

In this case we are interested in the projection of the Bellman error vector back into the representable space. This is the projected Bellman error vector $\Pi\bar{\delta}_{v_{\mathbf{w}}}$, shown in Figure 11.3 as PBE. The size of this vector, in the norm, is another measure of error in the approximate value function. For any approximate value function v, we define the *Mean Square Projected Bellman Error*, denoted $\overline{\text{PBE}}$, as

$$\overline{\text{PBE}}(\mathbf{w}) = \left\| \Pi\bar{\delta}_{\mathbf{w}} \right\|_\mu^2. \tag{11.22}$$

With linear function approximation there always exists an approximate value function (within the subspace) with zero $\overline{\text{PBE}}$; this is the TD fixed point, $\mathbf{w}_{\text{TD}}$, introduced in Section 9.4. As we have seen, this point is not always stable under semi-gradient TD methods and off-policy training. As shown in the figure, this value function is generally different from those minimizing $\overline{\text{VE}}$ or $\overline{\text{BE}}$. Methods that are guaranteed to converge to it are discussed in Sections 11.7 and 11.8.

11.5 Gradient Descent in the Bellman Error

Armed with a better understanding of value function approximation and its various objectives, we return now to the challenge of stability in off-policy learning. We would like to apply the approach of stochastic gradient descent (SGD, Section 9.3), in which updates are made that in expectation are equal to the negative gradient of an objective

function. These methods always go downhill (in expectation) in the objective and because of this are typically stable with excellent convergence properties. Among the algorithms investigated so far in this book, only the Monte Carlo methods are true SGD methods. These methods converge robustly under both on-policy and off-policy training as well as for general nonlinear (differentiable) function approximators, though they are often slower than semi-gradient methods with bootstrapping, which are not SGD methods. Semi-gradient methods may diverge under off-policy training, as we have seen earlier in this chapter, and under contrived cases of nonlinear function approximation (Tsitsiklis and Van Roy, 1997). With a true SGD method such divergence would not be possible.

The appeal of SGD is so strong that great effort has gone into finding a practical way of harnessing it for reinforcement learning. The starting place of all such efforts is the choice of an error or objective function to optimize. In this and the next section we explore the origins and limits of the most popular proposed objective function, that based on the *Bellman error* introduced in the previous section. Although this has been a popular and influential approach, the conclusion that we reach here is that it is a misstep and yields no good learning algorithms. On the other hand, this approach fails in an interesting way that offers insight into what might constitute a good approach.

To begin, let us consider not the Bellman error, but something more immediate and naive. Temporal difference learning is driven by the TD error. Why not take the minimization of the expected square of the TD error as the objective? In the general function-approximation case, the one-step TD error with discounting is

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t).$$

A possible objective function then is what one might call the *Mean Squared TD Error*:

$$\begin{aligned}
\overline{\text{TDE}}(\mathbf{w}) &= \sum_{s \in \mathcal{S}} \mu(s) \mathbb{E}\left[\delta_t^2 \;\middle|\; S_t = s, A_t \sim \pi \right] \\
&= \sum_{s \in \mathcal{S}} \mu(s) \mathbb{E}\left[\rho_t \delta_t^2 \;\middle|\; S_t = s, A_t \sim b \right] \\
&= \mathbb{E}_b\left[\rho_t \delta_t^2 \right]. \qquad \text{(if } \mu \text{ is the distribution encountered under } b\text{)}
\end{aligned}$$

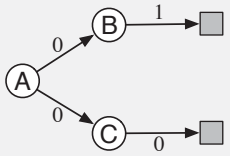
The last equation is of the form needed for SGD; it gives the objective as an expectation that can be sampled from experience (remember the experience is due to the behavior policy b). Thus, following the standard SGD approach, one can derive the per-step update based on a sample of this expected value:

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2} \alpha \nabla (\rho_t \delta_t^2) \\
&= \mathbf{w}_t - \alpha \rho_t \delta_t \nabla \delta_t \\
&= \mathbf{w}_t + \alpha \rho_t \delta_t \big(\nabla \hat{v}(S_t, \mathbf{w}_t) - \gamma \nabla \hat{v}(S_{t+1}, \mathbf{w}_t) \big),
\end{aligned} \tag{11.23}$$

which you will recognize as the same as the semi-gradient TD algorithm (11.2) except for the additional final term. This term completes the gradient and makes this a true SGD algorithm with excellent convergence guarantees. Let us call this algorithm the *naive*

residual-gradient algorithm (after Baird, 1995). Although the naive residual-gradient algorithm converges robustly, it does not necessarily converge to a desirable place.

Example 11.2: A-split example, showing the naiveté of the naive residual-gradient algorithm



Consider the three-state episodic MRP shown to the right. Episodes begin in state A and then 'split' stochastically, half the time going to B (and then invariably going on to terminate with a reward of 1) and half the time going to state C (and then invariably terminating with a reward of zero). Reward for the first transition, out of A, is always zero whichever way the episode goes. As this is an episodic problem, we can take γ to be 1. We also assume on-policy training, so that ρ_t is always 1, and tabular function approximation, so that the learning algorithms are free to give arbitrary, independent values to all three states. Thus, this should be an easy problem.

What should the values be? From A, half the time the return is 1, and half the time the return is 0; A should have value $\frac{1}{2}$. From B the return is always 1, so its value should be 1, and similarly from C the return is always 0, so its value should be 0. These are the true values and, as this is a tabular problem, all the methods presented previously converge to them exactly.

However, the naive residual-gradient algorithm finds different values for B and C. It converges with B having a value of $\frac{3}{4}$ and C having a value of $\frac{1}{4}$ (A converges correctly to $\frac{1}{2}$). These are in fact the values that minimize the $\overline{\text{TDE}}$.

Let us compute the $\overline{\text{TDE}}$ for these values. The first transition of each episode is either up from A's $\frac{1}{2}$ to B's $\frac{3}{4}$, a change of $\frac{1}{4}$, or down from A's $\frac{1}{2}$ to C's $\frac{1}{4}$, a change of $-\frac{1}{4}$. Because the reward is zero on these transitions, and $\gamma = 1$, these changes *are* the TD errors, and thus the squared TD error is always $\frac{1}{16}$ on the first transition. The second transition is similar; it is either up from B's $\frac{3}{4}$ to a reward of 1 (and a terminal state of value 0), or down from C's $\frac{1}{4}$ to a reward of 0 (again with a terminal state of value 0). Thus, the TD error is always $\pm\frac{1}{4}$, for a squared error of $\frac{1}{16}$ on the second step. Thus, for this set of values, the $\overline{\text{TDE}}$ on both steps is $\frac{1}{16}$.

Now let's compute the $\overline{\text{TDE}}$ for the true values (B at 1, C at 0, and A at $\frac{1}{2}$). In this case the first transition is either from $\frac{1}{2}$ up to 1, at B, or from $\frac{1}{2}$ down to 0, at C; in either case the absolute error is $\frac{1}{2}$ and the squared error is $\frac{1}{4}$. The second transition has zero error because the starting value, either 1 or 0 depending on whether the transition is from B or C, always exactly matches the immediate reward and return. Thus the squared TD error is $\frac{1}{4}$ on the first transition and 0 on the second, for a mean reward over the two transitions of $\frac{1}{8}$. As $\frac{1}{8}$ is bigger that $\frac{1}{16}$, this solution is worse according to the $\overline{\text{TDE}}$. On this simple problem, the true values do not have the smallest $\overline{\text{TDE}}$.

A tabular representation is used in the A-split example, so the true state values can be exactly represented, yet the naive residual-gradient algorithm finds different values, and these values have lower $\overline{\text{TDE}}$ than do the true values. Minimizing the $\overline{\text{TDE}}$ is naive; by penalizing all TD errors it achieves something more like temporal smoothing than accurate prediction.

A better idea would seem to be minimizing the Bellman error. If the exact values are learned, the Bellman error is zero everywhere. Thus, a Bellman-error-minimizing algorithm should have no trouble with the A-split example. We cannot expect to achieve zero Bellman error in general, as it would involve finding the true value function, which we presume is outside the space of representable value functions. But getting close to this ideal is a natural-seeming goal. As we have seen, the Bellman error is also closely related to the TD error. The Bellman error for a state is the expected TD error in that state. So let's repeat the derivation above with the expected TD error (all expectations here are implicitly conditional on S_t):

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla(\mathbb{E}_\pi[\delta_t]^2) \\
&= \mathbf{w}_t - \frac{1}{2}\alpha\nabla(\mathbb{E}_b[\rho_t\delta_t]^2) \\
&= \mathbf{w}_t - \alpha\mathbb{E}_b[\rho_t\delta_t]\,\nabla\mathbb{E}_b[\rho_t\delta_t] \\
&= \mathbf{w}_t - \alpha\mathbb{E}_b\big[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w}))\big]\,\mathbb{E}_b[\rho_t\nabla\delta_t] \\
&= \mathbf{w}_t + \alpha\Big[\mathbb{E}_b\big[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}))\big] - \hat{v}(S_t,\mathbf{w})\Big]\Big[\nabla\hat{v}(S_t,\mathbf{w}) - \gamma\mathbb{E}_b\big[\rho_t\nabla\hat{v}(S_{t+1},\mathbf{w})\big]\Big]
\end{aligned}$$

This update and various ways of sampling it are referred to as the *residual-gradient algorithm*. If you simply used the sample values in all the expectations, then the equation above reduces almost exactly to (11.23), the naive residual-gradient algorithm.[1] But this is naive, because the equation above involves the next state, S_{t+1}, appearing in two expectations that are multiplied together. To get an unbiased sample of the product, two independent samples of the next state are required, but during normal interaction with an external environment only one is obtained. One expectation or the other can be sampled, but not both.

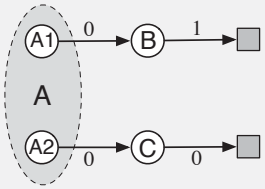
There are two ways to make the residual-gradient algorithm work. One is in the case of deterministic environments. If the transition to the next state is deterministic, then the two samples will necessarily be the same, and the naive algorithm is valid. The other way is to obtain *two* independent samples of the next state, S_{t+1}, from S_t, one for the first expectation and another for the second expectation. In real interaction with an environment, this would not seem possible, but when interacting with a simulated environment, it is. One simply rolls back to the previous state and obtains an alternate next state before proceeding forward from the first next state. In either of these cases the residual-gradient algorithm is guaranteed to converge to a minimum of the $\overline{\text{BE}}$ under the usual conditions on the step-size parameter. As a true SGD method, this convergence is

[1] For state values there remains a small difference in the treatment of the importance sampling ratio ρ_t. In the analagous action-value case (which is the most important case for control algorithms), the residual-gradient algorithm would reduce exactly to the naive version.

robust, applying to both linear and nonlinear function approximators. In the linear case, convergence is always to the *unique* $\mathbf{w}$ that minimizes the $\overline{\text{BE}}$.

However, there remain at least three ways in which the convergence of the residual-gradient method is unsatisfactory. The first of these is that empirically it is slow, much slower that semi-gradient methods. Indeed, proponents of this method have proposed increasing its speed by combining it with faster semi-gradient methods initially, then gradually switching over to residual gradient for the convergence guarantee (Baird and Moore, 1999). The second way in which the residual-gradient algorithm is unsatisfactory is that it still seems to converge to the wrong values. It does get the right values in all tabular cases, such as the A-split example, as for those an exact solution to the Bellman

Example 11.3: A-presplit example, a counterexample for the $\overline{\text{BE}}$



Consider the three-state episodic MRP shown to the right: Episodes start in either A1 or A2, with equal probability. These two states look exactly the same to the function approximator, like a single state A whose feature representation is distinct from and unrelated to the feature representation of the other two states, B and C, which are also distinct from each other. Specifically, the parameter of the function approximator has three components, one giving the value of state B, one giving the value of state C, and one giving the value of both states A1 and A2. Other than the selection of the initial state, the system is deterministic. If it starts in A1, then it transitions to B with a reward of 0 and then on to termination with a reward of 1. If it starts in A2, then it transitions to C, and then to termination, with both rewards zero.

To a learning algorithm, seeing only the features, the system looks identical to the A-split example. The system seems to always start in A, followed by either B or C with equal probability, and then terminating with a 1 or a 0 depending deterministically on the previous state. As in the A-split example, the true values of B and C are 1 and 0, and the best shared value of A1 and A2 is $\frac{1}{2}$, by symmetry.

Because this problem appears externally identical to the A-split example, we already know what values will be found by the algorithms. Semi-gradient TD converges to the ideal values just mentioned, while the naive residual-gradient algorithm converges to values of $\frac{3}{4}$ and $\frac{1}{4}$ for B and C respectively. All state transitions are deterministic, so the non-naive residual-gradient algorithm will also converge to these values (it is the same algorithm in this case). It follows then that this 'naive' solution must also be the one that minimizes the $\overline{\text{BE}}$, and so it is. On a deterministic problem, the Bellman errors and TD errors are all the same, so the $\overline{\text{BE}}$ is always the same as the $\overline{\text{TDE}}$. Optimizing the $\overline{\text{BE}}$ on this example gives rise to the same failure mode as with the naive residual-gradient algorithm on the A-split example.

equation is possible. But if we examine examples with genuine function approximation, then the residual-gradient algorithm, and indeed the $\overline{\text{BE}}$ objective, seem to find the wrong value functions. One of the most telling such examples is the variation on the A-split example known as the A-*pre*split example, shown on the preceding page, in which the residual-gradient algorithm finds the same poor solution as its naive version. This example shows intuitively that minimizing the $\overline{\text{BE}}$ (which the residual-gradient algorithm surely does) may not be a desirable goal.

The third way in which the convergence of the residual-gradient algorithm is not satisfactory is explained in the next section. Like the second way, the third way is also a problem with the $\overline{\text{BE}}$ objective itself rather than with any particular algorithm for achieving it.

11.6 The Bellman Error is Not Learnable

The concept of learnability that we introduce in this section is different from that commonly used in machine learning. There, a hypothesis is said to be "learnable" if it is *efficiently* learnable, meaning that it can be learned within a polynomial rather than an exponential number of examples. Here we use the term in a more basic way, to mean learnable at all, with any amount of experience. It turns out many quantities of apparent interest in reinforcement learning cannot be learned even from an infinite amount of experiential data. These quantities are well defined and can be computed given knowledge of the internal structure of the environment, but cannot be computed or estimated from the observed sequence of feature vectors, actions, and rewards.[2] We say that they are not *learnable*. It will turn out that the Bellman error objective ($\overline{\text{BE}}$) introduced in the last two sections is not learnable in this sense. That the Bellman error objective cannot be learned from the observable data is probably the strongest reason not to seek it.

To make the concept of learnability clear, let's start with some simple examples. Consider the two Markov reward processes[3] (MRPs) diagrammed below:

Where two edges leave a state, both transitions are assumed to occur with equal probability, and the numbers indicate the reward received. All the states appear the same; they all produce the same single-component feature vector $x = 1$ and have approximated value w. Thus, the only varying part of the data trajectory is the reward sequence. The left MRP stays in the same state and emits an endless stream of 0s and 2s at random, each with 0.5 probability. The right MRP, on every step, either stays in its current state or

[2] They would of course be estimated if the *state* sequence were observed rather than only the corresponding feature vectors.

[3] All MRPs can be considered MDPs with a single action in all states; what we conclude about MRPs here applies as well to MDPs.

switches to the other, with equal probability. The reward is deterministic in this MRP, always a 0 from one state and always a 2 from the other, but because the each state is equally likely on each step, the observable data is again an endless stream of 0s and 2s at random, identical to that produced by the left MRP. (We can assume the right MRP starts in one of two states at random with equal probability.) Thus, even given even an infinite amount of data, it would not be possible to tell which of these two MRPs was generating it. In particular, we could not tell if the MRP has one state or two, is stochastic or deterministic. These things are not learnable.

This pair of MRPs also illustrates that the $\overline{\text{VE}}$ objective (9.1) is not learnable. If $\gamma = 0$, then the true values of the three states (in both MRPs), left to right, are 1, 0, and 2. Suppose $w = 1$. Then the $\overline{\text{VE}}$ is 0 for the left MRP and 1 for the right MRP. Because the $\overline{\text{VE}}$ is different in the two problems, yet the data generated has the same distribution, the $\overline{\text{VE}}$ cannot be learned. The $\overline{\text{VE}}$ is not a unique function of the data distribution. And if it cannot be learned, then how could the $\overline{\text{VE}}$ possibly be useful as an objective for learning?

If an objective cannot be learned, it does indeed draw its utility into question. In the case of the $\overline{\text{VE}}$, however, there is a way out. Note that the same solution, $w = 1$, is optimal for both MRPs above (assuming μ is the same for the two indistinguishable states in the right MRP). Is this a coincidence, or could it be generally true that all MDPs with the same data distribution also have the same optimal parameter vector? If this is true—and we will show next that it is—then the $\overline{\text{VE}}$ remains a usable objective. The $\overline{\text{VE}}$ is not learnable, but the parameter that optimizes it is!

To understand this, it is useful to bring in another natural objective function, this time one that is clearly learnable. One error that is always observable is that between the value estimate at each time and the return from that time. The *Mean Square Return Error*, denoted $\overline{\text{RE}}$, is the expectation, under μ, of the square of this error. In the on-policy case the $\overline{\text{RE}}$ can be written

$$\begin{aligned}\overline{\text{RE}}(\mathbf{w}) &= \mathbb{E}\Big[\big(G_t - \hat{v}(S_t,\mathbf{w})\big)^2\Big] \\ &= \overline{\text{VE}}(\mathbf{w}) + \mathbb{E}\Big[\big(G_t - v_\pi(S_t)\big)^2\Big]. \end{aligned} \tag{11.24}$$

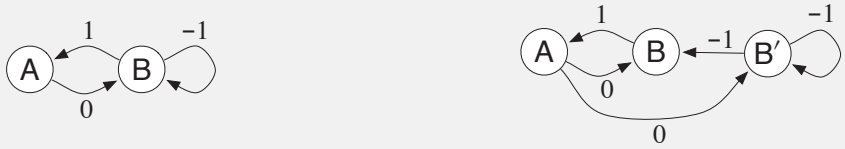
Thus, the two objectives are the same except for a variance term that does not depend on the parameter vector. The two objectives must therefore have the same optimal parameter value $\mathbf{w}^*$. The overall relationships are summarized in the left side of Figure 11.4.

**Exercise 11.4* Prove (11.24). Hint: Write the $\overline{\text{RE}}$ as an expectation over possible states s of the expectation of the squared error given that $S_t = s$. Then add and subtract the true value of state s from the error (before squaring), grouping the subtracted true value with the return and the added true value with the estimated value. Then, if you expand the square, the most complex term will end up being zero, leaving you with (11.24). □

Now let us return to the $\overline{\text{BE}}$. The $\overline{\text{BE}}$ is like the $\overline{\text{VE}}$ in that it can be computed from knowledge of the MDP but is not learnable from data. But it is not like the $\overline{\text{VE}}$ in that its minimum solution is not learnable. The box on the next page presents a counterexample—two MRPs that generate the same data distribution but whose minimizing parameter vector is different, proving that the optimal parameter vector is not a function of the

Example 11.4: Counterexample to the learnability of the Bellman error

To show the full range of possibilities we need a slightly more complex pair of Markov reward processes (MRPs) than those considered earlier. Consider the following two MRPs:



Where two edges leave a state, both transitions are assumed to occur with equal probability, and the numbers indicate the reward received. The MRP on the left has two states that are represented distinctly. The MRP on the right has three states, two of which, B and B′, appear the same and must be given the same approximate value. Specifically, $\mathbf{w}$ has two components and the value of state A is given by the first component and the value of B and B′ is given by the second. The second MRP has been designed so that equal time is spent in all three states, so we can take $\mu(s) = \frac{1}{3}$, for all s.

Note that the observable data distribution is identical for the two MRPs. In both cases the agent will see single occurrences of A followed by a 0, then some number of apparent Bs, each followed by a -1 except the last, which is followed by a 1, then we start all over again with a single A and a 0, etc. All the statistical details are the same as well; in both MRPs, the probability of a string of k Bs is 2^{-k}.

Now suppose $\mathbf{w} = \mathbf{0}$. In the first MRP, this is an exact solution, and the $\overline{\text{BE}}$ is zero. In the second MRP, this solution produces a squared error in both B and B′ of 1, such that $\overline{\text{BE}} = \mu(\text{B})1 + \mu(\text{B}')1 = \frac{2}{3}$. These two MRPs, which generate the same data distribution, have different $\overline{\text{BE}}$s; the $\overline{\text{BE}}$ is not learnable.

Moreover (and unlike the earlier example for the $\overline{\text{VE}}$) the minimizing value of $\mathbf{w}$ is different for the two MRPs. For the first MRP, $\mathbf{w} = \mathbf{0}$ minimizes the $\overline{\text{BE}}$ for any γ. For the second MRP, the minimizing $\mathbf{w}$ is a complicated function of γ, but in the limit, as $\gamma \to 1$, it is $(-\frac{1}{2}, 0)^\top$. Thus the solution that minimizes $\overline{\text{BE}}$ cannot be estimated from data alone; knowledge of the MRP beyond what is revealed in the data is required. In this sense, it is impossible in principle to pursue the $\overline{\text{BE}}$ as an objective for learning.

It may be surprising that in the second MRP the $\overline{\text{BE}}$-minimizing value of A is so far from zero. Recall that A has a dedicated weight and thus its value is unconstrained by function approximation. A is followed by a reward of 0 and transition to a state with a value of nearly 0, which suggests $v_{\mathbf{w}}(\text{A})$ should be 0; why is its optimal value substantially negative rather than 0? The answer is that making $v_{\mathbf{w}}(\text{A})$ negative reduces the error upon arriving in A from B. The reward on this deterministic transition is 1, which implies that B should have a value 1 more than A. Because B's value is approximately zero, A's value is driven toward -1. The $\overline{\text{BE}}$-minimizing value of $\approx -\frac{1}{2}$ for A is a compromise between reducing the errors on leaving and on entering A.

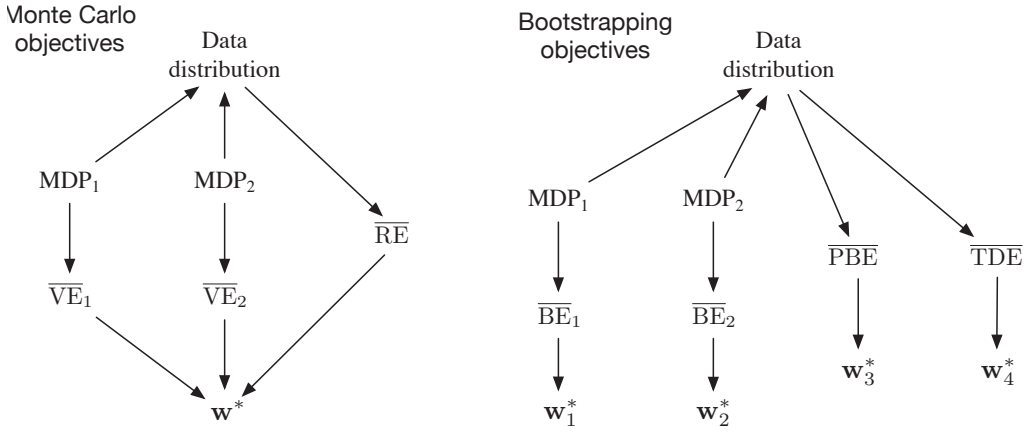


Figure 11.4: Causal relationships among the data distribution, MDPs, and various objectives. **Left, Monte Carlo objectives:** Two different MDPs can produce the same data distribution yet also produce different $\overline{\text{VE}}$s, proving that the $\overline{\text{VE}}$ objective cannot be determined from data and is not learnable. However, all such $\overline{\text{VE}}$s must have the same optimal parameter vector, $\mathbf{w}^*$! Moreover, this same $\mathbf{w}^*$ can be determined from another objective, the $\overline{\text{RE}}$, which *is* uniquely determined from the data distribution. Thus $\mathbf{w}^*$ and the $\overline{\text{RE}}$ are learnable even though the $\overline{\text{VE}}$s are not. **Right, Bootstrapping objectives:** Two different MDPs can produce the same data distribution yet also produce different $\overline{\text{BE}}$s *and* have different minimizing parameter vectors; these are not learnable from the data distribution. The $\overline{\text{PBE}}$ and $\overline{\text{TDE}}$ objectives and their (different) minima can be directly determined from data and thus are learnable.

data and thus cannot be learned from it. The other bootstrapping objectives that we have considered, the $\overline{\text{PBE}}$ and $\overline{\text{TDE}}$, can be determined from data (are learnable) and determine optimal solutions that are in general different from each other and the $\overline{\text{BE}}$ minimums. The general case is summarized in the right side of Figure 11.4.

Thus, the $\overline{\text{BE}}$ is not learnable; it cannot be estimated from feature vectors and other observable data. This limits the $\overline{\text{BE}}$ to model-based settings. There can be no algorithm that minimizes the $\overline{\text{BE}}$ without access to the underlying MDP states beyond the feature vectors. The residual-gradient algorithm is only able to minimize $\overline{\text{BE}}$ because it is allowed to double sample from the same state—not a state that has the same feature vector, but one that is guaranteed to be the same underlying state. We can see now that there is no way around this. Minimizing the $\overline{\text{BE}}$ requires some such access to the nominal, underlying MDP. This is an important limitation of the $\overline{\text{BE}}$ beyond that identified in the A-presplit example on page 273. All this directs more attention toward the $\overline{\text{PBE}}$.

11.7 Gradient-TD Methods

We now consider SGD methods for minimizing the $\overline{\text{PBE}}$. As true SGD methods, these *Gradient-TD methods* have robust convergence properties even under off-policy training and nonlinear function approximation. Remember that in the linear case there is always an exact solution, the TD fixed point $\mathbf{w}_{\text{TD}}$, at which the $\overline{\text{PBE}}$ is zero. This solution could be found by least-squares methods (Section 9.8), but only by methods of quadratic $O(d^2)$ complexity in the number of parameters. We seek instead an SGD method, which should be $O(d)$ and have robust convergence properties. Gradient-TD methods come close to achieving these goals, at the cost of a rough doubling of computational complexity.

To derive an SGD method for the $\overline{\text{PBE}}$ (assuming linear function approximation) we begin by expanding and rewriting the objective (11.22) in matrix terms:

$$\begin{aligned}
\overline{\text{PBE}}(\mathbf{w}) &= \left\|\Pi\bar{\delta}_{\mathbf{w}}\right\|_{\mu}^2 \\
&= (\Pi\bar{\delta}_{\mathbf{w}})^\top \mathbf{D}\Pi\bar{\delta}_{\mathbf{w}} && \text{(from (11.15))} \\
&= \bar{\delta}_{\mathbf{w}}^\top \Pi^\top \mathbf{D}\Pi\bar{\delta}_{\mathbf{w}} \\
&= \bar{\delta}_{\mathbf{w}}^\top \mathbf{D}\mathbf{X}\big(\mathbf{X}^\top\mathbf{D}\mathbf{X}\big)^{-1}\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}} && (11.25) \\
&\text{(using (11.14) and the identity } \Pi^\top\mathbf{D}\Pi = \mathbf{D}\mathbf{X}\left(\mathbf{X}^\top\mathbf{D}\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{D}\text{)} \\
&= \big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big)^\top\big(\mathbf{X}^\top\mathbf{D}\mathbf{X}\big)^{-1}\big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big). && (11.26)
\end{aligned}$$

The gradient with respect to $\mathbf{w}$ is

$$\nabla\overline{\text{PBE}}(\mathbf{w}) = 2\nabla\big[\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big]^\top\left(\mathbf{X}^\top\mathbf{D}\mathbf{X}\right)^{-1}\big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big).$$

To turn this into an SGD method, we have to sample something on every time step that has this quantity as its expected value. Let us take μ to be the distribution of states visited under the behavior policy. All three of the factors above can then be written in terms of expectations under this distribution. For example, the last factor can be written

$$\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}} = \sum_s \mu(s)\mathbf{x}(s)\bar{\delta}_{\mathbf{w}}(s) = \mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\,,$$

which is just the expectation of the semi-gradient TD(0) update (11.2). The first factor is the transpose of the gradient of this update:

$$\begin{aligned}
\nabla\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]^\top &= \mathbb{E}\big[\rho_t\nabla\delta_t^\top\mathbf{x}_t^\top\big] \\
&= \mathbb{E}\big[\rho_t\nabla(R_{t+1} + \gamma\mathbf{w}^\top\mathbf{x}_{t+1} - \mathbf{w}^\top\mathbf{x}_t)^\top\mathbf{x}_t^\top\big] && \text{(using episodic } \delta_t\text{)} \\
&= \mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\,.
\end{aligned}$$

Finally, the middle factor is the inverse of the expected outer-product matrix of the feature vectors:

$$\mathbf{X}^\top\mathbf{D}\mathbf{X} = \sum_s \mu(s)\mathbf{x}_s\mathbf{x}_s^\top = \mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]\,.$$

Substituting these expectations for the three factors in our expression for the gradient of the $\overline{\text{PBE}}$, we get

$$\nabla\overline{\text{PBE}}(\mathbf{w}) = 2\mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\,. \tag{11.27}$$

It might not be obvious that we have made any progress by writing the gradient in this form. It is a product of three expressions and the first and last are not independent. They both depend on the next feature vector $\mathbf{x}_{t+1}$; we cannot simply sample both of these expectations and then multiply the samples. This would give us a biased estimate of the gradient just as in the naive residual-gradient algorithm.

Another idea would be to estimate the three expectations separately and then combine them to produce an unbiased estimate of the gradient. This would work, but would require a lot of computational resources, particularly to store the first two expectations, which are $d \times d$ matrices, and to compute the inverse of the second. This idea can be improved. If two of the three expectations are estimated and stored, then the third could be sampled and used in conjunction with the two stored quantities. For example, you could store estimates of the second two quantities (using the increment inverse-updating techniques in Section 9.8) and then sample the first expression. Unfortunately, the overall algorithm would still be of quadratic complexity (of order $O(d^2)$).

The idea of storing some estimates separately and then combining them with samples is a good one and is also used in Gradient-TD methods. Gradient-TD methods estimate and store *the product* of the second two factors in (11.27). These factors are a $d \times d$ matrix and a d-vector, so their product is just a d-vector, like $\mathbf{w}$ itself. We denote this second learned vector as $\mathbf{v}$:

$$\mathbf{v} \approx \mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\,. \tag{11.28}$$

This form is familiar to students of linear supervised learning. It is the solution to a linear least-squares problem that tries to approximate $\rho_t\delta_t$ from the features. The standard SGD method for incrementally finding the vector $\mathbf{v}$ that minimizes the expected squared error $\big(\mathbf{v}^\top\mathbf{x}_t - \rho_t\delta_t\big)^2$ is known as the Least Mean Square (LMS) rule (here augmented with an importance sampling ratio):

$$\mathbf{v}_{t+1} \doteq \mathbf{v}_t + \beta\rho_t\,\big(\delta_t - \mathbf{v}_t^\top\mathbf{x}_t\big)\,\mathbf{x}_t,$$

where $\beta > 0$ is another step-size parameter. We can use this method to effectively achieve (11.28) with $O(d)$ storage and per-step computation.

Given a stored estimate $\mathbf{v}_t$ approximating (11.28), we can update our main parameter vector $\mathbf{w}_t$ using SGD methods based on (11.27). The simplest such rule is

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla\overline{\text{PBE}}(\mathbf{w}_t) && \text{(the general SGD rule)}\\
&= \mathbf{w}_t - \frac{1}{2}\alpha 2\mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t] && \text{(from (11.27))}\\
&= \mathbf{w}_t + \alpha\mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t] && (11.29)\\
&\approx \mathbf{w}_t + \alpha\mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\,\mathbf{v}_t && \text{(based on (11.28))}\\
&\approx \mathbf{w}_t + \alpha\rho_t\,(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\,\mathbf{x}_t^\top\mathbf{v}_t. && \text{(sampling)}
\end{aligned}$$

This algorithm is called *GTD2*. Note that if the final inner product $(\mathbf{x}_t^\top\mathbf{v}_t)$ is done first, then the entire algorithm is of $O(d)$ complexity.

A slightly better algorithm can be derived by doing a few more analytic steps before substituting in $\mathbf{v}_t$. Continuing from (11.29):

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\\
&= \mathbf{w}_t + \alpha\left(\mathbb{E}\big[\rho_t\mathbf{x}_t\mathbf{x}_t^\top\big] - \gamma\mathbb{E}\big[\rho_t\mathbf{x}_{t+1}\mathbf{x}_t^\top\big]\right)\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\\
&= \mathbf{w}_t + \alpha\left(\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big] - \gamma\mathbb{E}\big[\rho_t\mathbf{x}_{t+1}\mathbf{x}_t^\top\big]\right)\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\\
&= \mathbf{w}_t + \alpha\left(\mathbb{E}[\mathbf{x}_t\rho_t\delta_t] - \gamma\mathbb{E}\big[\rho_t\mathbf{x}_{t+1}\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\right)\\
&\approx \mathbf{w}_t + \alpha\left(\mathbb{E}[\mathbf{x}_t\rho_t\delta_t] - \gamma\mathbb{E}\big[\rho_t\mathbf{x}_{t+1}\mathbf{x}_t^\top\big]\,\mathbf{v}_t\right) && \text{(based on (11.28))}\\
&\approx \mathbf{w}_t + \alpha\rho_t\left(\delta_t\mathbf{x}_t - \gamma\mathbf{x}_{t+1}\mathbf{x}_t^\top\mathbf{v}_t\right), && \text{(sampling)}
\end{aligned}$$

which again is $O(d)$ if the final product $(\mathbf{x}_t^\top\mathbf{v}_t)$ is done first. This algorithm is known as either *TD(0) with gradient correction (TDC)* or, alternatively, as *GTD(0)*.

Figure 11.5 shows a sample and the expected behavior of TDC on Baird's counterexample. As intended, the $\overline{\text{PBE}}$ falls to zero, but note that the individual components of the parameter vector do not approach zero. In fact, these values are still far from

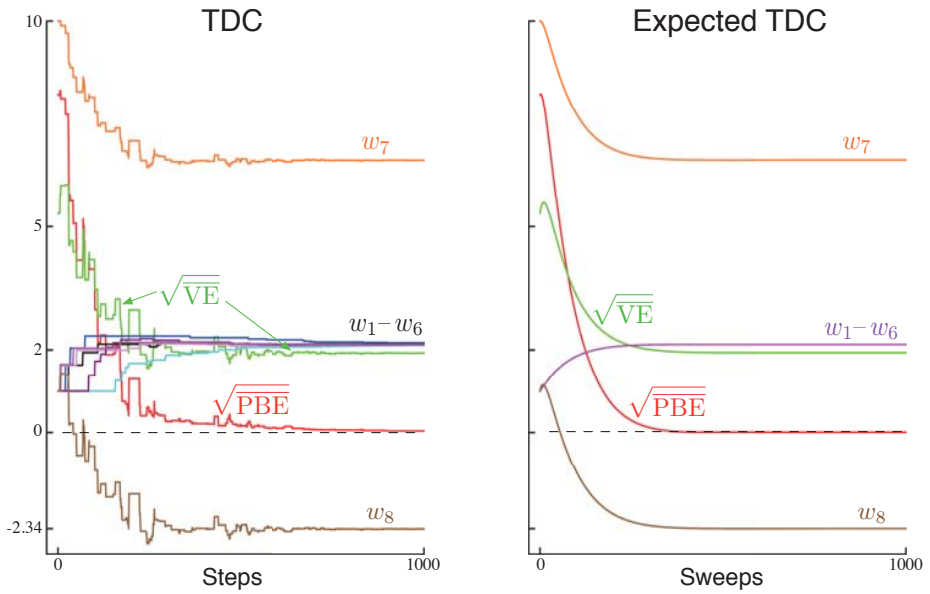


Figure 11.5: The behavior of the TDC algorithm on Baird's counterexample. On the left is shown a typical single run, and on the right is shown the expected behavior of this algorithm if the updates are done synchronously (analogous to (11.9), except for the two TDC parameter vectors). The step sizes were $\alpha = 0.005$ and $\beta = 0.05$.

an optimal solution, $\hat{v}(s) = 0$, for all s, for which $\mathbf{w}$ would have to be proportional to $(1, 1, 1, 1, 1, 1, 4, -2)^\top$. After 1000 iterations we are still far from an optimal solution, as we can see from the $\overline{\text{VE}}$, which remains almost 2. The system is actually converging to an optimal solution, but progress is extremely slow because the $\overline{\text{PBE}}$ is already so close to zero.

GTD2 and TDC both involve two learning processes, a primary one for $\mathbf{w}$ and a secondary one for $\mathbf{v}$. The logic of the primary learning process relies on the secondary learning process having finished, at least approximately, whereas the secondary learning process proceeds without being influenced by the first. We call this sort of asymmetrical dependence a *cascade*. In cascades we often assume that the secondary learning process is proceeding faster and thus is always at its asymptotic value, ready and accurate to assist the primary learning process. The convergence proofs for these methods often make this assumption explicitly. These are called *two-time-scale* proofs. The fast time scale is that of the secondary learning process, and the slower time scale is that of the primary learning process. If α is the step size of the primary learning process, and β is the step size of the secondary learning process, then these convergence proofs will typically require that in the limit $\beta \to 0$ and $\frac{\alpha}{\beta} \to 0$.

Gradient-TD methods are currently the most well understood and widely used stable off-policy methods. There are extensions to action values and control (GQ, Maei et al., 2010), to eligibility traces (GTD(λ) and GQ(λ), Maei, 2011; Maei and Sutton, 2010), and to nonlinear function approximation (Maei et al., 2009). There have also been proposed hybrid algorithms midway between semi-gradient TD and gradient TD (Hackman, 2012; White and White, 2016). Hybrid-TD algorithms behave like Gradient-TD algorithms in states where the target and behavior policies are very different, and behave like semi-gradient algorithms in states where the target and behavior policies are the same. Finally, the Gradient-TD idea has been combined with the ideas of proximal methods and control variates to produce more efficient methods (Mahadevan et al., 2014; Du et al., 2017).

11.8 Emphatic-TD Methods

We turn now to the second major strategy that has been extensively explored for obtaining a cheap and efficient off-policy learning method with function approximation. Recall that linear semi-gradient TD methods are efficient and stable when trained under the on-policy distribution, and that we showed in Section 9.4 that this has to do with the positive definiteness of the matrix $\mathbf{A}$ (9.11)[4] and the match between the on-policy state distribution μ_π and the state-transition probabilities $p(s|s,a)$ under the target policy. In off-policy learning, we reweight the state transitions using importance sampling so that they become appropriate for learning about the target policy, but the state distribution is still that of the behavior policy. There is a mismatch. A natural idea is to somehow reweight the states, emphasizing some and de-emphasizing others, so as to return the distribution of updates to the on-policy distribution. There would then be a match, and stability and convergence would follow from existing results. This is the idea of

[4] In the off-policy case, the matrix $\mathbf{A}$ is generally defined as $\mathbb{E}_{s\sim b}\big[\mathbf{x}(s)\mathbb{E}\big[\mathbf{x}(S_{t+1})^\top \big|\, S_t = s, A_t \sim \pi\big]\big]$.

Emphatic-TD methods, first introduced for on-policy training in Section 9.11.

Actually, the notion of "the on-policy distribution" is not quite right, as there are many on-policy distributions, and any one of these is sufficient to guarantee stability. Consider an undiscounted episodic problem. The way episodes terminate is fully determined by the transition probabilities, but there may be several different ways the episodes might begin. However the episodes start, if all state transitions are due to the target policy, then the state distribution that results is an on-policy distribution. You might start close to the terminal state and visit only a few states with high probability before ending the episode. Or you might start far away and pass through many states before terminating. Both are on-policy distributions, and training on both with a linear semi-gradient method would be guaranteed to be stable. However the process starts, an on-policy distribution results as long as all states encountered are updated up until termination.

If there is discounting, it can be treated as partial or probabilistic termination for these purposes. If $\gamma = 0.9$, then we can consider that with probability 0.1 the process terminates on every time step and then immediately restarts in the state that is transitioned to. A discounted problem is one that is continually terminating and restarting with probability $1 - \gamma$ on every step. This way of thinking about discounting is an example of a more general notion of *pseudo termination*—termination that does not affect the sequence of state transitions, but does affect the learning process and the quantities being learned. This kind of pseudo termination is important to off-policy learning because the restarting is optional—remember we can start any way we want to—and the termination relieves the need to keep including encountered states within the on-policy distribution. That is, if we don't consider the new states as restarts, then discounting quickly gives us a limited on-policy distribution.

The one-step Emphatic-TD algorithm for learning episodic state values is defined by:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t),$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha M_t \rho_t \delta_t \nabla \hat{v}(S_t, \mathbf{w}_t),$$

$$M_t = \gamma \rho_{t-1} M_{t-1} + I_t,$$

with I_t, the *interest*, being arbitrary and M_t, the *emphasis*, being initialized to $M_{t-1} = 0$. How does this algorithm perform on Baird's counterexample? Figure 11.6 shows the trajectory in expectation of the components of the parameter vector (for the case in which $I_t = 1$, for all t). There are some oscillations but eventually everything converges and the $\overline{\text{VE}}$ goes to zero. These trajectories are obtained by iteratively computing the expectation of the parameter vector trajectory without any of the variance due to sampling of transitions and rewards. We do not show the results of applying the Emphatic-TD algorithm directly because its variance on Baird's counterexample is so high that it is nigh impossible to get consistent results in computational experiments. The algorithm converges to the optimal solution in theory on this problem, but in practice it does not. We turn to the topic of reducing the variance of all these algorithms in the next section.

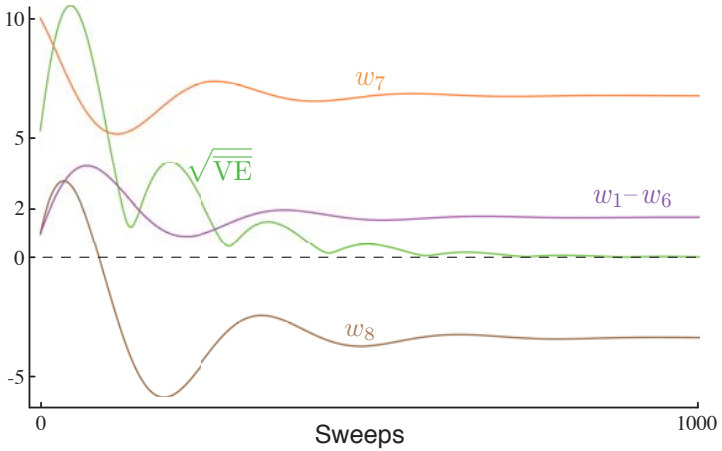


Figure 11.6: The behavior of the one-step Emphatic-TD algorithm in expectation on Baird's counterexample. The step size was $\alpha = 0.03$.

11.9 Reducing Variance

Off-policy learning is inherently of greater variance than on-policy learning. This is not surprising; if you receive data less closely related to a policy, you should expect to learn less about the policy's values. In the extreme, one may be able to learn nothing. You can't expect to learn how to drive by cooking dinner, for example. Only if the target and behavior policies are related, if they visit similar states and take similar actions, should one be able to make significant progress in off-policy training.

On the other hand, any policy has many neighbors, many similar policies with considerable overlap in states visited and actions chosen, and yet which are not identical. The raison d'être of off-policy learning is to enable generalization to this vast number of related-but-not-identical policies. The problem remains of how to make the best use of the experience. Now that we have some methods that are stable in expected value (if the step sizes are set right), attention naturally turns to reducing the variance of the estimates. There are many possible ideas, and we can just touch on a few of them in this introductory text.

Why is controlling variance especially critical in off-policy methods based on importance sampling? As we have seen, importance sampling often involves products of policy ratios. The ratios are always one in expectation (5.13), but their actual values may be very high or as low as zero. Successive ratios are uncorrelated, so their products are also always one in expected value, but they can be of very high variance. Recall that these ratios multiply the step size in SGD methods, so high variance means taking steps that vary greatly in their sizes. This is problematic for SGD because of the occasional very large steps. They must not be so large as to take the parameter to a part of the space with a very different gradient. SGD methods rely on averaging over multiple steps to get a good sense of the gradient, and if they make large moves from single samples they become unreliable.

If the step-size parameter is set small enough to prevent this, then the expected step can end up being very small, resulting in very slow learning. The notions of momentum (Derthick, 1984), of Polyak-Ruppert averaging (Polyak, 1990; Ruppert, 1988; Polyak and Juditsky, 1992), or further extensions of these ideas may significantly help. Methods for adaptively setting separate step sizes for different components of the parameter vector are also pertinent (e.g., Jacobs, 1988; Sutton, 1992b, c), as are the "importance weight aware" updates of Karampatziakis and Langford (2010).

In Chapter 5 we saw how weighted importance sampling is significantly better behaved, with lower variance updates, than ordinary importance sampling. However, adapting weighted importance sampling to function approximation is challenging and can probably only be done approximately with $O(d)$ complexity (Mahmood and Sutton, 2015).

The Tree Backup algorithm (Section 7.5) shows that it is possible to perform some off-policy learning without using importance sampling. This idea has been extended to the off-policy case to produce stable and more efficient methods by Munos, Stepleton, Harutyunyan, and Bellemare (2016) and by Mahmood, Yu and Sutton (2017).

Another, complementary strategy is to allow the target policy to be determined in part by the behavior policy, in such a way that it never can be so different from it to create large importance sampling ratios. For example, the target policy can be defined by reference to the behavior policy, as in the "recognizers" proposed by Precup et al. (2006).

11.10 Summary

Off-policy learning is a tempting challenge, testing our ingenuity in designing stable and efficient learning algorithms. Tabular Q-learning makes off-policy learning seem easy, and it has natural generalizations to Expected Sarsa and to the Tree Backup algorithm. But as we have seen in this chapter, the extension of these ideas to significant function approximation, even linear function approximation, involves new challenges and forces us to deepen our understanding of reinforcement learning algorithms.

Why go to such lengths? One reason to seek off-policy algorithms is to give flexibility in dealing with the tradeoff between exploration and exploitation. Another is to free behavior from learning, and avoid the tyranny of the target policy. TD learning appears to hold out the possibility of learning about multiple things in parallel, of using one stream of experience to solve many tasks simultaneously. We can certainly do this in special cases, just not in every case that we would like to or as efficiently as we would like to.

In this chapter we divided the challenge of off-policy learning into two parts. The first part, correcting the targets of learning for the behavior policy, is straightforwardly dealt with using the techniques devised earlier for the tabular case, albeit at the cost of increasing the variance of the updates and thereby slowing learning. High variance will probably always remains a challenge for off-policy learning.

The second part of the challenge of off-policy learning emerges as the instability of semi-gradient TD methods that involve bootstrapping. We seek powerful function

approximation, off-policy learning, and the efficiency and flexibility of bootstrapping TD methods, but it is challenging to combine all three aspects of this *deadly triad* in one algorithm without introducing the potential for instability. There have been several attempts. The most popular has been to seek to perform true stochastic gradient descent (SGD) in the Bellman error (a.k.a. the Bellman residual). However, our analysis concludes that this is not an appealing goal in many cases, and that anyway it is impossible to achieve with a learning algorithm—the gradient of the $\overline{\text{BE}}$ is not learnable from experience that reveals only feature vectors and not underlying states. Another approach, Gradient-TD methods, performs SGD in the *projected* Bellman error. The gradient of the $\overline{\text{PBE}}$ is learnable with $O(d)$ complexity, but at the cost of a second parameter vector with a second step size. The newest family of methods, Emphatic-TD methods, refine an old idea for reweighting updates, emphasizing some and de-emphasizing others. In this way they restore the special properties that make on-policy learning stable with computationally simple semi-gradient methods.

The whole area of off-policy learning is relatively new and unsettled. Which methods are best or even adequate is not yet clear. Are the complexities of the new methods introduced at the end of this chapter really necessary? Which of them can be combined effectively with variance reduction methods? The potential for off-policy learning remains tantalizing, the best way to achieve it still a mystery.

Bibliographical and Historical Remarks

11.1 The first semi-gradient method was linear TD(λ) (Sutton, 1988). The name "semi-gradient" is more recent (Sutton, 2015a). Semi-gradient off-policy TD(0) with general importance-sampling ratio may not have been explicitly stated until Sutton, Mahmood, and White (2016), but the action-value forms were introduced by Precup, Sutton, and Singh (2000), who also did eligibility trace forms of these algorithms (see Chapter 12). Their continuing, undiscounted forms have not been significantly explored. The n-step forms given here are new.

11.2 The earliest w-to-$2w$ example was given by Tsitsiklis and Van Roy (1996), who also introduced the specific counterexample in the box on page 263. Baird's counterexample is due to Baird (1995), though the version we present here is slightly modified. Averaging methods for function approximation were developed by Gordon (1995, 1996b). Other examples of instability with off-policy DP methods and more complex methods of function approximation are given by Boyan and Moore (1995). Bradtke (1993) gives an example in which Q-learning using linear function approximation in a linear quadratic regulation problem converges to a destabilizing policy.

11.3 The deadly triad was first identified by Sutton (1995b) and thoroughly analyzed by Tsitsiklis and Van Roy (1997). The name "deadly triad" is due to Sutton (2015a).

11.4 This kind of linear analysis was pioneered by Tsitsiklis and Van Roy (1996; 1997),

including the dynamic programming operator. Diagrams like Figure 11.3 were introduced by Lagoudakis and Parr (2003).

What we have called the Bellman operator, and denoted B_π, is more commonly denoted T^π and called a "dynamic programming operator," while a generalized form, denoted $T^{(\lambda)}$, is called the "TD(λ) operator" (Tsitsiklis and Van Roy, 1996, 1997).

11.5 The $\overline{\text{BE}}$ was first proposed as an objective function for dynamic programming by Schweitzer and Seidmann (1985). Baird (1995, 1999) extended it to TD learning based on stochastic gradient descent. In the literature, $\overline{\text{BE}}$ minimization is often referred to as Bellman residual minimization.

The earliest A-split example is due to Dayan (1992). The two forms given here were introduced by Sutton et al. (2009a).

11.6 The contents of this section are new to this text.

11.7 Gradient-TD methods were introduced by Sutton, Szepesvári, and Maei (2009b). The methods highlighted in this section were introduced by Sutton et al. (2009a) and Mahmood et al. (2014). A major extension to proximal TD methods was developed by Mahadeval et al. (2014). The most sensitive empirical investigations to date of Gradient-TD and related methods are given by Geist and Scherrer (2014), Dann, Neumann, and Peters (2014), White (2015), and Ghiassian, White, White, and Sutton (in preparation). Recent developments in the theory of Gradient-TD methods are presented by Yu (2017).

11.8 Emphatic-TD methods were introduced by Sutton, Mahmood, and White (2016). Full convergence proofs and other theory were later established by Yu (2015; 2016; Yu, Mahmood, and Sutton, 2017), Hallak, Tamar, and Mannor (2015), and Hallak, Tamar, Munos, and Mannor (2016).

Chapter 12

Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning. For example, in the popular TD(λ) algorithm, the λ refers to the use of an eligibility trace. Almost any temporal-difference (TD) method, such as Q-learning or Sarsa, can be combined with eligibility traces to obtain a more general method that may learn more efficiently.

Eligibility traces unify and generalize TD and Monte Carlo methods. When TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end ($\lambda=1$) and one-step TD methods at the other ($\lambda=0$). In between are intermediate methods that are often better than either extreme method. Eligibility traces also provide a way of implementing Monte Carlo methods online and on continuing problems without episodes.

Of course, we have already seen one way of unifying TD and Monte Carlo methods: the n-step TD methods of Chapter 7. What eligibility traces offer beyond these is an elegant algorithmic mechanism with significant computational advantages. The mechanism is a short-term memory vector, the *eligibility trace* $\mathbf{z}_t \in \mathbb{R}^d$, that parallels the long-term weight vector $\mathbf{w}_t \in \mathbb{R}^d$. The rough idea is that when a component of $\mathbf{w}_t$ participates in producing an estimated value, then the corresponding component of $\mathbf{z}_t$ is bumped up and then begins to fade away. Learning will then occur in that component of $\mathbf{w}_t$ if a nonzero TD error occurs before the trace falls back to zero. The trace-decay parameter $\lambda \in [0,1]$ determines the rate at which the trace falls.

The primary computational advantage of eligibility traces over n-step methods is that only a single trace vector is required rather than a store of the last n feature vectors. Learning also occurs continually and uniformly in time rather than being delayed and then catching up at the end of the episode. In addition learning can occur and affect behavior immediately after a state is encountered rather than being delayed n steps.

Eligibility traces illustrate that a learning algorithm can sometimes be implemented in a different way to obtain computational advantages. Many algorithms are most naturally formulated and understood as an update of a state's value based on events that follow that state over multiple future time steps. For example, Monte Carlo methods (Chapter 5) update a state based on all the future rewards, and n-step TD methods (Chapter 7)

update based on the next n rewards and state n steps in the future. Such formulations, based on looking forward from the updated state, are called *forward views*. Forward views are always somewhat complex to implement because the update depends on later things that are not available at the time. However, as we show in this chapter it is often possible to achieve nearly the same updates—and sometimes *exactly* the same updates—with an algorithm that uses the current TD error, looking backward to recently visited states using an eligibility trace. These alternate ways of looking at and implementing learning algorithms are called *backward views*. Backward views, transformations between forward views and backward views, and equivalences between them, date back to the introduction of temporal difference learning but have become much more powerful and sophisticated since 2014. Here we present the basics of the modern view.

As usual, first we fully develop the ideas for state values and prediction, then extend them to action values and control. We develop them first for the on-policy case then extend them to off-policy learning. Our treatment pays special attention to the case of linear function approximation, for which the results with eligibility traces are stronger. All these results apply also to the tabular and state aggregation cases because these are special cases of linear function approximation.

12.1 The λ-return

In Chapter 7 we defined an n-step return as the sum of the first n rewards plus the estimated value of the state reached in n steps, each appropriately discounted (7.1). The general form of that equation, for any parameterized function approximator, is

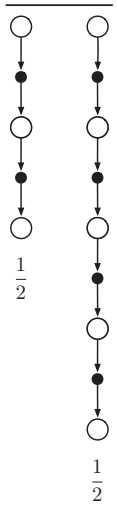
$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \le t \le T-n, \tag{12.1}$$

where $\hat{v}(s,\mathbf{w})$ is the approximate value of state s given weight vector $\mathbf{w}$ (Chapter 9), and T is the time of episode termination, if any. We noted in Chapter 7 that each n-step return, for $n \ge 1$, is a valid update target for a tabular learning update, just as it is for an approximate SGD learning update such as (9.7).

Now we note that a valid update can be done not just toward any n-step return, but toward any *average* of n-step returns for different ns. For example, an update can be done toward a target that is half of a two-step return and half of a four-step return: $\frac{1}{2}G_{t:t+2} + \frac{1}{2}G_{t:t+4}$. Any set of n-step returns can be averaged in this way, even an infinite set, as long as the weights on the component returns are positive and sum to 1. The composite return possesses an error reduction property similar to that of individual n-step returns (7.3) and thus can be used to construct updates with guaranteed convergence properties. Averaging produces a substantial new range of algorithms. For example, one could average one-step and infinite-step returns to obtain another way of interrelating TD and Monte Carlo methods. In principle, one could even average experience-based updates with DP updates to get a simple combination of experience-based and model-based methods (cf. Chapter 8).

An update that averages simpler component updates is called a *compound update*. The backup diagram for a compound update consists of the backup diagrams for each of the component updates with a horizontal line above them and the weighting fractions below.

For example, the compound update for the case mentioned at the start of this section, mixing half of a two-step return and half of a four-step return, has the diagram shown to the right. A compound update can only be done when the longest of its component updates is complete. The update at the right, for example, could only be done at time $t+4$ for the estimate formed at time t. In general one would like to limit the length of the longest component update because of the corresponding delay in the updates.



The TD(λ) algorithm can be understood as one particular way of averaging n-step updates. This average contains all the n-step updates, each weighted proportionally to λ^{n-1} (where $\lambda \in [0,1]$), and is normalized by a factor of $1-\lambda$ to ensure that the weights sum to 1 (Figure 12.1). The resulting update is toward a return, called the λ-*return*, defined in its state-based form by

$$G_t^\lambda \doteq (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n}. \tag{12.2}$$

Figure 12.2 further illustrates the weighting on the sequence of n-step returns in the λ-return. The one-step return is given the largest weight, $1-\lambda$; the two-step return is given the next largest weight, $(1-\lambda)\lambda$; the three-step return is given the weight $(1-\lambda)\lambda^2$; and so on. The weight fades by λ with each additional step. After a terminal state has been reached, all subsequent n-step returns are equal to the conventional return, G_t. If

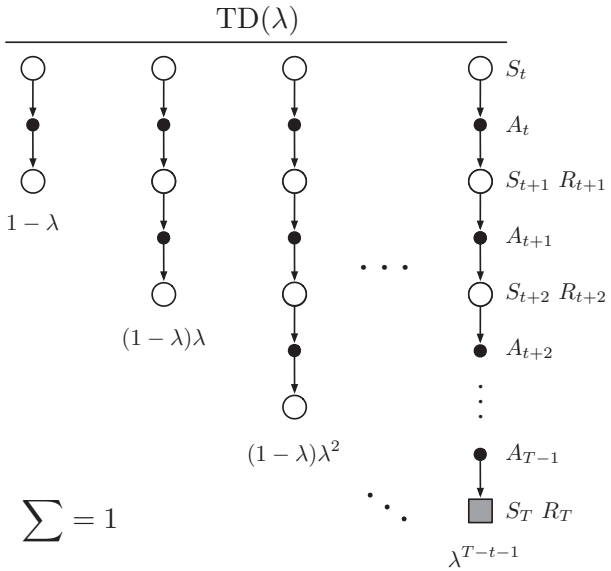


Figure 12.1: The backup digram for TD(λ). If $\lambda = 0$, then the overall update reduces to its first component, the one-step TD update, whereas if $\lambda = 1$, then the overall update reduces to its last component, the Monte Carlo update.

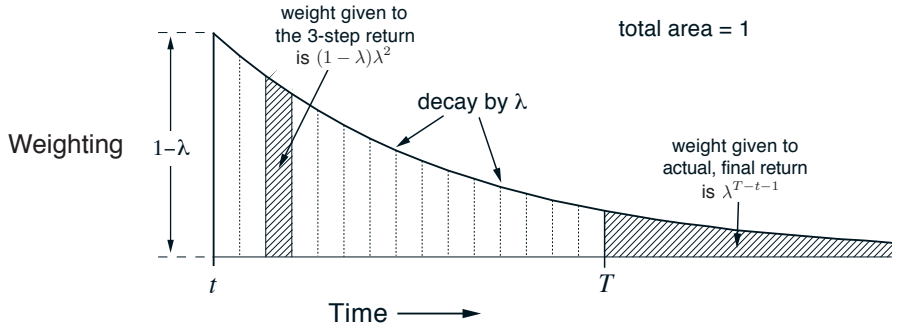


Figure 12.2: Weighting given in the λ-return to each of the n-step returns.

we want, we can separate these post-termination terms from the main sum, yielding

$$G_t^\lambda \;=\; (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} \;+\; \lambda^{T-t-1}G_t, \tag{12.3}$$

as indicated in the figures. This equation makes it clearer what happens when $\lambda = 1$. In this case the main sum goes to zero, and the remaining term reduces to the conventional return. Thus, for $\lambda = 1$, updating according to the λ-return is a Monte Carlo algorithm. On the other hand, if $\lambda = 0$, then the λ-return reduces to $G_{t:t+1}$, the one-step return. Thus, for $\lambda = 0$, updating according to the λ-return is a one-step TD method.

Exercise 12.1 Just as the return can be written recursively in terms of the first reward and itself one-step later (3.9), so can the λ-return. Derive the analogous recursive relationship from (12.2) and (12.1). □

Exercise 12.2 The parameter λ characterizes how fast the exponential weighting in Figure 12.2 falls off, and thus how far into the future the λ-return algorithm looks in determining its update. But a rate factor such as λ is sometimes an awkward way of characterizing the speed of the decay. For some purposes it is better to specify a time constant, or half-life. What is the equation relating λ and the half-life, τ_λ, the time by which the weighting sequence will have fallen to half of its initial value? □

We are now ready to define our first learning algorithm based on the λ-return: the *offline λ-return algorithm*. As an offline algorithm, it makes no changes to the weight vector during the episode. Then, at the end of the episode, a whole sequence of offline updates are made according to our usual semi-gradient rule, using the λ-return as the target:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[G_t^\lambda - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t), \quad t = 0,\ldots,T-1. \tag{12.4}$$

The λ-return gives us an alternative way of moving smoothly between Monte Carlo and one-step TD methods that can be compared with the n-step bootstrapping way developed in Chapter 7. There we assessed effectiveness on a 19-state random walk task (Example 7.1, page 144). Figure 12.3 shows the performance of the offline λ-return algorithm on this task alongside that of the n-step methods (repeated from Figure 7.2). The experiment was just as described earlier except that for the λ-return algorithm we varied λ instead of n. The performance measure used is the estimated root-mean-squared error between the correct and estimated values of each state measured at the end of the episode, averaged over the first 10 episodes and the 19 states. Note that overall performance of the offline λ-return algorithms is comparable to that of the n-step algorithms. In both cases we get best performance with an intermediate value of the bootstrapping parameter, n for n-step methods and λ for the offline λ-return algorithm.

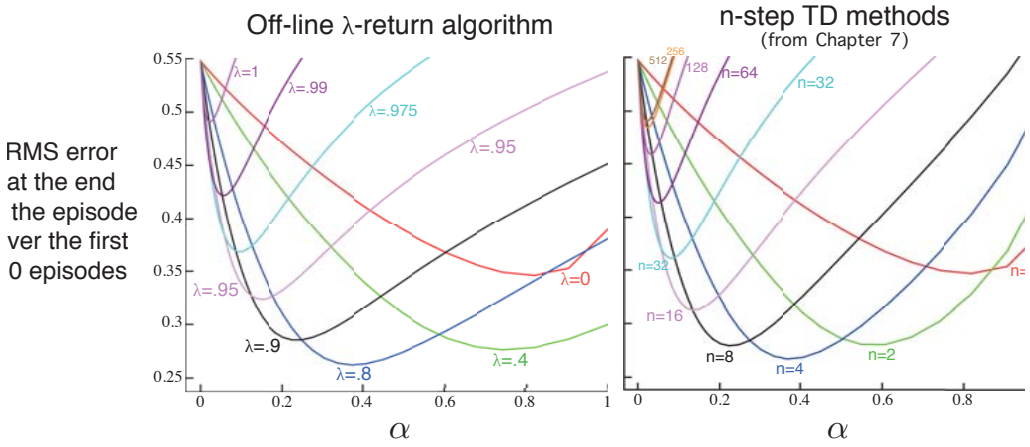


Figure 12.3: 19-state Random walk results (Example 7.1): Performance of the offline λ-return algorithm alongside that of the n-step TD methods. In both case, intermediate values of the bootstrapping parameter (λ or n) performed best. The results with the offline λ-return algorithm are slightly better at the best values of α and λ, and at high α.

The approach that we have been taking so far is what we call the theoretical, or *forward*, view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them. We might imagine ourselves riding the stream of states, looking forward from each state to determine its update, as suggested by Figure 12.4. After looking forward from and updating one state, we move on to the next and never have to work with the preceding state again. Future states, on the other hand, are viewed and processed repeatedly, once from each vantage point preceding them.

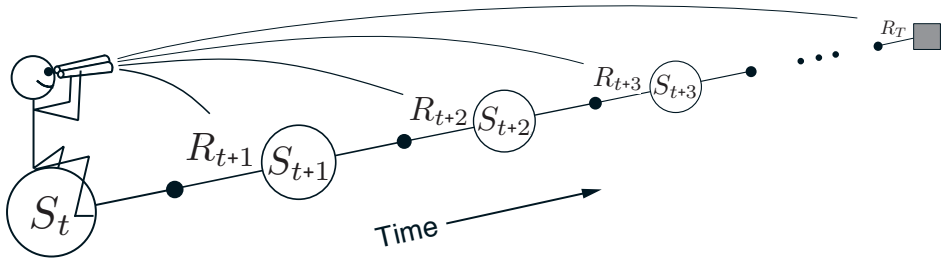


Figure 12.4: The forward view. We decide how to update each state by looking forward to future rewards and states.

12.2 TD(λ)

TD(λ) is one of the oldest and most widely used algorithms in reinforcement learning. It was the first algorithm for which a formal relationship was shown between a more theoretical forward view and a more computationally-congenial backward view using eligibility traces. Here we will show empirically that it approximates the offline λ-return algorithm presented in the previous section.

TD(λ) improves over the offline λ-return algorithm in three ways. First it updates the weight vector on every step of an episode rather than only at the end, and thus its estimates may be better sooner. Second, its computations are equally distributed in time rather than all at the end of the episode. And third, it can be applied to continuing problems rather than just to episodic problems. In this section we present the semi-gradient version of TD(λ) with function approximation.

With function approximation, the eligibility trace is a vector $\mathbf{z}_t \in \mathbb{R}^d$ with the same number of components as the weight vector $\mathbf{w}_t$. Whereas the weight vector is a long-term memory, accumulating over the lifetime of the system, the eligibility trace is a short-term memory, typically lasting less time than the length of an episode. Eligibility traces assist in the learning process; their only consequence is that they affect the weight vector, and then the weight vector determines the estimated value.

In TD(λ), the eligibility trace vector is initialized to zero at the beginning of the episode, is incremented on each time step by the value gradient, and then fades away by $\gamma\lambda$:

$$\begin{aligned} &\mathbf{z}_{-1} \doteq \mathbf{0}, \\ &\mathbf{z}_t \doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t,\mathbf{w}_t), \quad 0 \le t \le T, \end{aligned} \tag{12.5}$$

where γ is the discount rate and λ is the parameter introduced in the previous section, which we henceforth call the trace-decay parameter. The eligibility trace keeps track of which components of the weight vector have contributed, positively or negatively, to recent state valuations, where "recent" is defined in terms of $\gamma\lambda$. (Recall that in linear function approximation, $\nabla\hat{v}(S_t,\mathbf{w}_t)$ is just the feature vector, $\mathbf{x}_t$, in which case the eligibility trace vector is just a sum of past, fading, input vectors.) The trace is said to indicate the eligibility of each component of the weight vector for undergoing learning

changes should a reinforcing event occur. The reinforcing events we are concerned with are the moment-by-moment one-step TD errors. The TD error for state-value prediction is

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t). \tag{12.6}$$

In TD(λ), the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t. \tag{12.7}$$

Semi-gradient TD(λ) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
  Loop for each step of episode:
  | Choose $A \sim \pi(\cdot|S)$
  | Take action A, observe R, S'
  | $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S,\mathbf{w})$
  | $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
  | $S \leftarrow S'$
  until S' is terminal

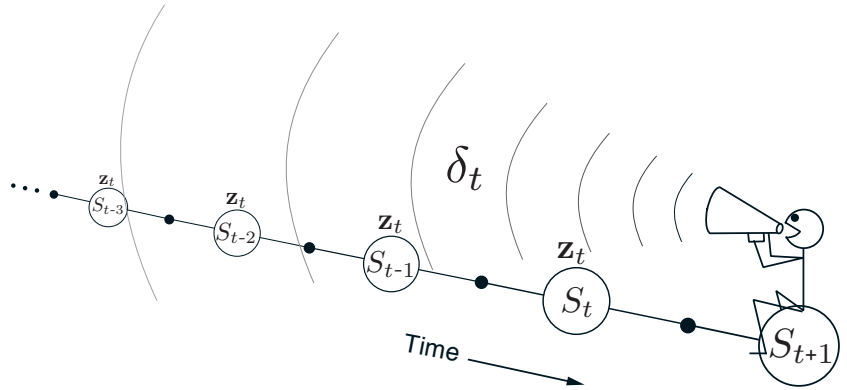


Figure 12.5: The backward or mechanistic view of TD(λ). Each update depends on the current TD error combined with the current eligibility traces of past events.

TD(λ) is oriented backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to how much that state contributed to the current eligibility trace at that time. We might imagine ourselves riding along the stream of states, computing TD errors, and shouting them back to the previously visited states, as suggested by Figure 12.5. Where the TD error and traces come together, we get the update given by (12.7), changing the values of those past states for when they occur again in the future.

To better understand the backward view of TD(λ), consider what happens at various values of λ. If $\lambda = 0$, then by (12.5) the trace at t is exactly the value gradient corresponding to S_t. Thus the TD(λ) update (12.7) reduces to the one-step semi-gradient TD update treated in Chapter 9 (and, in the tabular case, to the simple TD rule (6.2)). This is why that algorithm was called TD(0). In terms of Figure 12.5, TD(0) is the case in which only the one state preceding the current one is changed by the TD error. For larger values of λ, but still $\lambda < 1$, more of the preceding states are changed, but each more temporally distant state is changed less because the corresponding eligibility trace is smaller, as suggested by the figure. We say that the earlier states are given less *credit* for the TD error.

If $\lambda = 1$, then the credit given to earlier states falls only by γ per step. This turns out to be just the right thing to do to achieve Monte Carlo behavior. For example, remember that the TD error, δ_t, includes an undiscounted term of R_{t+1}. In passing this back k steps it needs to be discounted, like any reward in a return, by γ^k, which is just what the falling eligibility trace achieves. If $\lambda = 1$ and $\gamma = 1$, then the eligibility traces do not decay at all with time. In this case the method behaves like a Monte Carlo method for an undiscounted, episodic task. If $\lambda = 1$, the algorithm is also known as TD(1).

TD(1) is a way of implementing Monte Carlo algorithms that is more general than those presented earlier and that significantly increases their range of applicability. Whereas the earlier Monte Carlo methods were limited to episodic tasks, TD(1) can be applied to discounted continuing tasks as well. Moreover, TD(1) can be performed incrementally and online. One disadvantage of Monte Carlo methods is that they learn nothing from an episode until it is over. For example, if a Monte Carlo control method takes an action that produces a very poor reward but does not end the episode, then the agent's tendency to repeat the action will be undiminished during the episode. Online TD(1), on the other hand, learns in an n-step TD way from the incomplete ongoing episode, where the n steps are all the way up to the current step. If something unusually good or bad happens during an episode, control methods based on TD(1) can learn immediately and alter their behavior on that same episode.

It is revealing to revisit the 19-state random walk example (Example 7.1) to see how well TD(λ) does in approximating the offline λ-return algorithm. The results for both algorithms are shown in Figure 12.6. For each λ value, if α is selected optimally for it (or smaller), then the two algorithms perform virtually identically. If α is chosen larger than is optimal, however, then the λ-return algorithm is only a little worse whereas TD(λ) is much worse and may even be unstable. This is not catastrophic for TD(λ) on this problem, as these higher parameter values are not what one would want to use anyway, but for other problems it can be a significant weakness.

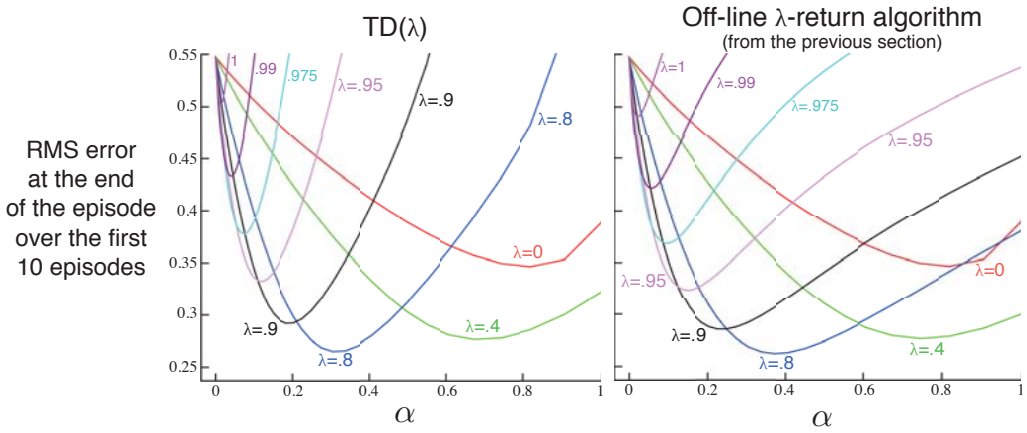


Figure 12.6: 19-state Random walk results (Example 7.1): Performance of TD(λ) alongside that of the offline λ-return algorithm. The two algorithms performed virtually identically at low (less than optimal) α values, but TD(λ) was worse at high α values.

Linear TD(λ) has been proved to converge in the on-policy case if the step-size parameter is reduced over time according to the usual conditions (2.7). Just as discussed in Section 9.4, convergence is not to the minimum-error weight vector, but to a nearby weight vector that depends on λ. The bound on solution quality presented in that section (9.14) can now be generalized to apply for any λ. For the continuing discounted case,

$$\overline{\text{VE}}(\mathbf{w}_\infty) \quad \leq \quad \frac{1-\gamma\lambda}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \tag{12.8}$$

That is, the asymptotic error is no more than $\frac{1-\gamma\lambda}{1-\gamma}$ times the smallest possible error. As λ approaches 1, the bound approaches the minimum error (and it is loosest at $\lambda = 0$). In practice, however, $\lambda = 1$ is often the poorest choice, as will be illustrated later in Figure 12.14.

Exercise 12.3 Some insight into how TD(λ) can closely approximate the offline λ-return algorithm can be gained by seeing that the latter's error term (in brackets in (12.4)) can be written as the sum of TD errors (12.6) for a single fixed $\mathbf{w}$. Show this, following the pattern of (6.6), and using the recursive relationship for the λ-return you obtained in Exercise 12.1. □

Exercise 12.4 Use your result from the preceding exercise to show that, if the weight updates over an episode were computed on each step but not actually used to change the weights ($\mathbf{w}$ remained fixed), then the sum of TD(λ)'s weight updates would be the same as the sum of the offline λ-return algorithm's updates. □

12.3 *n*-step Truncated λ-return Methods

The offline λ-return algorithm is an important ideal, but it is of limited utility because it uses the λ-return (12.2), which is not known until the end of the episode. In the

continuing case, the λ-return is technically never known, as it depends on n-step returns for arbitrarily large n, and thus on rewards arbitrarily far in the future. However, the dependence becomes weaker for longer-delayed rewards, falling by $\gamma\lambda$ for each step of delay. A natural approximation, then, would be to truncate the sequence after some number of steps. Our existing notion of n-step returns provides a natural way to do this in which the missing rewards are replaced with estimated values.

In general, we define the *truncated λ-return* for time t, given data only up to some later horizon, h, as

$$G^{\lambda}_{t:h} \doteq (1-\lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} \quad + \quad \lambda^{h-t-1} G_{t:h}, \qquad 0 \leq t < h \leq T. \tag{12.9}$$

If you compare this equation with the λ-return (12.3), it is clear that the horizon h is playing the same role as was previously played by T, the time of termination. Whereas in the λ-return there is a residual weight given to the conventional return G_t, here it is given to the longest available n-step return, $G_{t:h}$ (Figure 12.2).

The truncated λ-return immediately gives rise to a family of n-step λ-return algorithms similar to the n-step methods of Chapter 7. In all of these algorithms, updates are delayed by n steps and only take into account the first n rewards, but now all the k-step returns are included for $1 \leq k \leq n$ (whereas the earlier n-step algorithms used only the n-step return), weighted geometrically as in Figure 12.2. In the state-value case, this family of algorithms is known as truncated TD(λ), or TTD(λ). The compound backup diagram, shown in Figure 12.7, is similar to that for TD(λ) (Figure 12.1) except that the longest component update is at most n steps rather than always going all the way to the

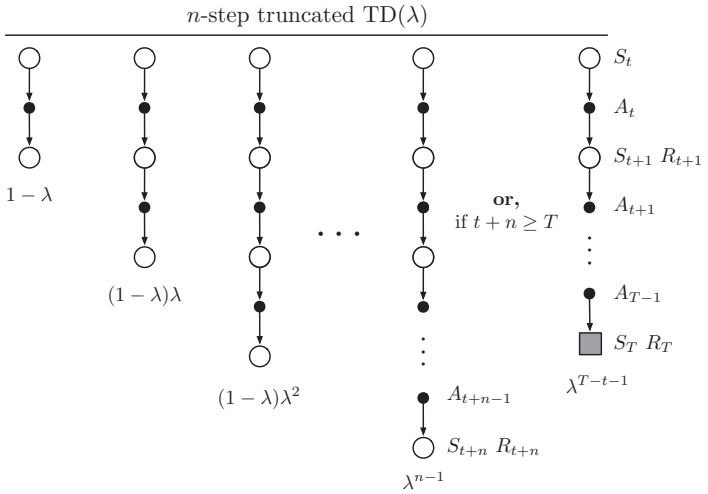


Figure 12.7: The backup diagram for truncated TD(λ).

end of the episode. TTD(λ) is defined by (cf. (9.15)):

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha \left[G^{\lambda}_{t:t+n} - \hat{v}(S_t,\mathbf{w}_{t+n-1}) \right] \nabla \hat{v}(S_t,\mathbf{w}_{t+n-1}), \qquad 0 \leq t < T.$$

This algorithm can be implemented efficiently so that per-step computation does not scale with n (though of course memory must). Much as in n-step TD methods, no updates are made on the first $n-1$ time steps of each episode, and $n-1$ additional updates are made upon termination. Efficient implementation relies on the fact that the k-step λ-return can be written exactly as

$$G^{\lambda}_{t:t+k} = \hat{v}(S_t,\mathbf{w}_{t-1}) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t} \delta'_i, \tag{12.10}$$

where

$$\delta'_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_{t-1}).$$

Exercise 12.5 Several times in this book (often in exercises) we have established that returns can be written as sums of TD errors if the value function is held constant. Why is (12.10) another instance of this? Prove (12.10). □

12.4 Redoing Updates: Online λ-return Algorithm

Choosing the truncation parameter n in truncated TD(λ) involves a tradeoff. n should be large so that the method closely approximates the offline λ-return algorithm, but it should also be small so that the updates can be made sooner and can influence behavior sooner. Can we get the best of both? Well, yes, in principle we can, albeit at the cost of computational complexity.

The idea is that, on each time step as you gather a new increment of data, you go back and redo all the updates since the beginning of the current episode. The new updates will be better than the ones you previously made because now they can take into account the time step's new data. That is, the updates are always towards an n-step truncated λ-return target, but they always use the latest horizon. In each pass over that episode you can use a slightly longer horizon and obtain slightly better results. Recall that the truncated λ-return is defined in (12.9) as

$$G^{\lambda}_{t:h} \doteq (1-\lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}.$$

Let us step through how this target could ideally be used if computational complexity was not an issue. The episode begins with an estimate at time 0 using the weights $\mathbf{w}_0$ from the end of the previous episode. Learning begins when the data horizon is extended to time step 1. The target for the estimate at step 0, given the data up to horizon 1, could only be the one-step return $G_{0:1}$, which includes R_1 and bootstraps from the estimate

$\hat{v}(S_1,\mathbf{w}_0)$. Note that this is exactly what $G^{\lambda}_{0:1}$ is, with the sum in the first term of the equation degenerating to zero. Using this update target, we construct $\mathbf{w}_1$. Then, after advancing the data horizon to step 2, what do we do? We have new data in the form of R_2 and S_2, as well as the new $\mathbf{w}_1$, so now we can construct a better update target $G^{\lambda}_{0:2}$ for the first update from S_0 as well as a better update target $G^{\lambda}_{1:2}$ for the second update from S_1. Using these improved targets, we redo the updates at S_1 and S_2, starting again from $\mathbf{w}_0$, to produce $\mathbf{w}_2$. Now we advance the horizon to step 3 and repeat, going all the way back to produce three new targets, redoing all updates starting from the original $\mathbf{w}_0$ to produce $\mathbf{w}_3$, and so on. Each time the horizon is advanced, all the updates are redone starting from $\mathbf{w}_0$ using the weight vector from the preceding horizon.

This conceptual algorithm involves multiple passes over the episode, one at each horizon, each generating a different sequence of weight vectors. To describe it clearly we have to distinguish between the weight vectors computed at the different horizons. Let us use $\mathbf{w}^h_t$ to denote the weights used to generate the value at time t in the sequence up to horizon h. The first weight vector $\mathbf{w}^h_0$ in each sequence is that inherited from the previous episode (so they are the same for all h), and the last weight vector $\mathbf{w}^h_h$ in each sequence defines the ultimate weight-vector sequence of the algorithm. At the final horizon $h = T$ we obtain the final weights $\mathbf{w}^T_T$ which will be passed on to form the initial weights of the next episode. With these conventions, the three first sequences described in the previous paragraph can be given explicitly:

$$\begin{aligned}
h = 1: \quad & \mathbf{w}^1_1 \doteq \mathbf{w}^1_0 + \alpha \left[G^{\lambda}_{0:1} - \hat{v}(S_0,\mathbf{w}^1_0) \right] \nabla \hat{v}(S_0,\mathbf{w}^1_0), \\
\\
h = 2: \quad & \mathbf{w}^2_1 \doteq \mathbf{w}^2_0 + \alpha \left[G^{\lambda}_{0:2} - \hat{v}(S_0,\mathbf{w}^2_0) \right] \nabla \hat{v}(S_0,\mathbf{w}^2_0), \\
& \mathbf{w}^2_2 \doteq \mathbf{w}^2_1 + \alpha \left[G^{\lambda}_{1:2} - \hat{v}(S_1,\mathbf{w}^2_1) \right] \nabla \hat{v}(S_1,\mathbf{w}^2_1), \\
\\
h = 3: \quad & \mathbf{w}^3_1 \doteq \mathbf{w}^3_0 + \alpha \left[G^{\lambda}_{0:3} - \hat{v}(S_0,\mathbf{w}^3_0) \right] \nabla \hat{v}(S_0,\mathbf{w}^3_0), \\
& \mathbf{w}^3_2 \doteq \mathbf{w}^3_1 + \alpha \left[G^{\lambda}_{1:3} - \hat{v}(S_1,\mathbf{w}^3_1) \right] \nabla \hat{v}(S_1,\mathbf{w}^3_1), \\
& \mathbf{w}^3_3 \doteq \mathbf{w}^3_2 + \alpha \left[G^{\lambda}_{2:3} - \hat{v}(S_2,\mathbf{w}^3_2) \right] \nabla \hat{v}(S_2,\mathbf{w}^3_2).
\end{aligned}$$

The general form for the update is

$$\mathbf{w}^h_{t+1} \doteq \mathbf{w}^h_t + \alpha \left[G^{\lambda}_{t:h} - \hat{v}(S_t,\mathbf{w}^h_t) \right] \nabla \hat{v}(S_t,\mathbf{w}^h_t), \quad 0 \leq t < h \leq T.$$

This update, together with $\mathbf{w}_t \doteq \mathbf{w}^t_t$ defines the *online λ-return algorithm*.

The online λ-return algorithm is fully online, determining a new weight vector $\mathbf{w}_t$ at each step t during an episode, using only information available at time t. Its main drawback is that it is computationally complex, passing over the portion of the episode experienced so far on every step. Note that it is strictly more complex than the offline λ-return algorithm, which passes through all the steps at the time of termination but does not make any updates during the episode. In return, the online algorithm can be expected to perform better than the offline one, not only during the episode when it makes an update while the offline algorithm makes none, but also at the end of the episode because

the weight vector used in bootstrapping (in $G^{\lambda}_{t:h}$) has had a larger number of informative updates. This effect can be seen if one looks carefully at Figure 12.8, which compares the two algorithms on the 19-state random walk task.

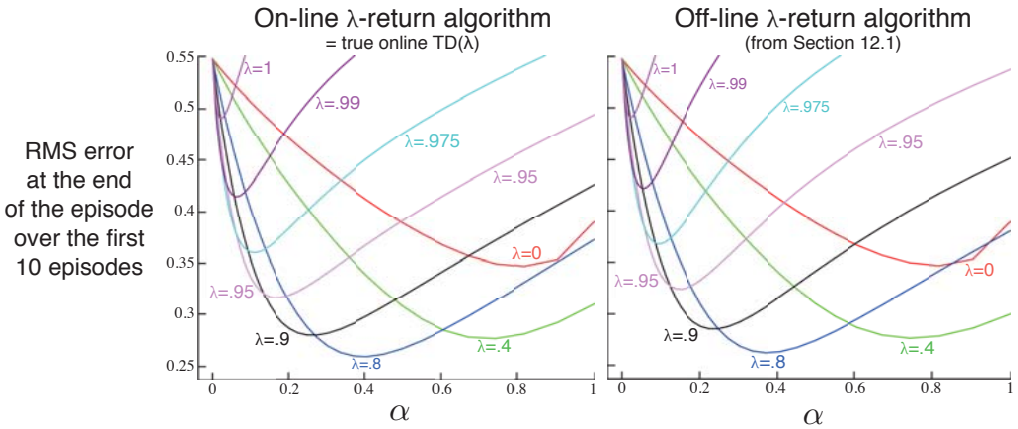


Figure 12.8: 19-state Random walk results (Example 7.1): Performance of online and offline λ-return algorithms. The performance measure here is the $\overline{\text{VE}}$ at the end of the episode, which should be the best case for the offline algorithm. Nevertheless, the online algorithm performs subtly better. For comparison, the $\lambda=0$ line is the same for both methods.

12.5 True Online TD(λ)

The online λ-return algorithm just presented is currently the best performing temporal-difference algorithm. It is an ideal which online TD(λ) only approximates. As presented, however, the online λ-return algorithm is very complex. Is there a way to invert this forward-view algorithm to produce an efficient backward-view algorithm using eligibility traces? It turns out that there is indeed an exact computationally congenial implementation of the online λ-return algorithm for the case of linear function approximation. This implementation is known as the true online TD(λ) algorithm because it is "truer" to the ideal of the online λ-return algorithm than the TD(λ) algorithm is.

The derivation of true online TD(λ) is a little too complex to present here (see the next section and the appendix to the paper by van Seijen et al., 2016) but its strategy is simple. The sequence of weight vectors produced by the online λ-return algorithm can be arranged in a triangle:

$$\begin{array}{ccccccc}
\mathbf{w}_0^0 & & & & & & \\
\mathbf{w}_0^1 & \mathbf{w}_1^1 & & & & & \\
\mathbf{w}_0^2 & \mathbf{w}_1^2 & \mathbf{w}_2^2 & & & & \\
\mathbf{w}_0^3 & \mathbf{w}_1^3 & \mathbf{w}_2^3 & \mathbf{w}_3^3 & & & \\
\vdots & \vdots & \vdots & \vdots & \ddots & & \\
\mathbf{w}_0^T & \mathbf{w}_1^T & \mathbf{w}_2^T & \mathbf{w}_3^T & \cdots & \mathbf{w}_T^T
\end{array}$$

One row of this triangle is produced on each time step. It turns out that the weight vectors on the diagonal, the $\mathbf{w}_t^t$, are the only ones really needed. The first, $\mathbf{w}_0^0$, is the initial weight vector of the episode, the last, $\mathbf{w}_T^T$, is the final weight vector, and each weight vector along the way, $\mathbf{w}_t^t$, plays a role in bootstrapping in the n-step returns of the updates. In the final algorithm the diagonal weight vectors are renamed without a superscript, $\mathbf{w}_t \doteq \mathbf{w}_t^t$. The strategy then is to find a compact, efficient way of computing each $\mathbf{w}_t^t$ from the one before. If this is done, for the linear case in which $\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s)$, then we arrive at the true online TD(λ) algorithm:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t \mathbf{z}_t + \alpha\left(\mathbf{w}_t^\top \mathbf{x}_t - \mathbf{w}_{t-1}^\top \mathbf{x}_t\right)(\mathbf{z}_t - \mathbf{x}_t),$$

where we have used the shorthand $\mathbf{x}_t \doteq \mathbf{x}(S_t)$, δ_t is defined as in TD(λ) (12.6), and $\mathbf{z}_t$ is defined by

$$\mathbf{z}_t \doteq \gamma\lambda \mathbf{z}_{t-1} + \left(1 - \alpha\gamma\lambda \mathbf{z}_{t-1}^\top \mathbf{x}_t\right)\mathbf{x}_t. \tag{12.11}$$

This algorithm has been proven to produce exactly the same sequence of weight vectors, $\mathbf{w}_t, 0 \le t \le T$, as the online λ-return algorithm (van Seijen et al. 2016). Thus the results on the random walk task on the left of Figure 12.8 are also its results on that task. Now, however, the algorithm is much less expensive. The memory requirements of true online TD(λ) are identical to those of conventional TD(λ), while the per-step computation is increased by about 50% (there is one more inner product in the eligibility-trace update). Overall, the per-step computational complexity remains of $O(d)$, the same as TD(λ). Pseudocode for the complete algorithm is given in the box.

True online TD(λ) for estimating $\mathbf{w}^\top\mathbf{x} \approx v_\pi$

Input: the policy π to be evaluated
Input: a feature function $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize state and obtain initial feature vector $\mathbf{x}$
  $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
  $V_{old} \leftarrow 0$ (a temporary scalar variable)
  Loop for each step of episode:
    Choose $A \sim \pi$
    Take action A, observe R, $\mathbf{x}'$ (feature vector of the next state)
    $V \leftarrow \mathbf{w}^\top\mathbf{x}$
    $V' \leftarrow \mathbf{w}^\top\mathbf{x}'$
    $\delta \leftarrow R + \gamma V' - V$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \left(1 - \alpha\gamma\lambda\mathbf{z}^\top\mathbf{x}\right)\mathbf{x}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + V - V_{old})\mathbf{z} - \alpha(V - V_{old})\mathbf{x}$
    $V_{old} \leftarrow V'$
    $\mathbf{x} \leftarrow \mathbf{x}'$
  until $\mathbf{x}' = \mathbf{0}$ (signaling arrival at a terminal state)

The eligibility trace (12.11) used in true online TD(λ) is called a *dutch trace* to distinguish it from the trace (12.5) used in TD(λ), which is called an *accumulating trace*. Earlier work often used a third kind of trace called the *replacing trace*, defined only for the tabular case or for binary feature vectors such as those produced by tile coding. The replacing trace is defined on a component-by-component basis depending on whether the component of the feature vector was 1 or 0:

$$z_{i,t} \doteq \begin{cases} 1 & \text{if } x_{i,t} = 1 \\ \gamma\lambda z_{i,t-1} & \text{otherwise.} \end{cases} \tag{12.12}$$

Nowadays, we see replacing traces as crude approximations to dutch traces, which largely supercede them. Dutch traces usually perform better than replacing traces and have a clearer theoretical basis. Accumulating traces remain of interest for nonlinear function approximations where dutch traces are not available.

12.6 *Dutch Traces in Monte Carlo Learning

Although eligibility traces are closely associated historically with TD learning, in fact they have nothing to do with it. In fact, eligibility traces arise even in Monte Carlo learning, as we show in this section. We show that the linear MC algorithm (Chapter 9), taken as a forward view, can be used to derive an equivalent yet computationally cheaper backward-view algorithm using dutch traces. This is the only equivalence of forward- and backward-views that we explicitly demonstrate in this book. It gives some of the flavor of the proof of equivalence of true online TD(λ) and the online λ-return algorithm, but is much simpler.

The linear version of the gradient Monte Carlo prediction algorithm (page 202) makes the following sequence of updates, one for each time step of the episode:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[G - \mathbf{w}_t^\top \mathbf{x}_t\right] \mathbf{x}_t, \quad 0 \le t < T. \tag{12.13}$$

To simplify the example, we assume here that the return G is a single reward received at the end of the episode (this is why G is not subscripted by time) and that there is no discounting. In this case the update is also known as the Least Mean Square (LMS) rule. As a Monte Carlo algorithm, all the updates depend on the final reward/return, so none can be made until the end of the episode. The MC algorithm is an offline algorithm and we do not seek to improve this aspect of it. Rather we seek merely an implementation of this algorithm with computational advantages. We will still update the weight vector only at the end of the episode, but we will do some computation during each step of the episode and less at its end. This will give a more equal distribution of computation—$O(d)$ per step—and also remove the need to store the feature vectors at each step for use later at the end of each episode. Instead, we will introduce an additional vector memory, the eligibility trace, keeping in it a summary of all the feature vectors seen so far. This will be sufficient to efficiently recreate exactly the same overall update as the sequence of MC

updates (12.13), by the end of the episode:

$$\begin{aligned}
\mathbf{w}_T &= \mathbf{w}_{T-1} + \alpha \left(G - \mathbf{w}_{T-1}^\top \mathbf{x}_{T-1}\right) \mathbf{x}_{T-1} \\
&= \mathbf{w}_{T-1} + \alpha \mathbf{x}_{T-1} \left(-\mathbf{x}_{T-1}^\top \mathbf{w}_{T-1}\right) + \alpha G \mathbf{x}_{T-1} \\
&= \left(\mathbf{I} - \alpha \mathbf{x}_{T-1} \mathbf{x}_{T-1}^\top\right) \mathbf{w}_{T-1} + \alpha G \mathbf{x}_{T-1} \\
&= \mathbf{F}_{T-1} \mathbf{w}_{T-1} + \alpha G \mathbf{x}_{T-1}
\end{aligned}$$

where $\mathbf{F}_t \doteq \mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^\top$ is a *forgetting*, or *fading*, matrix. Now, recursing,

$$\begin{aligned}
&= \mathbf{F}_{T-1} \left(\mathbf{F}_{T-2} \mathbf{w}_{T-2} + \alpha G \mathbf{x}_{T-2}\right) + \alpha G \mathbf{x}_{T-1} \\
&= \mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{w}_{T-2} + \alpha G \left(\mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&= \mathbf{F}_{T-1} \mathbf{F}_{T-2} \left(\mathbf{F}_{T-3} \mathbf{w}_{T-3} + \alpha G \mathbf{x}_{T-3}\right) + \alpha G \left(\mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&= \mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{F}_{T-3} \mathbf{w}_{T-3} + \alpha G \left(\mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{x}_{T-3} + \mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&\;\;\vdots \\
&= \underbrace{\mathbf{F}_{T-1} \mathbf{F}_{T-2} \cdots \mathbf{F}_0 \mathbf{w}_0}_{\mathbf{a}_{T-1}} \;+\; \underbrace{\alpha G \sum_{k=0}^{T-1} \mathbf{F}_{T-1} \mathbf{F}_{T-2} \cdots \mathbf{F}_{k+1} \mathbf{x}_k}_{\mathbf{z}_{T-1}} \\
&= \mathbf{a}_{T-1} + \alpha G \mathbf{z}_{T-1},
\end{aligned} \tag{12.14}$$

where $\mathbf{a}_{T-1}$ and $\mathbf{z}_{T-1}$ are the values at time $T-1$ of two auxilary memory vectors that can be updated incrementally without knowledge of G and with $O(d)$ complexity per time step. The $\mathbf{z}_t$ vector is in fact a dutch-style eligibility trace. It is initialized to $\mathbf{z}_0 = \mathbf{x}_0$ and then updated according to

$$\begin{aligned}
\mathbf{z}_t &\doteq \sum_{k=0}^{t} \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_{k+1} \mathbf{x}_k, \qquad 1 \le t < T \\
&= \sum_{k=0}^{t-1} \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_{k+1} \mathbf{x}_k + \mathbf{x}_t \\
&= \mathbf{F}_t \sum_{k=0}^{t-1} \mathbf{F}_{t-1} \mathbf{F}_{t-2} \cdots \mathbf{F}_{k+1} \mathbf{x}_k + \mathbf{x}_t \\
&= \mathbf{F}_t \mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \left(\mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^\top\right) \mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} - \alpha \mathbf{x}_t \mathbf{x}_t^\top \mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} - \alpha \left(\mathbf{z}_{t-1}^\top \mathbf{x}_t\right) \mathbf{x}_t + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} + \left(1 - \alpha \mathbf{z}_{t-1}^\top \mathbf{x}_t\right) \mathbf{x}_t,
\end{aligned}$$

which is the dutch trace for the case of $\gamma\lambda = 1$ (cf. Eq. 12.11). The $\mathbf{a}_t$ auxilary vector is initialized to $\mathbf{a}_0 = \mathbf{w}_0$ and then updated according to

$$\mathbf{a}_t \;\doteq\; \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_0 \mathbf{w}_0 \;=\; \mathbf{F}_t \mathbf{a}_{t-1} \;=\; \mathbf{a}_{t-1} - \alpha \mathbf{x}_t \mathbf{x}_t^\top \mathbf{a}_{t-1}, \quad 1 \le t < T.$$

The auxiliary vectors, $\mathbf{a}_t$ and $\mathbf{z}_t$, are updated on each time step $t < T$ and then, at time T when G is observed, they are used in (12.14) to compute $\mathbf{w}_T$. In this way we achieve exactly the same final result as the MC/LMS algorithm that has poor computational properties (12.13), but now with an incremental algorithm whose time and memory complexity per step is $O(d)$. This is surprising and intriguing because the notion of an eligibility trace (and the dutch trace in particular) has arisen in a setting without temporal-difference (TD) learning (in contrast to van Seijen and Sutton, 2014). It seems eligibility traces are not specific to TD learning at all; they are more fundamental than that. The need for eligibility traces seems to arise whenever one tries to learn long-term predictions in an efficient manner.

12.7 Sarsa(λ)

Very few changes in the ideas already presented in this chapter are required in order to extend eligibility-traces to action-value methods. To learn approximate action values, $\hat{q}(s, a, \mathbf{w})$, rather than approximate state values, $\hat{v}(s,\mathbf{w})$, we need to use the action-value form of the n-step return, from Chapter 10:

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \qquad t + n < T,$$

with $G_{t:t+n} \doteq G_t$ if $t + n \geq T$. Using this, we can form the action-value form of the truncated λ-return, which is otherwise identical to the state-value form (12.9). The action-value form of the offline λ-return algorithm (12.4) simply uses $\hat{q}$ rather than $\hat{v}$:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \Big[G_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}_t) \Big] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad t = 0, \ldots, T-1, \tag{12.15}$$

where $G_t^\lambda \doteq G_{t:\infty}^\lambda$. The compound backup diagram for this forward view is shown in Figure 12.9. Notice the similarity to the diagram of the TD(λ) algorithm (Figure 12.1). The first update looks ahead one full step, to the next state–action pair, the second looks ahead two steps, to the second state–action pair, and so on. A final update is based on the complete return. The weighting of each n-step update in the λ-return is just as in TD(λ) and the λ-return algorithm (12.3).

The temporal-difference method for action values, known as *Sarsa(λ)*, approximates this forward view. It has the same update rule as given earlier for TD(λ):

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t,$$

except, naturally, using the action-value form of the TD error:

$$\delta_t \doteq R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{12.16}$$

and the action-value form of the eligibility trace:

$$\mathbf{z}_{-1} \doteq \mathbf{0},$$
$$\mathbf{z}_t \doteq \gamma \lambda \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad 0 \leq t \leq T.$$

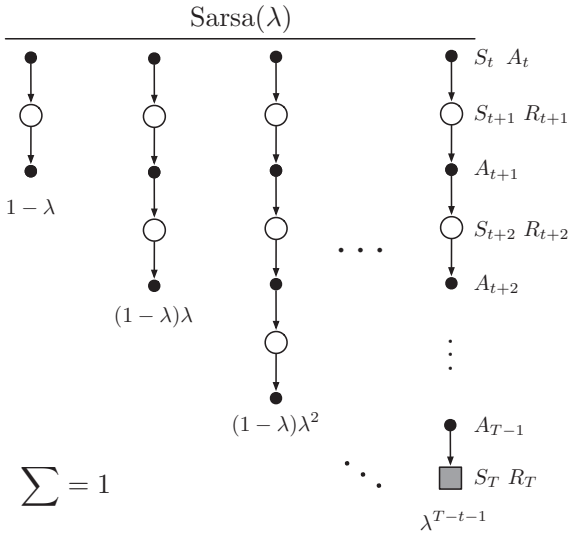
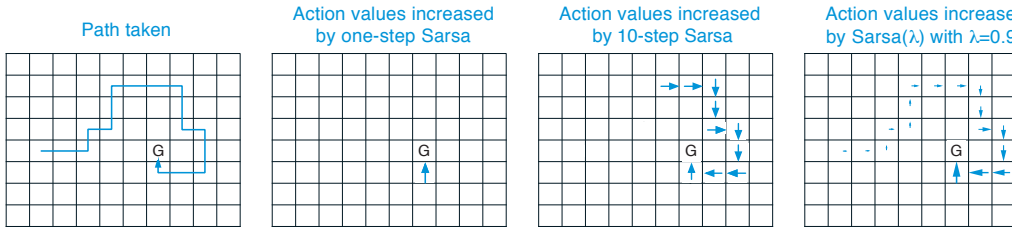


Figure 12.9: Sarsa(λ)'s backup diagram. Compare with Figure 12.1.

Complete pseudocode for Sarsa(λ) with linear function approximation, binary features, and either accumulating or replacing traces is given in the box on the next page. This pseudocode highlights a few optimizations possible in the special case of binary features (features are either active (=1) or inactive (=0).

Example 12.1: Traces in Gridworld The use of eligibility traces can substantially increase the efficiency of control algorithms over one-step methods and even over n-step methods. The reason for this is illustrated by the gridworld example below.



The first panel shows the path taken by an agent in a single episode. The initial estimated values were zero, and all rewards were zero except for a positive reward at the goal location marked by G. The arrows in the other panels show, for various algorithms, which action-values would be increased, and by how much, upon reaching the goal. A one-step method would increment only the last action value, whereas an n-step method would equally increment the last n actions' values, and an eligibility trace method would update all the action values up to the beginning of the episode, to different degrees, fading with recency. The fading strategy is often the best. ■

Sarsa(λ) with binary features and linear function approximation for estimating $\mathbf{w}^\top\mathbf{x} \approx q_\pi$ or q_*

Input: a function $\mathcal{F}(s,a)$ returning the set of (indices of) active features for s, a
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize: $\mathbf{w} = (w_1, \ldots, w_d)^\top \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$), $\mathbf{z} = (z_1, \ldots, z_d)^\top \in \mathbb{R}^d$

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or ε-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
  $\mathbf{z} \leftarrow \mathbf{0}$
  Loop for each step of episode:
    Take action A, observe R, S'
    $\delta \leftarrow R$
    Loop for i in $\mathcal{F}(S, A)$:
      $\delta \leftarrow \delta - w_i$
      $z_i \leftarrow z_i + 1$ (accumulating traces)
      or $z_i \leftarrow 1$ (replacing traces)
    If S' is terminal then:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
      Go to next episode
    Choose $A' \sim \pi(\cdot|S')$ or near greedily $\sim \hat{q}(S', \cdot, \mathbf{w})$
    Loop for i in $\mathcal{F}(S', A')$: $\delta \leftarrow \delta + \gamma w_i$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z}$
    $S \leftarrow S'; A \leftarrow A'$

Exercise 12.6 Modify the pseudocode for Sarsa(λ) to use dutch traces (12.11) without the other distinctive features of a true online algorithm. Assume linear function approximation and binary features. □

Example 12.2: Sarsa(λ) on Mountain Car Figure 12.10 (left) on the next page shows results with Sarsa(λ) on the Mountain Car task introduced in Example 10.1. The function approximation, action selection, and environmental details were exactly as in Chapter 10, and thus it is appropriate to numerically compare these results with the Chapter 10 results for n-step Sarsa (right side of the figure). The earlier results varied the update length n whereas here for Sarsa(λ) we vary the trace parameter λ, which plays a similar role. The fading-trace bootstrapping strategy of Sarsa(λ) appears to result in more efficient learning on this problem. ■

There is also an action-value version of our ideal TD method, the online λ-return algorithm (Section 12.4) and its efficient implementation as true online TD(λ) (Section 12.5). Everything in Section 12.4 goes through without change other than to use the action-value form of the n-step return given at the beginning of the current section. The analyses in Sections 12.5 and 12.6 also carry through for action values, the only change being the use

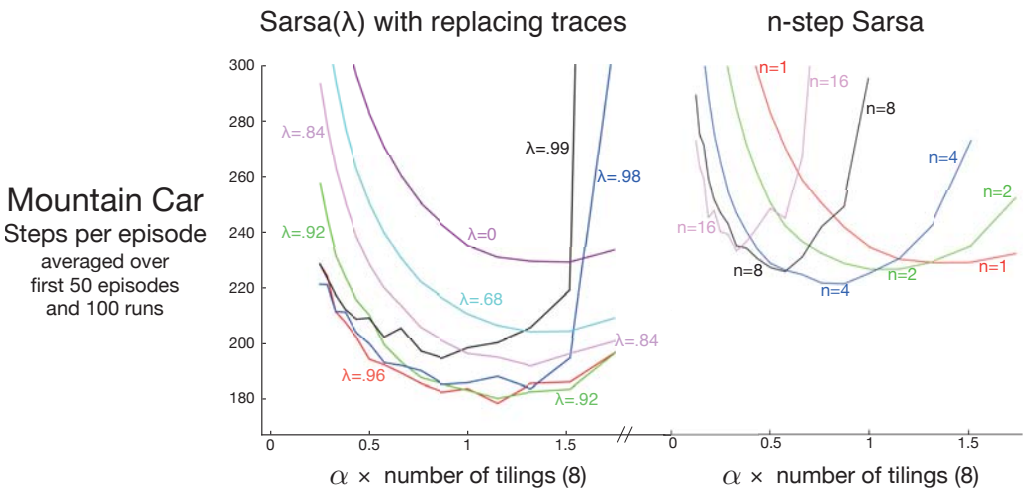


Figure 12.10: Early performance on the Mountain Car task of Sarsa(λ) with replacing traces and n-step Sarsa (copied from Figure 10.4) as a function of the step size, α.

of state–action feature vectors $\mathbf{x}_t = \mathbf{x}(S_t, A_t)$ instead of state feature vectors $\mathbf{x}_t = \mathbf{x}(S_t)$. Pseudocode for the resulting efficient algorithm, called *true online Sarsa(λ)* is given in the box on the next page. The figure below compares the performance of various versions of Sarsa(λ) on the Mountain Car example.

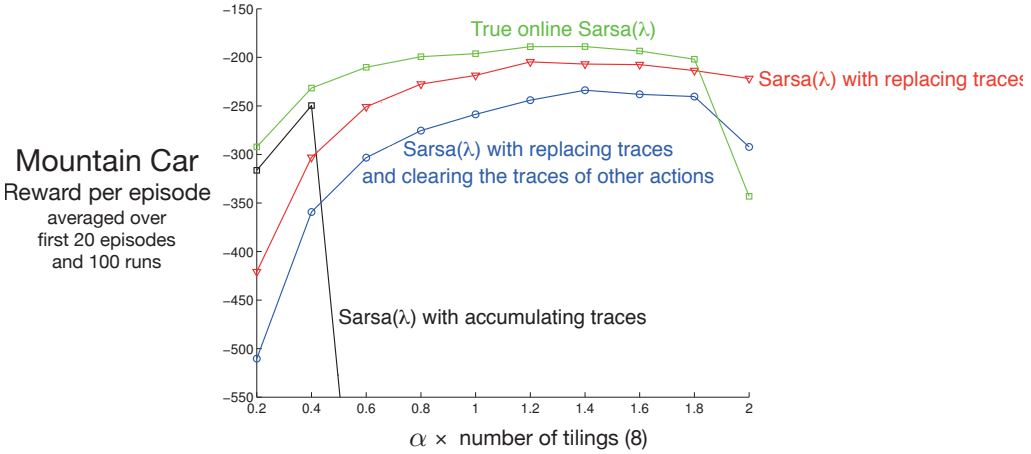


Figure 12.11: Summary comparison of Sarsa(λ) algorithms on the Mountain Car task. True online Sarsa(λ) performed better than regular Sarsa(λ) with both accumulating and replacing traces. Also included is a version of Sarsa(λ) with replacing traces in which, on each time step, the traces for the state and the actions not selected were set to zero.

True online Sarsa(λ) for estimating $\mathbf{w}^\top\mathbf{x} \approx q_\pi$ or q_*

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
Initialize: $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or near greedily from S using $\mathbf{w}$
  $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
  $\mathbf{z} \leftarrow \mathbf{0}$
  $Q_{old} \leftarrow 0$
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose $A' \sim \pi(\cdot|S')$ or near greedily from S' using $\mathbf{w}$
    $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
    $Q \leftarrow \mathbf{w}^\top\mathbf{x}$
    $Q' \leftarrow \mathbf{w}^\top\mathbf{x}'$
    $\delta \leftarrow R + \gamma Q' - Q$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \big(1 - \alpha\gamma\lambda\mathbf{z}^\top\mathbf{x}\big)\,\mathbf{x}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
    $Q_{old} \leftarrow Q'$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    $A \leftarrow A'$
  until S' is terminal

Finally, there is also a truncated version of Sarsa(λ), called *forward Sarsa(λ)* (van Seijen, 2016), which appears to be a particularly effective model-free control method for use in conjunction with multi-layer artificial neural networks.

12.8 Variable λ and γ

We are starting now to reach the end of our development of fundamental TD learning algorithms. To present the final algorithms in their most general forms, it is useful to generalize the degree of bootstrapping and discounting beyond constant parameters to functions potentially dependent on the state and action. That is, each time step will have a different λ and γ, denoted λ_t and γ_t. We change notation now so that $\lambda : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is now a function from states and actions to the unit interval such that $\lambda_t \doteq \lambda(S_t, A_t)$, and similarly, $\gamma : \mathcal{S} \to [0, 1]$ is a function from states to the unit interval such that $\gamma_t \doteq \gamma(S_t)$.

Introducing the function γ, the *termination function*, is particularly significant because it changes the return, the fundamental random variable whose expectation we seek to

estimate. Now the return is defined more generally as

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma_{t+1} G_{t+1} \\ &= R_{t+1} + \gamma_{t+1} R_{t+2} + \gamma_{t+1}\gamma_{t+2} R_{t+3} + \gamma_{t+1}\gamma_{t+2}\gamma_{t+3} R_{t+4} + \cdots \\ &= \sum_{k=t}^{\infty} \left(\prod_{i=t+1}^{k} \gamma_i \right) R_{k+1}, \end{aligned} \tag{12.17}$$

where, to assure the sums are finite, we require that $\prod_{k=t}^{\infty} \gamma_k = 0$ with probability one for all t. One convenient aspect of this definition is that it enables the episodic setting and its algorithms to be presented in terms of a single stream of experience, without special terminal states, start distributions, or termination times. An erstwhile terminal state becomes a state at which $\gamma(s) = 0$ and which transitions to the start distribution. In that way (and by choosing $\gamma(\cdot)$ as a constant in all other states) we can recover the classical episodic setting as a special case. State dependent termination includes other prediction cases such as *pseudo termination*, in which we seek to predict a quantity without altering the flow of the Markov process. Discounted returns can be thought of as such a quantity, in which case state dependent termination unifies the episodic and discounted-continuing cases. (The undiscounted-continuing case still needs some special treatment.)

The generalization to variable bootstrapping is not a change in the problem, like discounting, but a change in the solution strategy. The generalization affects the λ-returns for states and actions. The new state-based λ-return can be written recursively as

$$G_t^{\lambda s} \doteq R_{t+1} + \gamma_{t+1} \left((1 - \lambda_{t+1}) \hat{v}(S_{t+1}, \mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda s} \right), \tag{12.18}$$

where now we have added the "s" to the superscript λ to remind us that this is a return that bootstraps from state values, distinguishing it from returns that bootstrap from action values, which we present below with "a" in the superscript. This equation says that the λ-return is the first reward, undiscounted and unaffected by bootstrapping, plus possibly a second term to the extent that we are not discounting at the next state (that is, according to γ_{t+1}; recall that this is zero if the next state is terminal). To the extent that we aren't terminating at the next state, we have a second term which is itself divided into two cases depending on the degree of bootstrapping in the state. To the extent we are bootstrapping, this term is the estimated value at the state, whereas, to the extent that we not bootstrapping, the term is the λ-return for the next time step. The action-based λ-return is either the Sarsa form

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1} \Big((1 - \lambda_{t+1}) \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda a} \Big), \tag{12.19}$$

or the Expected Sarsa form,

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1} \Big((1 - \lambda_{t+1}) \bar{V}_t(S_{t+1}) + \lambda_{t+1} G_{t+1}^{\lambda a} \Big), \tag{12.20}$$

where (7.8) is generalized to function approximation as

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) \hat{q}(s, a, \mathbf{w}_t). \tag{12.21}$$

Exercise 12.7 Generalize the three recursive equations above to their truncated versions, defining $G^{\lambda s}_{t:h}$ and $G^{\lambda a}_{t:h}$. □

12.9 *Off-policy Traces with Control Variates

The final step is to incorporate importance sampling. Unlike in the case of n-step methods, for full non-truncated λ-returns one does not have a practical option in which the importance sampling is done outside the target return. Instead, we move directly to the bootstrapping generalization of per-decision importance sampling with control variates (Section 7.4). In the state case, our final definition of the λ-return generalizes (12.18), after the model of (7.13), to

$$G^{\lambda s}_t \doteq \rho_t\Big(R_{t+1}+\gamma_{t+1}\big((1-\lambda_{t+1})\hat{v}(S_{t+1},\mathbf{w}_t)+\lambda_{t+1}G^{\lambda s}_{t+1}\big)\Big)+(1-\rho_t)\hat{v}(S_t,\mathbf{w}_t) \quad (12.22)$$

where $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$ is the usual single-step importance sampling ratio. Much like the other returns we have seen in this book, the truncated version of this return can be approximated simply in terms of sums of the state-based TD error,

$$\delta^s_t \doteq R_{t+1}+\gamma_{t+1}\hat{v}(S_{t+1},\mathbf{w}_t)-\hat{v}(S_t,\mathbf{w}_t), \quad (12.23)$$

as

$$G^{\lambda s}_t \approx \hat{v}(S_t,\mathbf{w}_t)+\rho_t\sum_{k=t}^{\infty}\delta^s_k\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i \quad (12.24)$$

with the approximation becoming exact if the approximate value function does not change.

Exercise 12.8 Prove that (12.24) becomes exact if the value function does not change. To save writing, consider the case of $t=0$, and use the notation $V_k \doteq \hat{v}(S_k,\mathbf{w})$. □

Exercise 12.9 The truncated version of the general off-policy return is denoted $G^{\lambda s}_{t:h}$. Guess the correct equation, based on (12.24). □

The above form of the λ-return (12.24) is convenient to use in a forward-view update,

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t+\alpha\big(G^{\lambda s}_t-\hat{v}(S_t,\mathbf{w}_t)\big)\nabla\hat{v}(S_t,\mathbf{w}_t)\\ &\approx \mathbf{w}_t+\alpha\rho_t\left(\sum_{k=t}^{\infty}\delta^s_k\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\right)\nabla\hat{v}(S_t,\mathbf{w}_t),\end{aligned}$$

which to the experienced eye looks like an eligibility-based TD update—the product is like an eligibility trace and it is multiplied by TD errors. But this is just one time step of a forward view. The relationship that we are looking for is that the forward-view update, summed over time, is approximately equal to a backward-view update, summed over time (this relationship is only approximate because again we ignore changes in the value function). The sum of the forward-view update over time is

$$\begin{aligned}\sum_{t=1}^{\infty}(\mathbf{w}_{t+1}-\mathbf{w}_t) &\approx \sum_{t=1}^{\infty}\sum_{k=t}^{\infty}\alpha\rho_t\delta^s_k\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\\ &= \sum_{k=1}^{\infty}\sum_{t=1}^{k}\alpha\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\delta^s_k\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\\ &\qquad\qquad\text{(using the summation rule: } \textstyle\sum_{t=x}^{y}\sum_{k=t}^{y}=\sum_{k=x}^{y}\sum_{t=x}^{k}\text{)}\\ &= \sum_{k=1}^{\infty}\alpha\delta^s_k\sum_{t=1}^{k}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i,\end{aligned}$$

which would be in the form of the sum of a backward-view TD update if the entire expression from the second sum left could be written and updated incrementally as an eligibility trace, which we now show can be done. That is, we show that if this expression was the trace at time k, then we could update it from its value at time $k-1$ by:

$$\begin{aligned}\mathbf{z}_k &= \sum_{t=1}^{k}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\\ &= \sum_{t=1}^{k-1}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i \quad + \quad \rho_k\nabla\hat{v}(S_k,\mathbf{w}_k)\\ &= \gamma_k\lambda_k\rho_k\underbrace{\sum_{t=1}^{k-1}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k-1}\gamma_i\lambda_i\rho_i}_{\mathbf{z}_{k-1}} \quad + \quad \rho_k\nabla\hat{v}(S_k,\mathbf{w}_k)\\ &= \rho_k\big(\gamma_k\lambda_k\mathbf{z}_{k-1}+\nabla\hat{v}(S_k,\mathbf{w}_k)\big),\end{aligned}$$

which, changing the index from k to t, is the general accumulating trace update for state values:

$$\mathbf{z}_t \doteq \rho_t\big(\gamma_t\lambda_t\mathbf{z}_{t-1}+\nabla\hat{v}(S_t,\mathbf{w}_t)\big), \quad (12.25)$$

This eligibility trace, together with the usual semi-gradient parameter-update rule for TD(λ) (12.7), forms a general TD(λ) algorithm that can be applied to either on-policy or off-policy data. In the on-policy case, the algorithm is exactly TD(λ) because ρ_t is alway 1 and (12.25) becomes the usual accumulating trace (12.5) (extended to variable λ and γ). In the off-policy case, the algorithm often works well but, as a semi-gradient method, is not guaranteed to be stable. In the next few sections we will consider extensions of it that do guarantee stability.

A very similar series of steps can be followed to derive the off-policy eligibility traces for *action*-value methods and corresponding general Sarsa(λ) algorithms. One could start with either recursive form for the general action-based λ-return, (12.19) or (12.20), but the latter (the Expected Sarsa form) works out to be simpler. We extend (12.20) to the

off-policy case after the model of (7.14) to produce

$$G_t^{\lambda a} \doteq R_{t+1}+\gamma_{t+1}\Big((1-\lambda_{t+1})\bar{V}_t(S_{t+1})+\lambda_{t+1}\left[\rho_{t+1}G_{t+1}^{\lambda a}+\bar{V}_t(S_{t+1})-\rho_{t+1}\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t\right.$$
$$= R_{t+1}+\gamma_{t+1}\Big(\bar{V}_t(S_{t+1})+\lambda_{t+1}\rho_{t+1}\left[G_{t+1}^{\lambda a}-\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t)\right]\Big) \tag{12.26}$$

where $\bar{V}_t(S_{t+1})$ is as given by (12.21). Again the λ-return can be written approximately as the sum of TD errors,

$$G_t^{\lambda a} \approx \hat{q}(S_t,A_t,\mathbf{w}_t)+\sum_{k=t}^{\infty}\delta_k^a\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i, \tag{12.27}$$

using the expectation form of the action-based TD error:

$$\delta_t^a = R_{t+1}+\gamma_{t+1}\bar{V}_t(S_{t+1})-\hat{q}(S_t,A_t,\mathbf{w}_t). \tag{12.28}$$

As before, the approximation becomes exact if the approximate value function does not change.

Exercise 12.10 Prove that (12.27) becomes exact if the value function does not change. To save writing, consider the case of $t=0$, and use the notation $Q_k=\hat{q}(S_k,A_k,\mathbf{w})$. Hint: Start by writing out δ_0^a and $G_0^{\lambda a}$, then $G_0^{\lambda a}-Q_0$. □

Exercise 12.11 The truncated version of the general off-policy return is denoted $G_{t:h}^{\lambda a}$. Guess the correct equation for it, based on (12.27). □

Using steps entirely analogous to those for the state case, one can write a forward-view update based on (12.27), transform the sum of the updates using the summation rule, and finally derive the following form for the eligibility trace for action values:

$$\mathbf{z}_t \doteq \gamma_t\lambda_t\rho_t\mathbf{z}_{t-1}+\nabla\hat{q}(S_t,A_t,\mathbf{w}_t). \tag{12.29}$$

This eligibility trace, together with the expectation-based TD error (12.28) and the usual semi-gradient parameter-update rule (12.7), forms an elegant, efficient Expected Sarsa(λ) algorithm that can be applied to either on-policy or off-policy data. It is probably the best algorithm of this type at the current time (though of course it is not guaranteed to be stable until combined in some way with one of the methods presented in the following sections). In the on-policy case with constant λ and γ, and the usual state-action TD error (12.16), the algorithm would be identical to the Sarsa(λ) algorithm presented in Section 12.7.

Exercise 12.12 Show in detail the steps outlined above for deriving (12.29) from (12.27). Start with the update (12.15), substitute $G_t^{\lambda a}$ from (12.26) for G_t^{λ}, then follow similar steps as led to (12.25). □

At $\lambda=1$, these algorithms become closely related to corresponding Monte Carlo algorithms. One might expect that an exact equivalence would hold for episodic problems and offline updating, but in fact the relationship is subtler and slightly weaker than that. Under these most favorable conditions still there is not an episode by episode equivalence of updates, only of their expectations. This should not be surprising as these method

make irrevocable updates as a trajectory unfolds, whereas true Monte Carlo methods would make no update for a trajectory if any action within it has zero probability under the target policy. In particular, all of these methods, even at $\lambda=1$, still bootstrap in the sense that their targets depend on the current value estimates—it's just that the dependence cancels out in expected value. Whether this is a good or bad property in practice is another question. Recently, methods have been proposed that do achieve an exact equivalence (Sutton, Mahmood, Precup and van Hasselt, 2014). These methods require an additional vector of "provisional weights" that keep track of updates which have been made but may need to be retracted (or emphasized) depending on the actions taken later. The state and state–action versions of these methods are called PTD(λ) and PQ(λ) respectively, where the 'P' stands for Provisional.

The practical consequences of all these new off-policy methods have not yet been established. Undoubtedly, issues of high variance will arise as they do in all off-policy methods using importance sampling (Section 11.9).

If $\lambda<1$, then all these off-policy algorithms involve bootstrapping and the deadly triad applies (Section 11.3), meaning that they can be guaranteed stable only for the tabular case, for state aggregation, and for other limited forms of function approximation. For linear and more-general forms of function approximation the parameter vector may diverge to infinity as in the examples in Chapter 11. As we discussed there, the challenge of off-policy learning has two parts. Off-policy eligibility traces deal effectively with the first part of the challenge, correcting for the expected value of the targets, but not at all with the second part of the challenge, having to do with the distribution of updates. Algorithmic strategies for meeting the second part of the challenge of off-policy learning with eligibility traces are summarized in Section 12.11.

Exercise 12.13 What are the dutch-trace and replacing-trace versions of off-policy eligibility traces for state-value and action-value methods? □

12.10 Watkins's Q(λ) to Tree-Backup(λ)

Several methods have been proposed over the years to extend Q-learning to eligibility traces. The original is *Watkins's Q(λ)*, which decays its eligibility traces in the usual way as long as a greedy action was taken, then cuts the traces to zero after the first non-greedy action. The backup diagram for Watkins's Q(λ) is shown in Figure 12.12. In Chapter 6, we unified Q-learning and Expected Sarsa in the off-policy version of the latter, which includes Q-learning as a special case, and generalizes it to arbitrary target policies, and in the previous section of this chapter we completed our treatment of Expected Sarsa by generalizing it to off-policy eligibility traces. In Chapter 7, however, we distinguished n-step Expected Sarsa from n-step Tree Backup, where the latter retained the property of not using importance sampling. It remains then to present the eligibility trace version of Tree Backup, which we will call *Tree-Backup(λ)*, or *TB(λ)* for short. This is arguably the true successor to Q-learning because it retains its appealing absence of importance sampling even though it can be applied to off-policy data.

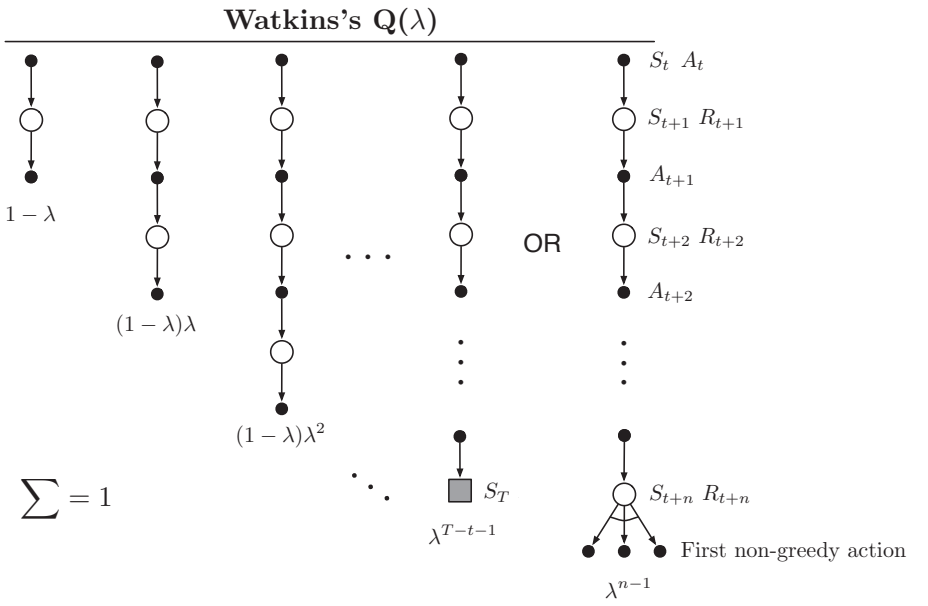


Figure 12.12: The backup diagram for Watkins's Q(λ). The series of component updates ends either with the end of the episode or with the first nongreedy action, whichever comes first.

The concept of TB(λ) is straightforward. As shown in its backup diagram in Figure 12.13, the tree-backup updates of each length (from Section 7.5) are weighted in the usual way dependent on the bootstrapping parameter λ. To get the detailed equations, with the right indices on the general bootstrapping and discounting parameters, it is best to start with a recursive form (12.20) for the λ-return using action values, and then expand the bootstrapping case of the target after the model of (7.16):

$$G_t^{\lambda a} \doteq R_{t+1}+\gamma_{t+1}\Big((1-\lambda_{t+1})\bar{V}_t(S_{t+1})+\lambda_{t+1}\Big[\sum_{a\neq A_{t+1}}\pi(a|S_{t+1})\hat{q}(S_{t+1},a,\mathbf{w}_t)+\pi(A_{t+1}|S_{t+1})G_t^{\lambda}$$

$$= R_{t+1}+\gamma_{t+1}\Big(\bar{V}_t(S_{t+1})+\lambda_{t+1}\pi(A_{t+1}|S_{t+1})\Big(G_{t+1}^{\lambda a}-\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t)\Big)\Big)$$

As per the usual pattern, it can also be written approximately (ignoring changes in the approximate value function) as a sum of TD errors,

$$G_t^{\lambda a} \approx \hat{q}(S_t,A_t,\mathbf{w}_t)+\sum_{k=t}^{\infty}\delta_k^a\prod_{i=t+1}^{k}\gamma_i\lambda_i\pi(A_i|S_i),$$

using the expectation form of the action-based TD error (12.28).

Following the same steps as in the previous section, we arrive at a special eligibility trace update involving the target-policy probabilities of the selected actions,

$$\mathbf{z}_t \doteq \gamma_t\lambda_t\pi(A_t|S_t)\mathbf{z}_{t-1}+\nabla\hat{q}(S_t,A_t,\mathbf{w}_t).$$

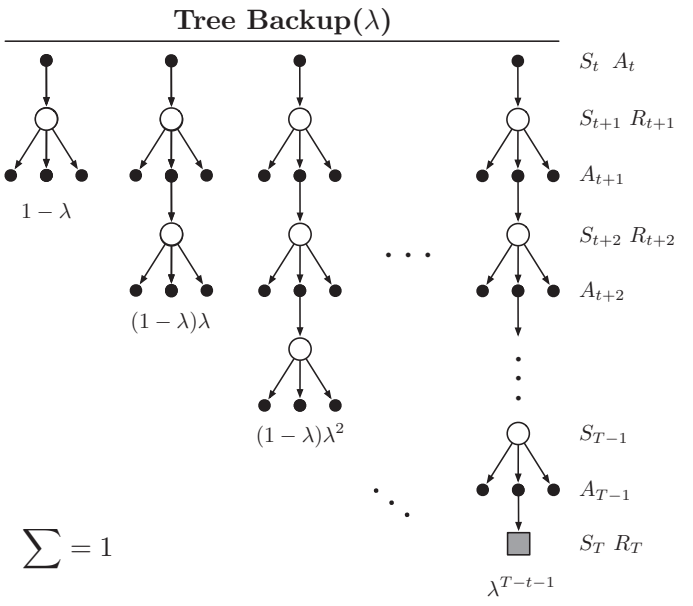


Figure 12.13: The backup diagram for the λ version of the Tree Backup algorithm.

This, together with the usual parameter-update rule (12.7), defines the TB(λ) algorithm. Like all semi-gradient algorithms, TB(λ) is not guaranteed to be stable when used with off-policy data and with a powerful function approximator. To obtain those assurances, TB(λ) would have to be combined with one of the methods presented in the next section.

**Exercise 12.14* How might Double Expected Sarsa be extended to eligibility traces? □

12.11 Stable Off-policy Methods with Traces

Several methods using eligibility traces have been proposed that achieve guarantees of stability under off-policy training, and here we present four of the most important using this book's standard notation, including general bootstrapping and discounting functions. All are based on either the Gradient-TD or the Emphatic-TD ideas presented in Sections 11.7 and 11.8. All the algorithms assume linear function approximation, though extensions to nonlinear function approximation can also be found in the literature.

GTD(λ) is the eligibility-trace algorithm analogous to TDC, the better of the two state-value Gradient-TD prediction algorithms discussed in Section 11.7. Its goal is to learn a parameter $\mathbf{w}_t$ such that $\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}_t^\top\mathbf{x}(s) \approx v_\pi(s)$, even from data that is due to following another policy b. Its update is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t+\alpha\delta_t^s\mathbf{z}_t-\alpha\gamma_{t+1}(1-\lambda_{t+1})\left(\mathbf{z}_t^\top\mathbf{v}_t\right)\mathbf{x}_{t+1},$$

with δ_t^s, $\mathbf{z}_t$, and ρ_t defined in the usual ways for state values (12.23) (12.25) (11.1), and

$$\mathbf{v}_{t+1} \doteq \mathbf{v}_t + \beta\delta_t^s\mathbf{z}_t - \beta\left(\mathbf{v}_t^\top\mathbf{x}_t\right)\mathbf{x}_t, \tag{12.30}$$

where, as in Section 11.7, $\mathbf{v} \in \mathbb{R}^d$ is a vector of the same dimension as $\mathbf{w}$, initialized to $\mathbf{v}_0 = \mathbf{0}$, and $\beta > 0$ is a second step-size parameter.

GQ(λ) is the Gradient-TD algorithm for action values with eligibility traces. Its goal is to learn a parameter $\mathbf{w}_t$ such that $\hat{q}(s,a,\mathbf{w}_t) \doteq \mathbf{w}_t^\top\mathbf{x}(s,a) \approx q_\pi(s,a)$ from off-policy data. If the target policy is ε-greedy, or otherwise biased toward the greedy policy for $\hat{q}$, then GQ(λ) can be used as a control algorithm. Its update is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t^a\mathbf{z}_t - \alpha\gamma_{t+1}(1-\lambda_{t+1})\left(\mathbf{z}_t^\top\mathbf{v}_t\right)\bar{\mathbf{x}}_{t+1},$$

where $\bar{\mathbf{x}}_t$ is the average feature vector for S_t under the target policy,

$$\bar{\mathbf{x}}_t \doteq \sum_a \pi(a|S_t)\mathbf{x}(S_t,a),$$

δ_t^a is the expectation form of the TD error, which can be written

$$\delta_t^a \doteq R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top\bar{\mathbf{x}}_{t+1} - \mathbf{w}_t^\top\mathbf{x}_t,$$

$\mathbf{z}_t$ is defined in the usual way for action values (12.29), and the rest is as in GTD(λ), including the update for $\mathbf{v}_t$ (12.30).

HTD(λ) is a hybrid state-value algorithm combining aspects of GTD(λ) and TD(λ). Its most appealing feature is that it is a strict generalization of TD(λ) to off-policy learning, meaning that if the behavior policy happens to be the same as the target policy, then HTD(λ) becomes the same as TD(λ), which is not true for GTD(λ). This is appealing because TD(λ) is often faster than GTD(λ) when both algorithms converge, and TD(λ) requires setting only a single step size. HTD(λ) is defined by

$$\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\delta_t^s\mathbf{z}_t + \alpha\left((\mathbf{z}_t - \mathbf{z}_t^b)^\top\mathbf{v}_t\right)(\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1}),\\
\mathbf{v}_{t+1} &\doteq \mathbf{v}_t + \beta\delta_t^s\mathbf{z}_t - \beta\left(\mathbf{z}_t^{b\top}\mathbf{v}_t\right)(\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1}), \quad \text{with } \mathbf{v}_0 \doteq \mathbf{0},\\
\mathbf{z}_t &\doteq \rho_t\big(\gamma_t\lambda_t\mathbf{z}_{t-1} + \mathbf{x}_t\big), \quad \text{with } \mathbf{z}_{-1} \doteq \mathbf{0},\\
\mathbf{z}_t^b &\doteq \gamma_t\lambda_t\mathbf{z}_{t-1}^b + \mathbf{x}_t, \quad \text{with } \mathbf{z}_{-1}^b \doteq \mathbf{0},
\end{aligned}$$

where $\beta > 0$ again is a second step-size parameter. In addition to the second set of weights, $\mathbf{v}_t$, HTD(λ) also has a second set of eligibility traces, $\mathbf{z}_t^b$. These are conventional accumulating eligibility traces for the behavior policy and become equal to $\mathbf{z}_t$ if all the ρ_t are 1, which causes the last term in the $\mathbf{w}_t$ update to be zero and the overall update to reduce to TD(λ).

Emphatic TD(λ) is the extension of the one-step Emphatic-TD algorithm (Sections 9.11 and 11.8) to eligibility traces. The resultant algorithm retains strong off-policy convergence guarantees while enabling any degree of bootstrapping, albeit at the cost of high variance and potentially slow convergence. Emphatic TD(λ) is defined by

$$\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t\\
\delta_t &\doteq R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top\mathbf{x}_{t+1} - \mathbf{w}_t^\top\mathbf{x}_t\\
\mathbf{z}_t &\doteq \rho_t\big(\gamma_t\lambda_t\mathbf{z}_{t-1} + M_t\mathbf{x}_t\big), \quad \text{with } \mathbf{z}_{-1} \doteq \mathbf{0},\\
M_t &\doteq \lambda_t I_t + (1-\lambda_t)F_t\\
F_t &\doteq \rho_{t-1}\gamma_t F_{t-1} + I_t, \quad \text{with } F_0 \doteq i(S_0),
\end{aligned}$$

where $M_t \geq 0$ is the general form of *emphasis*, $F_t \geq 0$ is termed the *followon trace*, and $I_t \geq 0$ is the *interest*, as described in Section 11.8. Note that M_t, like δ_t, is not really an additional memory variable. It can be removed from the algorithm by substituting its definition into the eligibility-trace equation. Pseudocode and software for the true online version of Emphatic-TD(λ) are available on the web (Sutton, 2015b).

In the on-policy case ($\rho_t = 1$, for all t), Emphatic-TD(λ) is similar to conventional TD(λ), but still significantly different. In fact, whereas Emphatic-TD(λ) is guaranteed to converge for all state-dependent λ functions, TD(λ) is not. TD(λ) is guaranteed convergent only for all constant λ. See Yu's counterexample (Ghiassian, Rafiee, and Sutton, 2016).

12.12 Implementation Issues

It might at first appear that tabular methods using eligibility traces are much more complex than one-step methods. A naive implementation would require every state (or state–action pair) to update both its value estimate and its eligibility trace on every time step. This would not be a problem for implementations on single-instruction, multiple-data, parallel computers or in plausible artificial neural network (ANN) implementations, but it is a problem for implementations on conventional serial computers. Fortunately, for typical values of λ and γ the eligibility traces of almost all states are almost always nearly zero; only those states that have recently been visited will have traces significantly greater than zero and only these few states need to be updated to closely approximate these algorithms.

In practice, then, implementations on conventional computers may keep track of and update only the few traces that are significantly greater than zero. Using this trick, the computational expense of using traces in tabular methods is typically just a few times that of a one-step method. The exact multiple of course depends on λ and γ and on the expense of the other computations. Note that the tabular case is in some sense the worst case for the computational complexity of eligibility traces. When function approximation is used, the computational advantages of not using traces generally decrease. For example, if ANNs and backpropagation are used, then eligibility traces generally cause only a doubling of the required memory and computation per step. truncated λ-return methods (Section 12.3) can be computationally efficient on conventional computers though they always require some additional memory.

12.13 Conclusions

Eligibility traces in conjunction with TD errors provide an efficient, incremental way of shifting and choosing between Monte Carlo and TD methods. The n-step methods of Chapter 7 also enabled this, but eligibility trace methods are more general, often faster to learn, and offer different computational complexity tradeoffs. This chapter has offered an introduction to the elegant, emerging theoretical understanding of eligibility traces for on- and off-policy learning and for variable bootstrapping and discounting. One aspect of this elegant theory is true online methods, which exactly reproduce the behavior of expensive ideal methods while retaining the computational congeniality of conventional TD methods. Another aspect is the possibility of derivations that automatically convert from intuitive forward-view methods to more efficient incremental backward-view algorithms. We illustrated this general idea in a derivation that started with a classical, expensive Monte Carlo algorithm and ended with a cheap incremental non-TD implementation using the same novel eligibility trace used in true online TD methods.

As we mentioned in Chapter 5, Monte Carlo methods may have advantages in non-Markov tasks because they do not bootstrap. Because eligibility traces make TD methods more like Monte Carlo methods, they also can have advantages in these cases. If one wants to use TD methods because of their other advantages, but the task is at least partially non-Markov, then the use of an eligibility trace method is indicated. Eligibility traces are the first line of defense against both long-delayed rewards and non-Markov tasks.

By adjusting λ, we can place eligibility trace methods anywhere along a continuum from Monte Carlo to one-step TD methods. Where shall we place them? We do not yet have a good theoretical answer to this question, but a clear empirical answer appears to be emerging. On tasks with many steps per episode, or many steps within the half-life of discounting, it appears significantly better to use eligibility traces than not to (e.g., see Figure 12.14). On the other hand, if the traces are so long as to produce a pure Monte Carlo method, or nearly so, then performance degrades sharply. An intermediate mixture appears to be the best choice. Eligibility traces should be used to bring us toward Monte Carlo methods, but not all the way there. In the future it may be possible to more finely vary the trade-off between TD and Monte Carlo methods by using variable λ, but at present it is not clear how this can be done reliably and usefully.

Methods using eligibility traces require more computation than one-step methods, but in return they offer significantly faster learning, particularly when rewards are delayed by many steps. Thus it often makes sense to use eligibility traces when data are scarce and cannot be repeatedly processed, as is often the case in online applications. On the other hand, in offline applications in which data can be generated cheaply, perhaps from an inexpensive simulation, then it often does not pay to use eligibility traces. In these cases the objective is not to get more out of a limited amount of data, but simply to process as much data as possible as quickly as possible. In these cases the speedup per datum due to traces is typically not worth their computational cost, and one-step methods are favored.

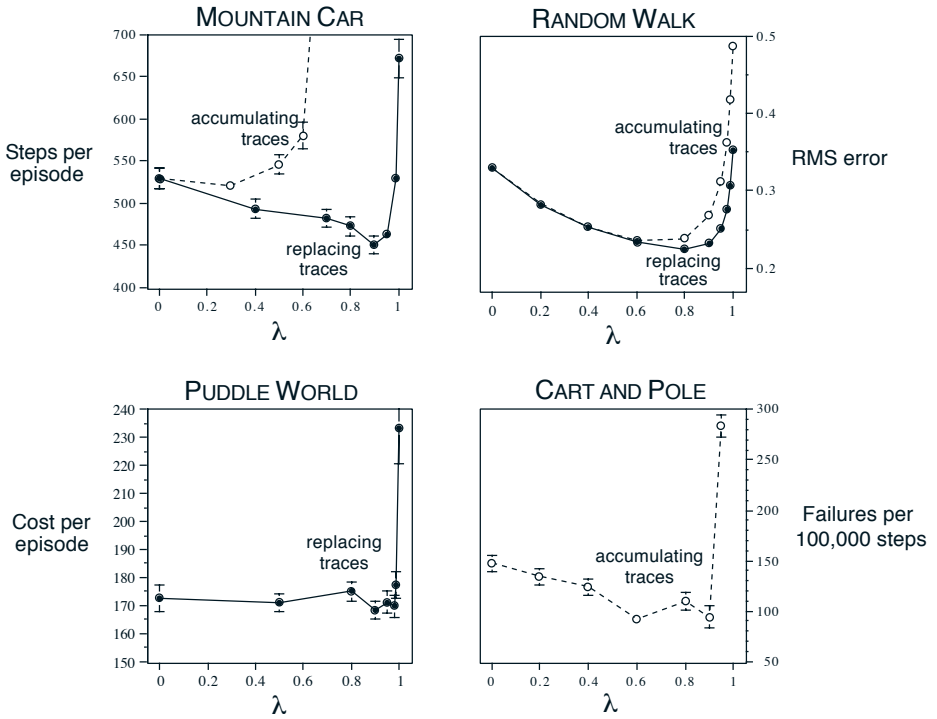


Figure 12.14: The effect of λ on reinforcement learning performance in four different test problems. In all cases, performance is generally best (a *lower* number in the graph) at an intermediate value of λ. The two left panels are applications to simple continuous-state control tasks using the Sarsa(λ) algorithm and tile coding, with either replacing or accumulating traces (Sutton, 1996). The upper-right panel is for policy evaluation on a random walk task using TD(λ) (Singh and Sutton, 1996). The lower right panel is unpublished data for the pole-balancing task (Example 3.4) from an earlier study (Sutton, 1984).

Bibliographical and Historical Remarks

Eligibility traces came into reinforcement learning via the fecund ideas of Klopf (1972). Our use of eligibility traces is based on Klopf's work (Sutton, 1978a, 1978b, 1978c; Barto and Sutton, 1981a, 1981b; Sutton and Barto, 1981a; Barto, Sutton, and Anderson, 1983; Sutton, 1984). We may have been the first to use the term "eligibility trace" (Sutton and Barto, 1981a). The idea that stimuli produce after effects in the nervous system that are important for learning is very old (see Chapter 14). Some of the earliest uses of eligibility traces were in the actor–critic methods discussed in Chapter 13 (Barto, Sutton, and Anderson, 1983; Sutton, 1984).

12.1 Compound updates were called "complex backups" in the first edition of this book.

The λ-return and its error-reduction properties were introduced by Watkins (1989) and further developed by Jaakkola, Jordan, and Singh (1994). The random walk results in this and subsequent sections are new to this text, as are the terms "forward view" and "backward view." The notion of a λ-return algorithm was introduced in the first edition of this text. The more refined treatment presented here was developed in conjunction with Harm van Seijen (e.g., van Seijen and Sutton, 2014).

12.2 TD(λ) with accumulating traces was introduced by Sutton (1988, 1984). Convergence in the mean was proved by Dayan (1992), and with probability 1 by many researchers, including Peng (1993), Dayan and Sejnowski (1994), Tsitsiklis (1994), and Gurvits, Lin, and Hanson (1994). The bound on the error of the asymptotic λ-dependent solution of linear TD(λ) is due to Tsitsiklis and Van Roy (1997).

12.3 Truncated TD methods were developed by Cichosz (1995) and van Seijen (2016).

12.4 The idea of redoing updates was extensively developed by van Seijen, originally under the name "best-match learning" (van Seijen, 2011; van Seijen, Whiteson, van Hasselt, and Weiring, 2011).

12.5 True online TD(λ) is primarily due to Harm van Seijen (van Seijen and Sutton, 2014; van Seijen et al., 2016) though some of its key ideas were discovered independently by Hado van Hasselt (personal communication). The name "dutch traces" is in recognition of the contributions of both scientists.

Replacing traces are due to Singh and Sutton (1996).

12.6 The material in this section is from van Hasselt and Sutton (2015).

12.7 Sarsa(λ) with accumulating traces was first explored as a control method by Rummery and Niranjan (1994; Rummery, 1995). True online Sarsa(λ) was introduced by van Seijen and Sutton (2014). The algorithm on page 307 was adapted from van Seijen et al. (2016). The Mountain Car results were made for this text, except for Figure 12.11 which is adapted from van Seijen and Sutton (2014).

12.8 Perhaps the first published discussion of variable λ was by Watkins (1989), who pointed out that the cutting off of the update sequence (Figure 12.12) in his Q(λ) when a nongreedy action was selected could be implemented by temporarily setting λ to 0.

Variable λ was introduced in the first edition of this text. The roots of variable γ are in the work on options (Sutton, Precup, and Singh, 1999) and its precursors (Sutton, 1995a), becoming explicit in the GQ(λ) paper (Maei and Sutton, 2010), which also introduced some of these recursive forms for the λ-returns.

A different notion of variable λ has been developed by Yu (2012).

12.9 Off-policy eligibility traces were introduced by Precup et al. (2000, 2001), then further developed by Bertsekas and Yu (2009), Maei (2011; Maei and Sutton, 2010), Yu (2012), and by Sutton, Mahmood, Precup, and van Hasselt (2014). The last reference in particular gives a powerful forward view for off-policy TD methods with general state-dependent λ and γ. The presentation here seems to be new.

This section ends with an elegant Expected Sarsa(λ) algorithm. Although it is a natural algorithm, to our knowledge it has not previously been described or tested in the literature.

12.10 Watkins's Q(λ) is due to Watkins (1989). The tabular, episodic, offline version has been proven convergent by Munos, Stepleton, Harutyunyan, and Bellemare (2016). Alternative Q(λ) algorithms were proposed by Peng and Williams (1994, 1996) and by Sutton, Mahmood, Precup, and van Hasselt (2014). Tree Backup(λ) is due to Precup, Sutton, and Singh (2000).

12.11 GTD(λ) is due to Maei (2011). GQ(λ) is due to Maei and Sutton (2010). HTD(λ) is due to White and White (2016) based on the one-step HTD algorithm introduced by Hackman (2012). The latest developments in the theory of Gradient-TD methods are by Yu (2017). Emphatic TD(λ) was introduced by Sutton, Mahmood, and White (2016), who proved its stability. Yu (2015, 2016) proved its convergence, and the algorithm was developed further by Hallak et al. (2015, 2016).

Chapter 13

Policy Gradient Methods

In this chapter we consider something new. So far in this book almost all the methods have been *action-value methods*; they learned the values of actions and then selected actions based on their estimated action values[1]; their policies would not even exist without the action-value estimates. In this chapter we consider methods that instead learn a *parameterized policy* that can select actions without consulting a value function. A value function may still be used to *learn* the policy parameter, but is not required for action selection. We use the notation $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ for the policy's parameter vector. Thus we write $\pi(a|s,\boldsymbol{\theta}) = \Pr\{A_t{=}a \mid S_t{=}s, \boldsymbol{\theta}_t{=}\boldsymbol{\theta}\}$ for the probability that action a is taken at time t given that the environment is in state s at time t with parameter $\boldsymbol{\theta}$. If a method uses a learned value function as well, then the value function's weight vector is denoted $\mathbf{w} \in \mathbb{R}^d$ as usual, as in $\hat{v}(s,\mathbf{w})$.

In this chapter we consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter. These methods seek to *maximize* performance, so their updates approximate gradient *ascent* in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)}, \tag{13.1}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^d$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$. All methods that follow this general schema we call *policy gradient methods*, whether or not they also learn an approximate value function. Methods that learn approximations to both policy and value functions are often called *actor–critic methods*, where 'actor' is a reference to the learned policy, and 'critic' refers to the learned value function, usually a state-value function. First we treat the episodic case, in which performance is defined as the value of the start state under the parameterized policy, before going on to consider the continuing case, in which performance is defined as the average reward rate, as in Section 10.3. In the end, we are able to express the algorithms for both cases in very similar terms.

[1] The lone exception is the gradient bandit algorithms of Section 2.8. In fact, that section goes through many of the same steps, in the single-state bandit case, as we go through here for full MDPs. Reviewing that section would be good preparation for fully understanding this chapter.

13.1 Policy Approximation and its Advantages

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s,\boldsymbol{\theta})$ is differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s,\boldsymbol{\theta})$ (the column vector of partial derivatives of $\pi(a|s,\boldsymbol{\theta})$ with respect to the components of $\boldsymbol{\theta}$) exists and is finite for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. In practice, to ensure exploration we generally require that the policy never becomes deterministic (i.e., that $\pi(a|s,\boldsymbol{\theta}) \in (0,1)$, for all $s,a,\boldsymbol{\theta}$). In this section we introduce the most common parameterization for discrete action spaces and point out the advantages it offers over action-value methods. Policy-based methods also offer useful ways of dealing with continuous action spaces, as we describe later in Section 13.7.

If the action space is discrete and not too large, then a natural and common kind of parameterization is to form parameterized numerical preferences $h(s,a,\boldsymbol{\theta}) \in \mathbb{R}$ for each state–action pair. The actions with the highest preferences in each state are given the highest probabilities of being selected, for example, according to an exponential soft-max distribution:

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}, \tag{13.2}$$

where $e \approx 2.71828$ is the base of the natural logarithm. Note that the denominator here is just what is required so that the action probabilities in each state sum to one. We call this kind of policy parameterization *soft-max in action preferences*.

The action preferences themselves can be parameterized arbitrarily. For example, they might be computed by a deep artificial neural network (ANN), where $\boldsymbol{\theta}$ is the vector of all the connection weights of the network (as in the AlphaGo system described in Section 16.6). Or the preferences could simply be linear in features,

$$h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^\top\mathbf{x}(s,a), \tag{13.3}$$

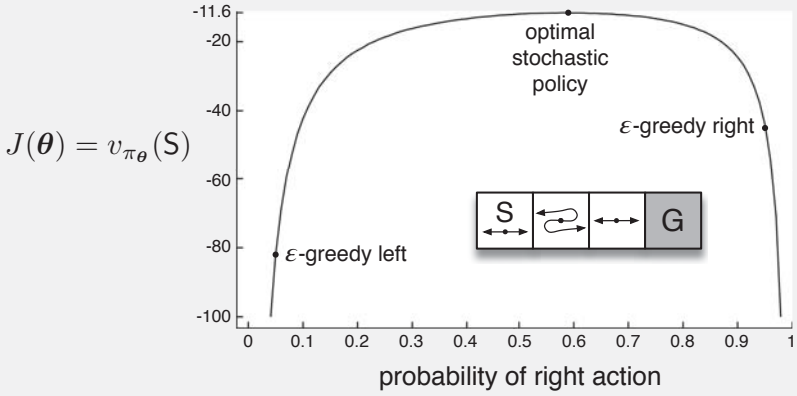
using feature vectors $\mathbf{x}(s,a) \in \mathbb{R}^{d'}$ constructed by any of the methods described in Chapter 9.

One advantage of parameterizing policies according to the soft-max in action preferences is that the approximate policy can approach a deterministic policy, whereas with ε-greedy action selection over action values there is always an ε probability of selecting a random action. Of course, one could select according to a soft-max distribution based on action values, but this alone would not allow the policy to approach a deterministic policy. Instead, the action-value estimates would converge to their corresponding true values, which would differ by a finite amount, translating to specific probabilities other than 0 and 1. If the soft-max distribution included a temperature parameter, then the temperature could be reduced over time to approach determinism, but in practice it would be difficult to choose the reduction schedule, or even the initial temperature, without more prior knowledge of the true action values than we would like to assume. Action preferences are different because they do not approach specific values; instead they are driven to produce the optimal stochastic policy. If the optimal policy is deterministic, then the preferences of the optimal actions will be driven infinitely higher than all suboptimal actions (if permitted by the parameterization).

A second advantage of parameterizing policies according to the soft-max in action preferences is that it enables the selection of actions with arbitrary probabilities. In problems with significant function approximation, the best approximate policy may be stochastic. For example, in card games with imperfect information the optimal play is often to do two different things with specific probabilities, such as when bluffing in Poker. Action-value methods have no natural way of finding stochastic optimal policies, whereas policy approximating methods can, as shown in Example 13.1.

Example 13.1 Short corridor with switched actions

Consider the small corridor gridworld shown inset in the graph below. The reward is -1 per step, as usual. In each of the three nonterminal states there are only two actions, right and left. These actions have their usual consequences in the first and third states (left causes no movement in the first state), but in the second state they are reversed, so that right moves to the left and left moves to the right. The problem is difficult because all the states appear identical under the function approximation. In particular, we define $\mathbf{x}(s, \text{right}) = [1, 0]^\top$ and $\mathbf{x}(s, \text{left}) = [0, 1]^\top$, for all s. An action-value method with ε-greedy action selection is forced to choose between just two policies: choosing right with high probability $1 - \varepsilon/2$ on all steps or choosing left with the same high probability on all time steps. If $\varepsilon = 0.1$, then these two policies achieve a value (at the start state) of less than -44 and -82, respectively, as shown in the graph. A method can do significantly better if it can learn a specific probability with which to select right. The best probability is about 0.59, which achieves a value of about -11.6.



Perhaps the simplest advantage that policy parameterization may have over action-value parameterization is that the policy may be a simpler function to approximate. Problems vary in the complexity of their policies and action-value functions. For some, the action-value function is simpler and thus easier to approximate. For others, the policy is simpler. In the latter case a policy-based method will typically learn faster and yield a superior asymptotic policy (as in Tetris; see Şimşek, Algórta, and Kothiyal, 2016).

Finally, we note that the choice of policy parameterization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the reinforcement learning system. This is often the most important reason for using a policy-based learning method.

Exercise 13.1 Use your knowledge of the gridworld and its dynamics to determine an *exact* symbolic expression for the optimal probability of selecting the right action in Example 13.1. □

13.2 The Policy Gradient Theorem

In addition to the practical advantages of policy parameterization over ε-greedy action selection, there is also an important theoretical advantage. With continuous policy parameterization the action probabilities change smoothly as a function of the learned parameter, whereas in ε-greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action values, if that change results in a different action having the maximal value. Largely because of this stronger convergence guarantees are available for policy-gradient methods than for action-value methods. In particular, it is the continuity of the policy dependence on the parameters that enables policy-gradient methods to approximate gradient ascent (13.1).

The episodic and continuing cases define the performance measure, $J(\boldsymbol{\theta})$, differently and thus have to be treated separately to some extent. Nevertheless, we will try to present both cases uniformly, and we develop a notation so that the major theoretical results can be described with a single set of equations.

In this section we treat the episodic case, for which we define the performance measure as the value of the start state of the episode. We can simplify the notation without losing any meaningful generality by assuming that every episode starts in some particular (non-random) state s_0. Then, in the episodic case we define performance as

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0), \tag{13.4}$$

where $v_{\pi_{\boldsymbol{\theta}}}$ is the true value function for $\pi_{\boldsymbol{\theta}}$, the policy determined by $\boldsymbol{\theta}$. From here on in our discussion we will assume no discounting ($\gamma = 1$) for the episodic case, although for completeness we do include the possibility of discounting in the boxed algorithms.

With function approximation, it may seem challenging to change the policy parameter in a way that ensures improvement. The problem is that performance depends on both the action selections and the distribution of states in which those selections are made, and that both of these are affected by the policy parameter. Given a state, the effect of the policy parameter on the actions, and thus on reward, can be computed in a relatively straightforward way from knowledge of the parameterization. But the effect of the policy on the state distribution is a function of the environment and is typically unknown. How can we estimate the performance gradient with respect to the policy parameter when the gradient depends on the unknown effect of policy changes on the state distribution?

Fortunately, there is an excellent theoretical answer to this challenge in the form of the *policy gradient theorem*, which provides an analytic expression for the gradient of

Proof of the Policy Gradient Theorem (episodic case)

With just elementary calculus and re-arranging of terms, we can prove the policy gradient theorem from first principles. To keep the notation simple, we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$, and all gradients are also implicitly with respect to $\boldsymbol{\theta}$. First note that the gradient of the state-value function can be written in terms of the action-value function as

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} && \text{(Exercise 3.18)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] && \text{(product rule of calculus)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r|s,a) \big(r + v_\pi(s')\big) \Big] \\
& && \text{(Exercise 3.19 and Equation 3.2)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla v_\pi(s') \Big] && \text{(Eq. 3.4)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) && \text{(unrolling)} \\
&\qquad \sum_{a'} \big[\nabla \pi(a'|s') q_\pi(s',a') + \pi(a'|s') \sum_{s''} p(s''|s',a') \nabla v_\pi(s'') \big] \Big] \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x,a),
\end{aligned}$$

after repeated unrolling, where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π. It is then immediate that

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(box page 199)} \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(Eq. 9.3)} \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(Q.E.D.)}
\end{aligned}$$

performance with respect to the policy parameter (which is what we need to approximate for gradient ascent (13.1)) that does *not* involve the derivative of the state distribution. The policy gradient theorem for the episodic case establishes that

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}), \tag{13.5}$$

where the gradients are column vectors of partial derivatives with respect to the components of $\boldsymbol{\theta}$, and π denotes the policy corresponding to parameter vector $\boldsymbol{\theta}$. The symbol $\propto$ here means "proportional to". In the episodic case, the constant of proportionality is the average length of an episode, and in the continuing case it is 1, so that the relationship is actually an equality. The distribution μ here (as in Chapters 9 and 10) is the on-policy distribution under π (see page 199). The policy gradient theorem is proved for the episodic case in the box on the previous page.

13.3 REINFORCE: Monte Carlo Policy Gradient

We are now ready to derive our first policy-gradient learning algorithm. Recall our overall strategy of stochastic gradient ascent (13.1), which requires a way to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter. The sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α, which is otherwise arbitrary. The policy gradient theorem gives an exact expression proportional to the gradient; all that is needed is some way of sampling whose expectation equals or approximates this expression. Notice that the right-hand side of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π; if π is followed, then states will be encountered in these proportions. Thus

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\
&= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right].
\end{aligned} \tag{13.6}$$

We could stop here and instantiate our stochastic gradient-ascent algorithm (13.1) as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \sum_a \hat{q}(S_t,a,\mathbf{w}) \nabla \pi(a|S_t,\boldsymbol{\theta}), \tag{13.7}$$

where $\hat{q}$ is some learned approximation to q_π. This algorithm, which has been called an *all-actions* method because its update involves all of the actions, is promising and deserving of further study, but our current interest is the classical REINFORCE algorithm (Willams, 1992) whose update at time t involves just A_t, the one action actually taken at time t.

We continue our derivation of REINFORCE by introducing A_t in the same way as we introduced S_t in (13.6)—by replacing a sum over the random variable's possible values

by an expectation under π, and then sampling the expectation. Equation (13.6) involves an appropriate sum over actions, but each term is not weighted by $\pi(a|S_t,\boldsymbol{\theta})$ as is needed for an expectation under π. So we introduce such a weighting, without changing the equality, by multiplying and then dividing the summed terms by $\pi(a|S_t,\boldsymbol{\theta})$. Continuing from (13.6), we have

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &= \mathbb{E}_\pi\left[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a)\frac{\nabla\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}\right] \\ &= \mathbb{E}_\pi\left[q_\pi(S_t,A_t)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right] \qquad \text{(replacing } a \text{ by the sample } A_t\sim\pi) \\ &= \mathbb{E}_\pi\left[G_t\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right], \qquad \text{(because } \mathbb{E}_\pi[G_t|S_t,A_t]=q_\pi(S_t,A_t))\end{aligned}$$

where G_t is the return as usual. The final expression in brackets is exactly what is needed, a quantity that can be sampled on each time step whose expectation is equal to the gradient. Using this sample to instantiate our generic stochastic gradient ascent algorithm (13.1) yields the REINFORCE update:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.8}$$

This update has an intuitive appeal. Each increment is proportional to the product of a return G_t and a vector, the gradient of the probability of taking the action actually taken divided by the probability of taking that action. The vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t. The update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability. The former makes sense because it causes the parameter to move most in the directions that favor actions that yield the highest return. The latter makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return.

Note that REINFORCE uses the complete return from time t, which includes all future rewards up until the end of the episode. In this sense REINFORCE is a Monte Carlo algorithm and is well defined only for the episodic case with all updates made in retrospect after the episode is completed (like the Monte Carlo algorithms in Chapter 5). This is shown explicitly in the boxed on the next page.

Notice that the update in the last line of pseudocode appears rather different from the REINFORCE update rule (13.8). One difference is that the pseudocode uses the compact expression $\nabla \ln \pi(A_t|S_t,\boldsymbol{\theta}_t)$ for the fractional vector $\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}$ in (13.8). That these two expressions for the vector are equivalent follows from the identity $\nabla \ln x = \frac{\nabla x}{x}$. This vector has been given several names and notations in the literature; we will refer to it simply as the *eligibility vector*. Note that it is the only place that the policy parameterization appears in the algorithm.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

The second difference between the pseudocode update and the REINFORCE update equation (13.8) is that the former includes a factor of γ^t. This is because, as mentioned earlier, in the text we are treating the non-discounted case ($\gamma = 1$) while in the boxed algorithms we are giving the algorithms for the general discounted case. All of the ideas go through in the discounted case with appropriate adjustments (including to the box on page 199) but involve additional complexity that distracts from the main ideas.

**Exercise 13.2* Generalize the box on page 199, the policy gradient theorem (13.5), the proof of the policy gradient theorem (page 325), and the steps leading to the REINFORCE update equation (13.8), so that (13.8) ends up with a factor of γ^t and thus aligns with the general algorithm given in the pseudocode. □

Figure 13.1 shows the performance of REINFORCE on the short-corridor gridworld from Example 13.1.

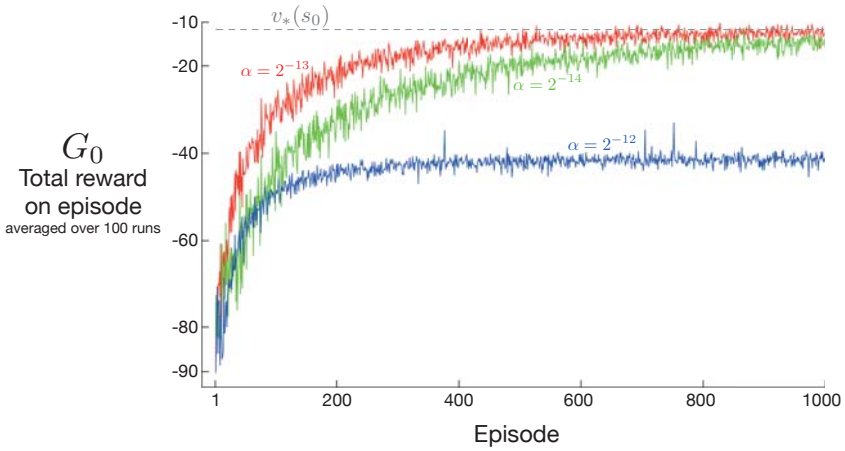


Figure 13.1: REINFORCE on the short-corridor gridworld (Example 13.1). With a good step size, the total reward per episode approaches the optimal value of the start state.

As a stochastic gradient method, REINFORCE has good theoretical convergence properties. By construction, the expected update over an episode is in the same direction as the performance gradient. This assures an improvement in expected performance for sufficiently small α, and convergence to a local optimum under standard stochastic approximation conditions for decreasing α. However, as a Monte Carlo method REINFORCE may be of high variance and thus produce slow learning.

Exercise 13.3 In Section 13.1 we considered policy parameterizations using the soft-max in action preferences (13.2) with linear action preferences (13.3). For this parameterization, prove that the eligibility vector is

$$\nabla \ln \pi(a|s,\boldsymbol{\theta}) = \mathbf{x}(s,a) - \sum_b \pi(b|s,\boldsymbol{\theta})\mathbf{x}(s,b), \tag{13.9}$$

using the definitions and elementary calculus. □

13.4 REINFORCE with Baseline

The policy gradient theorem (13.5) can be generalized to include a comparison of the action value to an arbitrary *baseline* $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \Big(q_\pi(s,a) - b(s)\Big)\nabla\pi(a|s,\boldsymbol{\theta}). \tag{13.10}$$

The baseline can be any function, even a random variable, as long as it does not vary with a; the equation remains valid because the subtracted quantity is zero:

$$\sum_a b(s)\nabla\pi(a|s,\boldsymbol{\theta}) \;=\; b(s)\nabla\sum_a \pi(a|s,\boldsymbol{\theta}) \;=\; b(s)\nabla 1 \;=\; 0.$$

The policy gradient theorem with baseline (13.10) can be used to derive an update rule using similar steps as in the previous section. The update rule that we end up with is a new version of REINFORCE that includes a general baseline:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big(G_t - b(S_t)\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.11}$$

Because the baseline could be uniformly zero, this update is a strict generalization of REINFORCE. In general, the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance. For example, we saw in Section 2.8 that an analogous baseline can significantly reduce the variance (and thus speed the learning) of gradient bandit algorithms. In the bandit algorithms the baseline was just a number (the average of the rewards seen so far), but for MDPs the baseline should vary with state. In some states all actions have high values and we need a high baseline to differentiate the higher valued actions from the less highly valued ones; in other states all actions will have low values and a low baseline is appropriate.

One natural choice for the baseline is an estimate of the state value, $\hat{v}(S_t,\mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^m$ is a weight vector learned by one of the methods presented in previous chapters.

Because REINFORCE is a Monte Carlo method for learning the policy parameter, $\boldsymbol{\theta}$, it seems natural to also use a Monte Carlo method to learn the state-value weights, $\mathbf{w}$. A complete pseudocode algorithm for REINFORCE with baseline using such a learned state-value function as the baseline is given in the box below.

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\gamma^t\delta\nabla\ln\pi(A_t|S_t,\boldsymbol{\theta})$

This algorithm has two step sizes, denoted $\alpha^{\boldsymbol{\theta}}$ and $\alpha^{\mathbf{w}}$ (where $\alpha^{\boldsymbol{\theta}}$ is the α in (13.11)). Choosing the step size for values (here $\alpha^{\mathbf{w}}$) is relatively easy; in the linear case we have rules of thumb for setting it, such as $\alpha^{\mathbf{w}} = 0.1/\mathbb{E}\big[\|\nabla\hat{v}(S_t,\mathbf{w})\|^2_\mu\big]$ (see Section 9.6). It is much less clear how to set the step size for the policy parameters, $\alpha^{\boldsymbol{\theta}}$, whose best value depends on the range of variation of the rewards and on the policy parameterization.

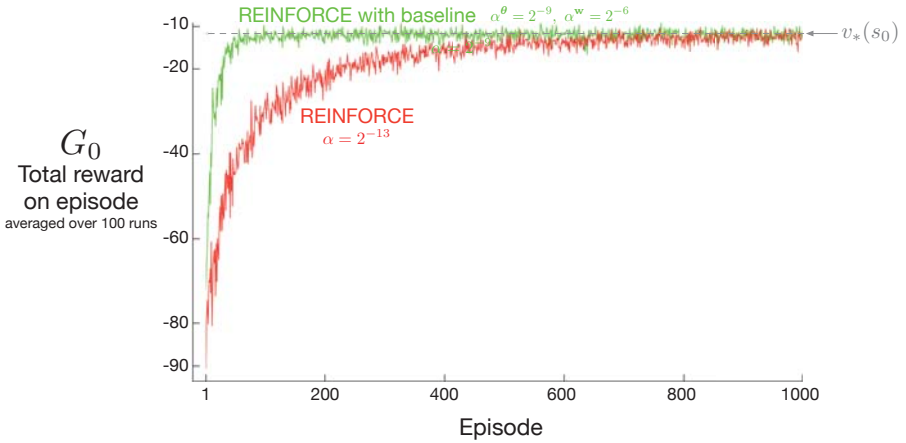


Figure 13.2: Adding a baseline to REINFORCE can make it learn much faster, as illustrated here on the short-corridor gridworld (Example 13.1). The step size used here for plain REINFORCE is that at which it performs best (to the nearest power of two; see Figure 13.1).

Figure 13.2 compares the behavior of REINFORCE with and without a baseline on the short-corridor gridword (Example 13.1). Here the approximate state-value function used in the baseline is $\hat{v}(s,\mathbf{w}) = w$. That is, $\mathbf{w}$ is a single component, w.

13.5 Actor–Critic Methods

Although the REINFORCE-with-baseline method learns both a policy and a state-value function, we do not consider it to be an actor–critic method because its state-value function is used only as a baseline, not as a critic. That is, it is not used for bootstrapping (updating the value estimate for a state from the estimated values of subsequent states), but only as a baseline for the state whose estimate is being updated. This is a useful distinction, for only through bootstrapping do we introduce bias and an asymptotic dependence on the quality of the function approximation. As we have seen, the bias introduced through bootstrapping and reliance on the state representation is often beneficial because it reduces variance and accelerates learning. REINFORCE with baseline is unbiased and will converge asymptotically to a local minimum, but like all Monte Carlo methods it tends to learn slowly (produce estimates of high variance) and to be inconvenient to implement online or for continuing problems. As we have seen earlier in this book, with temporal-difference methods we can eliminate these inconveniences, and through multi-step methods we can flexibly choose the degree of bootstrapping. In order to gain these advantages in the case of policy gradient methods we use actor–critic methods with a bootstrapping critic.

First consider one-step actor–critic methods, the analog of the TD methods introduced in Chapter 6 such as TD(0), Sarsa(0), and Q-learning. The main appeal of one-step methods is that they are fully online and incremental, yet avoid the complexities of eligibility traces. They are a special case of the eligibility trace methods, and not as general, but easier to understand. One-step actor–critic methods replace the full return of REINFORCE (13.11) with the one-step return (and use a learned state-value function as the baseline) as follows:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big(G_{t:t+1} - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{13.12}$$

$$= \boldsymbol{\theta}_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{13.13}$$

$$= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.14}$$

The natural state-value-function learning method to pair with this is semi-gradient TD(0). Pseudocode for the complete algorithm is given in the box at the top of the next page. Note that it is now a fully online, incremental algorithm, with states, actions, and rewards processed as they occur and then never revisited.

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

The generalizations to the forward view of n-step methods and then to a λ-return algorithm are straightforward. The one-step return in (13.12) is merely replaced by $G_{t:t+n}$ or G_t^λ respectively. The backward view of the λ-return algorithm is also straightforward, using separate eligibility traces for the actor and critic, each after the patterns in Chapter 12. Pseudocode for the complete algorithm is given in the box below.

Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma\lambda^{\mathbf{w}}\mathbf{z}^{\mathbf{w}} + \nabla\hat{v}(S,\mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma\lambda^{\boldsymbol{\theta}}\mathbf{z}^{\boldsymbol{\theta}} + I\nabla\ln\pi(A|S,\boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathbf{z}^{\boldsymbol{\theta}}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

13.6 Policy Gradient for Continuing Problems

As discussed in Section 10.3, for continuing problems without episode boundaries we need to define performance in terms of the average rate of reward per time step:

$$\begin{aligned} J(\boldsymbol{\theta}) \doteq r(\pi) &\doteq \lim_{h\to\infty} \frac{1}{h}\sum_{t=1}^{h} \mathbb{E}[R_t \mid S_0, A_{0:t-1}\sim\pi] \\ &= \lim_{t\to\infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1}\sim\pi] \\ &= \sum_s \mu(s)\sum_a \pi(a|s)\sum_{s',r} p(s',r\,|\,s,a)r, \end{aligned} \tag{13.15}$$

where μ is the steady-state distribution under π, $\mu(s) \doteq \lim_{t\to\infty}\Pr\{S_t{=}s|A_{0:t}\sim\pi\}$, which is assumed to exist and to be independent of S_0 (an ergodicity assumption). Remember that this is the special distribution under which, if you select actions according to π, you remain in the same distribution:

$$\sum_s \mu(s)\sum_a \pi(a|s,\boldsymbol{\theta})p(s'|s,a) = \mu(s'), \text{ for all } s'\in\mathcal{S}. \tag{13.16}$$

Complete pseudocode for the actor–critic algorithm in the continuing case (backward view) is given in the box below.

Actor–Critic with Eligibility Traces (continuing), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: $\lambda^{\mathbf{w}} \in [0,1]$, $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\alpha^{\mathbf{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\bar{R}} > 0$
Initialize $\bar{R} \in \mathbb{R}$ (e.g., to 0)
Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)
Initialize $S \in \mathcal{S}$ (e.g., to s_0)

$\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
$\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
Loop forever (for each time step):
  $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
  Take action A, observe S', R
  $\delta \leftarrow R - \bar{R} + \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}}\delta$
  $\mathbf{z}^{\mathbf{w}} \leftarrow \lambda^{\mathbf{w}}\mathbf{z}^{\mathbf{w}} + \nabla\hat{v}(S,\mathbf{w})$
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \lambda^{\boldsymbol{\theta}}\mathbf{z}^{\boldsymbol{\theta}} + \nabla \ln\pi(A|S,\boldsymbol{\theta})$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\mathbf{z}^{\mathbf{w}}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathbf{z}^{\boldsymbol{\theta}}$
  $S \leftarrow S'$

Naturally, in the continuing case, we define values, $v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t{=}s]$ and $q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t{=}s, A_t{=}a]$, with respect to the differential return:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \cdots. \tag{13.17}$$

With these alternate definitions, the policy gradient theorem as given for the episodic case (13.5) remains true for the continuing case. A proof is given in the box on the next page. The forward and backward view equations also remain the same.

Proof of the Policy Gradient Theorem (continuing case)

The proof of the policy gradient theorem for the continuing case begins similarly to the episodic case. Again we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$ and that the gradients are with respect to $\boldsymbol{\theta}$. Recall that in the continuing case $J(\boldsymbol{\theta}) = r(\pi)$ (13.15) and that v_π and q_π denote values with respect to the differential return (13.17). The gradient of the state-value function can be written, for any $s \in \mathcal{S}$, as

$$\begin{aligned} \nabla v_\pi(s) &= \nabla\left[\sum_a \pi(a|s)q_\pi(s,a)\right], \quad \text{for all } s\in\mathcal{S} && \text{(Exercise 3.18)} \\ &= \sum_a \Big[\nabla\pi(a|s)q_\pi(s,a) + \pi(a|s)\nabla q_\pi(s,a)\Big] && \text{(product rule of calculus)} \\ &= \sum_a \Big[\nabla\pi(a|s)q_\pi(s,a) + \pi(a|s)\nabla\sum_{s',r} p(s',r\,|\,s,a)\big(r - r(\boldsymbol{\theta}) + v_\pi(s')\big)\Big] \\ &= \sum_a \Big[\nabla\pi(a|s)q_\pi(s,a) + \pi(a|s)\big[-\nabla r(\boldsymbol{\theta}) + \sum_{s'} p(s'|s,a)\nabla v_\pi(s')\big]\Big]. \end{aligned}$$

After re-arranging terms, we obtain

$$\nabla r(\boldsymbol{\theta}) = \sum_a \Big[\nabla\pi(a|s)q_\pi(s,a) + \pi(a|s)\sum_{s'} p(s'|s,a)\nabla v_\pi(s')\Big] - \nabla v_\pi(s).$$

Notice that the left-hand side can be written $\nabla J(\boldsymbol{\theta})$, and that it does not depend on s. Thus the right-hand side does not depend on s either, and we can safely sum it over all $s \in \mathcal{S}$, weighted by $\mu(s)$, without changing it (because $\sum_s \mu(s) = 1$):

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &= \sum_s \mu(s)\left(\sum_a \Big[\nabla\pi(a|s)q_\pi(s,a) + \pi(a|s)\sum_{s'} p(s'|s,a)\nabla v_\pi(s')\Big] - \nabla v_\pi(s)\right) \\ &= \sum_s \mu(s)\sum_a \nabla\pi(a|s)q_\pi(s,a) \\ &\quad + \sum_s \mu(s)\sum_a \pi(a|s)\sum_{s'} p(s'|s,a)\nabla v_\pi(s') - \sum_s \mu(s)\nabla v_\pi(s) \end{aligned}$$

$$
\begin{aligned}
&= \sum_s \mu(s) \sum_a \nabla\pi(a|s) q_\pi(s,a) \\
&\quad + \sum_{s'} \underbrace{\sum_s \mu(s) \sum_a \pi(a|s) p(s'|s,a)}_{\mu(s') \text{ (13.16)}} \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla\pi(a|s) q_\pi(s,a) + \sum_{s'} \mu(s') \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla\pi(a|s) q_\pi(s,a). \qquad \text{Q.E.D.}
\end{aligned}
$$

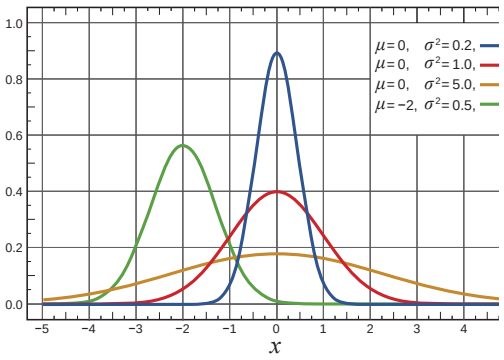
13.7 Policy Parameterization for Continuous Actions

Policy-based methods offer practical ways of dealing with large actions spaces, even continuous spaces with an infinite number of actions. Instead of computing learned probabilities for each of the many actions, we instead learn statistics of the probability distribution. For example, the action set might be the real numbers, with actions chosen from a normal (Gaussian) distribution.

The probability density function for the normal distribution is conventionally written

$$
p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{13.18}
$$

where μ and σ here are the mean and standard deviation of the normal distribution, and of course π here is just the number $\pi \approx 3.14159$. The probability density functions for several different means and standard deviations are shown to the right. The value $p(x)$ is the *density* of the probability at x, not the probability. It can be greater than 1; it is the total area under $p(x)$ that must sum to 1. In general, one can take the integral under $p(x)$ for any range of x values to get the probability of x falling within that range.



To produce a policy parameterization, the policy can be defined as the normal probability density over a real-valued scalar action, with mean and standard deviation given by parametric function approximators that depend on the state. That is,

$$
\pi(a|s,\boldsymbol{\theta}) \doteq \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}\right), \tag{13.19}
$$

where $\mu : \mathcal{S}\times\mathbb{R}^{d'} \to \mathbb{R}$ and $\sigma : \mathcal{S}\times\mathbb{R}^{d'} \to \mathbb{R}^+$ are two parameterized function approximators.

To complete the example we need only give a form for these approximators. For this we divide the policy's parameter vector into two parts, $\boldsymbol{\theta} = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma]^\top$, one part to be used for the approximation of the mean and one part for the approximation of the standard deviation. The mean can be approximated as a linear function. The standard deviation must always be positive and is better approximated as the exponential of a linear function. Thus

$$
\mu(s,\boldsymbol{\theta}) \doteq \boldsymbol{\theta}_\mu{}^\top \mathbf{x}_\mu(s) \quad \text{and} \quad \sigma(s,\boldsymbol{\theta}) \doteq \exp\left(\boldsymbol{\theta}_\sigma{}^\top \mathbf{x}_\sigma(s)\right), \tag{13.20}
$$

where $\mathbf{x}_\mu(s)$ and $\mathbf{x}_\sigma(s)$ are state feature vectors perhaps constructed by one of the methods described in Chapter 9. With these definitions, all the algorithms described in the rest of this chapter can be applied to learn to select real-valued actions.

Exercise 13.4 Show that for the gaussian policy parameterization (13.19) the eligibility vector has the following two parts:

$$
\nabla \ln \pi(a|s,\boldsymbol{\theta}_\mu) = \frac{\nabla\pi(a|s,\boldsymbol{\theta}_\mu)}{\pi(a|s,\boldsymbol{\theta})} = \frac{1}{\sigma(s,\boldsymbol{\theta})^2}\big(a-\mu(s,\boldsymbol{\theta})\big)\mathbf{x}_\mu(s), \text{ and}
$$

$$
\nabla \ln \pi(a|s,\boldsymbol{\theta}_\sigma) = \frac{\nabla\pi(a|s,\boldsymbol{\theta}_\sigma)}{\pi(a|s,\boldsymbol{\theta})} = \left(\frac{\big(a-\mu(s,\boldsymbol{\theta})\big)^2}{\sigma(s,\boldsymbol{\theta})^2} - 1\right)\mathbf{x}_\sigma(s). \qquad \square
$$

Exercise 13.5 A *Bernoulli-logistic unit* is a stochastic neuron-like unit used in some ANNs (Section 9.6). Its input at time t is a feature vector $\mathbf{x}(S_t)$; its output, A_t, is a random variable having two values, 0 and 1, with $\Pr\{A_t = 1\} = P_t$ and $\Pr\{A_t = 0\} = 1 - P_t$ (the Bernoulli distribution). Let $h(s,0,\boldsymbol{\theta})$ and $h(s,1,\boldsymbol{\theta})$ be the preferences in state s for the unit's two actions given policy parameter $\boldsymbol{\theta}$. Assume that the difference between the action preferences is given by a weighted sum of the unit's input vector, that is, assume that $h(s,1,\boldsymbol{\theta}) - h(s,0,\boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s)$, where $\boldsymbol{\theta}$ is the unit's weight vector.

(a) Show that if the exponential soft-max distribution (13.2) is used to convert action preferences to policies, then $P_t = \pi(1|S_t,\boldsymbol{\theta}_t) = 1/(1+\exp(-\boldsymbol{\theta}_t^\top \mathbf{x}(S_t)))$ (the logistic function).

(b) What is the Monte-Carlo REINFORCE update of $\boldsymbol{\theta}_t$ to $\boldsymbol{\theta}_{t+1}$ upon receipt of return G_t?

(c) Express the eligibility $\nabla \ln \pi(a|s,\boldsymbol{\theta})$ for a Bernoulli-logistic unit, in terms of a, $\mathbf{x}(s)$, and $\pi(a|s,\boldsymbol{\theta})$ by calculating the gradient.

Hint: separately for each action compute the derivative of the logarithm first with respect to $P_t = \pi(a|s,\boldsymbol{\theta}_t)$, combine the two results into one expression that depends on a and P_t, and then use the chain rule, noting that the derivative of the logistic function $f(x)$ is $f(x)(1-f(x))$. $\square$

13.8 Summary

Prior to this chapter, this book focused on *action-value methods*—meaning methods that learn action values and then use them to determine action selections. In this chapter, on the other hand, we considered methods that learn a parameterized policy that enables actions to be taken without consulting action-value estimates. In particular, we have considered *policy-gradient methods*—meaning methods that update the policy parameter on each step in the direction of an estimate of the gradient of performance with respect to the policy parameter.

Methods that learn and store a policy parameter have many advantages. They can learn specific probabilities for taking the actions. They can learn appropriate levels of exploration and approach deterministic policies asymptotically. They can naturally handle continuous action spaces. All these things are easy for policy-based methods but awkward or impossible for ε-greedy methods and for action-value methods in general. In addition, on some problems the policy is just simpler to represent parametrically than the value function; these problems are more suited to parameterized policy methods.

Parameterized policy methods also have an important theoretical advantage over action-value methods in the form of the *policy gradient theorem*, which gives an exact formula for how performance is affected by the policy parameter that does not involve derivatives of the state distribution. This theorem provides a theoretical foundation for all policy gradient methods.

The REINFORCE method follows directly from the policy gradient theorem. Adding a state-value function as a *baseline* reduces REINFORCE's variance without introducing bias. Using the state-value function for bootstrapping introduces bias but is often desirable for the same reason that bootstrapping TD methods are often superior to Monte Carlo methods (substantially reduced variance). The state-value function assigns credit to—critizes—the policy's action selections, and accordingly the former is termed the *critic* and the latter the *actor*, and these overall methods are termed *actor–critic* methods.

Overall, policy-gradient methods provide a significantly different set of strengths and weaknesses than action-value methods. Today they are less well understood in some respects, but a subject of excitement and ongoing research.

Bibliographical and Historical Remarks

Methods that we now see as related to policy gradients were actually some of the earliest to be studied in reinforcement learning (Witten, 1977; Barto, Sutton, and Anderson, 1983; Sutton, 1984; Williams, 1987, 1992) and in predecessor fields (Phansalkar and Thathachar, 1995). They were largely supplanted in the 1990s by the action-value methods that are the focus of the other chapters of this book. In recent years, however, attention has returned to actor–critic methods and to policy-gradient methods in general. Among the further developments beyond what we cover here are natural-gradient methods (Amari, 1998; Kakade, 2002, Peters, Vijayakumar and Schaal, 2005; Peters and Schall, 2008; Park, Kim and Kang, 2005; Bhatnagar, Sutton, Ghavamzadeh and Lee, 2009; see Grondman, Busoniu, Lopes and Babuska, 2012), deterministic policy gradient methods (Silver et al., 2014), off-policy policy-gradient methods (Degris, White, and Sutton, 2012; Maei, 2018), and entropy regularization (see Schulman, Chen, and Abbeel, 2017). Major applications include acrobatic helicopter autopilots and AlphaGo (Section 16.6).

Our presentation in this chapter is based primarily on that by Sutton, McAllester, Singh, and Mansour (2000), who introduced the term "policy gradient methods." A useful overview is provided by Bhatnagar et al. (2009). One of the earliest related works is by Aleksandrov, Sysoyev, and Shemeneva (1968). Thomas (2014) first realized that the factor of γ^t, as specified in the boxed algorithms of this chapter, was needed in the case of discounted episodic problems.

13.1 Example 13.1 and the results with it in this chapter were developed with Eric Graves.

13.2 The policy gradient theorem here and on page 334 was first obtained by Marbach and Tsitsiklis (1998, 2001) and then independently by Sutton et al. (2000). A similar expression was obtained by Cao and Chen (1997). Other early results are due to Konda and Tsitsiklis (2000, 2003), Baxter and Bartlett (2001), and Baxter, Bartlett, and Weaver (2001). Some additional results are developed by Sutton, Singh, and McAllester (2000).

13.3 REINFORCE is due to Williams (1987, 1992). Phansalkar and Thathachar (1995) proved both local and global convergence theorems for modified versions of REINFORCE algorithms.

The all-actions algorithm was first presented in an unpublished but widely circulated incomplete paper (Sutton, Singh, and McAllester, 2000) and then explored analytically and empirically by Asadi, Allen, Roderick, Mohamed, Konidaris, and Littman (2017), who called it the"mean actor critic" algorithm. An extension to continuous actions was developed by Ciosek and Whiteson (2018), which they termed "expected policy gradients."

13.4 The baseline was introduced in Williams's (1987, 1992) original work. Greensmith, Bartlett, and Baxter (2004) analyzed an arguably better baseline (see Dick, 2015). Thomas and Brunskill (2017) argue that an action-dependent baseline can be used without incurring bias.

13.5–6 Actor–critic methods were among the earliest to be investigated in reinforcement learning (Witten, 1977; Barto, Sutton, and Anderson, 1983; Sutton, 1984). The algorithms presented here are based on the work of Degris, White, and Sutton (2012). Actor–critic methods are sometimes referred to as advantage actor–critic methods in the literature.

13.7 The first to show how continuous actions could be handled this way appears to have been Williams (1987, 1992). The figure on page 335 is adapted from Wikipedia.

Part III: Looking Deeper

In this last part of the book we look beyond the standard reinforcement learning ideas presented in the first two parts of the book to briefly survey their relationships with psychology and neuroscience, a sampling of reinforcement learning applications, and some of the active frontiers for future reinforcement learning research.

Chapter 14

Psychology

In previous chapters we developed ideas for algorithms based on computational considerations alone. In this chapter we look at some of these algorithms from another perspective: the perspective of psychology and its study of how animals learn. The goals of this chapter are, first, to discuss ways that reinforcement learning ideas and algorithms correspond to what psychologists have discovered about animal learning, and second, to explain the influence reinforcement learning is having on the study of animal learning. The clear formalism provided by reinforcement learning that systemizes tasks, returns, and algorithms is proving to be enormously useful in making sense of experimental data, in suggesting new kinds of experiments, and in pointing to factors that may be critical to manipulate and to measure. The idea of optimizing return over the long term that is at the core of reinforcement learning is contributing to our understanding of otherwise puzzling features of animal learning and behavior.

Some of the correspondences between reinforcement learning and psychological theories are not surprising because the development of reinforcement learning drew inspiration from psychological learning theories. However, as developed in this book, reinforcement learning explores idealized situations from the perspective of an artificial intelligence researcher or engineer, with the goal of solving computational problems with efficient algorithms, rather than to replicate or explain in detail how animals learn. As a result, some of the correspondences we describe connect ideas that arose independently in their respective fields. We believe these points of contact are specially meaningful because they expose computational principles important to learning, whether it is learning by artificial or by natural systems.

For the most part, we describe correspondences between reinforcement learning and learning theories developed to explain how animals like rats, pigeons, and rabbits learn in controlled laboratory experiments. Thousands of these experiments were conducted throughout the 20th century, and many are still being conducted today. Although sometimes dismissed as irrelevant to wider issues in psychology, these experiments probe subtle properties of animal learning, often motivated by precise theoretical questions. As psychology shifted its focus to more cognitive aspects of behavior, that is, to mental processes such as thought and reasoning, animal learning experiments came to play less of a role in psychology than they once did. But this experimentation led to the discovery of learning principles that are elemental and widespread throughout the animal kingdom, principles that should not be neglected in designing artificial learning systems. In addition, as we shall see, some aspects of cognitive processing connect naturally to the computational perspective provided by reinforcement learning.

This chapter's final section includes references relevant to the connections we discuss as well as to connections we neglect. We hope this chapter encourages readers to probe all of these connections more deeply. Also included in this final section is a discussion of how the terminology used in reinforcement learning relates to that of psychology. Many of the terms and phrases used in reinforcement learning are borrowed from animal learning theories, but the computational/engineering meanings of these terms and phrases do not always coincide with their meanings in psychology.

14.1 Prediction and Control

The algorithms we describe in this book fall into two broad categories: algorithms for *prediction* and algorithms for *control*.[1] These categories arise naturally in solution methods for the reinforcement learning problem presented in Chapter 3. In many ways these categories respectively correspond to categories of learning extensively studied by psychologists: *classical*, or *Pavlovian*, *conditioning* and *instrumental*, or *operant*, *conditioning*. These correspondences are not completely accidental because of psychology's influence on reinforcement learning, but they are nevertheless striking because they connect ideas arising from different objectives.

The prediction algorithms presented in this book estimate quantities that depend on how features of an agent's environment are expected to unfold over the future. We specifically focus on estimating the amount of reward an agent can expect to receive over the future while it interacts with its environment. In this role, prediction algorithms are *policy evaluation algorithms*, which are integral components of algorithms for improving policies. But prediction algorithms are not limited to predicting future reward; they can predict any feature of the environment (see, for example, Modayil, White, and Sutton, 2014). The correspondence between prediction algorithms and classical conditioning rests on their common property of predicting upcoming stimuli, whether or not those stimuli are rewarding (or punishing).

The situation in an instrumental, or operant, conditioning experiment is different. Here, the experimental apparatus is set up so that an animal is given something it likes (a reward) or something it dislikes (a penalty) depending on what the animal did. The animal learns to increase its tendency to produce rewarded behavior and to decrease its tendency to produce penalized behavior. The reinforcing stimulus is said to be *contingent* on the animal's behavior, whereas in classical conditioning it is not (although it is difficult to remove all behavior contingencies in a classical conditioning experiment). Instrumental conditioning experiments are like those that inspired Thorndike's Law of Effect that

[1] What control means for us is different from what it typically means in animal learning theories; there the environment controls the agent instead of the other way around. See our comments on terminology at the end of this chapter.

we briefly discuss in Chapter 1. *Control* is at the core of this form of learning, which corresponds to the operation of reinforcement learning's policy-improvement algorithms.

Thinking of classical conditioning in terms of prediction, and instrumental conditioning in terms of control, is a starting point for connecting our computational view of reinforcement learning to animal learning, but in reality, the situation is more complicated than this. There is more to classical conditioning than prediction; it also involves action, and so is a mode of control, sometimes called *Pavlovian control*. Further, classical and instrumental conditioning interact in interesting ways, with both sorts of learning likely being engaged in most experimental situations. Despite these complications, aligning the classical/instrumental distinction with the prediction/control distinction is a convenient first approximation in connecting reinforcement learning to animal learning.

In psychology, the term reinforcement is used to describe learning in both classical and instrumental conditioning. Originally referring only to the strengthening of a pattern of behavior, it is frequently also used for the weakening of a pattern of behavior. A stimulus considered to be the cause of the change in behavior is called a reinforcer, whether or not it is contingent on the animal's previous behavior. At the end of this chapter we discuss this terminology in more detail and how it relates to terminology used in machine learning.

14.2 Classical Conditioning

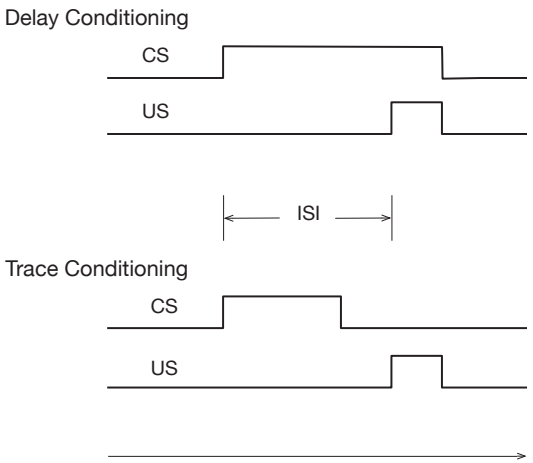
While studying the activity of the digestive system, the celebrated Russian physiologist Ivan Pavlov found that an animal's innate responses to certain triggering stimuli can come to be triggered by other stimuli that are quite unrelated to the inborn triggers. His experimental subjects were dogs that had undergone minor surgery to allow the intensity of their salivary reflex to be accurately measured. In one case he describes, the dog did not salivate under most circumstances, but about 5 seconds after being presented with food it produced about six drops of saliva over the next several seconds. After several repetitions of presenting another stimulus, one not related to food, in this case the sound of a metronome, shortly before the introduction of food, the dog salivated in response to the sound of the metronome in the same way it did to the food. "The activity of the salivary gland has thus been called into play by impulses of sound—a stimulus quite alien to food" (Pavlov, 1927, p. 22). Summarizing the significance of this finding, Pavlov wrote:

> It is pretty evident that under natural conditions the normal animal must respond not only to stimuli which themselves bring immediate benefit or harm, but also to other physical or chemical agencies—waves of sound, light, and the like—which in themselves only *signal* the approach of these stimuli; though it is not the sight and sound of the beast of prey which is in itself harmful to the smaller animal, but its teeth and claws. (Pavlov, 1927, p. 14)

Connecting new stimuli to innate reflexes in this way is now called classical, or Pavlovian, conditioning. Pavlov (or more exactly, his translators) called inborn responses (e.g., salivation in his demonstration described above) "unconditioned responses" (URs), their

natural triggering stimuli (e.g., food) "unconditioned stimuli" (USs), and new responses triggered by predictive stimuli (e.g., here also salivation) "conditioned responses" (CRs). A stimulus that is initially neutral, meaning that it does not normally elicit strong responses (e.g., the metronome sound), becomes a "conditioned stimulus" (CS) as the animal learns that it predicts the US and so comes to produce a CR in response to the CS. These terms are still used in describing classical conditioning experiments (though better translations would have been "conditional" and "unconditional" instead of conditioned and unconditioned). The US is called a reinforcer because it reinforces producing a CR in response to the CS.

The arrangement of stimuli in two common types of classical conditioning experiments is shown to the right. In *delay conditioning*, the CS extends throughout the interstimulus interval, or ISI, which is the time interval between the CS onset and the US onset (with the CS ending when the US ends in a common version shown here). In *trace conditioning*, the US begins after the CS ends, and the time interval between CS offset and US onset is called the trace interval.



The salivation of Pavlov's dogs to the sound of a metronome is just one example of classical conditioning, which has been intensively studied across many response systems of many species of animals. URs are often preparatory in some way, like the salivation of Pavlov's dog, or protective in some way, like an eye blink in response to something irritating to the eye, or freezing in response to seeing a predator. Experiencing the CS-US predictive relationship over a series of trials causes the animal to learn that the CS predicts the US so that the animal can respond to the CS with a CR that prepares the animal for, or protects it from, the predicted US. Some CRs are similar to the UR but begin earlier and differ in ways that increase their effectiveness. In one intensively studied type of experiment, for example, a tone CS reliably predicts a puff of air (the US) to a rabbit's eye, triggering a UR consisting of the closure of a protective inner eyelid called the nictitating membrane. After one or more trials, the tone comes to trigger a CR consisting of membrane closure that begins before the air puff and eventually becomes timed so that peak closure occurs just when the air puff is likely to occur. This CR, being initiated in anticipation of the air puff and appropriately timed, offers better protection than simply initiating closure as a reaction to the irritating US. The ability to act in anticipation of important events by learning about predictive relationships among stimuli is so beneficial that it is widely present across the animal kingdom.

14.2.1 Blocking and Higher-order Conditioning

Many interesting properties of classical conditioning have been observed in experiments. Beyond the anticipatory nature of CRs, two widely observed properties figured prominently in the development of classical conditioning models: *blocking* and *higher-order conditioning*. Blocking occurs when an animal fails to learn a CR when a potential CS is presented along with another CS that had been used previously to condition the animal to produce that CR. For example, in the first stage of a blocking experiment involving rabbit nictitating membrane conditioning, a rabbit is first conditioned with a tone CS and an air puff US to produce the CR of closing its nictitating membrane in anticipation of the air puff. The experiment's second stage consists of additional trials in which a second stimulus, say a light, is added to the tone to form a compound tone/light CS followed by the same air puff US. In the experiment's third phase, the second stimulus alone—the light—is presented to the rabbit to see if the rabbit has learned to respond to it with a CR. It turns out that the rabbit produces very few, or no, CRs in response to the light: learning to the light had been *blocked* by the previous learning to the tone.[2] Blocking results like this challenged the idea that conditioning depends only on simple temporal contiguity, that is, that a necessary and sufficient condition for conditioning is that a US frequently follows a CS closely in time. In the next section we describe the *Rescorla–Wagner model* (Rescorla and Wagner, 1972) that offered an influential explanation for blocking.

Higher-order conditioning occurs when a previously-conditioned CS acts as a US in conditioning another initially neutral stimulus. Pavlov described an experiment in which his assistant first conditioned a dog to salivate to the sound of a metronome that predicted a food US, as described above. After this stage of conditioning, a number of trials were conducted in which a black square, to which the dog was initially indifferent, was placed in the dog's line of vision followed by the sound of the metronome—and this was *not* followed by food. In just ten trials, the dog began to salivate merely upon seeing the black square, despite the fact that the sight of it had never been followed by food. The sound of the metronome itself acted as a US in conditioning a salivation CR to the black square CS. This was second-order conditioning. If the black square had been used as a US to establish salivation CRs to another otherwise neutral CS, it would have been third-order conditioning, and so on. Higher-order conditioning is difficult to demonstrate, especially above the second order, in part because a higher-order reinforcer loses its reinforcing value due to not being repeatedly followed by the original US during higher-order conditioning trials. But under the right conditions, such as intermixing first-order trials with higher-order trials or by providing a general energizing stimulus, higher-order conditioning beyond the second order can be demonstrated. As we describe below, the *TD model of classical conditioning* uses the bootstrapping idea that is central to our approach to extend the Rescorla–Wagner model's account of blocking to include both the anticipatory nature of CRs and higher-order conditioning.

Higher-order instrumental conditioning occurs as well. In this case, a stimulus that

[2]Comparison with a control group is necessary to show that the previous conditioning to the tone is responsible for blocking learning to the light. This is done by trials with the tone/light CS but with no prior conditioning to the tone. Learning to the light in this case is unimpaired. Moore and Schmajuk (2008) give a full account of this procedure.

consistently predicts primary reinforcement becomes a reinforcer itself, where reinforcement is primary if its rewarding or penalizing quality has been built into the animal by evolution. The predicting stimulus becomes a *secondary reinforcer*, or more generally, a *higher-order* or *conditioned reinforcer*—the latter being a better term when the predicted reinforcing stimulus is itself a secondary, or an even higher-order, reinforcer. A conditioned reinforcer delivers *conditioned reinforcement*: conditioned reward or conditioned penalty. Conditioned reinforcement acts like primary reinforcement in increasing an animal's tendency to produce behavior that leads to conditioned reward, and to decrease an animal's tendency to produce behavior that leads to conditioned penalty. (See our comments at the end of this chapter that explain how our terminology sometimes differs, as it does here, from terminology used in psychology.)

Conditioned reinforcement is a key phenomenon that explains, for instance, why we work for the conditioned reinforcer money, whose worth derives solely from what is predicted by having it. In actor–critic methods described in Section 13.5 (and discussed in the context of neuroscience in Sections 15.7 and 15.8), the critic uses a TD method to evaluate the actor's policy, and its value estimates provide conditioned reinforcement to the actor, allowing the actor to improve its policy. This analog of higher-order instrumental conditioning helps address the credit-assignment problem mentioned in Section 1.7 because the critic gives moment-by-moment reinforcement to the actor when the primary reward signal is delayed. We discuss this more below in Section 14.4.

14.2.2 The Rescorla–Wagner Model

Rescorla and Wagner created their model mainly to account for blocking. The core idea of the Rescorla–Wagner model is that an animal only learns when events violate its expectations, in other words, only when the animal is surprised (although without necessarily implying any *conscious* expectation or emotion). We first present Rescorla and Wagner's model using their terminology and notation before shifting to the terminology and notation we use to describe the TD model.

Here is how Rescorla and Wagner described their model. The model adjusts the "associative strength" of each component stimulus of a compound CS, which is a number representing how strongly or reliably that component is predictive of a US. When a compound CS consisting of several component stimuli is presented in a classical conditioning trial, the associative strength of each component stimulus changes in a way that depends on an associative strength associated with the entire stimulus compound, called the "aggregate associative strength," and not just on the associative strength of each component itself.

Rescorla and Wagner considered a compound CS AX, consisting of component stimuli A and X, where the animal may have already experienced stimulus A, and stimulus X might be new to the animal. Let V_A, V_X, and V_{AX} respectively denote the associative strengths of stimuli A, X, and the compound AX. Suppose that on a trial the compound CS AX is followed by a US, which we label stimulus Y. Then the associative strengths of

the stimulus components change according to these expressions:

$$\Delta V_{\mathrm{A}} = \alpha_{\mathrm{A}}\beta_{\mathrm{Y}}(R_{\mathrm{Y}} - V_{\mathrm{AX}})$$
$$\Delta V_{\mathrm{X}} = \alpha_{\mathrm{X}}\beta_{\mathrm{Y}}(R_{\mathrm{Y}} - V_{\mathrm{AX}}),$$

where $\alpha_{\mathrm{A}}\beta_{\mathrm{Y}}$ and $\alpha_{\mathrm{X}}\beta_{\mathrm{Y}}$ are the step-size parameters, which depend on the identities of the CS components and the US, and R_{Y} is the asymptotic level of associative strength that the US Y can support. (Rescorla and Wagner used λ here instead of R, but we use R to avoid confusion with our use of λ and because we usually think of this as the magnitude of a reward signal, with the caveat that the US in classical conditioning is not necessarily rewarding or penalizing.) A key assumption of the model is that the aggregate associative strength V_{AX} is equal to $V_{\mathrm{A}} + V_{\mathrm{X}}$. The associative strengths as changed by these Δs become the associative strengths at the beginning of the next trial.

To be complete, the model needs a response-generation mechanism, which is a way of mapping values of Vs to CRs. Because this mapping would depend on details of the experimental situation, Rescorla and Wagner did not specify a mapping but simply assumed that larger Vs would produce stronger or more likely CRs, and that negative Vs would mean that there would be no CRs.

The Rescorla–Wagner model accounts for the acquisition of CRs in a way that explains blocking. As long as the aggregate associative strength, V_{AX}, of the stimulus compound is below the asymptotic level of associative strength, R_{Y}, that the US Y can support, the prediction error $R_{\mathrm{Y}} - V_{\mathrm{AX}}$ is positive. This means that over successive trials the associative strengths V_{A} and V_{X} of the component stimuli increase until the aggregate associative strength V_{AX} equals R_{Y}, at which point the associative strengths stop changing (unless the US changes). When a new component is added to a compound CS to which the animal has already been conditioned, further conditioning with the augmented compound produces little or no increase in the associative strength of the added CS component because the error has already been reduced to zero, or to a low value. The occurrence of the US is already predicted nearly perfectly, so little or no error—or surprise—is introduced by the new CS component. Prior learning blocks learning to the new component.

To transition from Rescorla and Wagner's model to the TD model of classical conditioning (which we just call the TD model), we first recast their model in terms of the concepts that we are using throughout this book. Specifically, we match the notation we use for learning with linear function approximation (Section 9.4), and we think of the conditioning process as one of learning to predict the "magnitude of the US" on a trial on the basis of the compound CS presented on that trial, where the magnitude of a US Y is the R_{Y} of the Rescorla–Wagner model as given above. We also introduce states. Because the Rescorla–Wagner model is a *trial-level* model, meaning that it deals with how associative strengths change from trial to trial without considering any details about what happens within and between trials, we do not have to consider how states change during a trial until we present the full TD model in the following section. Instead, here we simply think of a state as a way of labeling a trial in terms of the collection of component CSs that are present on the trial.

Therefore, assume that trial-type, or state, s is described by a real-valued vector of features $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^\top$ where $x_i(s) = 1$ if CS_i, the i^{th} component of a compound CS, is present on the trial and 0 otherwise. Then if the d-dimensional vector of associative strengths is $\mathbf{w}$, the aggregate associative strength for trial-type s is

$$\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s). \tag{14.1}$$

This corresponds to a *value estimate* in reinforcement learning, and we think of it as the *US prediction*.

Now temporally let t denote the number of a complete trial and not its usual meaning as a time step (we revert to t's usual meaning when we extend this to the TD model below), and assume that S_t is the state corresponding to trial t. Conditioning trial t updates the associative strength vector $\mathbf{w}_t$ to $\mathbf{w}_{t+1}$ as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\,\delta_t\,\mathbf{x}(S_t), \tag{14.2}$$

where α is the step-size parameter, and—because here we are describing the Rescorla–Wagner model—δ_t is the *prediction error*

$$\delta_t = R_t - \hat{v}(S_t,\mathbf{w}_t). \tag{14.3}$$

R_t is the target of the prediction on trial t, that is, the magnitude of the US, or in Rescorla and Wagner's terms, the associative strength that the US on the trial can support. Note that because of the factor $\mathbf{x}(S_t)$ in (14.2), only the associative strengths of CS components present on a trial are adjusted as a result of that trial. You can think of the prediction error as a measure of surprise, and the aggregate associative strength as the animal's expectation that is violated when it does not match the target US magnitude.

From the perspective of machine learning, the Rescorla–Wagner model is an error-correction supervised learning rule. It is essentially the same as the Least Mean Square (LMS), or Widrow-Hoff, learning rule (Widrow and Hoff, 1960) that finds the weights—here the associative strengths—that make the average of the squares of all the errors as close to zero as possible. It is a "curve-fitting," or regression, algorithm that is widely used in engineering and scientific applications (see Section 9.4).[3]

The Rescorla–Wagner model was very influential in the history of animal learning theory because it showed that a "mechanistic" theory could account for the main facts about blocking without resorting to more complex cognitive theories involving, for example, an animal's explicit recognition that another stimulus component had been added and then scanning its short-term memory backward to reassess the predictive relationships involving the US. The Rescorla–Wagner model showed how traditional contiguity theories of conditioning—that temporal contiguity of stimuli was a necessary and sufficient condition for learning—could be adjusted in a simple way to account for blocking (Moore and Schmajuk, 2008).

The Rescorla–Wagner model provides a simple account of blocking and some other features of classical conditioning, but it is not a complete or perfect model of classical

[3]The only differences between the LMS rule and the Rescorla–Wagner model are that for LMS the input vectors $\mathbf{x}_t$ can have any real numbers as components, and—at least in the simplest version of the LMS rule—the step-size parameter α does not depend on the input vector or the identity of the stimulus setting the prediction target.

conditioning. Different ideas account for a variety of other observed effects, and progress is still being made toward understanding the many subtleties of classical conditioning. The TD model, which we describe next, though also not a complete or perfect model model of classical conditioning, extends the Rescorla–Wagner model to address how within-trial and between-trial timing relationships among stimuli can influence learning and how higher-order conditioning might arise.

14.2.3 The TD Model

The TD model is a *real-time* model, as opposed to a trial-level model like the Rescorla–Wagner model. A single step t in the Rescorla–Wagner model represents an entire conditioning trial. The model does not apply to details about what happens during the time a trial is taking place, or what might happen between trials. Within each trial an animal might experience various stimuli whose onsets occur at particular times and that have particular durations. These timing relationships strongly influence learning. The Rescorla–Wagner model also does not include a mechanism for higher-order conditioning, whereas for the TD model, higher-order conditioning is a natural consequence of the bootstrapping idea that is at the base of TD algorithms.

To describe the TD model we begin with the formulation of the Rescorla–Wagner model above, but t now labels time steps within or between trials instead of complete trials. Think of the time between t and $t+1$ as a small time interval, say .01 second, and think of a trial as a sequences of states, one associated with each time step, where the state at step t now represents details of how stimuli are represented at t instead of just a label for the CS components present on a trial. In fact, we can completely abandon the idea of trials. From the point of view of the animal, a trial is just a fragment of its continuing experience interacting with its world. Following our usual view of an agent interacting with its environment, imagine that the animal is experiencing an endless sequence of states s, each represented by a feature vector $\mathbf{x}(s)$. That said, it is still often convenient to refer to trials as fragments of time during which patterns of stimuli repeat in an experiment.

State features are not restricted to describing the external stimuli that an animal experiences; they can describe neural activity patterns that external stimuli produce in an animal's brain, and these patterns can be history-dependent, meaning that they can be persistent patterns produced by sequences of external stimuli. Of course, we do not know exactly what these neural activity patterns are, but a real-time model like the TD model allows one to explore the consequences on learning of different hypotheses about the internal representations of external stimuli. For these reasons, the TD model does not commit to any particular state representation. In addition, because the TD model includes discounting and eligibility traces that span time intervals between stimuli, the model also makes it possible to explore how discounting and eligibility traces interact with stimulus representations in making predictions about the results of classical conditioning experiments.

Below we describe some of the state representations that have been used with the TD model and some of their implications, but for the moment we stay agnostic about

the representation and just assume that each state s is represented by a feature vector $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_n(s))^\top$. Then the aggregate associative strength corresponding to a state s is given by (14.1), the same as for the Rescorla-Wgner model, but the TD model updates the associative strength vector, $\mathbf{w}$, differently. With t now labeling a time step instead of a complete trial, the TD model governs learning according to this update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t, \tag{14.4}$$

which replaces $\mathbf{x}_t(S_t)$ in the Rescorla–Wagner update (14.2) with $\mathbf{z}_t$, a vector of eligibility traces, and instead of the δ_t of (14.3), here δ_t is a TD error:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \tag{14.5}$$

where γ is a discount factor (between 0 and 1), R_t is the prediction target at time t, and $\hat{v}(S_{t+1}, \mathbf{w}_t)$ and $\hat{v}(S_t, \mathbf{w}_t)$ are aggregate associative strengths at $t+1$ and t as defined by (14.1).

Each component i of the eligibility-trace vector $\mathbf{z}_t$ increments or decrements according to the component $x_i(S_t)$ of the feature vector $\mathbf{x}(S_t)$, and otherwise decays with a rate determined by $\gamma\lambda$:

$$\mathbf{z}_{t+1} = \gamma\lambda \mathbf{z}_t + \mathbf{x}(S_t). \tag{14.6}$$

Here λ is the usual eligibility trace decay parameter.

Note that if $\gamma = 0$, the TD model reduces to the Rescorla–Wagner model with the exceptions that: the meaning of t is different in each case (a trial number for the Rescorla–Wagner model and a time step for the TD model), and in the TD model there is a one-time-step lead in the prediction target R. The TD model is equivalent to the backward view of the semi-gradient TD(λ) algorithm with linear function approximation (Chapter 12), except that R_t in the model does not have to be a reward signal as it does when the TD algorithm is used to learn a value function for policy-improvement.

14.2.4 TD Model Simulations

Real-time conditioning models like the TD model are interesting primarily because they make predictions for a wide range of situations that cannot be represented by trial-level models. These situations involve the timing and durations of conditionable stimuli, the timing of these stimuli in relation to the timing of the US, and the timing and shapes of CRs. For example, the US generally must begin after the onset of a neutral stimulus for conditioning to occur, with the rate and effectiveness of learning depending on the inter-stimulus interval, or ISI, the interval between the onsets of the CS and the US. When CRs appear, they generally begin before the appearance of the US and their temporal profiles change during learning. In conditioning with compound CSs, the component stimuli of the compound CSs may not all begin and end at the same time, sometimes forming what is called a *serial compound* in which the component stimuli occur in a sequence over time. Timing considerations like these make it important to consider how stimuli are represented, how these representations unfold over time during and between trials, and how they interact with discounting and eligibility traces.

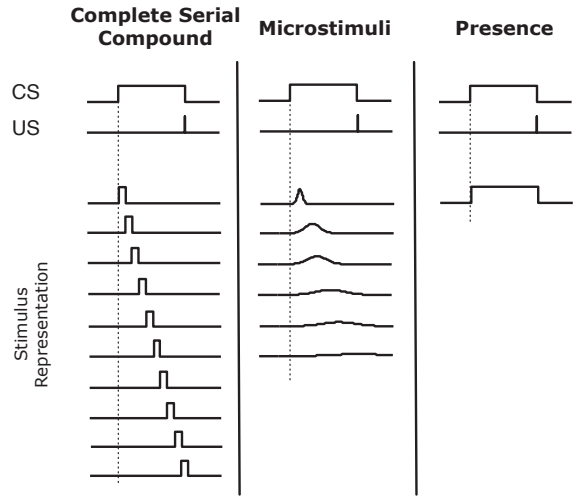


Figure 14.1: Three stimulus representations (in columns) sometimes used with the TD model. Each row represents one element of the stimulus representation. The three representations vary along a temporal generalization gradient, with no generalization between nearby time points in the complete serial compound (left column) and complete generalization between nearby time points in the presence representation (right column). The microstimulus representation occupies a middle ground. The degree of temporal generalization determines the temporal granularity with which US predictions are learned. Adapted with minor changes from *Learning & Behavior*, Evaluating the TD Model of Classical Conditioning, volume 40, 2012, p. 311, E. A. Ludvig, R. S. Sutton, E. J. Kehoe. With permission of Springer.

Figure 14.1 shows three of the stimulus representations that have been used in exploring the behavior of the TD model: the *complete serial compound* (CSC), the *microstimulus* (MS), and the *presence* representations (Ludvig, Sutton, and Kehoe, 2012). These representations differ in the degree to which they force generalization among nearby time points during which a stimulus is present.

The simplest of the representations shown in Figure 14.1 is the presence representation in the figure's right column. This representation has a single feature for each component CS present on a trial, where the feature has value 1 whenever that component is present, and 0 otherwise.[4] The presence representation is not a realistic hypothesis about how stimuli are represented in an animal's brain, but as we describe below, the TD model with this representation can produce many of the timing phenomena seen in classical conditioning.

For the CSC representation (left column of Figure 14.1), the onset of each external stimulus initiates a sequence of precisely-timed short-duration internal signals that

[4]In our formalism, there is a different state, S_t, for each time step t during a trial, and for a trial in which a compound CS consists of n component CSs of various durations occurring at various times throughout the trial, there is a feature, x_i, for each component CS_i, $i = 1, \ldots, n$, where $x_i(S_t) = 1$ for all times t when the CS_i is present, and equals zero otherwise.

continues until the external stimulus ends.[5] This is like assuming the animal's nervous system has a clock that keeps precise track of time during stimulus presentations; it is what engineers call a "tapped delay line." Like the presence representation, the CSC representation is unrealistic as a hypothesis about how the brain internally represents stimuli, but Ludvig et al. (2012) call it a "useful fiction" because it can reveal details of how the TD model works when relatively unconstrained by the stimulus representation. The CSC representation is also used in most TD models of dopamine-producing neurons in the brain, a topic we take up in Chapter 15. The CSC representation is often viewed as an essential part of the TD model, although this view is mistaken.

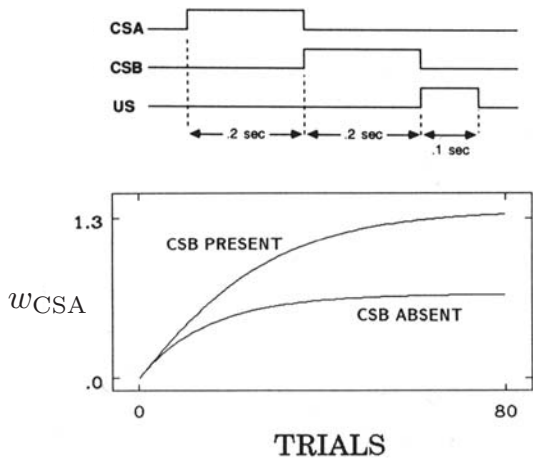
The MS representation (center column of Figure 14.1) is like the CSC representation in that each external stimulus initiates a cascade of internal stimuli, but in this case the internal stimuli—the microstimuli—are not of such limited and non-overlapping form; they are extended over time and overlap. As time elapses from stimulus onset, different sets of microstimuli become more or less active, and each subsequent microstimulus becomes progressively wider in time and reaches a lower maximal level. Of course, there are many MS representations depending on the nature of the microstimuli, and a number of examples of MS representations have been studied in the literature, in some cases along with proposals for how an animal's brain might generate them (see the Bibliographic and Historical Comments at the end of this chapter). MS representations are more realistic than the presence or CSC representations as hypotheses about neural representations of stimuli, and they allow the behavior of the TD model to be related to a broader collection of phenomena observed in animal experiments. In particular, by assuming that cascades of microstimuli are initiated by USs as well as by CSs, and by studying the significant effects on learning of interactions between microstimuli, eligibility traces, and discounting, the TD model is helping to frame hypotheses to account for many of the subtle phenomena of classical conditioning and how an animal's brain might produce them. We say more about this below, particularly in Chapter 15 where we discuss reinforcement learning and neuroscience.

Even with the simple presence representation, however, the TD model produces all the basic properties of classical conditioning that are accounted for by the Rescorla–Wagner model, plus features of conditioning that are beyond the scope of trial-level models. For example, as we have already mentioned, a conspicuous feature of classical conditioning is that the US generally must begin *after* the onset of a neutral stimulus for conditioning to occur, and that after conditioning, the CR begins *before* the appearance of the US. In other words, conditioning generally requires a positive ISI, and the CR generally anticipates the US. How the strength of conditioning (e.g., the percentage of CRs elicited by a CS) depends on the ISI varies substantially across species and response systems, but it typically has the following properties: it is negligible for a zero or negative ISI, i.e., when the US onset occurs simultaneously with, or earlier than, the CS onset (although research has found that associative strengths sometimes increase slightly or become negative with

[5]In our formalism, for each CS component CS_i present on a trial, and for each time step t during a trial, there is a separate feature x_i^t, where $x_i^t(S_{t'}) = 1$ if $t = t'$ for any t' at which CS_i is present, and equals 0 otherwise. This is different from the CSC representation in Sutton and Barto (1990) in which there are the same distinct features for each time step but no reference to external stimuli; hence the name complete serial compound.

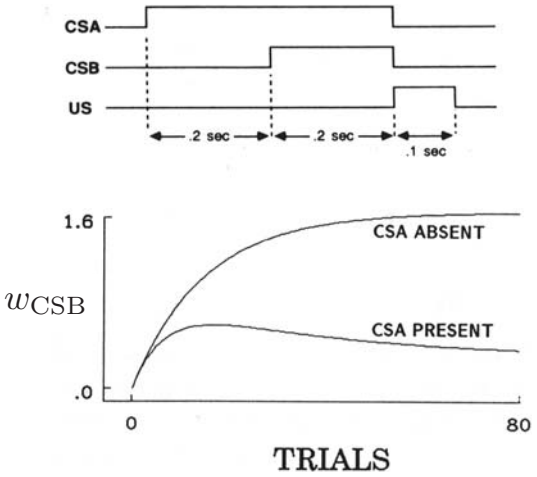
negative ISIs); it increases to a maximum at a positive ISI where conditioning is most effective; and it then decreases to zero after an interval that varies widely with response systems. The precise shape of this dependency for the TD model depends on the values of its parameters and details of the stimulus representation, but these basic features of ISI-dependency are core properties of the TD model.

One of the theoretical issues arising with serial-compound conditioning, that is, conditioning with a compound CS whose components occur in a sequence, concerns the facilitation of remote associations. It has been found that if the empty trace interval between a first CS (CSA) and the US is filled with a second CS (CSB) to form a serial-compound stimulus, then conditioning to CSA is facilitated. Shown to the right is the behavior of the TD model with the presence representation in a simulation of such an experiment whose timing details are shown above. Consistent with the experimental results (Kehoe, 1982), the model shows facilitation of both the rate of conditioning and the asymptotic level of conditioning of the first CS due to the presence of the second CS.



Facilitation of remote associations in the TD model

A well-known demonstration of the effects on conditioning of temporal relationships among stimuli within a trial is an experiment by Egger and Miller (1962) that involved two overlapping CSs in a delay configuration as shown to the right (top). Although CSB was in a better temporal relationship with the US, the presence of CSA substantially reduced conditioning to CSB as compared to controls in which CSA was absent. Directly to the right is shown the same result being generated by the TD model in a simulation of this experiment with the presence representation.



The Egger-Miller effect in the TD model

The TD model accounts for blocking because it is an error-correcting learning rule like the Rescorla–Wagner model. Beyond accounting for basic blocking results, however, the TD model predicts (with the presence representation and more complex representations as well) that blocking is reversed if the blocked stimulus is moved earlier in time so that its onset occurs before the onset of the blocking stimulus (like CSA in the diagram to the right). This feature of the TD model's behavior deserves attention because it had not been observed at the time of the model's introduction. Recall that in blocking, if an animal has already learned that one CS predicts a US, then learning that a newly-added second CS also predicts the US is much reduced, i.e., is blocked. But if the newly-added second CS begins earlier than the pretrained CS, then—according to the TD model—learning to the newly-added CS is not blocked. In fact, as training continues and the newly-added CS gains associative strength, the pretrained CS loses associative strength. The behavior of the TD model under these conditions is shown in the lower part of Figure 14.2.

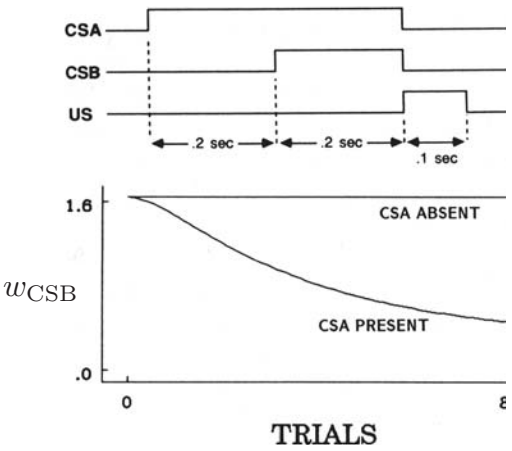


Figure 14.2: Temporal primacy overriding blocking in the TD model.

This simulation experiment differed from the Egger-Miller experiment (bottom of the preceding page) in that the shorter CS with the later onset was given prior training until it was fully associated with the US. This surprising prediction led Kehoe, Schreurs, and Graham (1987) to conduct the experiment using the well-studied rabbit nictitating membrane preparation. Their results confirmed the model's prediction, and they noted that non-TD models have considerable difficulty explaining their data.

With the TD model, an earlier predictive stimulus takes precedence over a later predictive stimulus because, like all the prediction methods described in this book, the TD model is based on the backing-up or bootstrapping idea: updates to associative strengths shift the strengths at a particular state toward the strength at later states. Another consequence of bootstrapping is that the TD model provides an account of higher-order conditioning, a feature of classical conditioning that is beyond the scope of the Rescoral-Wagner and similar models. As we described above, higher-order conditioning is the phenomenon in which a previously-conditioned CS can act as a US in conditioning another initially neutral stimulus. Figure 14.3 shows the behavior of the TD model (again with the presence representation) in a higher-order conditioning experiment—in this case it is second-order conditioning. In the first phase (not shown in the figure), CSB is trained to predict a US so that its associative strength increases, here to 1.65. In the second phase, CSA is paired with CSB in the absence of the US, in the sequential arrangement shown at the top of the figure. CSA acquires associative strength even though it is never paired with the US. With continued training, CSA's associative strength reaches a peak and then decreases because the associative strength of CSB, the secondary reinforcer, decreases so that it loses its ability to provide secondary reinforcement. CSB's associative

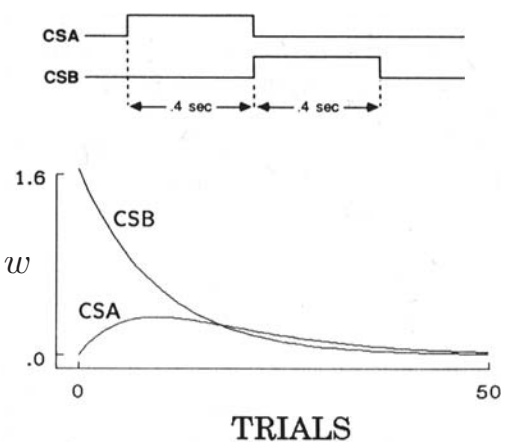


Figure 14.3: Second-order conditioning with the TD model.

strength decreases because the US does not occur in these higher-order conditioning trials. These are *extinction trials* for CSB because its predictive relationship to the US is disrupted so that its ability to act as a reinforcer decreases. This same pattern is seen in animal experiments. This extinction of conditioned reinforcement in higher-order conditioning trials makes it difficult to demonstrate higher-order conditioning unless the original predictive relationships are periodically refreshed by occasionally inserting first-order trials.

The TD model produces an analog of second- and higher-order conditioning because $\gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t)$ appears in the TD error δ_t (14.5). This means that as a result of previous learning, $\gamma\hat{v}(S_{t+1},\mathbf{w}_t)$ can differ from $\hat{v}(S_t,\mathbf{w}_t)$, making δ_t non-zero (a temporal difference). This difference has the same status as R_{t+1} in (14.5), implying that as far as learning is concerned there is no difference between a temporal difference and the occurrence of a US. In fact, this feature of the TD algorithm is one of the major reasons for its development, which we now understand through its connection to dynamic programming as described in Chapter 6. Bootstrapping values is intimately related to second-order, and higher-order, conditioning.

In the examples of the TD model's behavior described above, we examined only the changes in the associative strengths of the CS components; we did not look at what the model predicts about properties of an animal's conditioned responses (CRs): their timing, shape, and how they develop over conditioning trials. These properties depend on the species, the response system being observed, and parameters of the conditioning trials, but in many experiments with different animals and different response systems, the magnitude of the CR, or the probability of a CR, increases as the expected time of the US approaches. For example, in classical conditioning of a rabbit's nictitating membrane response that we mentioned above, over conditioning trials the delay from CS onset to when the nictitating membrane begins to move across the eye decreases over trials, and the amplitude of this anticipatory closure gradually increases over the interval between the CS and the US until the membrane reaches maximal closure at the expected time of the US. The timing and shape of this CR is critical to its adaptive significance—covering the eye too early reduces vision (even though the nictitating membrane is translucent), while covering it too late is of little protective value. Capturing CR features like these is challenging for models of classical conditioning.

The TD model does not include as part of its definition any mechanism for translating the time course of the US prediction, $\hat{v}(S_t,\mathbf{w}_t)$, into a profile that can be compared with the properties of an animal's CR. The simplest choice is to let the time course of

a simulated CR equal the time course of the US prediction. In this case, features of simulated CRs and how they change over trials depend only on the stimulus representation chosen and the values of the model's parameters α, γ, and λ.

Figure 14.4 shows the time courses of US predictions at different points during learning with the three representations shown in Figure 14.1. For these simulations the US occurred 25 time steps after the onset of the CS, and $\alpha = .05$, $\lambda = .95$ and $\gamma = .97$. With the CSC representation (Figure 14.4 left), the curve of the US prediction formed by the TD model increases exponentially throughout the interval between the CS and the US until it reaches a maximum exactly when the US occurs (at time step 25). This exponential increase is the result of discounting in the TD model learning rule. With the presence representation (Figure 14.4 middle), the US prediction is nearly constant while the stimulus is present because there is only one weight, or associative strength, to be learned for each stimulus. Consequently, the TD model with the presence representation cannot recreate many features of CR timing. With an MS representation (Figure 14.4 right), the development of the TD model's US prediction is more complicated. After 200 trials the prediction's profile is a reasonable approximation of the US prediction curve produced with the CSC representation.

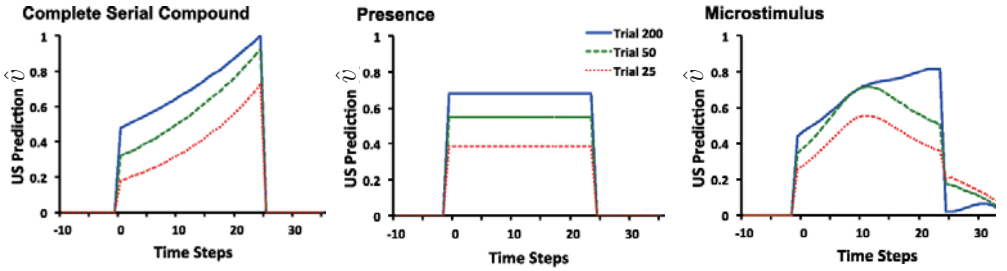


Figure 14.4: Time course of US prediction over the course of acquisition for the TD model with three different stimulus representations. Left: With the complete serial compound (CSC), the US prediction increases exponentially through the interval, peaking at the time of the US. At asymptote (trial 200), the US prediction peaks at the US intensity (1 in these simulations). Middle: With the presence representation, the US prediction converges to an almost constant level. This constant level is determined by the US intensity and the length of the CS–US interval. Right: With the microstimulus representation, at asymptote, the TD model approximates the exponentially increasing time course depicted with the CSC through a linear combination of the different microstimuli. Adapted with minor changes from *Learning & Behavior*, Evaluating the TD Model of Classical Conditioning, volume 40, 2012, E. A. Ludvig, R. S. Sutton, E. J. Kehoe. With permission of Springer.

The US prediction curves shown in Figure 14.4 were not intended to precisely match profiles of CRs as they develop during conditioning in any particular animal experiment, but they illustrate the strong influence that the stimulus representation has on predictions derived from the TD model. Further, although we can only mention it here, how the stimulus representation interacts with discounting and eligibility traces is important

in determining properties of the US prediction profiles produced by the TD model. Another dimension beyond what we can discuss here is the influence of different response-generation mechanisms that translate US predictions into CR profiles; the profiles shown in Figure 14.4 are "raw" US prediction profiles. Even without any special assumption about how an animal's brain might produce overt responses from US predictions, however, the profiles in Figure 14.4 for the CSC and MS representations increase as the time of the US approaches and reach a maximum at the time of the US, as is seen in many animal conditioning experiments.

The TD model, when combined with particular stimulus representations and response-generation mechanisms, is able to account for a surprisingly wide range of phenomena observed in animal classical conditioning experiments, but it is far from being a perfect model. To generate other details of classical conditioning the model needs to be extended, perhaps by adding model-based elements and mechanisms for adaptively altering some of its parameters. Other approaches to modeling classical conditioning depart significantly from the Rescorla–Wagner-style error-correction process. Bayesian models, for example, work within a probabilistic framework in which experience revises probability estimates. All of these models usefully contribute to our understanding of classical conditioning.

Perhaps the most notable feature of the TD model is that it is based on a theory—the theory we have described in this book—that suggests an account of what an animal's nervous system is *trying to do* while undergoing conditioning: it is trying to form accurate *long-term predictions*, consistent with the limitations imposed by the way stimuli are represented and how the nervous system works. In other words, it suggests a *normative account* of classical conditioning in which long-term, instead of immediate, prediction is a key feature.

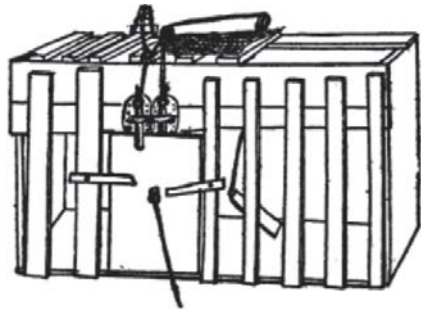
The development of the TD model of classical conditioning is one instance in which the explicit goal was to model some of the details of animal learning behavior. In addition to its standing as an *algorithm*, then, TD learning is also the basis of this *model* of aspects of biological learning. As we discuss in Chapter 15, TD learning has also turned out to underlie an influential model of the activity of neurons that produce dopamine, a chemical in the brain of mammals that is deeply involved in reward processing. These are instances in which reinforcement learning theory makes detailed contact with animal behavioral and neural data.

We now turn to considering correspondences between reinforcement learning and animal behavior in instrumental conditioning experiments, the other major type of laboratory experiment studied by animal learning psychologists.

14.3 Instrumental Conditioning

In *instrumental conditioning* experiments learning depends on the consequences of behavior: the delivery of a reinforcing stimulus is contingent on what the animal does. In classical conditioning experiments, in contrast, the reinforcing stimulus—the US—is delivered independently of the animal's behavior. Instrumental conditioning is usually considered to be the same as *operant conditioning*, the term B. F. Skinner (1938, 1963) introduced for experiments with behavior-contingent reinforcement, though the experiments and theories of those who use these two terms differ in a number of ways, some of which we touch on below. We will exclusively use the term instrumental conditioning for experiments in which reinforcement is contingent upon behavior. The roots of instrumental conditioning go back to experiments performed by the American psychologist Edward Thorndike one hundred years before publication of the first edition of this book.

Thorndike observed the behavior of cats when they were placed in "puzzle boxes," such as the one at the right, from which they could escape by appropriate actions. For example, a cat could open the door of one box by performing a sequence of three separate actions: depressing a platform at the back of the box, pulling a string by clawing at it, and pushing a bar up or down. When first placed in a puzzle box, with food visible outside, all but a few of Thorndike's cats displayed "evident signs of discomfort" and extraordinarily vigorous activity "to strive instinctively to escape from confinement" (Thorndike, 1898).



One of Thorndike's puzzle boxes.
Reprinted from Thorndike, Animal Intelligence: An Experimental Study of the Associative Processes in Animals, *The Psychological Review, Series of Monograph Supplements* II(4), Macmillan, New York, 1898.

In experiments with different cats and boxes with different escape mechanisms, Thorndike recorded the amounts of time each cat took to escape over multiple experiences in each box. He observed that the time almost invariably decreased with successive experiences, for example, from 300 seconds to 6 or 7 seconds. He described cats' behavior in a puzzle box like this:

> The cat that is clawing all over the box in her impulsive struggle will probably claw the string or loop or button so as to open the door. And gradually all the other non-successful impulses will be stamped out and the particular impulse leading to the successful act will be stamped in by the resulting pleasure, until, after many trials, the cat will, when put in the box, immediately claw the button or loop in a definite way. (Thorndike 1898, p. 13)

These and other experiments (some with dogs, chicks, monkeys, and even fish) led Thorndike to formulate a number of "laws" of learning, the most influential being the *Law of Effect*, a version of which we quoted in Chapter 1 (page 15). This law describes what is generally known as learning by trial and error. As mentioned in Chapter 1, many aspects of the Law of Effect have generated controversy, and its details have been modified over the years. Still the law—in one form or another—expresses an enduring principle of learning.

Essential features of reinforcement learning algorithms correspond to features of animal learning described by the Law of Effect. First, reinforcement learning algorithms are *selectional*, meaning that they try alternatives and select among them by comparing their consequences. Second, reinforcement learning algorithms are *associative*, meaning that the alternatives found by selection are associated with particular situations, or states, to form the agent's policy. Like learning described by the Law of Effect, reinforcement

learning is not just the process of *finding* actions that produce a lot of reward, but also of *connecting* these actions to situations or states. Thorndike used the phrase learning by "selecting and connecting" (Hilgard, 1956). Natural selection in evolution is a prime example of a selectional process, but it is not associative (at least as it is commonly understood); supervised learning is associative, but it is not selectional because it relies on instructions that directly tell the agent how to change its behavior.

In computational terms, the Law of Effect describes an elementary way of combining *search* and *memory*: search in the form of trying and selecting among many actions in each situation, and memory in the form of associations linking situations with the actions found—so far—to work best in those situations. Search and memory are essential components of all reinforcement learning algorithms, whether memory takes the form of an agent's policy, value function, or environment model.

A reinforcement learning algorithm's need to search means that it has to explore in some way. Animals clearly explore as well, and early animal learning researchers disagreed about the degree of guidance an animal uses in selecting its actions in situations like Thorndike's puzzle boxes. Are actions the result of "absolutely random, blind groping" (Woodworth, 1938, p. 777), or is there some degree of guidance, either from prior learning, reasoning, or other means? Although some thinkers, including Thorndike, seem to have taken the former position, others favored more deliberate exploration. Reinforcement learning algorithms allow wide latitude for how much guidance an agent can employ in selecting actions. The forms of exploration we have used in the algorithms presented in this book, such as ε-greedy and upper-confidence-bound action selection, are merely among the simplest. More sophisticated methods are possible, with the only stipulation being that there has to be *some* form of exploration for the algorithms to work effectively.

The feature of our treatment of reinforcement learning allowing the set of actions available at any time to depend on the environment's current state echoes something Thorndike observed in his cats' puzzle-box behaviors. The cats selected actions from those that they instinctively perform in their current situation, which Thorndike called their "instinctual impulses." First placed in a puzzle box, a cat instinctively scratches, claws, and bites with great energy: a cat's instinctual responses to finding itself in a confined space. Successful actions are selected from these and not from every possible action or activity. This is like the feature of our formalism where the action selected from a state s belongs to a set of admissible actions, $\mathcal{A}(s)$. Specifying these sets is an important aspect of reinforcement learning because it can radically simplify learning. They are like an animal's instinctual impulses. On the other hand, Thorndike's cats might have been exploring according to an instinctual context-specific *ordering* over actions rather than by just selecting from a set of instinctual impulses. This is another way to make reinforcement learning easier.

Among the most prominent animal learning researchers influenced by the Law of Effect were Clark Hull (e.g., Hull, 1943) and B. F. Skinner (e.g., Skinner, 1938). At the center of their research was the idea of selecting behavior on the basis of its consequences. Reinforcement learning has features in common with Hull's theory, which included eligibility-like mechanisms and secondary reinforcement to account for the ability to learn when there is a significant time interval between an action and the consequent reinforcing

stimulus (see Section 14.4). Randomness also played a role in Hull's theory through what he called "behavioral oscillation" to introduce exploratory behavior.

Skinner did not fully subscribe to the memory aspect of the Law of Effect. Being averse to the idea of associative linkages, he instead emphasized selection from spontaneously-emitted behavior. He introduced the term "operant" to emphasize the key role of an action's effects on an animal's environment. Unlike the experiments of Thorndike and others, which consisted of sequences of separate trials, Skinner's operant conditioning experiments allowed animal subjects to behave for extended periods of time without interruption. He invented the operant conditioning chamber, now called a "Skinner box," the most basic version of which contains a lever or key that an animal can press to obtain a reward, such as food or water, which would be delivered according to a well-defined rule, called a reinforcement schedule. By recording the cumulative number of lever presses as a function of time, Skinner and his followers could investigate the effect of different reinforcement schedules on the animal's rate of lever-pressing. Modeling results from experiments likes these using the reinforcement learning principles we present in this book is not well developed, but we mention some exceptions in the Bibliographic and Historical Remarks section at the end of this chapter.

Another of Skinner's contributions resulted from his recognition of the effectiveness of training an animal by reinforcing successive approximations of the desired behavior, a process he called *shaping*. Although this technique had been used by others, including Skinner himself, its significance was impressed upon him when he and colleagues were attempting to train a pigeon to bowl by swiping a wooden ball with its beak. After waiting for a long time without seeing any swipe that they could reinforce, they

> ... decided to reinforce any response that had the slightest resemblance to a swipe—perhaps, at first, merely the behavior of looking at the ball—and then to select responses which more closely approximated the final form. The result amazed us. In a few minutes, the ball was caroming off the walls of the box as if the pigeon had been a champion squash player. (Skinner, 1958, p. 94)

Not only did the pigeon learn a behavior that is unusual for pigeons, it learned quickly through an interactive process in which its behavior and the reinforcement contingencies changed in response to each other. Skinner compared the process of altering reinforcement contingencies to the work of a sculptor shaping clay into a desired form. Shaping is a powerful technique for computational reinforcement learning systems as well. When it is difficult for an agent to receive any non-zero reward signal at all, either due to sparseness of rewarding situations or their inaccessibility given initial behavior, starting with an easier problem and incrementally increasing its difficulty as the agent learns can be an effective, and sometimes indispensable, strategy.

A concept from psychology that is especially relevant in the context of instrumental conditioning is *motivation*, which refers to processes that influence the direction and strength, or vigor, of behavior. Thorndike's cats, for example, were motivated to escape from puzzle boxes because they wanted the food that was sitting just outside. Obtaining this goal was rewarding to them and reinforced the actions allowing them to escape. It is difficult to link the concept of motivation, which has many dimensions, in a precise

way to reinforcement learning's computational perspective, but there are clear links with some of its dimensions.

In one sense, a reinforcement learning agent's reward signal is at the base of its motivation: the agent is motivated to maximize the total reward it receives over the long run. A key facet of motivation, then, is what makes an agent's experience rewarding. In reinforcement learning, reward signals depend on the state of the reinforcement learning agent's environment and the agent's actions. Further, as pointed out in Chapter 1, the state of the agent's environment not only includes information about what is external to the machine, like an organism or a robot, that houses the agent, but also what is internal to this machine. Some internal state components correspond to what psychologists call an animal's *motivational state*, which influences what is rewarding to the animal. For example, an animal will be more rewarded by eating when it is hungry than when it has just finished a satisfying meal. The concept of state dependence is broad enough to allow for many types of modulating influences on the generation of reward signals.

Value functions provide a further link to psychologists' concept of motivation. If the most basic motive for selecting an action is to obtain as much reward as possible, for a reinforcement learning agent that selects actions using a value function, a more proximal motive is to *ascend the gradient of its value function*, that is, to select actions expected to lead to the most highly-valued next states (or what is essentially the same thing, to select actions with the greatest action-values). For these agents, value functions are the main driving force determining the direction of their behavior.

Another dimension of motivation is that an animal's motivational state not only influences learning, but also influences the strength, or vigor, of the animal's behavior after learning. For example, after learning to find food in the goal box of a maze, a hungry rat will run faster to the goal box than one that is not hungry. This aspect of motivation does not link so cleanly to the reinforcement learning framework we present here, but in the Bibliographical and Historical Remarks section at the end of this chapter we cite several publications that propose theories of behavioral vigor based on reinforcement learning.

We turn now to the subject of learning when reinforcing stimuli occur well after the events they reinforce. The mechanisms used by reinforcement learning algorithms to enable learning with delayed reinforcement—eligibility traces and TD learning—closely correspond to psychologists' hypotheses about how animals can learn under these conditions.

14.4 Delayed Reinforcement

The Law of Effect requires a backward effect on connections, and some early critics of the law could not conceive of how the present could affect something that was in the past. This concern was amplified by the fact that learning can even occur when there is a considerable delay between an action and the consequent reward or penalty. Similarly, in classical conditioning, learning can occur when US onset follows CS offset by a non-negligible time interval. We call this the problem of delayed reinforcement, which is related to what Minsky (1961) called the "credit-assignment problem for learning systems": how do you

distribute credit for success among the many decisions that may have been involved in producing it? The reinforcement learning algorithms presented in this book include two basic mechanisms for addressing this problem. The first is the use of eligibility traces, and the second is the use of TD methods to learn value functions that provide nearly immediate evaluations of actions (in tasks like instrumental conditioning experiments) or that provide immediate prediction targets (in tasks like classical conditioning experiments). Both of these methods correspond to similar mechanisms proposed in theories of animal learning.

Pavlov (1927) pointed out that every stimulus must leave a trace in the nervous system that persists for some time after the stimulus ends, and he proposed that stimulus traces make learning possible when there is a temporal gap between the CS offset and the US onset. To this day, conditioning under these conditions is called *trace conditioning* (page 344). Assuming a trace of the CS remains when the US arrives, learning occurs through the simultaneous presence of the trace and the US. We discuss some proposals for trace mechanisms in the nervous system in Chapter 15.

Stimulus traces were also proposed as a means for bridging the time interval between actions and consequent rewards or penalties in instrumental conditioning. In Hull's influential learning theory, for example, "molar stimulus traces" accounted for what he called an animal's *goal gradient*, a description of how the maximum strength of an instrumentally-conditioned response decreases with increasing delay of reinforcement (Hull, 1932, 1943). Hull hypothesized that an animal's actions leave internal stimuli whose traces decay exponentially as functions of time since an action was taken. Looking at the animal learning data available at the time, he hypothesized that the traces effectively reach zero after 30 to 40 seconds.

The eligibility traces used in the algorithms described in this book are like Hull's traces: they are decaying traces of past state visitations, or of past state–action pairs. Eligibility traces were introduced by Klopf (1972) in his neuronal theory in which they are temporally-extended traces of past activity at synapses, the connections between neurons. Klopf's traces are more complex than the exponentially-decaying traces our algorithms use, and we discuss this more when we take up his theory in Section 15.9.

To account for goal gradients that extend over longer time periods than spanned by stimulus traces, Hull (1943) proposed that longer gradients result from conditioned reinforcement passing backwards from the goal, a process acting in conjunction with his molar stimulus traces. Animal experiments showed that if conditions favor the development of conditioned reinforcement during a delay period, learning does not decrease with increased delay as much as it does under conditions that obstruct secondary reinforcement. Conditioned reinforcement is favored if there are stimuli that regularly occur during the delay interval. Then it is as if reward is not actually delayed because there is more immediate conditioned reinforcement. Hull therefore envisioned that there is a primary gradient based on the delay of the primary reinforcement mediated by stimulus traces, and that this is progressively modified, and lengthened, by conditioned reinforcement.

Algorithms presented in this book that use both eligibility traces and value functions to enable learning with delayed reinforcement correspond to Hull's hypothesis about how

animals are able to learn under these conditions. The actor–critic architecture discussed in Sections 13.5, 15.7, and 15.8 illustrates this correspondence most clearly. The critic uses a TD algorithm to learn a value function associated with the system's current behavior, that is, to predict the current policy's return. The actor updates the current policy based on the critic's predictions, or more exactly, on changes in the critic's predictions. The TD error produced by the critic acts as a conditioned reinforcement signal for the actor, providing an immediate evaluation of performance even when the primary reward signal itself is considerably delayed. Algorithms that estimate action-value functions, such as Q-learning and Sarsa, similarly use TD learning principles to enable learning with delayed reinforcement by means of conditioned reinforcement. The close parallel between TD learning and the activity of dopamine producing neurons that we discuss in Chapter 15 lends additional support to links between reinforcement learning algorithms and this aspect of Hull's learning theory.

14.5 Cognitive Maps

Model-based reinforcement learning algorithms use environment models that have elements in common with what psychologists call *cognitive maps*. Recall from our discussion of planning and learning in Chapter 8 that by an environment model we mean anything an agent can use to predict how its environment will respond to its actions in terms of state transitions and rewards, and by planning we mean any process that computes a policy from such a model. Environment models consist of two parts: the state-transition part encodes knowledge about the effect of actions on state changes, and the reward-model part encodes knowledge about the reward signals expected for each state or each state–action pair. A model-based algorithm selects actions by using a model to predict the consequences of possible courses of action in terms of future states and the reward signals expected to arise from those states. The simplest kind of planning is to compare the predicted consequences of collections of "imagined" sequences of decisions.

Questions about whether or not animals use environment models, and if so, what are the models like and how are they learned, have played influential roles in the history of animal learning research. Some researchers challenged the then-prevailing stimulus-response (S–R) view of learning and behavior, which corresponds to the simplest model-free way of learning policies, by demonstrating *latent learning*. In the earliest latent learning experiment, two groups of rats were run in a maze. For the experimental group, there was no reward during the first stage of the experiment, but food was suddenly introduced into the goal box of the maze at the start of the second stage. For the control group, food was in the goal box throughout both stages. The question was whether or not rats in the experimental group would have learned anything during the first stage in the absence of food reward. Although the experimental rats did not *appear* to learn much during the first, unrewarded, stage, as soon as they discovered the food that was introduced in the second stage, they rapidly caught up with the rats in the control group. It was concluded that "during the non-reward period, the rats [in the experimental group] were developing a latent learning of the maze which they were able to utilize as soon as reward was introduced" (Blodgett, 1929).

Latent learning is most closely associated with the psychologist Edward Tolman, who interpreted this result, and others like it, as showing that animals could learn a "cognitive map of the environment" in the absence of rewards or penalties, and that they could use the map later when they were motivated to reach a goal (Tolman, 1948). A cognitive map could also allow a rat to plan a route to the goal that was different from the route the rat had used in its initial exploration. Explanations of results like these led to the enduring controversy lying at the heart of the behaviorist/cognitive dichotomy in psychology. In modern terms, cognitive maps are not restricted to models of spatial layouts but are more generally environment models, or models of an animal's "task space" (e.g., Wilson, Takahashi, Schoenbaum, and Niv, 2014). The cognitive map explanation of latent learning experiments is analogous to the claim that animals use model-based algorithms, and that environment models can be learned even without explicit rewards or penalties. Models are then used for planning when the animal is motivated by the appearance of rewards or penalties.

Tolman's account of how animals learn cognitive maps was that they learn stimulus-stimulus, or S–S, associations by experiencing successions of stimuli as they explore an environment. In psychology this is called *expectancy theory*: given S–S associations, the occurrence of a stimulus generates an expectation about the stimulus to come next. This is much like what control engineers call *system identification*, in which a model of a system with unknown dynamics is learned from labeled training examples. In the simplest discrete-time versions, training examples are S–S′ pairs, where S is a state and S′, the subsequent state, is the label. When S is observed, the model creates the "expectation" that S′ will be observed next. Models more useful for planning involve actions as well, so that examples look like SA–S′, where S′ is expected when action A is executed in state S. It is also useful to learn how the environment generates rewards. In this case, examples are of the form S–R or SA–R, where R is a reward signal associated with S or the SA pair. These are all forms of supervised learning by which an agent can acquire cognitive-like maps whether or not it receives any non-zero reward signals while exploring its environment.

14.6 Habitual and Goal-directed Behavior

The distinction between model-free and model-based reinforcement learning algorithms corresponds to the distinction psychologists make between *habitual* and *goal-directed* control of learned behavioral patterns. Habits are behavior patterns triggered by appropriate stimuli and then performed more-or-less automatically. Goal-directed behavior, according to how psychologists use the phrase, is purposeful in the sense that it is controlled by knowledge of the value of goals and the relationship between actions and their consequences. Habits are sometimes said to be controlled by antecedent stimuli, whereas goal-directed behavior is said to be controlled by its consequences (Dickinson, 1980, 1985). Goal-directed control has the advantage that it can rapidly change an animal's behavior when the environment changes its way of reacting to the animal's actions. While habitual behavior responds quickly to input from an accustomed environment, it is unable to quickly adjust to changes in the environment. The development of goal-directed

behavioral control was likely a major advance in the evolution of animal intelligence.

Figure 14.5 illustrates the difference between model-free and model-based decision strategies in a hypothetical task in which a rat has to navigate a maze that has distinctive goal boxes, each delivering an associated reward of the magnitude shown (Figure 14.5 top). Starting at S_1, the rat has to first select left (L) or right (R) and then has to select L or R again at S_2 or S_3 to reach one of the goal boxes. The goal boxes are the terminal states of each episode of the rat's episodic task. A model-free strategy (Figure 14.5 lower left) relies on stored values for state–action pairs. These action values are estimates of the highest return the rat can expect for each action taken from each (nonterminal) state. They are obtained over many trials of running the maze from start to finish. When the action values have become good enough estimates of the optimal returns, the rat just has to select at each state the action with the largest action value in order to make optimal

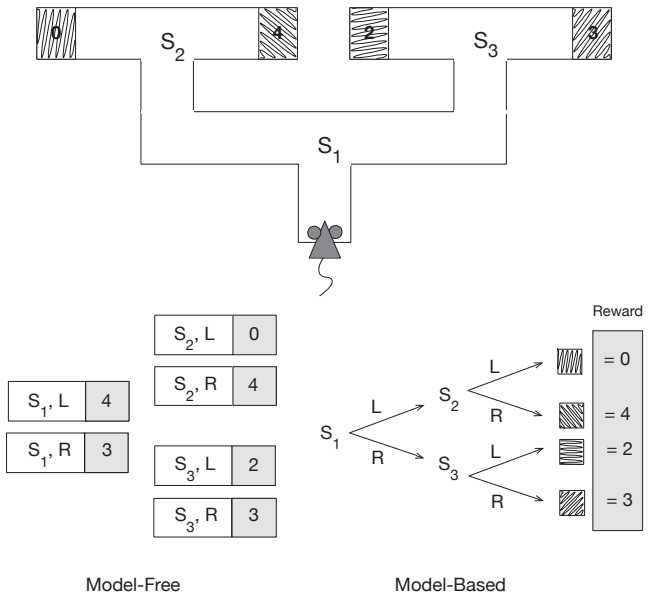


Figure 14.5: Model-based and model-free strategies to solve a hypothetical sequential action-selection problem. Top: a rat navigates a maze with distinctive goal boxes, each associated with a reward having the value shown. Lower left: a model-free strategy relies on stored action values for all the state–action pairs obtained over many learning trials. To make decisions the rat just has to select at each state the action with the largest action value for that state. Lower right: in a model-based strategy, the rat learns an environment model, consisting of knowledge of state–action-next-state transitions and a reward model consisting of knowledge of the reward associated with each distinctive goal box. The rat can decide which way to turn at each state by using the model to simulate sequences of action choices to find a path yielding the highest return. Adapted from *Trends in Cognitive Science*, volume 10, number 8, Y. Niv, D. Joel, and P. Dayan, A Normative Perspective on Motivation, p. 376, 2006, with permission from Elsevier.

decisions. In this case, when the action-value estimates become accurate enough, the rat selects L from S_1 and R from S_2 to obtain the maximum return of 4. A different model-free strategy might simply rely on a cached policy instead of action values, making direct links from S_1 to L and from S_2 to R. In neither of these strategies do decisions rely on an environment model. There is no need to consult a state-transition model, and no connection is required between the features of the goal boxes and the rewards they deliver.

Figure 14.5 (lower right) illustrates a model-based strategy. It uses an environment model consisting of a state-transition model and a reward model. The state-transition model is shown as a decision tree, and the reward model associates the distinctive features of the goal boxes with the rewards to be found in each. (The rewards associated with states S_1, S_2, and S_3 are also part of the reward model, but here they are zero and are not shown.) A model-based agent can decide which way to turn at each state by using the model to simulate sequences of action choices to find a path yielding the highest return. In this case the return is the reward obtained from the outcome at the end of the path. Here, with a sufficiently accurate model, the rat would select L and then R to obtain reward of 4. Comparing the predicted returns of simulated paths is a simple form of planning, which can be done in a variety of ways as discussed in Chapter 8.

When the environment of a model-free agent changes the way it reacts to the agent's actions, the agent has to acquire new experience in the changed environment during which it can update its policy and/or value function. In the model-free strategy shown in Figure 14.5 (lower left), for example, if one of the goal boxes were to somehow shift to delivering a different reward, the rat would have to traverse the maze, possibly many times, to experience the new reward upon reaching that goal box, all the while updating either its policy or its action-value function (or both) based on this experience. The key point is that for a model-free agent to change the action its policy specifies for a state, or to change an action value associated with a state, it has to move to that state, act from it, possibly many times, and experience the consequences of its actions.

A model-based agent can accommodate changes in its environment without this kind of 'personal experience' with the states and actions affected by the change. A change in its model automatically (through planning) changes its policy. Planning can determine the consequences of changes in the environment that have never been linked together in the agent's own experience. For example, again referring to the maze task of Figure 14.5, imagine that a rat with a previously learned transition and reward model is placed directly in the goal box to the right of S_2 to find that the reward available there now has value 1 instead of 4. The rat's reward model will change even though the action choices required to find that goal box in the maze were not involved. The planning process will bring knowledge of the new reward to bear on maze running without the need for additional experience in the maze; in this case changing the policy to right turns at both S_1 and S_3 to obtain a return of 3.

Exactly this logic is the basis of *outcome-devaluation experiments* with animals. Results from these experiments provide insight into whether an animal has learned a habit or if its behavior is under goal-directed control. Outcome-devaluation experiments are like latent-learning experiments in that the reward changes from one stage to the next. After

an initial rewarded stage of learning, the reward value of an outcome is changed, including being shifted to zero or even to a negative value.

An early important experiment of this type was conducted by Adams and Dickinson (1981). They trained rats via instrumental conditioning until the rats energetically pressed a lever for sucrose pellets in a training chamber. The rats were then placed in the same chamber with the lever retracted and allowed non-contingent food, meaning that pellets were made available to them independently of their actions. After 15-minutes of this free-access to the pellets, rats in one group were injected with the nausea-inducing poison lithium chloride. This was repeated for three sessions, in the last of which none of the injected rats consumed any of the non-contingent pellets, indicating that the reward value of the pellets had been decreased—the pellets had been devalued. In the next stage taking place a day later, the rats were again placed in the chamber and given a session of extinction training, meaning that the response lever was back in place but disconnected from the pellet dispenser so that pressing it did not release pellets. The question was whether the rats that had the reward value of the pellets decreased would lever-press less than rats that did not have the reward value of the pellets decreased, even without experiencing the devalued reward as a result of lever-pressing. It turned out that the injected rats had significantly lower response rates than the non-injected rats *right from the start of the extinction trials*.

Adams and Dickinson concluded that the injected rats associated lever pressing with consequent nausea by means of a cognitive map linking lever pressing to pellets, and pellets to nausea. Hence, in the extinction trials, the rats "knew" that the consequences of pressing the lever would be something they did not want, and so they reduced their lever-pressing right from the start. The important point is that they reduced lever-pressing without ever having experienced lever-pressing directly followed by being sick: no lever was present when they were made sick. They seemed able to combine knowledge of the outcome of a behavioral choice (pressing the lever will be followed by getting a pellet) with the reward value of the outcome (pellets are to be avoided) and hence could alter their behavior accordingly. Not every psychologist agrees with this "cognitive" account of this kind of experiment, and it is not the only possible way to explain these results, but the model-based planning explanation is widely accepted.

Nothing prevents an agent from using both model-free and model-based algorithms, and there are good reasons for using both. We know from our own experience that with enough repetition, goal-directed behavior tends to turn into habitual behavior. Experiments show that this happens for rats too. Adams (1982) conducted an experiment to see if extended training would convert goal-directed behavior into habitual behavior. He did this by comparing the effect of outcome devaluation on rats that experienced different amounts of training. If extended training made the rats less sensitive to devaluation compared to rats that received less training, this would be evidence that extended training made the behavior more habitual. Adams' experiment closely followed the Adams and Dickinson (1981) experiment just described. Simplifying a bit, rats in one group were trained until they made 100 rewarded lever-presses, and rats in the other group—the overtrained group—were trained until they made 500 rewarded lever-presses. After this training, the reward value of the pellets was decreased (using lithium chloride injections) for rats

in both groups. Then both groups of rats were given a session of extinction training. Adams' question was whether devaluation would effect the rate of lever-pressing for the overtrained rats less than it would for the non-overtrained rats, which would be evidence that extended training reduces sensitivity to outcome devaluation. It turned out that devaluation strongly decreased the lever-pressing rate of the non-overtrained rats. For the overtrained rats, in contrast, devaluation had little effect on their lever-pressing; in fact, if anything, it made it more vigorous. (The full experiment included control groups showing that the different amounts of training did not by themselves significantly effect lever-pressing rates after learning.) This result suggested that while the non-overtrained rats were acting in a goal-directed manner sensitive to their knowledge of the outcome of their actions, the overtrained rats had developed a lever-pressing habit.

Viewing this and other results like it from a computational perspective provides insight as to why one might expect animals to behave habitually in some circumstances, in a goal-directed way in others, and why they shift from one mode of control to another as they continue to learn. While animals undoubtedly use algorithms that do not exactly match those we have presented in this book, one can gain insight into animal behavior by considering the tradeoffs that various reinforcement learning algorithms imply. An idea developed by computational neuroscientists Daw, Niv, and Dayan (2005) is that animals use both model-free and model-based processes. Each process proposes an action, and the action chosen for execution is the one proposed by the process judged to be the more trustworthy of the two as determined by measures of confidence that are maintained throughout learning. Early in learning the planning process of a model-based system is more trustworthy because it chains together short-term predictions which can become accurate with less experience than long-term predictions of the model-free process. But with continued experience, the model-free process becomes more trustworthy because planning is prone to making mistakes due to model inaccuracies and short-cuts necessary to make planning feasible, such as various forms of "tree-pruning": the removal of unpromising search tree branches. According to this idea one would expect a shift from goal-directed behavior to habitual behavior as more experience accumulates. Other ideas have been proposed for how animals arbitrate between goal-directed and habitual control, and both behavioral and neuroscience research continues to examine this and related questions.

The distinction between model-free and model-based algorithms is proving to be useful for this research. One can examine the computational implications of these types of algorithms in abstract settings that expose basic advantages and limitations of each type. This serves both to suggest and to sharpen questions that guide the design of experiments necessary for increasing psychologists' understanding of habitual and goal-directed behavioral control.

14.7 Summary

Our goal in this chapter has been to discuss correspondences between reinforcement learning and the experimental study of animal learning in psychology. We emphasized at the outset that reinforcement learning as described in this book is not intended

to model details of animal behavior. It is an abstract computational framework that explores idealized situations from the perspective of artificial intelligence and engineering. But many of the basic reinforcement learning algorithms were inspired by psychological theories, and in some cases, these algorithms have contributed to the development of new animal learning models. This chapter described the most conspicuous of these correspondences.

The distinction in reinforcement learning between algorithms for prediction and algorithms for control parallels animal learning theory's distinction between classical, or Pavlovian, conditioning and instrumental conditioning. The key difference between instrumental and classical conditioning experiments is that in the former the reinforcing stimulus is contingent upon the animal's behavior, whereas in the latter it is not. Learning to predict via a TD algorithm corresponds to classical conditioning, and we described the *TD model of classical conditioning* as one instance in which reinforcement learning principles account for some details of animal learning behavior. This model generalizes the influential Rescorla–Wagner model by including the temporal dimension where events within individual trials influence learning, and it provides an account of second-order conditioning, where predictors of reinforcing stimuli become reinforcing themselves. It also is the basis of an influential view of the activity of dopamine neurons in the brain, something we take up in Chapter 15.

Learning by trial and error is at the base of the control aspect of reinforcement learning. We presented some details about Thorndike's experiments with cats and other animals that led to his *Law of Effect*, which we discussed here and in Chapter 1 (page 15). We pointed out that in reinforcement learning, exploration does not have to be limited to "blind groping"; trials can be generated by sophisticated methods using innate and previously learned knowledge as long as there is *some* exploration. We discussed the training method B. F. Skinner called *shaping* in which reward contingencies are progressively altered to train an animal to successively approximate a desired behavior. Shaping is not only indispensable for animal training, it is also an effective tool for training reinforcement learning agents. There is also a connection to the idea of an animal's motivational state, which influences what an animal will approach or avoid and what events are rewarding or punishing for the animal.

The reinforcement learning algorithms presented in this book include two basic mechanisms for addressing the problem of delayed reinforcement: eligibility traces and value functions learned via TD algorithms. Both mechanisms have antecedents in theories of animal learning. Eligibility traces are similar to stimulus traces of early theories, and value functions correspond to the role of secondary reinforcement in providing nearly immediate evaluative feedback.

The next correspondence the chapter addressed is that between reinforcement learning's environment models and what psychologists call *cognitive maps*. Experiments conducted in the mid 20th century purported to demonstrate the ability of animals to learn cognitive maps as alternatives to, or as additions to, state–action associations, and later use them to guide behavior, especially when the environment changes unexpectedly. Environment models in reinforcement learning are like cognitive maps in that they can be learned by supervised learning methods without relying on reward signals, and then they can be used later to plan behavior.

Reinforcement learning's distinction between *model-free* and *model-based* algorithms corresponds to the distinction in psychology between *habitual* and *goal-directed* behavior. Model-free algorithms make decisions by accessing information that has been stored in a policy or an action-value function, whereas model-based methods select actions as the result of planning ahead using a model of the agent's environment. Outcome-devaluation experiments provide information about whether an animal's behavior is habitual or under goal-directed control. Reinforcement learning theory has helped clarify thinking about these issues.

Animal learning clearly informs reinforcement learning, but as a type of machine learning, reinforcement learning is directed toward designing and understanding effective learning algorithms, not toward replicating or explaining details of animal behavior. We focused on aspects of animal learning that relate in clear ways to methods for solving prediction and control problems, highlighting the fruitful two-way flow of ideas between reinforcement learning and psychology without venturing deeply into many of the behavioral details and controversies that have occupied the attention of animal learning researchers. Future development of reinforcement learning theory and algorithms will likely exploit links to many other features of animal learning as the computational utility of these features becomes better appreciated. We expect that a flow of ideas between reinforcement learning and psychology will continue to bear fruit for both disciplines.

Many connections between reinforcement learning and areas of psychology and other behavioral sciences are beyond the scope of this chapter. We largely omit discussing links to the psychology of decision making, which focuses on how actions are selected, or how decisions are made, *after* learning has taken place. We also do not discuss links to ecological and evolutionary aspects of behavior studied by ethologists and behavioral ecologists: how animals relate to one another and to their physical surroundings, and how their behavior contributes to evolutionary fitness. Optimization, MDPs, and dynamic programming figure prominently in these fields, and our emphasis on agent interaction with dynamic environments connects to the study of agent behavior in complex "ecologies." Multi-agent reinforcement learning, omitted in this book, has connections to social aspects of behavior. Despite the lack of treatment here, reinforcement learning should by no means be interpreted as dismissing evolutionary perspectives. Nothing about reinforcement learning implies a *tabula rasa* view of learning and behavior. Indeed, experience with engineering applications has highlighted the importance of building into reinforcement learning systems knowledge that is analogous to what evolution provides to animals.

Bibliographical and Historical Remarks

Ludvig, Bellemare, and Pearson (2011) and Shah (2012) review reinforcement learning in the contexts of psychology and neuroscience. These publications are useful companions to this chapter and the following chapter on reinforcement learning and neuroscience.

14.1 Dayan, Niv, Seymour, and Daw (2006) focused on interactions between classical and instrumental conditioning, particularly situations where classically-conditioned and instrumental responses are in conflict. They proposed a Q-learning framework for modeling aspects of this interaction. Modayil and Sutton (2014) used a mobile robot to demonstrate the effectiveness of a control method combining a fixed response with online prediction learning. Calling this *Pavlovian control*, they emphasized that it differs from the usual control methods of reinforcement learning, being based on predictively executing fixed responses and not on reward maximization. The electro-mechanical machine of Ross (1933) and especially the learning version of Walter's turtle (Walter, 1951) were very early illustrations of Pavlovian control.

14.2.1 Kamin (1968) first reported blocking, now commonly known as Kamin blocking, in classical conditioning. Moore and Schmajuk (2008) provide an excellent summary of the blocking phenomenon, the research it stimulated, and its lasting influence on animal learning theory. Gibbs, Cool, Land, Kehoe, and Gormezano (1991) describe second-order conditioning of the rabbit's nictitating membrane response and its relationship to conditioning with serial-compound stimuli. Finch and Culler (1934) reported obtaining fifth-order conditioning of a dog's foreleg withdrawal "when the *motivation* of the animal is maintained through the various orders."

14.2.2 The idea built into the Rescorla–Wagner model that learning occurs when animals are surprised is derived from Kamin (1969). Models of classical conditioning other than Rescorla and Wagner's include the models of Klopf (1988), Grossberg (1975), Mackintosh (1975), Moore and Stickney (1980), Pearce and Hall (1980), and Courville, Daw, and Touretzky (2006). Schmajuk (2008) review models of classical conditioning. Wagner (2008) provides a modern psychological perspective on the Rescorla-Wagner model and similar elemental theories of learning.

14.2.3 An early version of the TD model of classical conditioning appeared in Sutton and Barto (1981a), which also included the early model's prediction that temporal primacy overrides blocking, later shown by Kehoe, Schreurs, and Graham (1987) to occur in the rabbit nictitating membrane preparation. Sutton and Barto (1981a) contains the earliest recognition of the near identity between the Rescorla–Wagner model and the Least-Mean-Square (LMS), or Widrow-Hoff, learning rule (Widrow and Hoff, 1960). This early model was revised following Sutton's development of the TD algorithm (Sutton, 1984, 1988) and was first presented as the TD model in Sutton and Barto (1987) and more completely in Sutton and Barto (1990), upon which this section is largely based. Additional exploration

of the TD model and its possible neural implementation was conducted by Moore and colleagues (Moore, Desmond, Berthier, Blazis, Sutton, and Barto, 1986; Moore and Blazis, 1989; Moore, Choi, and Brunzell, 1998; Moore, Marks, Castagna, and Polewan, 2001). Klopf's (1988) drive-reinforcement theory of classical conditioning extends the TD model to address additional experimental details, such as the S-shape of acquisition curves. In some of these publications TD is taken to mean Time Derivative instead of Temporal Difference.

14.2.4 Ludvig, Sutton, and Kehoe (2012) evaluated the performance of the TD model in previously unexplored tasks involving classical conditioning and examined the influence of various stimulus representations, including the microstimulus representation that they introduced earlier (Ludvig, Sutton, and Kehoe, 2008). Earlier investigations of the influence of various stimulus representations and their possible neural implementations on response timing and topography in the context of the TD model are those of Moore and colleagues cited above. Although not in the context of the TD model, representations like the microstimulus representation of Ludvig et al. (2012) have been proposed and studied by Grossberg and Schmajuk (1989), Brown, Bullock, and Grossberg (1999), Buhusi and Schmajuk (1999), and Machado (1997). The figures on pages 353–355 are adapted from Sutton and Barto (1990).

14.3 Section 1.7 includes comments on the history of trial-and-error learning and the Law of Effect. The idea that Thorndike's cats might have been exploring according to an instinctual context-specific ordering over actions rather than by just selecting from a set of instinctual impulses was suggested by Peter Dayan (personal communication). Selfridge, Sutton, and Barto (1985) illustrated the effectiveness of shaping in a pole-balancing reinforcement learning task. Other examples of shaping in reinforcement learning are Gullapalli and Barto (1992), Mahadevan and Connell (1992), Mataric (1994), Dorigo and Colombette (1994), Saksida, Raymond, and Touretzky (1997), and Randløv and Alstrøm (1998). Ng (2003) and Ng, Harada, and Russell (1999) used the term shaping in a sense somewhat different from Skinner's, focussing on the problem of how to alter the reward signal without altering the set of optimal policies.

Dickinson and Balleine (2002) discuss the complexity of the interaction between learning and motivation. Wise (2004) provides an overview of reinforcement learning and its relation to motivation. Daw and Shohamy (2008) link motivation and learning to aspects of reinforcement learning theory. See also McClure, Daw, and Montague (2003), Niv, Joel, and Dayan (2006), Rangel, Camerer, and Montague (2008), and Dayan and Berridge (2014). McClure et al. (2003), Niv, Daw, and Dayan (2006), and Niv, Daw, Joel, and Dayan (2007) present theories of behavioral vigor related to the reinforcement learning framework.

14.4 Spence, Hull's student and collaborator at Yale, elaborated the role of higher-order reinforcement in addressing the problem of delayed reinforcement (Spence, 1947). Learning over very long delays, as in taste-aversion conditioning with

delays up to several hours, led to interference theories as alternatives to decaying-trace theories (e.g., Revusky and Garcia, 1970; Boakes and Costa, 2014). Other views of learning under delayed reinforcement invoke roles for awareness and working memory (e.g., Clark and Squire, 1998; Seo, Barraclough, and Lee, 2007).

14.5 Thistlethwaite (1951) provides an extensive review of latent learning experiments up to the time of its publication. Ljung (1998) provides an overview of model learning, or system identification, techniques in engineering. Gopnik, Glymour, Sobel, Schulz, Kushnir, and Danks (2004) present a Bayesian theory about how children learn models.

14.6 Connections between habitual and goal-directed behavior and model-free and model-based reinforcement learning were first proposed by Daw, Niv, and Dayan (2005). The hypothetical maze task used to explain habitual and goal-directed behavioral control is based on the explanation of Niv, Joel, and Dayan (2006). Dolan and Dayan (2013) review four generations of experimental research related to this issue and discuss how it can move forward on the basis of reinforcement learning's model-free/model-based distinction. Dickinson (1980, 1985) and Dickinson and Balleine (2002) discuss experimental evidence related to this distinction. Donahoe and Burgos (2000) alternatively argue that model-free processes can account for the results of outcome-devaluation experiments. Dayan and Berridge (2014) argue that classical conditioning involves model-based processes. Rangel, Camerer, and Montague (2008) review many of the outstanding issues involving habitual, goal-directed, and Pavlovian modes of control.

Comments on Terminology— The traditional meaning of *reinforcement* in psychology is the strengthening of a pattern of behavior (by increasing either its intensity or frequency) as a result of an animal receiving a stimulus (or experiencing the omission of a stimulus) in an appropriate temporal relationship with another stimulus or with a response. Reinforcement produces changes that remain in future behavior. Sometimes in psychology reinforcement refers to the process of producing lasting changes in behavior, whether the changes strengthen or weaken a behavior pattern (Mackintosh, 1983). Letting reinforcement refer to weakening in addition to strengthening is at odds with the everyday meaning of reinforce, and its traditional use in psychology, but it is a useful extension that we have adopted here. In either case, a stimulus considered to be the cause of the behavioral change is called a *reinforcer*.

Psychologists do not generally use the specific phrase *reinforcement learning* as we do. Animal learning pioneers probably regarded reinforcement and learning as being synonymous, so it would be redundant to use both words. Our use of the phrase follows its use in computational and engineering research, influenced mostly by Minsky (1961). But the phrase is lately gaining currency in psychology and neuroscience, likely because strong parallels have surfaced between reinforcement learning algorithms and animal learning—parallels described in this chapter and the next.

According to common usage, a *reward* is an object or event that an animal will approach and work for. A reward may be given to an animal in recognition of its 'good' behavior, or given in order to make the animal's behavior 'better.' Similarly, a *penalty* is an object or event that the animal usually avoids and that is given as a consequence of 'bad' behavior, usually in order to change that behavior. *Primary reward* is reward due to machinery built into an animal's nervous system by evolution to improve its chances of survival and reproduction, for example, reward produced by the taste of nourishing food, sexual contact, successful escape, and many other stimuli and events that predicted reproductive success over the animal's ancestral history. As explained in Section 14.2.1, *higher-order reward* is reward delivered by stimuli that predict primary reward, either directly or indirectly by predicting other stimuli that predict primary reward. Reward is *secondary* if its rewarding quality is the result of directly predicting primary reward.

In this book we call R_t the 'reward signal at time t' or sometimes just the 'reward at time t,' but we do not think of it as an object or event in the agent's environment. Because R_t is a number—not an object or an event—it is more like a reward signal in neuroscience, which is a signal internal to the brain, like the activity of neurons, that influences decision making and learning. This signal might be triggered when the animal perceives an attractive (or an aversive) object, but it can also be triggered by things that do not physically exist in the animal's external environment, such as memories, ideas, or hallucinations. Because our R_t can be positive, negative, or zero, it might be better to call a negative R_t a penalty, and an R_t equal to zero a neutral signal, but for simplicity we generally avoid these terms.

In reinforcement learning, the process that generates all the R_ts defines the problem the agent is trying to solve. The agent's objective is to keep the magnitude of R_t as large as possible over time. In this respect, R_t is like primary reward for an animal if we think of the problem the animal faces as the problem of obtaining as much primary reward as possible over its lifetime (and thereby, through the prospective "wisdom" of evolution, improve its chances of solving its real problem, which is to pass its genes on to future generations). However, as we suggest in Chapter 15, it is unlikely that there is a single "master" reward signal like R_t in an animal's brain.

Not all reinforcers are rewards or penalties. Sometimes reinforcement is not the result of an animal receiving a stimulus that evaluates its behavior by labeling the behavior good or bad. A behavior pattern can be reinforced by a stimulus that arrives to an animal no matter how the animal behaved. As described in Section 14.1, whether the delivery of reinforcer depends, or does not depend, on preceding behavior is the defining difference between instrumental, or operant, conditioning experiments and classical, or Pavlovian, conditioning experiments. Reinforcement is at work in both types of experiments, but only in the former is it feedback that evaluates past behavior. (Though it has often been pointed out that even when the reinforcing US in a classical conditioning experiment is not contingent on the subject's preceding behavior, its reinforcing value can be influenced by this behavior, an example being that a closed eye makes an air puff to the eye less aversive.)

The distinction between reward signals and reinforcement signals is a crucial point when we discuss neural correlates of these signals in the next chapter. Like a reward signal, for us, the reinforcement signal at any specific time is a positive or negative number, or zero. A reinforcement signal is the major factor directing changes a learning algorithm

makes in an agent's policy, value estimates, or environment models. The definition that makes the most sense to us is that a reinforcement signal at any time is a number that multiplies (possibly along with some constants) a vector to determine parameter updates in some learning algorithm.

For some algorithms, the reward signal alone is the critical multiplier in the parameter-update equation. For these algorithms the reinforcement signal is the same as the reward signal. But for most of the algorithms we discuss in this book, reinforcement signals include terms in addition to the reward signal, an example being a TD error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, which is the reinforcement signal for TD state-value learning (and analogous TD errors for action-value learning). In this reinforcement signal, R_{t+1} is the *primary reinforcement* contribution, and the temporal difference in predicted values, $\gamma V(S_{t+1}) - V(S_t)$ (or an analogous temporal difference for action values), is the *conditioned reinforcement* contribution. Thus, whenever $\gamma V(S_{t+1}) - V(S_t) = 0$, δ_t signals 'pure' primary reinforcement; and whenever $R_{t+1} = 0$, it signals 'pure' conditioned reinforcement, but it often signals a mixture of these. Note as we mentioned in Section 6.1, this δ_t is not available until time $t + 1$. We therefore think of δ_t as the reinforcement signal at time $t + 1$, which is fitting because it reinforces predictions and/or actions made earlier at step t.

A possible source of confusion is the terminology used by the famous psychologist B. F. Skinner and his followers. For Skinner, positive reinforcement occurs when the consequences of an animal's behavior increase the frequency of that behavior; punishment occurs when the behavior's consequences decrease that behavior's frequency. Negative reinforcement occurs when behavior leads to the removal of an aversive stimulus (that is, a stimulus the animal does not like), thereby increasing the frequency of that behavior. Negative punishment, on the other hand, occurs when behavior leads to the removal of an appetitive stimulus (that is, a stimulus the animal likes), thereby decreasing the frequency of that behavior. We find no critical need for these distinctions because our approach is more abstract than this, with both reward and reinforcement signals allowed to take on both positive and negative values. (But note especially that when our reinforcement signal is negative, it is not the same as Skinner's negative reinforcement.)

On the other hand, it has often been pointed out that using a single number as a reward or a penalty signal, depending only on its sign, is at odds with the fact that animals' appetitive and aversive systems have qualitatively different properties and involve different brain mechanisms. This points to a direction in which the reinforcement learning framework might be developed in the future to exploit computational advantages of separate appetitive and aversive systems, but for now we are passing over these possibilities.

Another discrepancy in terminology is how we use the word *action*. To many cognitive scientists, an action is purposeful in the sense of being the result of an animal's knowledge about the relationship between the behavior in question and the consequences of that behavior. An action is goal-directed and the result of a decision, in contrast to a response, which is triggered by a stimulus; the result of a reflex or a habit. We use the word action without differentiating among what others call actions, decisions, and responses. These are important distinctions, but for us they are encompassed by differences between

model-free and model-based reinforcement learning algorithms, which we discussed above in relation to habitual and goal-directed behavior in Section 14.6. Dickinson (1985) discusses the distinction between responses and actions.

A term used a lot in this book is *control*. What we mean by control is entirely different from what it means to animal learning psychologists. By control we mean that an agent influences its environment to bring about states or events that the agent prefers: the agent exerts control over its environment. This is the sense of control used by control engineers. In psychology, on the other hand, control typically means that an animal's behavior is influenced by—is controlled by—the stimuli the animal receives (stimulus control) or the reinforcement schedule it experiences. Here the environment is controlling the agent. Control in this sense is the basis of behavior modification therapy. Of course, both of these directions of control are at play when an agent interacts with its environment, but our focus is on the agent as controller; not the environment as controller. A view equivalent to ours, and perhaps more illuminating, is that the agent is actually controlling the input it receives from its environment (Powers, 1973). This is *not* what psychologists mean by stimulus control.

Sometimes reinforcement learning is understood to refer solely to learning policies directly from rewards (and penalties) without the involvement of value functions or environment models. This is what psychologists call stimulus-response, or S-R, learning. But for us, along with most of today's psychologists, reinforcement learning is much broader than this, including in addition to S-R learning, methods involving value functions, environment models, planning, and other processes that are commonly thought to belong to the more cognitive side of mental functioning.

Chapter 15

Neuroscience

Neuroscience is the multidisciplinary study of nervous systems: how they regulate bodily functions; control behavior; change over time as a result of development, learning, and aging; and how cellular and molecular mechanisms make these functions possible. One of the most exciting aspects of reinforcement learning is the mounting evidence from neuroscience that the nervous systems of humans and many other animals implement algorithms that correspond in striking ways to reinforcement learning algorithms. The main objective of this chapter is to explain these parallels and what they suggest about the neural basis of reward-related learning in animals.

The most remarkable point of contact between reinforcement learning and neuroscience involves dopamine, a chemical deeply involved in reward processing in the brains of mammals. Dopamine appears to convey temporal-difference (TD) errors to brain structures where learning and decision making take place. This parallel is expressed by the *reward prediction error hypothesis of dopamine neuron activity*, a hypothesis that resulted from the convergence of computational reinforcement learning and results of neuroscience experiments. In this chapter we discuss this hypothesis, the neuroscience findings that led to it, and why it is a significant contribution to understanding brain reward systems. We also discuss parallels between reinforcement learning and neuroscience that are less striking than this dopamine/TD-error parallel but that provide useful conceptual tools for thinking about reward-based learning in animals. Other elements of reinforcement learning have the potential to impact the study of nervous systems, but their connections to neuroscience are still relatively undeveloped. We discuss several of these evolving connections that we think will grow in importance over time.

As we outlined in the history section of this book's introductory chapter (Section 1.7), many aspects of reinforcement learning were influenced by neuroscience. A second objective of this chapter is to acquaint readers with ideas about brain function that have contributed to our approach to reinforcement learning. Some elements of reinforcement learning are easier to understand when seen in light of theories of brain function. This is particularly true for the idea of the eligibility trace, one of the basic mechanisms of reinforcement learning, that originated as a conjectured property of synapses, the structures by which nerve cells—neurons—communicate with one another.

In this chapter we do not delve very deeply into the enormous complexity of the neural systems underlying reward-based learning in animals: this chapter is too short, and we are not neuroscientists. We do not try to describe—or even to name—the very many brain structures and pathways, or any of the molecular mechanisms, believed to be involved in these processes. We also do not do justice to hypotheses and models that are alternatives to those that align so well with reinforcement learning. It should not be surprising that there are differing views among experts in the field. We can only provide a glimpse into this fascinating and developing story. We hope, though, that this chapter convinces you that a very fruitful channel has emerged connecting reinforcement learning and its theoretical underpinnings to the neuroscience of reward-based learning in animals.

Many excellent publications cover links between reinforcement learning and neuroscience, some of which we cite in this chapter's final section. Our treatment differs from most of these because we assume familiarity with reinforcement learning as presented in the earlier chapters of this book, but we do not assume knowledge of neuroscience. We begin with a brief introduction to the neuroscience concepts needed for a basic understanding of what is to follow.

15.1 Neuroscience Basics

Some basic information about nervous systems is helpful for following what we cover in this chapter. Terms that we refer to later are italicized. Skipping this section will not be a problem if you already have an elementary knowledge of neuroscience.

Neurons, the main components of nervous systems, are cells specialized for processing and transmitting information using electrical and chemical signals. They come in many forms, but a neuron typically has a cell body, *dendrites*, and a single *axon*. Dendrites are structures that branch from the cell body to receive input from other neurons (or to also receive external signals in the case of sensory neurons). A neuron's axon is a fiber that carries the neuron's output to other neurons (or to muscles or glands). A neuron's output consists of sequences of electrical pulses called *action potentials* that travel along the axon. Action potentials are also called *spikes*, and a neuron is said to *fire* when it generates a spike. In models of neural networks it is common to use real numbers to represent a neuron's *firing rate*, the average number of spikes per some unit of time.

A neuron's axon can branch widely so that the neuron's action potentials reach many targets. The branching structure of a neuron's axon is called the neuron's *axonal arbor*. Because the conduction of an action potential is an active process, not unlike the burning of a fuse, when an action potential reaches an axonal branch point it "lights up" action potentials on all of the outgoing branches (although propagation to a branch can sometimes fail). As a result, the activity of a neuron with a large axonal arbor can influence many target sites.

A *synapse* is a structure generally at the termination of an axon branch that mediates the communication of one neuron to another. A synapse transmits information from the *presynaptic* neuron's axon to a dendrite or cell body of the *postsynaptic* neuron. With a few exceptions, synapses release a chemical *neurotransmitter* upon the arrival of an action potential from the presynaptic neuron. (The exceptions are cases of direct electric coupling between neurons, but these will not concern us here.) Neurotransmitter molecules released from the presynaptic side of the synapse diffuse across the *synaptic cleft*, the very small space between the presynaptic ending and the postsynaptic neuron, and then bind to receptors on the surface of the postsynaptic neuron to excite or inhibit its spike-generating activity, or to modulate its behavior in other ways. A particular neurotransmitter may bind to several different types of receptors, with each producing a different effect on the postsynaptic neuron. For example, there are at least five different receptor types by which the neurotransmitter dopamine can affect a postsynaptic neuron. Many different chemicals have been identified as neurotransmitters in animal nervous systems.

A neuron's *background* activity is its level of activity, usually its firing rate, when the neuron does not appear to be driven by synaptic input related to the task of interest to the experimenter, for example, when the neuron's activity is not correlated with a stimulus delivered to a subject as part of an experiment. Background activity can be irregular due to input from the wider network, or due to noise within the neuron or its synapses. Sometimes background activity is the result of dynamic processes intrinsic to the neuron. A neuron's *phasic* activity, in contrast to its background activity, consists of bursts of spiking activity usually caused by synaptic input. Activity that varies slowly and often in a graded manner, whether as background activity or not, is called a neuron's *tonic* activity.

The strength or effectiveness by which the neurotransmitter released at a synapse influences the postsynaptic neuron is the synapse's *efficacy*. One way a nervous system can change through experience is through changes in synaptic efficacies as a result of combinations of the activities of the presynaptic and postsynaptic neurons, and sometimes by the presence of a *neuromodulator*, which is a neurotransmitter having effects other than, or in addition to, direct fast excitation or inhibition.

Brains contain several different neuromodulation systems consisting of clusters of neurons with widely branching axonal arbors, with each system using a different neurotransmitter. Neuromodulation can alter the function of neural circuits, mediate motivation, arousal, attention, memory, mood, emotion, sleep, and body temperature. Important here is that a neuromodulatory system can distribute something like a scalar signal, such as a reinforcement signal, to alter the operation of synapses in widely distributed sites critical for learning.

The ability of synaptic efficacies to change is called *synaptic plasticity*. It is one of the primary mechanisms responsible for learning. The parameters, or weights, adjusted by learning algorithms correspond to synaptic efficacies. As we detail below, modulation of synaptic plasticity via the neuromodulator dopamine is a plausible mechanism for how the brain might implement learning algorithms like many of those described in this book.

15.2 Reward Signals, Reinforcement Signals, Values, and Prediction Errors

Links between neuroscience and computational reinforcement learning begin as parallels between signals in the brain and signals playing prominent roles in reinforcement learning theory and algorithms. In Chapter 3 we said that any problem of learning goal-directed behavior can be reduced to the three signals representing actions, states, and rewards. However, to explain links that have been made between neuroscience and reinforcement learning, we have to be less abstract than this and consider other reinforcement learning signals that correspond, in certain ways, to signals in the brain. In addition to reward signals, these include reinforcement signals (which we argue are different from reward signals), value signals, and signals conveying prediction errors. When we label a signal by its function in this way, we are doing it in the context of reinforcement learning theory in which the signal corresponds to a term in an equation or an algorithm. On the other hand, when we refer to a signal in the brain, we mean a physiological event such as a burst of action potentials or the secretion of a neurotransmitter. Labeling a neural signal by its function, for example calling the phasic activity of a dopamine neuron a reinforcement signal, means that the neural signal behaves like, and is conjectured to function like, the corresponding theoretical signal.

Uncovering evidence for these correspondences involves many challenges. Neural activity related to reward processing can be found in nearly every part of the brain, and it is difficult to interpret results unambiguously because representations of different reward-related signals tend to be highly correlated with one another. Experiments need to be carefully designed to allow one type of reward-related signal to be distinguished with any degree of certainty from others—or from an abundance of other signals not related to reward processing. Despite these difficulties, many experiments have been conducted with the aim of reconciling aspects of reinforcement learning theory and algorithms with neural signals, and some compelling links have been established. To prepare for examining these links, in the rest of this section we remind the reader of what various reward-related signals mean according to reinforcement learning theory.

In our Comments on Terminology at the end of the previous chapter, we said that R_t is like a reward signal in an animal's brain and not an object or event in the animal's environment. In reinforcement learning, the reward signal (along with an agent's environment) defines the problem a reinforcement learning agent is trying to solve. In this respect, R_t is like a signal in an animal's brain that distributes primary reward to sites throughout the brain. But it is unlikely that a unitary master reward signal like R_t exists in an animal's brain. It is best to think of R_t as an abstraction summarizing the overall effect of a multitude of neural signals generated by many systems in the brain that assess the rewarding or punishing qualities of sensations and states.

Reinforcement signals in reinforcement learning are different from reward signals. The function of a reinforcement signal is to direct the changes a learning algorithm makes in an agent's policy, value estimates, or environment models. For a TD method, for instance, the reinforcement signal at time t is the TD error $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$.[1] The

[1] As we mentioned in Section 6.1, δ_t in our notation is defined to be $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, so δ_t

reinforcement signal for some algorithms could be just the reward signal, but for most of the algorithms we consider the reinforcement signal is the reward signal adjusted by other information, such as the value estimates in TD errors.

Estimates of state values or of action values, that is, V or Q, specify what is good or bad for the agent over the long run. They are predictions of the total reward an agent can expect to accumulate over the future. Agents make good decisions by selecting actions leading to states with the largest estimated state values, or by selecting actions with the largest estimated action values.

Prediction errors measure discrepancies between expected and actual signals or sensations. Reward prediction errors (RPEs) specifically measure discrepancies between the expected and the received reward signal, being positive when the reward signal is greater than expected, and negative otherwise. TD errors like (6.5) are special kinds of RPEs that signal discrepancies between current and earlier expectations of reward over the long-term. When neuroscientists refer to RPEs they generally (though not always) mean TD RPEs, which we simply call TD errors throughout this chapter. Also in this chapter, a TD error is generally one that does not depend on actions, as opposed to TD errors used in learning action-values by algorithms like Sarsa and Q-learning. This is because the most well-known links to neuroscience are stated in terms of action-free TD errors, but we do not mean to rule out possible similar links involving action-dependent TD errors. (TD errors for predicting signals other than rewards are useful too, but that case will not concern us here. See, for example, Modayil, White, and Sutton, 2014.)

One can ask many questions about links between neuroscience data and these theoretically defined signals. Is an observed signal more like a reward signal, a value signal, a prediction error, a reinforcement signal, or something altogether different? And if it is an error signal, is it an RPE, a TD error, or a simpler error like the Rescorla–Wagner error (14.3)? And if it is a TD error, does it depend on actions like the TD error of Q-learning or Sarsa? As indicated above, probing the brain to answer questions like these is extremely difficult. But experimental evidence suggests that one neurotransmitter, specifically the neurotransmitter dopamine, signals RPEs, and further, that the phasic activity of dopamine-producing neurons in fact conveys TD errors (see Section 15.1 for a definition of phasic activity). This evidence led to the *reward prediction error hypothesis of dopamine neuron activity*, which we describe next.

15.3 The Reward Prediction Error Hypothesis

The *reward prediction error hypothesis of dopamine neuron activity* proposes that one of the functions of the phasic activity of dopamine-producing neurons in mammals is to deliver an error between an old and a new estimate of expected future reward to target areas throughout the brain. This hypothesis (though not in these exact words) was first explicitly stated by Montague, Dayan, and Sejnowski (1996), who showed how the TD error concept from reinforcement learning accounts for many features of the phasic

is not available until time $t+1$. The TD error *available* at t is actually $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$. Because we are thinking of time steps as very small, or even infinitesimal, time intervals, one should not attribute undue importance to this one-step time shift.

activity of dopamine neurons in mammals. The experiments that led to this hypothesis were performed in the 1980s and early 1990s in the laboratory of neuroscientist Wolfram Schultz. Section 15.4 describes these influential experiments, Section 15.6 explains how the results of these experiments align with TD errors, and the Bibliographical and Historical Remarks section at the end of this chapter includes a guide to the literature surrounding the development of this influential hypothesis.

Montague et al. (1996) compared the TD errors of the TD model of classical conditioning with the phasic activity of dopamine-producing neurons during classical conditioning experiments. Recall from Section 14.2 that the TD model of classical conditioning is basically the semi-gradient-descent TD(λ) algorithm with linear function approximation. Montague et al. made several assumptions to set up this comparison. First, because a TD error can be negative but neurons cannot have a negative firing rate, they assumed that the quantity corresponding to dopamine neuron activity is $\delta_{t-1} + b_t$, where b_t is the background firing rate of the neuron. A negative TD error corresponds to a drop in a dopamine neuron's firing rate below its background rate.[2]

A second assumption was needed about the states visited in each classical conditioning trial and how they are represented as inputs to the learning algorithm. This is the same issue we discussed in Section 14.2.4 for the TD model. Montague et al. chose a complete serial compound (CSC) representation as shown in the left column of Figure 14.1, but where the sequence of short-duration internal signals continues until the onset of the US, which here is the arrival of a non-zero reward signal. This representation allows the TD error to mimic the fact that dopamine neuron activity not only predicts a future reward, but that it is also sensitive to *when* after a predictive cue that reward is expected to arrive. There has to be some way to keep track of the time between sensory cues and the arrival of reward. If a stimulus initiates a sequence of internal signals that continues after the stimulus ends, and if there is a different signal for each time step following the stimulus, then each time step after the stimulus is represented by a distinct state. Thus, the TD error, being state-dependent, can be sensitive to the timing of events within a trial.

In simulated trials with these assumptions about background firing rate and input representation, TD errors of the TD model are remarkably similar to dopamine neuron phasic activity. Previewing our description of details about these similarities in Section 15.4 below, the TD errors parallel the following features of dopamine neuron activity: 1) the phasic response of a dopamine neuron only occurs when a rewarding event is unpredicted; 2) early in learning, neutral cues that precede a reward do not cause substantial phasic dopamine responses, but with continued learning these cues gain predictive value and come to elicit phasic dopamine responses; 3) if an even earlier cue reliably precedes a cue that has already acquired predictive value, the phasic dopamine response shifts to the earlier cue, ceasing for the later cue; and 4) if after learning, the predicted rewarding event is omitted, a dopamine neuron's response decreases below its baseline level shortly after the expected time of the rewarding event.

Although not every dopamine neuron monitored in the experiments of Schultz and

[2] In the literature relating TD errors to the activity of dopamine neurons, their δ_t is the same as our $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$.

colleagues behaved in all of these ways, the striking correspondence between the activities of most of the monitored neurons and TD errors lends strong support to the reward prediction error hypothesis. There are situations, however, in which predictions based on the hypothesis do not match what is observed in experiments. The choice of input representation is critical to how closely TD errors match some of the details of dopamine neuron activity, particularly details about the timing of dopamine neuron responses. Different ideas, some of which we discuss below, have been proposed about input representations and other features of TD learning to make the TD errors fit the data better, though the main parallels appear with the CSC representation that Montague et al. used. Overall, the reward prediction error hypothesis has received wide acceptance among neuroscientists studying reward-based learning, and it has proven to be remarkably resilient in the face of accumulating results from neuroscience experiments.

To prepare for our description of the neuroscience experiments supporting the reward prediction error hypothesis, and to provide some context so that the significance of the hypothesis can be appreciated, we next present some of what is known about dopamine, the brain structures it influences, and how it is involved in reward-based learning.

15.4 Dopamine

Dopamine is produced as a neurotransmitter by neurons whose cell bodies lie mainly in two clusters of neurons in the midbrain of mammals: the substantia nigra pars compacta (SNpc) and the ventral tegmental area (VTA). Dopamine plays essential roles in many processes in the mammalian brain. Prominent among these are motivation, learning, action-selection, most forms of addiction, and the disorders schizophrenia and Parkinson's disease. Dopamine is called a neuromodulator because it performs many functions other than direct fast excitation or inhibition of targeted neurons. Although much remains unknown about dopamine's functions and details of its cellular effects, it is clear that it is fundamental to reward processing in the mammalian brain. Dopamine is not the only neuromodulator involved in reward processing, and its role in aversive situations—punishment—remains controversial. Dopamine also can function differently in non-mammals. But no one doubts that dopamine is essential for reward-related processes in mammals, including humans.

An early, traditional view is that dopamine neurons broadcast a reward signal to multiple brain regions implicated in learning and motivation. This view followed from a famous 1954 paper by James Olds and Peter Milner that described the effects of electrical stimulation on certain areas of a rat's brain. They found that electrical stimulation to particular regions acted as a very powerful reward in controlling the rat's behavior: "...the control exercised over the animal's behavior by means of this reward is extreme, possibly exceeding that exercised by any other reward previously used in animal experimentation" (Olds and Milner, 1954). Later research revealed that the sites at which stimulation was most effective in producing this rewarding effect excited dopamine pathways, either directly or indirectly, that ordinarily are excited by natural rewarding stimuli. Effects similar to these were also observed with human subjects. These observations strongly suggested that dopamine neuron activity signals reward.

But if the reward prediction error hypothesis is correct—even if it accounts for only some features of a dopamine neuron's activity—this traditional view of dopamine neuron activity is not entirely correct: phasic responses of dopamine neurons signal reward prediction errors, not reward itself. In reinforcement learning's terms, a dopamine neuron's phasic response at a time t corresponds to $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$, not to R_t.

Reinforcement learning theory and algorithms help reconcile the reward-prediction-error view with the conventional notion that dopamine signals reward. In many of the algorithms we discuss in this book, δ functions as a reinforcement signal, meaning that it is the main driver of learning. For example, δ is the critical factor in the TD model of classical conditioning, and δ is the reinforcement signal for learning both a value function and a policy in an actor–critic architecture (Sections 13.5 and 15.7). Action-dependent forms of δ are reinforcement signals for Q-learning and Sarsa. The reward signal R_t is a crucial component of δ_{t-1}, but it is not the complete determinant of its reinforcing effect in these algorithms. The additional term $\gamma V(S_t) - V(S_{t-1})$ is the higher-order reinforcement part of δ_{t-1}, and even if reward occurs ($R_t \neq 0$), the TD error can be silent if the reward is fully predicted (which is fully explained in Section 15.6 below).

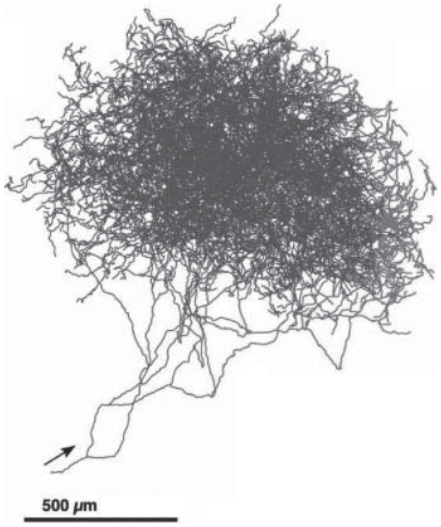
A closer look at Olds' and Milner's 1954 paper, in fact, reveals that it is mainly about the reinforcing effect of electrical stimulation in an instrumental conditioning task. Electrical stimulation not only energized the rats' behavior—through dopamine's effect on motivation—it also led to the rats quickly learning to stimulate themselves by pressing a lever, which they would do frequently for long periods of time. The activity of dopamine neurons triggered by electrical stimulation reinforced the rats' lever pressing.

More recent experiments using optogenetic methods clinch the role of phasic responses of dopamine neurons as reinforcement signals. These methods allow neuroscientists to precisely control the activity of selected neuron types at a millisecond timescale in awake behaving animals. Optogenetic methods introduce light-sensitive proteins into selected neuron types so that these neurons can be activated or silenced by means of flashes of laser light. The first experiment using optogenetic methods to study dopamine neurons showed that optogenetic stimulation producing phasic activation of dopamine neurons in mice was enough to condition the mice to prefer the side of a chamber where they received this stimulation as compared to the chamber's other side where they received no, or lower-frequency, stimulation (Tsai et al. 2009). In another example, Steinberg et al. (2013) used optogenetic activation of dopamine neurons to create artificial bursts of dopamine neuron activity in rats at the times when rewarding stimuli were expected but omitted—times when dopamine neuron activity normally pauses. With these pauses replaced by artificial bursts, responding was sustained when it would ordinarily decrease due to lack of reinforcement (in extinction trials), and learning was enabled when it would ordinarily be blocked due to the reward being already predicted (the blocking paradigm; Section 14.2.1).

Additional evidence for the reinforcing function of dopamine comes from optogenetic experiments with fruit flies, except in these animals dopamine's effect is the opposite of its effect in mammals: optically triggered bursts of dopamine neuron activity act just like electric foot shock in reinforcing avoidance behavior, at least for the population

of dopamine neurons activated (Claridge-Chang et al. 2009). Although none of these optogenetic experiments showed that phasic dopamine neuron activity is specifically like a TD error, they convincingly demonstrated that phasic dopamine neuron activity acts just like δ acts (or perhaps like *minus* δ acts in fruit flies) as the reinforcement signal in algorithms for both prediction (classical conditioning) and control (instrumental conditioning).

Dopamine neurons are particularly well suited to broadcasting a reinforcement signal to many areas of the brain. These neurons have huge axonal arbors, each releasing dopamine at 100 to 1,000 times more synaptic sites than reached by the axons of typical neurons. Shown to the right is the axonal arbor of a single dopamine neuron whose cell body is in the SNpc of a rat's brain. Each axon of a SNpc or VTA dopamine neuron makes roughly 500,000 synaptic contacts on the dendrites of neurons in targeted brain areas.



Axonal arbor of a single neuron producing dopamine as a neurotransmitter. These axons make synaptic contacts with a huge number of dendrites of neurons in targeted brain areas.

Adapted from *The Journal of Neuroscience*, Matsuda, Furuta, Nakamura, Hioki, Fujiyama, Arai, and Kaneko, volume 29, 2009, page 451.

If dopamine neurons broadcast a reinforcement signal like reinforcement learning's δ, then because this is a scalar signal, i.e., a single number, all dopamine neurons in both the SNpc and VTA would be expected to activate more-or-less identically so that they would act in near synchrony to send the same signal to all of the sites their axons target. Although it has been a common belief that dopamine neurons do act together like this, modern evidence is pointing to the more complicated picture that different subpopulations of dopamine neurons respond to input differently depending on the structures to which they send their signals and the different ways these signals act on their target structures. Dopamine has functions other than signaling RPEs, and even for dopamine neurons that do signal RPEs, it can make sense to send different RPEs to different structures depending on the roles these structures play in producing reinforced behavior. This is beyond what we treat in any detail in this book, but vector-valued RPE signals make sense from the perspective of reinforcement learning when decisions can be decomposed into separate sub-decisions, or more generally, as a way to address the *structural* version of the credit assignment problem: How do you distribute credit for success (or blame for failure) of a decision among the many component structures that could have been involved in producing it? We say a bit more about this in Section 15.10 below.

The axons of most dopamine neurons make synaptic contact with neurons in the frontal cortex and the basal ganglia, areas of the brain involved in voluntary movement, decision making, learning, and cognitive functions such as planning. Because most ideas relating

dopamine to reinforcement learning focus on the basal ganglia, and the connections from dopamine neurons are particularly dense there, we focus on the basal ganglia here. The basal ganglia are a collection of neuron groups, or nuclei, lying at the base of the forebrain. The main input structure of the basal ganglia is called the striatum. Essentially all of the cerebral cortex, among other structures, provides input to the striatum. The activity of cortical neurons conveys a wealth of information about sensory input, internal states, and motor activity. The axons of cortical neurons make synaptic contacts on the dendrites of the main input/output neurons of the striatum, called medium spiny neurons. Output from the striatum loops back via other basal ganglia nuclei and the thalamus to frontal areas of cortex, and to motor areas, making it possible for the striatum to influence movement, abstract decision processes, and reward processing. Two main subdivisions of the striatum are important for reinforcement learning: the dorsal striatum, primarily implicated in influencing action selection, and the ventral striatum, thought to be critical for different aspects of reward processing, including the assignment of affective value to sensations.

The dendrites of medium spiny neurons are covered with spines on whose tips the axons of neurons in the cortex make synaptic contact. Also making synaptic contact with these spines—in this case contacting the spine stems—are axons of dopamine neurons (Figure 15.1). This arrangement brings together presynaptic activity of cortical neurons,

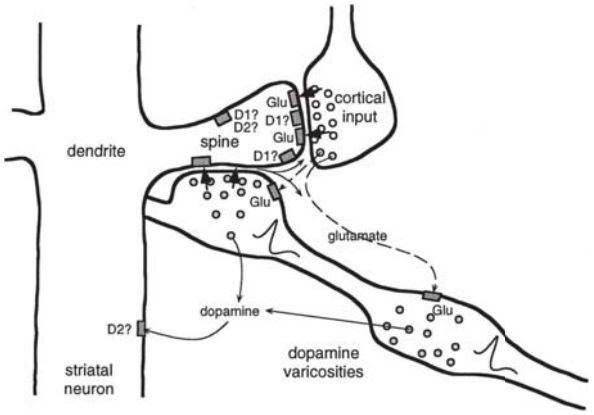


Figure 15.1: Spine of a striatal neuron showing input from both cortical and dopamine neurons. Axons of cortical neurons influence striatal neurons via corticostriatal synapses releasing the neurotransmitter glutamate at the tips of spines covering the dendrites of striatal neurons. An axon of a VTA or SNpc dopamine neuron is shown passing by the spine (from the lower right). "Dopamine varicosities" on this axon release dopamine at or near the spine stem, in an arrangement that brings together presynaptic input from cortex, postsynaptic activity of the striatal neuron, and dopamine, making it possible that several types of learning rules govern the plasticity of corticostriatal synapses. Each axon of a dopamine neuron makes synaptic contact with the stems of roughly 500,000 spines. Some of the complexity omitted from our discussion is shown here by other neurotransmitter pathways and multiple receptor types, such as D1 an D2 dopamine receptors by which dopamine can produce different effects at spines and other postsynaptic sites. From *Journal of Neurophysiology*, W. Schultz, vol. 80, 1998, page 10.

postsynaptic activity of medium spiny neurons, and input from dopamine neurons. What actually occurs at these spines is complex and not completely understood. Figure 15.1 hints at the complexity by showing two types of receptors for dopamine, receptors for glutamate—the neurotransmitter of the cortical inputs—and multiple ways that the various signals can interact. But evidence is mounting that changes in the efficacies of the synapses on the pathway from the cortex to the striatum, which neuroscientists call *corticostriatal synapses*, depend critically on appropriately-timed dopamine signals.

15.5 Experimental Support for the Reward Prediction Error Hypothesis

Dopamine neurons respond with bursts of activity to intense, novel, or unexpected visual and auditory stimuli that trigger eye and body movements, but very little of their activity is related to the movements themselves. This is surprising because degeneration of dopamine neurons is a cause of Parkinson's disease, whose symptoms include motor disorders, particularly deficits in self-initiated movement. Motivated by the weak relationship between dopamine neuron activity and stimulus-triggered eye and body movements, Romo and Schultz (1990) and Schultz and Romo (1990) took the first steps toward the reward prediction error hypothesis by recording the activity of dopamine neurons and muscle activity while monkeys moved their arms.

They trained two monkeys to reach from a resting hand position into a bin containing a bit of apple, a piece of cookie, or a raisin, when the monkey saw and heard the bin's door open. The monkey could then grab and bring the food to its mouth. After a monkey became good at this, it was trained on two additional tasks. The purpose of the first task was to see what dopamine neurons do when movements are self-initiated. The bin was left open but covered from above so that the monkey could not see inside but could reach in from below. No triggering stimuli were presented, and after the monkey reached for and ate the food morsel, the experimenter usually (though not always), silently and unseen by the monkey, replaced food in the bin by sticking it onto a rigid wire. Here too, the activity of the dopamine neurons Romo and Schultz monitored was not related to the monkey's movements, but a large percentage of these neurons produced phasic responses whenever the monkey first touched a food morsel. These neurons did not respond when the monkey touched just the wire or explored the bin when no food was there. This was good evidence that the neurons were responding to the food and not to other aspects of the task.

The purpose of Romo and Schultz's second task was to see what happens when movements are triggered by stimuli. This task used a different bin with a movable cover. The sight and sound of the bin opening triggered reaching movements to the bin. In this case, Romo and Schultz found that after some period of training, the dopamine neurons no longer responded to the touch of the food but instead responded to the sight and sound of the opening cover of the food bin. The phasic responses of these neurons had shifted from the reward itself to stimuli predicting the availability of the reward. In a followup study, Romo and Schultz found that most of the dopamine neurons whose activity they

monitored did not respond to the sight and sound of the bin opening outside the context of the behavioral task. These observations suggested that the dopamine neurons were responding neither to the initiation of a movement nor to the sensory properties of the stimuli, but were rather signaling an expectation of reward.

Schultz's group conducted many additional studies involving both SNpc and VTA dopamine neurons. A particular series of experiments was influential in suggesting that the phasic responses of dopamine neurons correspond to TD errors and not to simpler errors like those in the Rescorla–Wagner model (14.3). In the first of these experiments (Ljungberg, Apicella, and Schultz, 1992), monkeys were trained to depress a lever after a light was illuminated as a 'trigger cue' to obtain a drop of apple juice. As Romo and Schultz had observed earlier, many dopamine neurons initially responded to the reward—the drop of juice (Figure 15.2, top panel). But many of these neurons lost that reward response as training continued and developed responses instead to the illumination of the light that predicted the reward (Figure 15.2, middle panel). With continued training, lever pressing became faster while the number of dopamine neurons responding to the trigger cue decreased.

Following this study, the same monkeys were trained on a new task (Schultz, Apicella, and Ljungberg, 1993). Here the monkeys faced two levers, each with a light above it. Illuminating one of these lights was an 'instruction cue' indicating which of the two levers

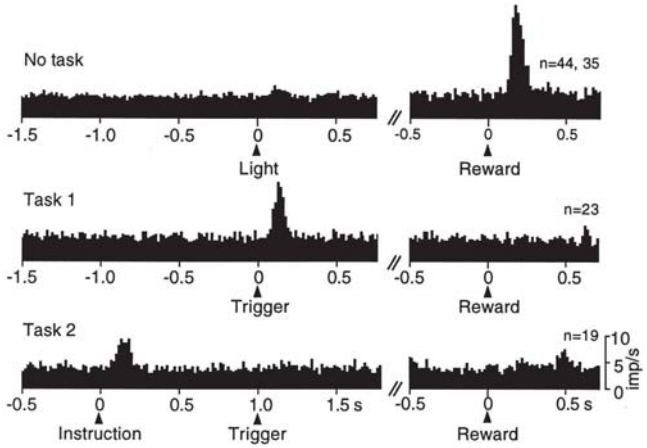


Figure 15.2: The response of dopamine neurons shifts from initial responses to primary reward to earlier predictive stimuli. These are plots of the number of action potentials produced by monitored dopamine neurons within small time intervals, averaged over all the monitored dopamine neurons (ranging from 23 to 44 neurons for these data). Top: dopamine neurons are activated by the unpredicted delivery of drop of apple juice. Middle: with learning, dopamine neurons developed responses to the reward-predicting trigger cue and lost responsiveness to the delivery of reward. Bottom: with the addition of an instruction cue preceding the trigger cue by 1 second, dopamine neurons shifted their responses from the trigger cue to the earlier instruction cue. From Schultz et al. (1995), MIT Press.

would produce a drop of apple juice. In this task, the instruction cue preceded the trigger cue of the previous task by a fixed interval of 1 second. The monkeys learned to withhold reaching until seeing the trigger cue, and dopamine neuron activity increased, but now the responses of the monitored dopamine neurons occurred almost exclusively to the earlier instruction cue and not to the trigger cue (Figure 15.2, bottom panel). Here again the number of dopamine neurons responding to the instruction cue was much reduced when the task was well learned. During learning across these tasks, dopamine neuron activity shifted from initially responding to the reward to responding to the earlier predictive stimuli, first progressing to the trigger stimulus then to the still earlier instruction cue. As responding moved earlier in time it disappeared from the later stimuli. This shifting of responses to earlier reward predictors, while losing responses to later predictors is a hallmark of TD learning (see, for example, Figure 14.2).

The task just described revealed another property of dopamine neuron activity shared with TD learning. The monkeys sometimes pressed the wrong key, that is, the key other than the instructed one, and consequently received no reward. In these trials, many of the dopamine neurons showed a sharp decrease in their firing rates below baseline shortly after the reward's usual time of delivery, and this happened without the availability of any external cue to mark the usual time of reward delivery (Figure 15.3). Somehow the

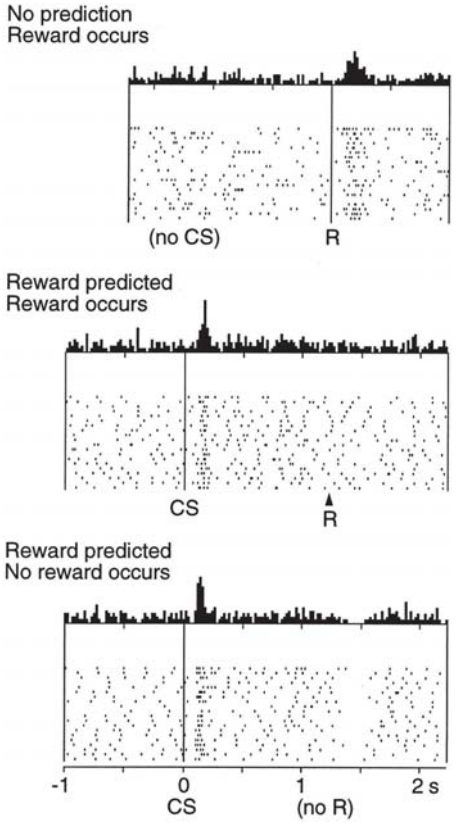


Figure 15.3: The response of dopamine neurons drops below baseline shortly after the time when an expected reward fails to occur. Top: dopamine neurons are activated by the unpredicted delivery of a drop of apple juice. Middle: dopamine neurons respond to a conditioned stimulus (CS) that predicts reward and do not respond to the reward itself. Bottom: when the reward predicted by the CS fails to occur, the activity of dopamine neurons drops below baseline shortly after the time the reward is expected to occur. At the top of each of these panels is shown the average number of action potentials produced by monitored dopamine neurons within small time intervals around the indicated times. The raster plots below show the activity patterns of the individual dopamine neurons that were monitored; each dot represents an action potential. From Schultz, Dayan, and Montague, A Neural Substrate of Prediction and Reward, *Science*, vol. 275, issue 5306, pages 1593-1598, March 14, 1997. Reprinted with permission from AAAS.

monkeys were internally keeping track of the timing of the reward. (Response timing is one area where the simplest version of TD learning needs to be modified to account for some of the details of the timing of dopamine neuron responses. We consider this issue in the following section.)

The observations from the studies described above led Schultz and his group to conclude that dopamine neurons respond to unpredicted rewards, to the earliest predictors of reward, and that dopamine neuron activity decreases below baseline if a reward, or a predictor of reward, does not occur at its expected time. Researchers familiar with reinforcement learning were quick to recognize that these results are strikingly similar to how the TD error behaves as the reinforcement signal in a TD algorithm. The next section explores this similarity by working through a specific example in detail.

15.6 TD Error/Dopamine Correspondence

This section explains the correspondence between the TD error δ and the phasic responses of dopamine neurons observed in the experiments just described. We examine how δ changes over the course of learning in a task something like the one described above where a monkey first sees an instruction cue and then a fixed time later has to respond correctly to a trigger cue in order to obtain reward. We use a simple idealized version of this task, but we go into a lot more detail than is usual because we want to emphasize the theoretical basis of the parallel between TD errors and dopamine neuron activity.

The first simplifying assumption is that the agent has already learned the actions required to obtain reward. Then its task is just to learn accurate predictions of future reward for the sequence of states it experiences. This is then a prediction task, or more technically, a policy-evaluation task: learning the value function for a fixed policy (Sections 4.1 and 6.1). The value function to be learned assigns to each state a value that predicts the return that will follow that state if the agent selects actions according to the given policy, where the return is the (possibly discounted) sum of all the future rewards. This is unrealistic as a model of the monkey's situation because the monkey would likely learn these predictions at the same time that it is learning to act correctly (as would a reinforcement learning algorithm that learns policies as well as value functions, such as an actor–critic algorithm), but this scenario is simpler to describe than one in which a policy and a value function are learned simultaneously.

Now imagine that the agent's experience divides into multiple trials, in each of which the same sequence of states repeats, with a distinct state occurring on each time step during the trial. Further imagine that the return being predicted is limited to the return over a trial, which makes a trial analogous to a reinforcement learning episode as we have defined it. In reality, of course, the returns being predicted are not confined to single trials, and the time interval between trials is an important factor in determining what an animal learns. This is true for TD learning as well, but here we assume that returns do not accumulate over multiple trials. Given this, then, a trial in experiments like those conducted by Schultz and colleagues is equivalent to an episode of reinforcement learning. (Though in this discussion, we will use the term trial instead of episode to relate better to the experiments.)

As usual, we also need to make an assumption about how states are represented as inputs to the learning algorithm, an assumption that influences how closely the TD error corresponds to dopamine neuron activity. We discuss this issue later, but for now we assume the same CSC representation used by Montague et al. (1996) in which there is a separate internal stimulus for each state visited at each time step in a trial. This reduces the process to the tabular case covered in the first part of this book. Finally, we assume that the agent uses TD(0) to learn a value function, V, stored in a lookup table initialized to be zero for all the states. We also assume that this is a deterministic task and that the discount factor, γ, is very nearly one so that we can ignore it.

Figure 15.4 shows the time courses of R, V, and δ at several stages of learning in this policy-evaluation task. The time axes represent the time interval over which a sequence of states is visited in a trial (where for clarity we omit showing individual states). The reward signal is zero throughout each trial except when the agent reaches the rewarding state, shown near the right end of the time line, when the reward signal becomes some positive number, say $R^\star$. The goal of TD learning is to predict the return for each state visited in a trial, which in this undiscounted case and given our assumption that predictions are confined to individual trials, is simply $R^\star$ for each state.

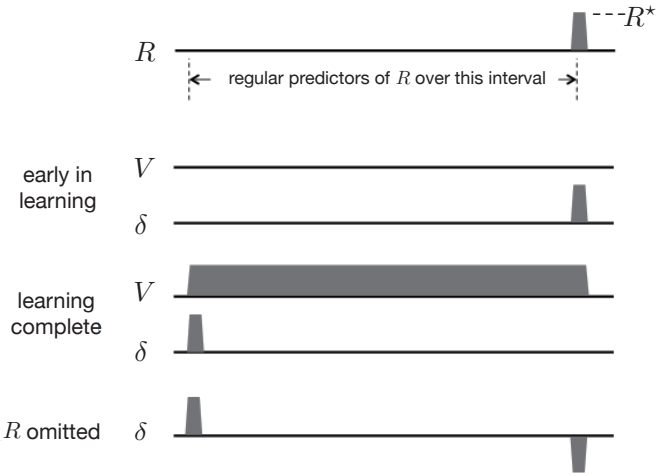


Figure 15.4: The behavior of the TD error δ during TD learning is consistent with features of the phasic activation of dopamine neurons. (Here δ is the TD error *available* at time t, i.e., δ_{t-1}). *Top*: a sequence of states, shown as an interval of regular predictors, is followed by a non-zero reward $R^\star$. *Early in learning*: the initial value function, V, and initial δ, which at first is equal to $R^\star$. *Learning complete*: the value function accurately predicts future reward, δ is positive at the earliest predictive state, and $\delta = 0$ at the time of the non-zero reward. $R^\star$ *omitted*: at the time the predicted reward is omitted, δ becomes negative. See text for a complete explanation of why this happens.

Preceding the rewarding state is a sequence of reward-predicting states, with the *earliest reward-predicting state* shown near the left end of the time line. This is like the state near the start of a trial, for example like the state marked by the instruction cue in a trial of the monkey experiment of Schultz et al. (1993) described above. It is the first state in a trial that reliably predicts that trial's reward. (Of course, in reality states visited on preceding trials are even earlier reward-predicting states, but because we are confining predictions to individual trials, these do not qualify as predictors of *this* trial's reward. Below we give a more satisfactory, though more abstract, description of an earliest reward-predicting state.) The *latest reward-predicting state* in a trial is the state immediately preceding the trial's rewarding state. This is the state near the far right end of the time line in Figure 15.4. Note that the rewarding state of a trial does not predict the return for that trial: the value of this state would come to predict the return over all the *following* trials, which here we are assuming to be zero in this episodic formulation.

Figure 15.4 shows the first-trial time courses of V and δ as the graphs labeled 'early in learning.' Because the reward signal is zero throughout the trial except when the rewarding state is reached, and all the V-values are zero, the TD error is also zero until it becomes $R^\star$ at the rewarding state. This follows because $\delta_{t-1} = R_t + V_t - V_{t-1} = R_t + 0 - 0 = R_t$, which is zero until it equals $R^\star$ when the reward occurs. Here V_t and V_{t-1} are respectively the estimated values of the states visited at times t and $t-1$ in a trial. The TD error at this stage of learning is analogous to a dopamine neuron responding to an unpredicted reward (e.g., a drop of apple juice) at the start of training.

Throughout this first trial and all successive trials, TD(0) updates occur at each state transition as described in Chapter 6. This successively increases the values of the reward-predicting states, with the increases spreading backwards from the rewarding state, until the values converge to the correct return predictions. In this case (because we are assuming no discounting) the correct predictions are equal to $R^\star$ for all the reward-predicting states. This can be seen in Figure 15.4 as the graph of V labeled 'learning complete' where the values of all the states from the earliest to the latest reward-predicting states all equal $R^\star$. The values of the states preceding the earliest reward-predicting state remain low (which Figure 15.4 shows as zero) because they are not reliable predictors of reward.

When learning is complete, that is, when V attains its correct values, the TD errors associated with transitions *from* any reward-predicting state are zero because the predictions are now accurate. This is because for a transition from a reward-predicting state to another reward-predicting state, we have $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + R^\star - R^\star = 0$, and for the transition from the latest reward-predicting state to the rewarding state, we have $\delta_{t-1} = R_t + V_t - V_{t-1} = R^\star + 0 - R^\star = 0$. On the other hand, the TD error on a transition from any state *to* the earliest reward-predicting state is positive because of the mismatch between this state's low value and the larger value of the following reward-predicting state. Indeed, if the value of a state preceding the earliest reward-predicting state were zero, then after the transition to the earliest reward-predicting state, we would have that $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + R^\star - 0 = R^\star$. The 'learning complete' graph of δ in Figure 15.4 shows this positive value at the earliest reward-predicting state, and zeros everywhere else.

The positive TD error upon transitioning to the earliest reward-predicting state is analogous to the persistence of dopamine responses to the earliest stimuli predicting reward. By the same token, when learning is complete, a transition from the latest reward-predicting state to the rewarding state produces a zero TD error because the latest reward-predicting state's value, being correct, cancels the reward. This parallels the observation that fewer dopamine neurons generate a phasic response to a fully predicted reward than to an unpredicted reward.

After learning, if the reward is suddenly omitted, the TD error goes negative at the usual time of reward because the value of the latest reward-predicting state is then too high: $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + 0 - R^\star = -R^\star$, as shown at the right end of the 'R omitted' graph of δ in Figure 15.4. This is like dopamine neuron activity decreasing below baseline at the time an expected reward is omitted as seen in the experiment of Schultz et al. (1993) described above and shown in Figure 15.3.

The idea of an *earliest reward-predicting state* deserves more attention. In the scenario described above, because experience is divided into trials, and we assumed that predictions are confined to individual trials, the earliest reward-predicting state is always the first state of a trial. Clearly this is artificial. A more general way to think of an earliest reward-predicting state is that it is an *unpredicted predictor* of reward, and there can be many such states. In an animal's life, many different states may precede an earliest reward-predicting state. However, because these states are more often followed by *other* states that do not predict reward, their reward-predicting powers, that is, their values, remain low. A TD algorithm, if operating throughout the animal's life, would update the values of these states too, but the updates would not consistently accumulate because, by assumption, none of these states reliably precedes an earliest reward-predicting state. If any of them did, they would be reward-predicting states as well. This might explain why with overtraining, dopamine responses decrease to even the earliest reward-predicting stimulus in a trial. With overtraining one would expect that even a formerly-unpredicted predictor state would become predicted by stimuli associated with earlier states: the animal's interaction with its environment both inside and outside of an experimental task would become commonplace. Upon breaking this routine with the introduction of a new task, however, one would see TD errors reappear, as indeed is observed in dopamine neuron activity.

The example described above explains why the TD error shares key features with the phasic activity of dopamine neurons when the animal is learning in a task similar to the idealized task of our example. But not every property of the phasic activity of dopamine neurons coincides so neatly with properties of δ. One of the most troubling discrepancies involves what happens when a reward occurs *earlier* than expected. We have seen that the omission of an expected reward produces a negative prediction error at the reward's expected time, which corresponds to the activity of dopamine neurons decreasing below baseline when this happens. If the reward arrives later than expected, it is then an unexpected reward and generates a positive prediction error. This happens with both TD errors and dopamine neuron responses. But when reward arrives earlier than expected, dopamine neurons do not do what the TD error does—at least with the CSC representation used by Montague et al. (1996) and by us in our example. Dopamine

neurons do respond to the early reward, which is consistent with a positive TD error because the reward is not predicted to occur then. However, at the later time when the reward is expected but omitted, the TD error is negative whereas, in contrast to this prediction, dopamine neuron activity does not drop below baseline in the way the TD model predicts (Hollerman and Schultz, 1998). Something more complicated is going on in the animal's brain than simply TD learning with a CSC representation.

Some of the mismatches between the TD error and dopamine neuron activity can be addressed by selecting suitable parameter values for the TD algorithm and by using stimulus representations other than the CSC representation. For instance, to address the early-reward mismatch just described, Suri and Schultz (1999) proposed a CSC representation in which the sequences of internal signals initiated by earlier stimuli are cancelled by the occurrence of a reward. Another proposal by Daw, Courville, and Touretzky (2006) is that the brain's TD system uses representations produced by statistical modeling carried out in sensory cortex rather than simpler representations based on raw sensory input. Ludvig, Sutton, and Kehoe (2008) found that TD learning with a microstimulus (MS) representation (Figure 14.1) fits the activity of dopamine neurons in the early-reward and other situations better than when a CSC representation is used. Pan, Schmidt, Wickens, and Hyland (2005) found that even with the CSC representation, prolonged eligibility traces improve the fit of the TD error to some aspects of dopamine neuron activity. In general, many fine details of TD-error behavior depend on subtle interactions between eligibility traces, discounting, and stimulus representations. Findings like these elaborate and refine the reward prediction error hypothesis without refuting its core claim that the phasic activity of dopamine neurons is well characterized as signaling TD errors.

On the other hand, there are other discrepancies between the TD theory and experimental data that are not so easily accommodated by selecting parameter values and stimulus representations (we mention some of these discrepancies in the Bibliographical and Historical Remarks section at the end of this chapter), and more mismatches are likely to be discovered as neuroscientists conduct ever more refined experiments. But the reward prediction error hypothesis has been functioning very effectively as a catalyst for improving our understanding of how the brain's reward system works. Intricate experiments have been designed to validate or refute predictions derived from the hypothesis, and experimental results have, in turn, led to refinement and elaboration of the TD error/dopamine hypothesis.

A remarkable aspect of these developments is that the reinforcement learning algorithms and theory that connect so well with properties of the dopamine system were developed from a computational perspective in total absence of any knowledge about the relevant properties of dopamine neurons—remember, TD learning and its connections to optimal control and dynamic programming were developed many years before any of the experiments were conducted that revealed the TD-like nature of dopamine neuron activity. This unplanned correspondence, despite not being perfect, suggests that the TD error/dopamine parallel captures something significant about brain reward processes.

In addition to accounting for many features of the phasic activity of dopamine neurons, the reward prediction error hypothesis links neuroscience to other aspects of reinforcement

learning, in particular, to learning algorithms that use TD errors as reinforcement signals. Neuroscience is still far from reaching complete understanding of the circuits, molecular mechanisms, and functions of the phasic activity of dopamine neurons, but evidence supporting the reward prediction error hypothesis, along with evidence that phasic dopamine responses are reinforcement signals for learning, suggest that the brain might implement something like an actor–critic algorithm in which TD errors play critical roles. Other reinforcement learning algorithms are plausible candidates too, but actor–critic algorithms fit the anatomy and physiology of the mammalian brain particularly well, as we describe in the following two sections.

15.7 Neural Actor–Critic

Actor–critic algorithms learn both policies and value functions. The 'actor' is the component that learns policies, and the 'critic' is the component that learns about whatever policy is currently being followed by the actor in order to 'criticize' the actor's action choices. The critic uses a TD algorithm to learn the state-value function for the actor's current policy. The value function allows the critic to critique the actor's action choices by sending TD errors, δ, to the actor. A positive δ means that the action was 'good' because it led to a state with a better-than-expected value; a negative δ means that the action was 'bad' because it led to a state with a worse-than-expected value. Based on these critiques, the actor continually updates its policy.

Two distinctive features of actor–critic algorithms are responsible for thinking that the brain might implement an algorithm like this. First, the two components of an actor–critic algorithm—the actor and the critic—suggest that two parts of the striatum—the dorsal and ventral subdivisions (Section 15.4), both critical for reward-based learning—may function respectively something like an actor and a critic. A second property of actor–critic algorithms that suggests a brain implementation is that the TD error has the dual role of being the reinforcement signal for both the actor and the critic, though it has a different influence on learning in each of these components. This fits well with several properties of the neural circuitry: axons of dopamine neurons target both the dorsal and ventral subdivisions of the striatum; dopamine appears to be critical for modulating synaptic plasticity in both structures; and how a neuromodulator such as dopamine acts on a target structure depends on properties of the target structure and not just on properties of the neuromodulator.

Section 13.5 presents actor–critic algorithms as policy gradient methods, but the actor–critic algorithm of Barto, Sutton, and Anderson (1983) was simpler and was presented as an artificial neural network (ANN). Here we describe an ANN implementation something like that of Barto et al., and we follow Takahashi, Schoenbaum, and Niv (2008) in giving a schematic proposal for how this ANN might be implemented by real neural networks in the brain. We postpone discussion of the actor and critic learning rules until Section 15.8, where we present them as special cases of the policy-gradient formulation and discuss what they suggest about how dopamine might modulate synaptic plasticity.

Figure 15.5a shows an implementation of an actor–critic algorithm as an ANN with component networks implementing the actor and the critic. The critic consists of a single neuron-like unit, V, whose output activity represents state values, and a component shown as the diamond labeled TD that computes TD errors by combining V's output with reward signals and with previous state values (as suggested by the loop from the TD diamond to itself). The actor network has a single layer of k actor units labeled A_i, $i = 1, \ldots, k$. The output of each actor unit is a component of a k-dimensional action vector. An alternative is that there are k separate actions, one commanded by each actor unit, that compete with one another to be executed, but here we will think of the entire A-vector as an action.

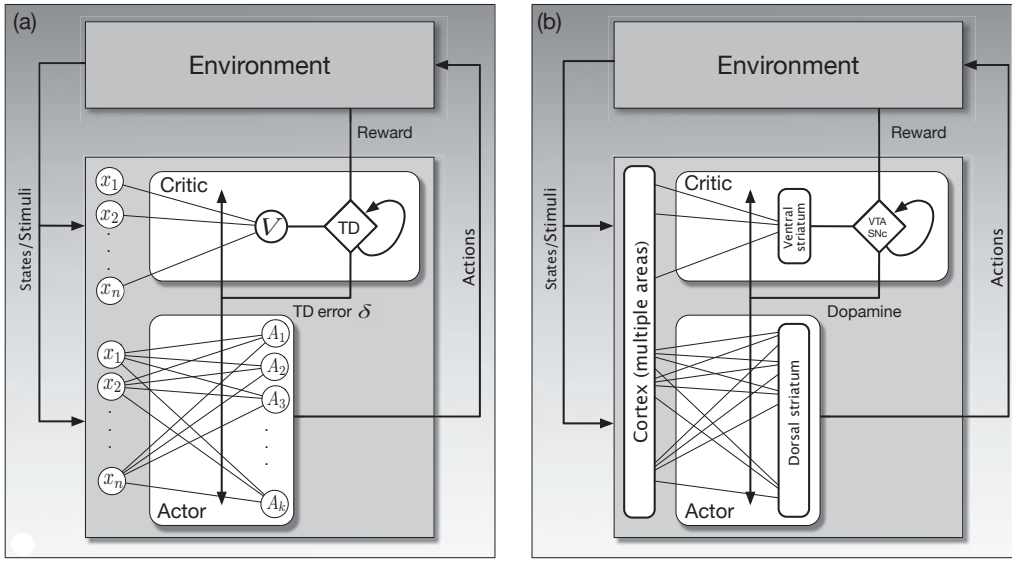


Figure 15.5: Actor–critic ANN and a hypothetical neural implementation. a) Actor–critic algorithm as an ANN. The actor adjusts a policy based on the TD error δ it receives from the critic; the critic adjusts state-value parameters using the same δ. The critic produces a TD error from the reward signal, R, and the current change in its estimate of state values. The actor does not have direct access to the reward signal, and the critic does not have direct access to the action. b) Hypothetical neural implementation of an actor–critic algorithm. The actor and the value-learning part of the critic are respectively placed in the dorsal and ventral subdivisions of the striatum. The TD error is transmitted by dopamine neurons located in the VTA and SNpc to modulate changes in synaptic efficacies of input from cortical areas to the ventral and dorsal striatum. Adapted from Frontiers in Neuroscience, vol. 2(1), 2008, Y. Takahashi, G. Schoenbaum, and Y. Niv, Silencing the critics: Understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an Actor/Critic model.

Both the critic and actor networks receive input consisting of multiple features representing the state of the agent's environment. (Recall from Chapter 1 that the environment of a reinforcement learning agent includes components both inside and outside of the 'organism' containing the agent.) The figure shows these features as the circles labeled $x_1, x_2, \ldots, x_n$, shown twice just to keep the figure simple. A weight representing the efficacy of a synapse is associated with each connection from each feature x_i to the critic unit, V, and to each of the action units, A_i. The weights in the critic network parameterize the value function, and the weights in the actor network parameterize the policy. The networks learn as these weights change according to the critic and actor learning rules that we describe in the following section.

The TD error produced by circuitry in the critic is the reinforcement signal for changing the weights in both the critic and the actor networks. This is shown in Figure 15.5a by the line labeled 'TD error δ' extending across all of the connections in the critic and actor networks. This aspect of the network implementation, together with the reward prediction error hypothesis and the fact that the activity of dopamine neurons is so widely distributed by the extensive axonal arbors of these neurons, suggests that an actor–critic network something like this may not be too farfetched as a hypothesis about how reward-related learning might happen in the brain.

Figure 15.5b suggests—very schematically—how the ANN on the figure's left might map onto structures in the brain according to the hypothesis of Takahashi et al. (2008). The hypothesis puts the actor and the value-learning part of the critic respectively in the dorsal and ventral subdivisions of the striatum, the input structure of the basal ganglia. Recall from Section 15.4 that the dorsal striatum is primarily implicated in influencing action selection, and the ventral striatum is thought to be critical for different aspects of reward processing, including the assignment of affective value to sensations. The cerebral cortex, along with other structures, sends input to the striatum conveying information about stimuli, internal states, and motor activity.

In this hypothetical actor–critic brain implementation, the ventral striatum sends value information to the VTA and SNpc, where dopamine neurons in these nuclei combine it with information about reward to generate activity corresponding to TD errors (though exactly how dopaminergic neurons calculate these errors is not yet understood). The 'TD error δ' line in Figure 15.5a becomes the line labeled 'Dopamine' in Figure 15.5b, which represents the widely branching axons of dopamine neurons whose cell bodies are in the VTA and SNpc. Referring back to Figure 15.1, these axons make synaptic contact with the spines on the dendrites of medium spiny neurons, the main input/output neurons of both the dorsal and ventral divisions of the striatum. Axons of the cortical neurons that send input to the striatum make synaptic contact on the tips of these spines. According to the hypothesis, it is at these spines where changes in the efficacies of the synapses from cortical regions to the striatum are governed by learning rules that critically depend on a reinforcement signal supplied by dopamine.

An important implication of the hypothesis illustrated in Figure 15.5b is that the dopamine signal is not the 'master' reward signal like the scalar R_t of reinforcement learning. In fact, the hypothesis implies that one should not necessarily be able to probe the brain and record any signal like R_t in the activity of any single neuron.

Many interconnected neural systems generate reward-related information, with different structures being recruited depending on different types of rewards. Dopamine neurons receive information from many different brain areas, so the input to the SNpc and VTA labeled 'Reward' in Figure 15.5b should be thought of as vector of reward-related information arriving to neurons in these nuclei along multiple input channels. What the theoretical scalar reward signal R_t might correspond to, then, is the net contribution of all reward-related information to dopamine neuron activity. It is the result of a pattern of activity across many neurons in different areas of the brain.

Although the actor–critic neural implementation illustrated in Figure 15.5b may be correct on some counts, it clearly needs to be refined, extended, and modified to qualify as a full-fledged model of the function of the phasic activity of dopamine neurons. The Historical and Bibliographic Remarks section at the end of this chapter cites publications that discuss in more detail both empirical support for this hypothesis and places where it falls short. We now look in detail at what the actor and critic learning algorithms suggest about the rules governing changes in synaptic efficacies of corticostriatal synapses.

15.8 Actor and Critic Learning Rules

If the brain does implement something like the actor–critic algorithm—and assuming populations of dopamine neurons broadcast a common reinforcement signal to the corticostriatal synapses of both the dorsal and ventral striatum as illustrated in Figure 15.5b (which is likely an oversimplification as we mentioned above)—then this reinforcement signal affects the synapses of these two structures in different ways. The learning rules for the critic and the actor use the same reinforcement signal, the TD error δ, but its effect on learning is different for these two components. The TD error (combined with eligibility traces) tells the actor how to update action probabilities in order to reach higher-valued states. Learning by the actor is like instrumental conditioning using a Law-of-Effect-type learning rule (Section 1.7): the actor works to keep δ as positive as possible. On the other hand, the TD error (when combined with eligibility traces) tells the critic the direction and magnitude in which to change the parameters of the value function in order to improve its predictive accuracy. The critic works to reduce δ's magnitude to be as close to zero as possible using a learning rule like the TD model of classical conditioning (Section 14.2). The difference between the critic and actor learning rules is relatively simple, but this difference has a profound effect on learning and is essential to how the actor–critic algorithm works. The difference lies solely in the eligibility traces each type of learning rule uses.

More than one set of learning rules can be used in actor–critic neural networks like those in Figure 15.5b but, to be specific, here we focus on the actor–critic algorithm for continuing problems with eligibility traces presented in Section 13.6. On each transition from state S_t to state S_{t+1}, taking action A_t and receiving action R_{t+1}, that algorithm computes the TD error (δ) and then updates the eligibility trace vectors ($\mathbf{z}_t^{\mathbf{w}}$ and $\mathbf{z}_t^{\boldsymbol{\theta}}$) and

the parameters for the critic and actor ($\mathbf{w}$ and $\boldsymbol{\theta}$), according to

$$\begin{aligned}
\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w}),\\
\mathbf{z}^{\mathbf{w}}_t &= \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}}_{t-1} + \nabla \hat{v}(S_t,\mathbf{w}),\\
\mathbf{z}^{\boldsymbol{\theta}}_t &= \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}}_{t-1} + \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta}),\\
\mathbf{w} &\leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\, \delta_t\, \mathbf{z}^{\mathbf{w}}_t,\\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\, \delta\, \mathbf{z}^{\boldsymbol{\theta}}_t,
\end{aligned}$$

where $\gamma \in [0,1)$ is a discount-rate parameter, $\lambda^w c \in [0,1]$ and $\lambda^w a \in [0,1]$ are bootstrapping parameters for the critic and the actor respectively, and $\alpha^{\mathbf{w}} > 0$ and $\alpha^{\boldsymbol{\theta}} > 0$ are analogous step-size parameters.

Think of the approximate value function $\hat{v}$ as the output of a single linear neuron-like unit, called the *critic unit* and labeled V in Figure 15.5a. Then the value function is a linear function of the feature-vector representation of state s, $\mathbf{x}(s) = (x_1(s), \ldots, x_n(s))^\top$, parameterized by a weight vector $\mathbf{w} = (w_1, \ldots, w_n)^\top$:

$$\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s). \tag{15.1}$$

Each $x_i(s)$ is like the presynaptic signal to a neuron's synapse whose efficacy is w_i. The weights of the critic are incremented according to the rule above by $\alpha^{\mathbf{w}} \delta_t \mathbf{z}^{\mathbf{w}}_t$, where the reinforcement signal, δ_t, corresponds to a dopamine signal being broadcast to all of the critic unit's synapses. The eligibility trace vector, $\mathbf{z}^{\mathbf{w}}_t$, for the critic unit is a trace (average of recent values) of $\nabla \hat{v}(S_t,\mathbf{w})$. Because $\hat{v}(s,\mathbf{w})$ is linear in the weights, $\nabla \hat{v}(S_t,\mathbf{w}) = \mathbf{x}(S_t)$.

In neural terms, this means that each synapse has its own eligibility trace, which is one component of the vector $\mathbf{z}^{\mathbf{w}}_t$. A synapse's eligibility trace accumulates according to the level of activity arriving at that synapse, that is, the level of presynaptic activity, represented here by the component of the feature vector $\mathbf{x}(S_t)$ arriving at that synapse. The trace otherwise decays toward zero at a rate governed by the fraction $\lambda^{\mathbf{w}}$. A synapse is *eligible for modification* as long as its eligibility trace is non-zero. How the synapse's efficacy is actually modified depends on the reinforcement signals that arrive while the synapse is eligible. We call eligibility traces like these of the critic unit's synapses *non-contingent eligibility traces* because they only depend on presynaptic activity and are not contingent in any way on postsynaptic activity.

The non-contingent eligibility traces of the critic unit's synapses mean that the critic unit's learning rule is essentially the TD model of classical conditioning described in Section 14.2. With the definition we have given above of the critic unit and its learning rule, the critic in Figure 15.5a is the same as the critic in the ANN actor–critic of Barto et al. (1983). Clearly, a critic like this consisting of just one linear neuron-like unit is the simplest starting point; this critic unit is a proxy for a more complicated neural network able to learn value functions of greater complexity.

The actor in Figure 15.5a is a one-layer network of k neuron-like actor units, each receiving at time t the same feature vector, $\mathbf{x}(S_t)$, that the critic unit receives. Each actor unit j, $j = 1, \ldots, k$, has its own weight vector, $\boldsymbol{\theta}_j$, but because the actor units are all identical, we describe just one of the units and omit the subscript. One way for these

units to follow the actor–critic algorithm given in the equations above is for each to be a *Bernoulli-logistic unit*. This means that the output of each actor unit at each time is a random variable, A_t, taking value 0 or 1. Think of value 1 as the neuron firing, that is, emitting an action potential. The weighted sum, $\boldsymbol{\theta}^\top \mathbf{x}(S_t)$, of a unit's input vector determines the unit's action probabilities via the exponential soft-max distribution (13.2), which for two actions is the logistic function:

$$\pi(1|s,\boldsymbol{\theta}) = 1 - \pi(0|s,\boldsymbol{\theta}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^\top \mathbf{x}(s))}. \tag{15.2}$$

The weights of each actor unit are incremented, as above, by: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta_t \mathbf{z}^{\boldsymbol{\theta}}_t$, where δ again corresponds to the dopamine signal: the same reinforcement signal that is sent to all the critic unit's synapses. Figure 15.5a shows δ_t being broadcast to all the synapses of all the actor units (which makes this actor network a *team* of reinforcement learning agents, something we discuss in Section 15.10 below). The actor eligibility trace vector $\mathbf{z}^{\boldsymbol{\theta}}_t$ is a trace (average of recent values) of $\nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$. To understand this eligibility trace refer to Exercise 13.5, which defines this kind of unit and asks you to give a learning rule for it. That exercise asked you to express $\nabla \ln \pi(a|s,\boldsymbol{\theta})$ in terms of a, $\mathbf{x}(s)$, and $\pi(a|s,\boldsymbol{\theta})$ (for arbitrary state s and action a) by calculating the gradient. For the action and state actually occurring at time t, the answer is

$$\nabla \pi(A_t|S_t,\boldsymbol{\theta}) = \big(A_t - \pi(A_t|S_t,\boldsymbol{\theta})\big)\mathbf{x}(S_t). \tag{15.3}$$

Unlike the non-contingent eligibility trace of a critic synapse that only accumulates the presynaptic activity $\mathbf{x}(S_t)$, the eligibility trace of an actor unit's synapse in addition depends on the activity of the actor unit itself. We call this a *contingent eligibility trace* because it is contingent on this postsynaptic activity. The eligibility trace at each synapse continually decays, but increments or decrements depending on the activity of the presynaptic neuron *and* whether or not the postsynaptic neuron fires. The factor $A_t - \pi(A_t|S_t,\boldsymbol{\theta})$ in (15.3) is positive when $A_t = 1$ and negative otherwise. *The postsynaptic contingency in the eligibility traces of actor units is the only difference between the critic and actor learning rules.* By keeping information about what actions were taken in what states, contingent eligibility traces allow credit for reward (positive δ), or blame for punishment (negative δ), to be apportioned among the policy parameters (the efficacies of the actor units' synapses) according to the contributions these parameters made to the units' outputs that could have influenced later values of δ. Contingent eligibility traces mark the synapses as to how they should be modified to alter the units' future responses to favor positive values of δ.

What do the critic and actor learning rules suggest about how efficacies of corticostriatal synapses change? Both learning rules are related to Donald Hebb's classic proposal that whenever a presynaptic signal participates in activating the postsynaptic neuron, the synapse's efficacy increases (Hebb, 1949). The critic and actor learning rules share with Hebb's proposal the idea that changes in a synapse's efficacy depend on the interaction of several factors. In the critic learning rule the interaction is between the reinforcement signal δ and eligibility traces that depend only on presynaptic signals. Neuroscientists call this a *two-factor learning rule* because the interaction is between two signals or

quantities. The actor learning rule, on the other hand, is a *three-factor learning rule* because, in addition to depending on δ, its eligibility traces depend on both presynaptic and postsynaptic activity. Unlike Hebb's proposal, however, the relative timing of the factors is critical to how synaptic efficacies change, with eligibility traces intervening to allow the reinforcement signal to affect synapses that were active in the recent past.

Some subtleties about signal timing for the actor and critic learning rules deserve closer attention. In defining the neuron-like actor and critic units, we ignored the small amount of time it takes synaptic input to effect the firing of a real neuron. When an action potential from the presynaptic neuron arrives at a synapse, neurotransmitter molecules are released that diffuse across the synaptic cleft to the postsynaptic neuron, where they bind to receptors on the postsynaptic neuron's surface; this activates molecular machinery that causes the postsynaptic neuron to fire (or to inhibit its firing in the case of inhibitory synaptic input). This process can take several tens of milliseconds. According to (15.1) and (15.2), though, the input to a critic and actor unit instantaneously produces the unit's output. Ignoring activation time like this is common in abstract models of Hebbian-style plasticity in which synaptic efficacies change according to a simple product of simultaneous pre- and postsynaptic activity. More realistic models must take activation time into account.

Activation time is especially important for a more realistic actor unit because it influences how contingent eligibility traces have to work in order to properly apportion credit for reinforcement to the appropriate synapses. The expression $\big(A_t - \pi(A_t|S_t,\boldsymbol{\theta})\big)\mathbf{x}(S_t)$ defining contingent eligibility traces for the actor unit's learning rule given above includes the postsynaptic factor $\big(A_t - \pi(A_t|S_t,\boldsymbol{\theta})\big)$ and the presynaptic factor $\mathbf{x}(S_t)$. This works because by ignoring activation time, the presynaptic activity $\mathbf{x}(S_t)$ participates in *causing* the postsynaptic activity appearing in $\big(A_t - \pi(A_t|S_t,\boldsymbol{\theta})\big)$. To assign credit for reinforcement correctly, the presynaptic factor defining the eligibility trace must be a cause of the postsynaptic factor that also defines the trace. Contingent eligibility traces for a more realistic actor unit would have to take activation time into account. (Activation time should not be confused with the time required for a neuron to receive a reinforcement signal influenced by that neuron's activity. The function of eligibility traces is to span this time interval which is generally much longer than the activation time. We discuss this further in the following section.)

There are hints from neuroscience for how this process might work in the brain. Neuroscientists have discovered a form of Hebbian plasticity called *spike-timing-dependent plasticity* (STDP) that lends plausibility to the existence of actor-like synaptic plasticity in the brain. STDP is a Hebbian-style plasticity, but changes in a synapse's efficacy depend on the relative timing of presynaptic and postsynaptic action potentials. The dependence can take different forms, but in the one most studied, a synapse increases in strength if spikes incoming via that synapse arrive shortly before the postsynaptic neuron fires. If the timing relation is reversed, with a presynaptic spike arriving shortly after the postsynaptic neuron fires, then the strength of the synapse decreases. STDP is a type of Hebbian plasticity that takes the activation time of a neuron into account, which is one of the ingredients needed for actor-like learning.

The discovery of STDP has led neuroscientists to investigate the possibility of a three-factor form of STDP in which neuromodulatory input must follow appropriately-timed pre- and postsynaptic spikes. This form of synaptic plasticity, called *reward-modulated STDP*, is much like the actor learning rule discussed here. Synaptic changes that would be produced by regular STDP only occur if there is neuromodulatory input within a time window after a presynaptic spike is closely followed by a postsynaptic spike. Evidence is accumulating that reward-modulated STDP occurs at the spines of medium spiny neurons of the dorsal striatum, with dopamine providing the neuromodulatory factor—the sites where actor learning takes place in the hypothetical neural implementation of an actor–critic algorithm illustrated in Figure 15.5b. Experiments have demonstrated reward-modulated STDP in which lasting changes in the efficacies of corticostriatal synapses occur only if a neuromodulatory pulse arrives within a time window that can last up to 10 seconds after a presynaptic spike is closely followed by a postsynaptic spike (Yagishita et al. 2014). Although the evidence is indirect, these experiments point to the existence of contingent eligibility traces having prolonged time courses. The molecular mechanisms producing these traces, as well as the much shorter traces that likely underly STDP, are not yet understood, but research focusing on time-dependent and neuromodulator-dependent synaptic plasticity is continuing.

The neuron-like actor unit that we have described here, with its Law-of-Effect-style learning rule, appeared in somewhat simpler form in the actor–critic network of Barto et al. (1983). That network was inspired by the "hedonistic neuron" hypothesis proposed by physiologist A. H. Klopf (1972, 1982). Not all the details of Klopf's hypothesis are consistent with what has been learned about synaptic plasticity, but the discovery of STDP and the growing evidence for a reward-modulated form of STDP suggest that Klopf's ideas may not have been far off the mark. We discuss Klopf's hedonistic neuron hypothesis next.

15.9 Hedonistic Neurons

In his hedonistic neuron hypothesis, Klopf (1972, 1982) conjectured that individual neurons seek to maximize the difference between synaptic input treated as rewarding and synaptic input treated as punishing by adjusting the efficacies of their synapses on the basis of rewarding or punishing consequences of their own action potentials. In other words, individual neurons can be trained with response-contingent reinforcement like an animal can be trained in an instrumental conditioning task. His hypothesis included the idea that rewards and punishments are conveyed to a neuron via the same synaptic input that excites or inhibits the neuron's spike-generating activity. (Had Klopf known what we know today about neuromodulatory systems, he might have assigned the reinforcing role to neuromodulatory input, but he wanted to avoid any centralized source of training information.) Synaptically-local traces of past pre- and postsynaptic activity had the key function in Klopf's hypothesis of making synapses *eligible*—the term he introduced—for modification by later reward or punishment. He conjectured that these traces are implemented by molecular mechanisms local to each synapse and therefore different from the electrical activity of both the pre- and the postsynaptic neurons. In

the Bibliographical and Historical Remarks section of this chapter we bring attention to some similar proposals made by others.

Klopf specifically conjectured that synaptic efficacies change in the following way. When a neuron fires an action potential, all of its synapses that were active in contributing to that action potential become eligible to undergo changes in their efficacies. If the action potential is followed within an appropriate time period by an increase of reward, the efficacies of all the eligible synapses increase. Symmetrically, if the action potential is followed within an appropriate time period by an increase of punishment, the efficacies of eligible synapses decrease. This is implemented by triggering an eligibility trace at a synapse upon a coincidence of presynaptic and postsynaptic activity (or more exactly, upon pairing of presynaptic activity with the postsynaptic activity that that presynaptic activity participates in causing)—what we call a contingent eligibility trace. This is essentially the three-factor learning rule of an actor unit described in the previous section.

The shape and time course of an eligibility trace in Klopf's theory reflects the durations of the many feedback loops in which the neuron is embedded, some of which lie entirely within the brain and body of the organism, while others extend out through the organism's external environment as mediated by its motor and sensory systems. His idea was that the shape of a synaptic eligibility trace is like a histogram of the durations of the feedback loops in which the neuron is embedded. The peak of an eligibility trace would then occur at the duration of the most prevalent feedback loops in which that neuron participates. The eligibility traces used by algorithms described in this book are simplified versions of Klopf's original idea, being exponentially (or geometrically) decreasing functions controlled by the parameters λ and γ. This simplifies simulations as well as theory, but we regard these simple eligibility traces as a placeholders for traces closer to Klopf's original conception, which would have computational advantages in complex reinforcement learning systems by refining the credit-assignment process.

Klopf's hedonistic neuron hypothesis is not as implausible as it may at first appear. A well-studied example of a single cell that seeks some stimuli and avoids others is the bacterium *Escherichia coli*. The movement of this single-cell organism is influenced by chemical stimuli in its environment, behavior known as chemotaxis. It swims in its liquid environment by rotating hairlike structures called flagella attached to its surface. (Yes, it rotates them!) Molecules in the bacterium's environment bind to receptors on its surface. Binding events modulate the frequency with which the bacterium reverses flagellar rotation. Each reversal causes the bacterium to tumble in place and then head off in a random new direction. A little chemical memory and computation causes the frequency of flagellar reversal to decrease when the bacterium swims toward higher concentrations of molecules it needs to survive (attractants) and increase when the bacterium swims toward higher concentrations of molecules that are harmful (repellants). The result is that the bacterium tends to persist in swimming up attractant gradients and tends to avoid swimming up repellant gradients.

The chemotactic behavior just described is called klinokinesis. It is a kind of trial-and-error behavior, although it is unlikely that learning is involved: the bacterium needs a modicum of short-term memory to detect molecular concentration gradients, but it probably does not maintain long-term memories. Artificial intelligence pioneer Oliver

Selfridge called this strategy "run and twiddle," pointing out its utility as a basic adaptive strategy: "keep going in the same way if things are getting better, and otherwise move around" (Selfridge, 1978, 1984). Similarly, one might think of a neuron "swimming" (not literally of course) in a medium composed of the complex collection of feedback loops in which it is embedded, acting to obtain one type of input signal and to avoid others. Unlike the bacterium, however, the neuron's synaptic strengths retain information about its past trial-and-error behavior. If this view of the behavior of a neuron (or just one type of neuron) is plausible, then the closed-loop nature of how the neuron interacts with its environment is important for understanding its behavior, where the neuron's environment consists of the rest of the animal together with the environment with which the animal as a whole interacts.

Klopf's hedonistic neuron hypothesis extended beyond the idea that individual neurons are reinforcement learning agents. He argued that many aspects of intelligent behavior can be understood as the result of the collective behavior of a population of self-interested hedonistic neurons interacting with one another in an immense society or economic system making up an animal's nervous system. Whether or not this view of nervous systems is useful, the collective behavior of reinforcement learning agents has implications for neuroscience. We take up this subject next.

15.10 Collective Reinforcement Learning

The behavior of populations of reinforcement learning agents is deeply relevant to the study of social and economic systems, and if anything like Klopf's hedonistic neuron hypothesis is correct, to neuroscience as well. The hypothesis described above about how an actor–critic algorithm might be implemented in the brain only narrowly addresses the implications of the fact that the dorsal and ventral subdivisions of the striatum, the respective locations of the actor and the critic according to the hypothesis, each contain millions of medium spiny neurons whose synapses undergo change modulated by phasic bursts of dopamine neuron activity.

The actor in Figure 15.5a is a single-layer network of k actor units. The actions produced by this network are vectors $(A_1, A_2, \cdots, A_k)^\top$ presumed to drive the animal's behavior. Changes in the efficacies of the synapses of all of these units depend on the reinforcement signal δ. Because actor units attempt to make δ as large as possible, δ effectively acts as a reward signal for them (so in this case reinforcement is the same as reward). Thus, each actor unit is itself a reinforcement learning agent—a hedonistic neuron if you will. Now, to make the situation as simple as possible, assume that each of these units receives the same reward signal at the same time (although, as indicated above, the assumption that dopamine is released at all the corticostriatal synapses under the same conditions and at the same times is likely an oversimplification).

What can reinforcement learning theory tell us about what happens when all members of a population of reinforcement learning agents learn according to a common reward signal? The field of *multi-agent reinforcement learning* considers many aspects of learning by populations of reinforcement learning agents. Although this field is beyond the scope of this book, we believe that some of its basic concepts and results are relevant to thinking

about the brain's diffuse neuromodulatory systems. In multi-agent reinforcement learning (and in game theory), the scenario in which all the agents try to maximize a common reward signal that they simultaneously receive is known as a *cooperative game* or a *team problem*.

What makes a team problem interesting and challenging is that the common reward signal sent to each agent evaluates the *pattern* of activity produced by the entire population, that is, it evaluates the *collective action* of the team members. This means that any individual agent has only limited ability to affect the reward signal because any single agent contributes just one component of the collective action evaluated by the common reward signal. Effective learning in this scenario requires addressing a *structural credit assignment problem*: which team members, or groups of team members, deserve credit for a favorable reward signal, or blame for an unfavorable reward signal? It is a *cooperative* game, or a team problem, because the agents are united in seeking to increase the same reward signal: there are no conflicts of interest among the agents. The scenario would be a *competitive game* if different agents receive different reward signals, where each reward signal again evaluates the collective action of the population, and the objective of each agent is to increase its own reward signal. In this case there might be conflicts of interest among the agents, meaning that actions that are good for some agents are bad for others. Even deciding what the best collective action should be is a non-trivial aspect of game theory. This competitive setting might be relevant to neuroscience too (for example, to account for heterogeneity of dopamine neuron activity), but here we focus only on the cooperative, or team, case.

How can each reinforcement learning agent in a team learn to "do the right thing" so that the collective action of the team is highly rewarded? An interesting result is that if each agent can learn effectively despite its reward signal being corrupted by a large amount of noise, and despite its lack of access to complete state information, then the population as a whole will learn to produce collective actions that improve as evaluated by the common reward signal, even when the agents cannot communicate with one another. Each agent faces its own reinforcement learning task in which its influence on the reward signal is deeply buried in the noise created by the influences of other agents. In fact, for any agent, all the other agents are part of its environment because its input, both the part conveying state information and the reward part, depends on how all the other agents are behaving. Furthermore, lacking access to the actions of the other agents, indeed lacking access to the parameters determining their policies, each agent can only partially observe the state of its environment. This makes each team member's learning task very difficult, but if each uses a reinforcement learning algorithm able to increase a reward signal even under these difficult conditions, teams of reinforcement learning agents can learn to produce collective actions that improve over time as evaluated by the team's common reward signal.

If the team members are neuron-like units, then each unit has to have the goal of increasing the amount of reward it receives over time, as the actor unit does that we described in Section 15.8. Each unit's learning algorithm has to have two essential features. First, it has to use contingent eligibility traces. Recall that a contingent eligibility trace, in neural terms, is initiated (or increased) at a synapse when its presynaptic input

participates in causing the postsynaptic neuron to fire. A non-contingent eligibility trace, in contrast, is initiated or increased by presynaptic input independently of what the postsynaptic neuron does. As explained in Section 15.8, by keeping information about what actions were taken in what states, contingent eligibility traces allow credit for reward, or blame for punishment, to be apportioned to an agent's policy parameters according to the contribution the values of these parameters made in determining the agent's action. By similar reasoning, a team member must remember its recent action so that it can either increase or decrease the likelihood of producing that action according to the reward signal that is subsequently received. The action component of a contingent eligibility trace implements this action memory. Because of the complexity of the learning task, however, contingent eligibility is merely a preliminary step in the credit assignment process: the relationship between a single team member's action and changes in the team's reward signal is a statistical correlation that has to be estimated over many trials. Contingent eligibility is an essential but preliminary step in this process.

Learning with non-contingent eligibility traces does not work at all in the team setting because it does not provide a way to correlate actions with consequent changes in the reward signal. Non-contingent eligibility traces are adequate for learning to predict, as the critic component of the actor–critic algorithm does, but they do not support learning to control, as the actor component must do. The members of a population of critic-like agents may still receive a common reinforcement signal, but they would all learn to predict the same quantity (which in the case of an actor–critic method, would be the expected return for the current policy). How successful each member of the population would be in learning to predict the expected return would depend on the information it receives, which could be very different for different members of the population. There would be no need for the population to produce differentiated patterns of activity. This is not a team problem as defined here.

A second requirement for collective learning in a team problem is that there has to be variability in the actions of the team members in order for the team to explore the space of collective actions. The simplest way for a team of reinforcement learning agents to do this is for each member to independently explore its own action space through persistent variability in its output. This will cause the team as a whole to vary its collective actions. For example, a team of the actor units described in Section 15.8 explores the space of collective actions because the output of each unit, being a Bernoulli-logistic unit, probabilistically depends on the weighted sum of its input vector's components. The weighted sum biases firing probability up or down, but there is always variability. Because each unit uses a REINFORCE policy gradient algorithm (Chapter 13), each unit adjusts its weights with the goal of maximizing the average reward rate it experiences while stochastically exploring its own action space. One can show, as Williams (1992) did, that a team of Bernoulli-logistic REINFORCE units implements a policy gradient algorithm *as a whole* with respect to average rate of the team's common reward signal, where the actions are the collective actions of the team.

Further, Williams (1992) showed that a team of Bernoulli-logistic units using REINFORCE ascends the average reward gradient when the units in the team are interconnected to form a multilayer ANN. In this case, the reward signal is broadcast to all the units in

the network, though reward may depend only on the collective actions of the network's output units. This means that a multilayer team of Bernoulli-logistic REINFORCE units learns like a multilayer network trained by the widely-used error backpropagation method, but in this case the backpropagation process is replaced by the broadcasted reward signal. In practice, the error backpropagation method is considerably faster, but the reinforcement learning team method is more plausible as a neural mechanism, especially in light of what is being learned about reward-modulated STDP as discussed in Section 15.8.

Exploration through independent exploration by team members is only the simplest way for a team to explore; more sophisticated methods are possible if the team members coordinate their actions to focus on particular parts of the collective action space, either by communicating with one another or by responding to common inputs. There are also mechanisms more sophisticated than contingent eligibility traces for addressing structural credit assignment, which is easier in a team problem when the set of possible collective actions is restricted in some way. An extreme case is a winner-take-all arrangement (for example, the result of lateral inhibition in the brain) that restricts collective actions to those to which only one, or a few, team members contribute. In this case the winners get the credit or blame for resulting reward or punishment.

Details of learning in cooperative games (or team problems) and non-cooperative game problems are beyond the scope of this book. The Bibliographical and Historical Remarks section at the end of this chapter cites a selection of the relevant publications, including extensive references to research on implications for neuroscience of collective reinforcement learning.

15.11 Model-based Methods in the Brain

Reinforcement learning's distinction between model-free and model-based algorithms is proving to be useful for thinking about animal learning and decision processes. Section 14.6 discusses how this distinction aligns with that between habitual and goal-directed animal behavior. The hypothesis discussed above about how the brain might implement an actor–critic algorithm is relevant only to an animal's habitual mode of behavior because the basic actor–critic method is model-free. What neural mechanisms are responsible for producing goal-directed behavior, and how do they interact with those underlying habitual behavior?

One way to investigate questions about the brain structures involved in these modes of behavior is to inactivate an area of a rat's brain and then observe what the rat does in an outcome-devaluation experiment (Section 14.6). Results from experiments like these indicate that the actor–critic hypothesis described above is too simple in placing the actor in the dorsal striatum. Inactivating one part of the dorsal striatum, the dorsolateral striatum (DLS), impairs habit learning, causing the animal to rely more on goal-directed processes. On the other hand, inactivating the dorsomedial striatum (DMS) impairs goal-directed processes, requiring the animal to rely more on habit learning. Results like these support the view that the DLS in rodents is more involved in model-free processes, whereas their DMS is more involved in model-based processes. Results of

studies with human subjects in similar experiments using functional neuroimaging, and with non-human primates, support the view that the analogous structures in the primate brain are differentially involved in habitual and goal-directed modes of behavior.

Other studies identify activity associated with model-based processes in the prefrontal cortex of the human brain, the front-most part of the frontal cortex implicated in executive function, including planning and decision making. Specifically implicated is the orbitofrontal cortex (OFC), the part of the prefrontal cortex immediately above the eyes. Functional neuroimaging in humans, and also recordings of the activities of single neurons in monkeys, reveals strong activity in the OFC related to the subjective reward value of biologically significant stimuli, as well as activity related to the reward expected as a consequence of actions. Although not free of controversy, these results suggest significant involvement of the OFC in goal-directed choice. It may be critical for the reward part of an animal's environment model.

Another structure involved in model-based behavior is the hippocampus, a structure critical for memory and spatial navigation. A rat's hippocampus plays a critical role in the rat's ability to navigate a maze in the goal-directed manner that led Tolman to the idea that animals use models, or cognitive maps, in selecting actions (Section 14.5). The hippocampus may also be a critical component of our human ability to imagine new experiences (Hassabis and Maguire, 2007; Ólafsdóttir, Barry, Saleem, Hassabis, and Spiers, 2015).

The findings that most directly implicate the hippocampus in planning—the process needed to enlist an environment model in making decisions—come from experiments that decode the activity of neurons in the hippocampus to determine what part of space hippocampal activity is representing on a moment-to-moment basis. When a rat pauses at a choice point in a maze, the representation of space in the hippocampus sweeps forward (and not backwards) along the possible paths the animal can take from that point (Johnson and Redish, 2007). Furthermore, the spatial trajectories represented by these sweeps closely correspond to the rat's subsequent navigational behavior (Pfeiffer and Foster, 2013). These results suggest that the hippocampus is critical for the state-transition part of an animal's environment model, and that it is part of a system that uses the model to simulate possible future state sequences to assess the consequences of possible courses of action: a form of planning.

The results described above add to a voluminous literature on neural mechanisms underlying goal-directed, or model-based, learning and decision making, but many questions remain unanswered. For example, how can areas as structurally similar as the DLS and DMS be essential components of modes of learning and behavior that are as different as model-free and model-based algorithms? Are separate structures responsible for (what we call) the transition and reward components of an environment model? Is all planning conducted at decision time via simulations of possible future courses of action as the forward sweeping activity in the hippocampus suggests? In other words, is all planning something like a rollout algorithm (Section 8.10)? Or are models sometimes engaged in the background to refine or recompute value information as illustrated by the Dyna architecture (Section 8.2)? How does the brain arbitrate between the use of the habit and goal-directed systems? Is there, in fact, a clear separation between the neural

substrates of these systems?

The evidence is not pointing to a positive answer to this last question. Summarizing the situation, Doll, Simon, and Daw (2012) wrote that "model-based influences appear ubiquitous more or less wherever the brain processes reward information," and this is true even in the regions thought to be critical for model-free learning. This includes the dopamine signals themselves, which can exhibit the influence of model-based information in addition to the reward prediction errors thought to be the basis of model-free processes.

Continuing neuroscience research informed by reinforcement learning's model-free and model-based distinction has the potential to sharpen our understanding of habitual and goal-directed processes in the brain. A better grasp of these neural mechanisms may lead to algorithms combining model-free and model-based methods in ways that have not yet been explored in computational reinforcement learning.

15.12 Addiction

Understanding the neural basis of drug abuse is a high-priority goal of neuroscience with the potential to produce new treatments for this serious public health problem. One view is that drug craving is the result of the same motivation and learning processes that lead us to seek natural rewarding experiences that serve our biological needs. Addictive substances, by being intensely reinforcing, effectively co-opt our natural mechanisms of learning and decision making. This is plausible given that many—though not all—drugs of abuse increase levels of dopamine either directly or indirectly in regions around terminals of dopamine neuron axons in the striatum, a brain structure firmly implicated in normal reward-based learning (Section 15.7). But the self-destructive behavior associated with drug addiction is not characteristic of normal learning. What is different about dopamine-mediated learning when the reward is the result of an addictive drug? Is addiction the result of normal learning in response to substances that were largely unavailable throughout our evolutionary history, so that evolution could not select against their damaging effects? Or do addictive substances somehow interfere with normal dopamine-mediated learning?

The reward prediction error hypothesis of dopamine neuron activity and its connection to TD learning are the basis of a model due to Redish (2004) of some—but certainly not all—features of addiction. The model is based on the observation that administration of cocaine and some other addictive drugs produces a transient increase in dopamine. In the model, this dopamine surge is assumed to increase the TD error, δ, in a way that cannot be cancelled out by changes in the value function. In other words, whereas δ is reduced to the degree that a normal reward is predicted by antecedent events (Section 15.6), the contribution to δ due to an addictive stimulus does not decrease as the reward signal becomes predicted: drug rewards cannot be "predicted away." The model does this by preventing δ from ever becoming negative when the reward signal is due to an addictive drug, thus eliminating the error-correcting feature of TD learning for states associated with administration of the drug. The result is that the values of these states increase without bound, making actions leading to these states preferred above all others.

Addictive behavior is much more complicated than this result from Redish's model, but the model's main idea may be a piece of the puzzle. Or the model might be misleading. Dopamine appears not to play a critical role in all forms of addiction, and not everyone is equally susceptible to developing addictive behavior. Moreover, the model does not include the changes in many circuits and brain regions that accompany chronic drug taking, for example, changes that lead to a drug's diminishing effect with repeated use. It is also likely that addiction involves model-based processes. Still, Redish's model illustrates how reinforcement learning theory can be enlisted in the effort to understand a major health problem. In a similar manner, reinforcement learning theory has been influential in the development of the new field of computational psychiatry, which aims to improve understanding of mental disorders through mathematical and computational methods.

15.13 Summary

The neural pathways involved in the brain's reward system are complex and incompletely understood, but neuroscience research directed toward understanding these pathways and their roles in behavior is progressing rapidly. This research is revealing striking correspondences between the brain's reward system and the theory of reinforcement learning as presented in this book.

The *reward prediction error hypothesis of dopamine neuron activity* was proposed by scientists who recognized striking parallels between the behavior of TD errors and the activity of neurons that produce dopamine, a neurotransmitter essential in mammals for reward-related learning and behavior. Experiments conducted in the late 1980s and 1990s in the laboratory of neuroscientist Wolfram Schultz showed that dopamine neurons respond to rewarding events with substantial bursts of activity, called phasic responses, only if the animal does not expect those events, suggesting that dopamine neurons are signaling reward prediction errors instead of reward itself. Further, these experiments showed that as an animal learns to predict a rewarding event on the basis of preceding sensory cues, the phasic activity of dopamine neurons shifts to earlier predictive cues while decreasing to later predictive cues. This parallels the backing-up effect of the TD error as a reinforcement learning agent learns to predict reward.

Other experimental results firmly establish that the phasic activity of dopamine neurons is a reinforcement signal for learning that reaches multiple areas of the brain by means of profusely branching axons of dopamine producing neurons. These results are consistent with the distinction we make between a reward signal, R_t, and a reinforcement signal, which is the TD error δ_t in most of the algorithms we present. Phasic responses of dopamine neurons are reinforcement signals, not reward signals.

A prominent hypothesis is that the brain implements something like an actor–critic algorithm. Two structures in the brain (the dorsal and ventral subdivisions of the striatum), both of which play critical roles in reward-based learning, may function respectively like an actor and a critic. That the TD error is the reinforcement signal for both the actor and the critic fits well with the facts that dopamine neuron axons target both the dorsal and ventral subdivisions of the striatum; that dopamine appears to be

critical for modulating synaptic plasticity in both structures; and that the effect on a target structure of a neuromodulator such as dopamine depends on properties of the target structure and not just on properties of the neuromodulator.

The actor and the critic can be implemented by ANNs consisting of neuron-like units having learning rules based on the policy-gradient actor–critic method described in Section 13.5. Each connection in these networks is like a synapse between neurons in the brain, and the learning rules correspond to rules governing how synaptic efficacies change as functions of the activities of the presynaptic and the postsynaptic neurons, together with neuromodulatory input corresponding to input from dopamine neurons. In this setting, each synapse has its own eligibility trace that records past activity involving that synapse. The only difference between the actor and critic learning rules is that they use different kinds of eligibility traces: the critic unit's traces are *non-contingent* because they do not involve the critic unit's output, whereas the actor unit's traces are *contingent* because in addition to the actor unit's input, they depend on the actor unit's output. In the hypothetical implementation of an actor–critic system in the brain, these learning rules respectively correspond to rules governing plasticity of corticostriatal synapses that convey signals from the cortex to the principal neurons in the dorsal and ventral striatal subdivisions, synapses that also receive inputs from dopamine neurons.

The learning rule of an actor unit in the actor–critic network closely corresponds to *reward-modulated spike-timing-dependent plasticity*. In spike-timing-dependent plasticity (STDP), the relative timing of pre- and postsynaptic activity determines the direction of synaptic change. In reward-modulated STDP, changes in synapses in addition depend on a neuromodulator, such as dopamine, arriving within a time window that can last up to 10 seconds after the conditions for STDP are met. Evidence is accumulating that reward-modulated STDP occurs at corticostriatal synapses, where the actor's learning takes place in the hypothetical neural implementation of an actor–critic system, adds to the plausibility of the hypothesis that something like an actor–critic system exists in the brains of some animals.

The idea of synaptic eligibility and basic features of the actor learning rule derive from Klopf's hypothesis of the "hedonistic neuron" (Klopf, 1972, 1981). He conjectured that individual neurons seek to obtain reward and to avoid punishment by adjusting the efficacies of their synapses on the basis of rewarding or punishing consequences of their action potentials. A neuron's activity can affect its later input because the neuron is embedded in many feedback loops, some within the animal's nervous system and body and others passing through the animal's external environment. Klopf's idea of eligibility is that synapses are temporarily marked as eligible for modification if they participated in the neuron's firing (making this the contingent form of eligibility trace). A synapse's efficacy is modified if a reinforcing signal arrives while the synapse is eligible. We alluded to the chemotactic behavior of a bacterium as an example of a single cell that directs its movements in order to seek some molecules and to avoid others.

A conspicuous feature of the dopamine system is that fibers releasing dopamine project widely to multiple parts of the brain. Although it is likely that only some populations of dopamine neurons broadcast the same reinforcement signal, if this signal reaches the synapses of many neurons involved in actor-type learning, then the situation can

be modeled as a *team problem*. In this type of problem, each agent in a collection of reinforcement learning agents receives the same reinforcement signal, where that signal depends on the activities of all members of the collection, or team. If each team member uses a sufficiently capable learning algorithm, the team can learn collectively to improve performance of the entire team as evaluated by the globally-broadcast reinforcement signal, even if the team members do not directly communicate with one another. This is consistent with the wide dispersion of dopamine signals in the brain and provides a neurally plausible alternative to the widely-used error-backpropagation method for training multilayer networks.

The distinction between model-free and model-based reinforcement learning is helping neuroscientists investigate the neural bases of habitual and goal-directed learning and decision making. Research so far points to their being some brain regions more involved in one type of process than the other, but the picture remains unclear because model-free and model-based processes do not appear to be neatly separated in the brain. Many questions remain unanswered. Perhaps most intriguing is evidence that the hippocampus, a structure traditionally associated with spatial navigation and memory, appears to be involved in simulating possible future courses of action as part of an animal's decision-making process. This suggests that it is part of a system that uses an environment model for planning.

Reinforcement learning theory is also influencing thinking about neural processes underlying drug abuse. A model of some features of drug addiction is based on the reward prediction error hypothesis. It proposes that an addicting stimulant, such as cocaine, destabilizes TD learning to produce unbounded growth in the values of actions associated with drug intake. This is far from a complete model of addiction, but it illustrates how a computational perspective suggests theories that can be tested with further research. The new field of computational psychiatry similarly focuses on the use of computational models, some derived from reinforcement learning, to better understand mental disorders.

This chapter only touched the surface of how the neuroscience of reinforcement learning and the development of reinforcement learning in computer science and engineering have influenced one another. Most features of reinforcement learning algorithms owe their design to purely computational considerations, but some have been influenced by hypotheses about neural learning mechanisms. Remarkably, as experimental data has accumulated about the brain's reward processes, many of the purely computationally-motivated features of reinforcement learning algorithms are turning out to be consistent with neuroscience data. Other features of computational reinforcement learning, such as eligibility traces and the ability of teams of reinforcement learning agents to learn to act collectively under the influence of a globally-broadcast reinforcement signal, may also turn out to parallel experimental data as neuroscientists continue to unravel the neural basis of reward-based animal learning and behavior.

Bibliographical and Historical Remarks

The number of publications treating parallels between the neuroscience of learning and decision making and the approach to reinforcement learning presented in this book is enormous. We can cite only a small selection. Niv (2009), Dayan and Niv (2008), Gimcher (2011), Ludvig, Bellemare, and Pearson (2011), and Shah (2012) are good places to start.

Together with economics, evolutionary biology, and mathematical psychology, reinforcement learning theory is helping to formulate quantitative models of the neural mechanisms of choice in humans and non-human primates. With its focus on learning, this chapter only lightly touches upon the neuroscience of decision making. Glimcher (2003) introduced the field of "neuroeconomics," in which reinforcement learning contributes to the study of the neural basis of decision making from an economics perspective. See also Glimcher and Fehr (2013). The text on computational and mathematical modeling in neuroscience by Dayan and Abbott (2001) includes reinforcement learning's role in these approaches. Sterling and Laughlin (2015) examined the neural basis of learning in terms of general design principles that enable efficient adaptive behavior.

15.1 There are many good expositions of basic neuroscience. Kandel, Schwartz, Jessell, Siegelbaum, and Hudspeth (2013) is an authoritative and very comprehensive source.

15.2 Berridge and Kringelbach (2008) reviewed the neural basis of reward and pleasure, pointing out that reward processing has many dimensions and involves many neural systems. Space prevents discussion of the influential research of Berridge and Robinson (1998), who distinguish between the hedonic impact of a stimulus, which they call "liking," and the motivational effect, which they call "wanting." Hare, O'Doherty, Camerer, Schultz, and Rangel (2008) examined the neural basis of value-related signals from an economic perspective, distinguishing between goal values, decision values, and prediction errors. Decision value is goal value minus action cost. See also Rangel, Camerer, and Montague (2008), Rangel and Hare (2010), and Peters and Büchel (2010).

15.3 The reward prediction error hypothesis of dopamine neuron activity is most prominently discussed by Schultz, Dayan, and Montague (1997). The hypothesis was first explicitly put forward by Montague, Dayan, and Sejnowski (1996). As they stated the hypothesis, it referred to reward prediction errors (RPEs) but not specifically to TD errors; however, their development of the hypothesis made it clear that they were referring to TD errors. The earliest recognition of the TD-error/dopamine connection of which we are aware is that of Montague, Dayan, Nowlan, Pouget, and Sejnowski (1993), who proposed a TD-error-modulated Hebbian learning rule motivated by results on dopamine signaling from Schultz's group. The connection was also pointed out in an abstract by Quartz, Dayan, Montague, and Sejnowski (1992). Montague and Sejnowski (1994) emphasized the importance of prediction in the brain and outlined how predictive Hebbian learning modulated by TD errors could be implemented via a diffuse neuromodulatory system, such as the dopamine system. Friston, Tononi, Reeke, Sporns, and Edelman (1994) presented a model of value-dependent learning in the brain in which synaptic changes are mediated by a TD-like error provided by a global neuromodulatory signal (although they did not single out dopamine). Montague, Dayan, Person, and Sejnowski (1995) presented a model of honeybee foraging using the TD error. The model is based on research by Hammer, Menzel, and colleagues (Hammer and Menzel, 1995; Hammer, 1997) showing that the neuromodulator octopamine acts as a reinforcement signal in the honeybee. Montague et al. (1995) pointed out that dopamine likely plays a similar role in the vertebrate brain. Barto (1995a) related the actor–critic architecture to basal-ganglionic circuits and discussed the relationship between TD learning and the main results from Schultz's group. Houk, Adams, and Barto (1995) suggested how TD learning and the actor–critic architecture might map onto the anatomy, physiology, and molecular mechanism of the basal ganglia. Doya and Sejnowski (1998) extended their earlier paper on a model of birdsong learning (Doya and Sejnowski, 1995) by including a TD-like error identified with dopamine to reinforce the selection of auditory input to be memorized. O'Reilly and Frank (2006) and O'Reilly, Frank, Hazy, and Watz (2007) argued that phasic dopamine signals are RPEs but not TD errors. In support of their theory they cited results with variable interstimulus intervals that do not match predictions of a simple TD model, as well as the observation that higher-order conditioning beyond second-order conditioning is rarely observed, while TD learning is not so limited. Dayan and Niv (2008) discussed "the good, the bad, and the ugly" of how reinforcement learning theory and the reward prediction error hypothesis align with experimental data. Glimcher (2011) reviewed the empirical findings that support the reward prediction error hypothesis and emphasized the significance of the hypothesis for contemporary neuroscience.

15.4 Graybiel (2000) is a brief primer on the basal ganglia. The experiments mentioned that involve optogenetic activation of dopamine neurons were conducted by Tsai, Zhang, Adamantidis, Stuber, Bonci, de Lecea, and Deisseroth (2009), Steinberg, Keiflin, Boivin, Witten, Deisseroth, and Janak (2013), and Claridge-Chang, Roorda, Vrontou, Sjulson, Li, Hirsh, and Miesenböck (2009). Fiorillo, Yun, and Song (2013), Lammel, Lim, and Malenka (2014), and Saddoris, Cacciapaglia, Wightmman, and Carelli (2015) are among studies showing that the signaling properties of dopamine neurons are specialized for different target regions. RPE-signaling neurons may belong to one among multiple populations of dopamine neurons having different targets and subserving different functions. Eshel, Tian, Bukwich, and Uchida (2016) found homogeneity of reward prediction error responses of dopamine neurons in the lateral VTA during classical conditioning in mice, though their results do not rule out response diversity across wider areas. Gershman, Pesaran, and Daw (2009) studied reinforcement learning tasks that can be decomposed into independent components with separate reward signals, finding evidence in human neuroimaging data suggesting that the brain exploits this kind of structure.

15.5 Schultz's 1998 survey article is a good entrée into the very extensive literature on reward predicting signaling of dopamine neurons. Berns, McClure, Pagnoni, and Montague (2001), Breiter, Aharon, Kahneman, Dale, and Shizgal (2001), Pagnoni, Zink, Montague, and Berns (2002), and O'Doherty, Dayan, Friston, Critchley, and Dolan (2003) described functional brain imaging studies supporting the existence of signals like TD errors in the human brain.

15.6 This section roughly follows Barto (1995a) in explaining how TD errors mimic the main results from Schultz's group on the phasic responses of dopamine neurons.

15.7 This section is largely based on Takahashi, Schoenbaum, and Niv (2008) and Niv (2009). To the best of our knowledge, Barto (1995a) and Houk, Adams, and Barto (1995) first speculated about possible implementations of actor–critic algorithms in the basal ganglia. On the basis of functional magnetic resonance imaging of human subjects while engaged in instrumental conditioning, O'Doherty, Dayan, Schultz, Deichmann, Friston, and Dolan (2004) suggested that the actor and the critic are most likely located respectively in the dorsal and ventral striatum. Gershman, Moustafa, and Ludvig (2014) focused on how time is represented in reinforcement learning models of the basal ganglia, discussing evidence for, and implications of, various computational approaches to time representation.

The hypothetical neural implementation of the actor–critic architecture described in this section includes very little detail about known basal ganglia anatomy and physiology. In addition to the more detailed hypothesis of Houk, Adams, and Barto (1995), a number of other hypotheses include more specific connections to anatomy and physiology and are claimed to explain additional data. These include hypotheses proposed by Suri and Schultz (1998, 1999), Brown, Bullock, and Grossberg (1999), Contreras-Vidal and Schultz (1999), Suri, Bargas, and Arbib (2001), O'Reilly and Frank (2006), and O'Reilly, Frank, Hazy, and Watz (2007). Joel, Niv, and Ruppin (2002) critically evaluated the anatomical plausibility of several of these models and present an alternative intended to accommodate some neglected features of basal ganglionic circuitry.

15.8 The actor learning rule discussed here is more complicated than the one in the early actor–critic network of Barto et al. (1983). Actor-unit eligibility traces in that network were traces of just $A_t \times \mathbf{x}(S_t)$ instead of the full $(A_t - \pi(A_t|S_t, \boldsymbol{\theta}))\mathbf{x}(S_t)$. That work did not benefit from the policy-gradient theory presented in Chapter 13 or the contributions of Williams (1986, 1992), who showed how an ANN of Bernoulli-logistic units could implement a policy-gradient method.

Reynolds and Wickens (2002) proposed a three-factor rule for synaptic plasticity in the corticostriatal pathway in which dopamine modulates changes in corticostriatal synaptic efficacy. They discussed the experimental support for this kind of learning rule and its possible molecular basis. The definitive demonstration of spike-timing-dependent plasticity (STDP) is attributed to Markram, Lübke, Frotscher, and Sakmann (1997), with evidence from earlier experiments

by Levy and Steward (1983) and others that the relative timing of pre- and postsynaptic spikes is critical for inducing changes in synaptic efficacy. Rao and Sejnowski (2001) suggested how STDP could be the result of a TD-like mechanism at synapses with non-contingent eligibility traces lasting about 10 milliseconds. Dayan (2002) commented that this would require an error as in Sutton and Barto's (1981a) early model of classical conditioning and not a true TD error. Representative publications from the extensive literature on reward-modulated STDP are Wickens (1990), Reynolds and Wickens (2002), and Calabresi, Picconi, Tozzi and Di Filippo (2007). Pawlak and Kerr (2008) showed that dopamine is necessary to induce STDP at the corticostriatal synapses of medium spiny neurons. See also Pawlak, Wickens, Kirkwood, and Kerr (2010). Yagishita, Hayashi-Takagi, Ellis-Davies, Urakubo, Ishii, and Kasai (2014) found that dopamine promotes spine enlargement of the medium spiny neurons of mice only during a time window of from 0.3 to 2 seconds after STDP stimulation. Izhikevich (2007) proposed and explored the idea of using STDP timing conditions to trigger contingent eligibility traces. Frémaux, Sprekeler, and Gerstner (2010) proposed theoretical conditions for successful learning by rules based on reward-modulated STDP.

15.9 Klopf's hedonistic neuron hypothesis (Klopf 1972, 1982) inspired our actor–critic algorithm implemented as an ANN with a single neuron-like unit, called the actor unit, implementing a Law-of-Effect-like learning rule (Barto, Sutton, and Anderson, 1983). Ideas related to Klopf's synaptically-local eligibility have been proposed by others. Crow (1968) proposed that changes in the synapses of cortical neurons are sensitive to the consequences of neural activity. Emphasizing the need to address the time delay between neural activity and its consequences in a reward-modulated form of synaptic plasticity, he proposed a contingent form of eligibility, but associated with entire neurons instead of individual synapses. According to his hypothesis, a wave of neuronal activity

> leads to a short-term change in the cells involved in the wave such that they are picked out from a background of cells not so activated. ... such cells are rendered sensitive by the short-term change to a reward signal ... in such a way that if such a signal occurs before the end of the decay time of the change the synaptic connexions between the cells are made more effective. (Crow, 1968)

Crow argued against previous proposals that reverberating neural circuits play this role by pointing out that the effect of a reward signal on such a circuit would "...establish the synaptic connexions leading to the reverberation (that is to say, those involved in activity at the time of the reward signal) and not those on the path which led to the adaptive motor output." Crow further postulated that reward signals are delivered via a "distinct neural fiber system," presumably the one into which Olds and Milner (1954) tapped, that would transform synaptic connections "from a short into a long-term form."

In another farsighted hypothesis, Miller proposed a Law-of-Effect-like learning rule that includes synaptically-local contingent eligibility traces:

> ... it is envisaged that in a particular sensory situation neurone B, by chance, fires a 'meaningful burst' of activity, which is then translated into motor acts, which then change the situation. It must be supposed that the meaningful burst has an influence, *at the neuronal level*, on all of its own synapses which are active at the time ... thereby making a preliminary selection of the synapses to be strengthened, though not yet actually strengthening them. ...The strengthening signal ... makes the final selection ... and accomplishes the definitive change in the appropriate synapses. (Miller, 1981, p. 81)

Miller's hypothesis also included a critic-like mechanism, which he called a "sensory analyzer unit," that worked according to classical conditioning principles to provide reinforcement signals to neurons so that they would learn to move from lower- to higher-valued states, thus anticipating the use of the TD error as a reinforcement signal in the actor–critic architecture. Miller's idea not only parallels Klopf's (with the exception of its explicit invocation of a distinct "strengthening signal"), it also anticipated the general features of reward-modulated STDP.

A related though different idea, which Seung (2003) called the "hedonistic synapse," is that synapses individually adjust the probability that they release neurotransmitter in the manner of the Law of Effect: if reward follows release, the release probability increases, and decreases if reward follows failure to release. This is essentially the same as the learning scheme Minsky used in his 1954 Princeton Ph.D. dissertation, where he called the synapse-like learning element a SNARC (Stochastic Neural-Analog Reinforcement Calculator). Contingent eligibility is involved in these ideas too, although it is contingent on the activity of an individual synapse instead of the postsynaptic neuron. Also related is the proposal of Unnikrishnan and Venugopal (1994) that uses the correlation-based method of Harth and Tzanakou (1974) to adjust ANN weights.

Frey and Morris (1997) proposed the idea of a "synaptic tag" for the induction of long-lasting strengthening of synaptic efficacy. Though not unlike Klopf's eligibility, their tag was hypothesized to consist of a temporary strengthening of a synapse that could be transformed into a long-lasting strengthening by subsequent neuron activation. The model of O'Reilly and Frank (2006) and O'Reilly, Frank, Hazy, and Watz (2007) uses working memory to bridge temporal intervals instead of eligibility traces. Wickens and Kotter (1995) discuss possible mechanisms for synaptic eligibility. He, Huertas, Hong, Tie, Hell, Shouval, Kirkwood (2015) provide evidence supporting the existence of contingent eligibility traces in synapses of cortical neurons with time courses like those of the eligibility traces Klopf postulated.

The metaphor of a neuron using a learning rule related to bacterial chemotaxis was discussed by Barto (1989). Koshland's extensive study of bacterial chemotaxis was in part motivated by similarities between features of bacteria and features of neurons (Koshland, 1980). See also Berg (1975). Shimansky (2009) proposed a

synaptic learning rule somewhat similar to Seung's mentioned above in which each synapse individually acts like a chemotactic bacterium. In this case a collection of synapses "swims" toward attractants in the high-dimensional space of synaptic weight values. Montague, Dayan, Person, and Sejnowski (1995) proposed a chemotactic-like model of the bee's foraging behavior involving the neuromodulator octopamine.

15.10 Research on the behavior of reinforcement learning agents in team and game problems has a long history roughly occurring in three phases. To the best of our knowledge, the first phase began with investigations by the Russian mathematician and physicist M. L. Tsetlin. A collection of his work was published as Tsetlin (1973) after his death in 1966. Our Sections 1.7 and 4.8 refer to his study of learning automata in connection to bandit problems. The Tsetlin collection also includes studies of learning automata in team and game problems, which led to later work in this area using stochastic learning automata as described by Narendra and Thathachar (1974, 1989), Viswanathan and Narendra (1974), Lakshmivarahan and Narendra (1982), Narendra and Wheeler (1983), and Thathachar and Sastry (2002). Thathachar and Sastry (2011) is a more recent comprehensive account. These studies were mostly restricted to non-associative learning automata, meaning that they did not address associative, or contextual, bandit problems (Section 2.9).

The second phase began with the extension of learning automata to the associative, or contextual, case. Barto, Sutton, and Brouwer (1981) and Barto and Sutton (1981b) experimented with associative stochastic learning automata in single-layer ANNs to which a global reinforcement signal was broadcast. The learning algorithm was an associative extension of the Alopex algorithm of Harth and Tzanakou (1974). Barto et al. called neuron-like elements implementing this kind of learning *associative search elements* (ASEs). Barto and Anandan (1985) introduced an associative reinforcement learning algorithm called the *associative reward-penalty* (A_{R-P}) algorithm. They proved a convergence result by combining theory of stochastic learning automata with theory of pattern classification. Barto (1985, 1986) and Barto and Jordan (1987) described results with teams of A_{R-P} units connected into multi-layer ANNs, showing that they could learn nonlinear functions, such as XOR and others, with a globally-broadcast reinforcement signal. Barto (1985) extensively discussed this approach to ANNs and how this type of learning rule is related to others in the literature at that time. Williams (1992) mathematically analyzed and broadened this class of learning rules and related their use to the error backpropagation method for training multilayer ANNs. Williams (1988) described several ways that backpropagation and reinforcement learning can be combined for training ANNs. Williams (1992) showed that a special case of the A_{R-P} algorithm is a REINFORCE algorithm, although better results were obtained with the general A_{R-P} algorithm (Barto, 1985).

The third phase of interest in teams of reinforcement learning agents was influenced by increased understanding of the role of dopamine as a widely broadcast neuromodulator and speculation about the existence of reward-modulated STDP. Much more so than earlier research, this research considers details of synaptic plasticity and other constraints from neuroscience. Publications include the following (chronologically and alphabetically): Bartlett and Baxter (1999, 2000), Xie and Seung (2004), Baras and Meir (2007), Farries and Fairhall (2007), Florian (2007), Izhikevich (2007), Pecevski, Maass, and Legenstein (2008), Legenstein, Pecevski, and Maass (2008), Kolodziejski, Porr, and Wörgötter (2009), Urbanczik and Senn (2009), and Vasilaki, Frémaux, Urbanczik, Senn, and Gerstner (2009). Nowé, Vrancx, and De Hauwere (2012) reviewed more recent developments in the wider field of multi-agent reinforcement learning

15.11 Yin and Knowlton (2006) reviewed findings from outcome-devaluation experiments with rodents supporting the view that habitual and goal-directed behavior (as psychologists use the phrase) are respectively most associated with processing in the dorsolateral striatum (DLS) and the dorsomedial striatum (DMS). Results of functional imaging experiments with human subjects in the outcome-devaluation setting by Valentin, Dickinson, and O'Doherty (2007) suggest that the orbitofrontal cortex (OFC) is an important component of goal-directed choice. Single unit recordings in monkeys by Padoa-Schioppa and Assad (2006) support the role of the OFC in encoding values guiding choice behavior. Rangel, Camerer, and Montague (2008) and Rangel and Hare (2010) reviewed findings from the perspective of neuroeconomics about how the brain makes goal-directed decisions. Pezzulo, van der Meer, Lansink, and Pennartz (2014) reviewed the neuroscience of internally generated sequences and presented a model of how these mechanisms might be components of model-based planning. Daw and Shohamy (2008) proposed that while dopamine signaling connects well to habitual, or model-free, behavior, other processes are involved in goal-directed, or model-based, behavior. Data from experiments by Bromberg-Martin, Matsumoto, Hong, and Hikosaka (2010) indicate that dopamine signals contain information pertinent to both habitual and goal-directed behavior. Doll, Simon, and Daw (2012) argued that there may not a clear separation in the brain between mechanisms that subserve habitual and goal-directed learning and choice.

15.12 Keiflin and Janak (2015) reviewed connections between TD errors and addiction. Nutt, Lingford-Hughes, Erritzoe, and Stokes (2015) critically evaluated the hypothesis that addiction is due to a disorder of the dopamine system. Montague, Dolan, Friston, and Dayan (2012) outlined the goals and early efforts in the field of computational psychiatry, and Adams, Huys, and Roiser (2015) reviewed more recent progress.

Chapter 16

Applications and Case Studies

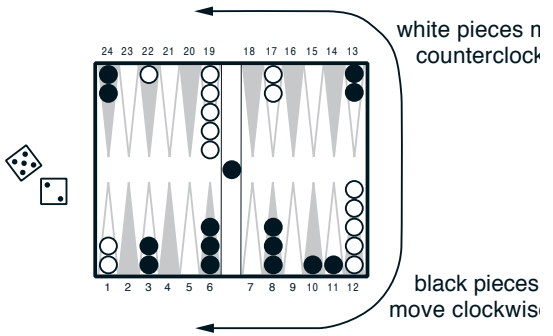
In this chapter we present a few case studies of reinforcement learning. Several of these are substantial applications of potential economic significance. One, Samuel's checkers player, is primarily of historical interest. Our presentations are intended to illustrate some of the trade-offs and issues that arise in real applications. For example, we emphasize how domain knowledge is incorporated into the formulation and solution of the problem. We also highlight the representation issues that are so often critical to successful applications. The algorithms used in some of these case studies are substantially more complex than those we have presented in the rest of the book. Applications of reinforcement learning are still far from routine and typically require as much art as science. Making applications easier and more straightforward is one of the goals of current research in reinforcement learning.

16.1 TD-Gammon

One of the most impressive applications of reinforcement learning to date is that by Gerald Tesauro to the game of backgammon (Tesauro, 1992, 1994, 1995, 2002). Tesauro's program, *TD-Gammon*, required little backgammon knowledge, yet learned to play extremely well, near the level of the world's strongest grandmasters. The learning algorithm in TD-Gammon was a straightforward combination of the TD(λ) algorithm and nonlinear function approximation using a multilayer artificial neural network (ANN) trained by backpropagating TD errors.

Backgammon is a major game in the sense that it is played throughout the world, with numerous tournaments and regular world championship matches. It is in part a game of chance, and it is a popular vehicle for waging significant sums of money. There are probably more professional backgammon players than there are professional chess players. The game is played with 15 white and 15 black pieces on a board of 24 locations, called *points*. To the right on the next page is shown a typical position early in the game, seen from the perspective of the white player. White here has just rolled the dice and obtained a 5 and a 2. This means that he can move one of his pieces 5 steps and one

(possibly the same piece) 2 steps. For example, he could move two pieces from the 12 point, one to the 17 point, and one to the 14 point. White's objective is to advance all of his pieces into the last quadrant (points 19–24) and then off the board. The first player to remove all his pieces wins. One complication is that the pieces interact as they pass each other going in different directions. For example, if it were black's move, he could use the dice roll of 2 to move a piece from the 24 point to the 22 point, "hitting" the white piece there. Pieces that have been hit are placed on the "bar" in the middle of the board (where we already see one previously hit black piece), from whence they reenter the race from the start. However, if there are two pieces on a point, then the opponent cannot move to that point; the pieces are protected from being hit. Thus, white cannot use his 5–2 dice roll to move either of his pieces on the 1 point, because their possible resulting points are occupied by groups of black pieces. Forming contiguous blocks of occupied points to block the opponent is one of the elementary strategies of the game.



A backgammon position

Backgammon involves several further complications, but the above description gives the basic idea. With 30 pieces and 24 possible locations (26, counting the bar and off-the-board) it should be clear that the number of possible backgammon positions is enormous, far more than the number of memory elements one could have in any physically realizable computer. The number of moves possible from each position is also large. For a typical dice roll there might be 20 different ways of playing. In considering future moves, such as the response of the opponent, one must consider the possible dice rolls as well. The result is that the game tree has an effective branching factor of about 400. This is far too large to permit effective use of the conventional heuristic search methods that have proved so effective in games like chess and checkers.

On the other hand, the game is a good match to the capabilities of TD learning methods. Although the game is highly stochastic, a complete description of the game's state is available at all times. The game evolves over a sequence of moves and positions until finally ending in a win for one player or the other, ending the game. The outcome can be interpreted as a final reward to be predicted. On the other hand, the theoretical results we have described so far cannot be usefully applied to this task. The number of states is so large that a lookup table cannot be used, and the opponent is a source of uncertainty and time variation.

TD-Gammon used a nonlinear form of TD(λ). The estimated value, $\hat{v}(s,\mathbf{w})$, of any state (board position) s was meant to estimate the probability of winning starting from state s. To achieve this, rewards were defined as zero for all time steps except those on which the game is won. To implement the value function, TD-Gammon used a standard multilayer ANN, much like that shown to the right on the next page. (The real

network had two additional units in its final layer to estimate the probability of each player's winning in a special way called a "gammon" or "backgammon.") The network consisted of a layer of input units, a layer of hidden units, and a final output unit. The input to the network was a representation of a backgammon position, and the output was an estimate of the value of that position.

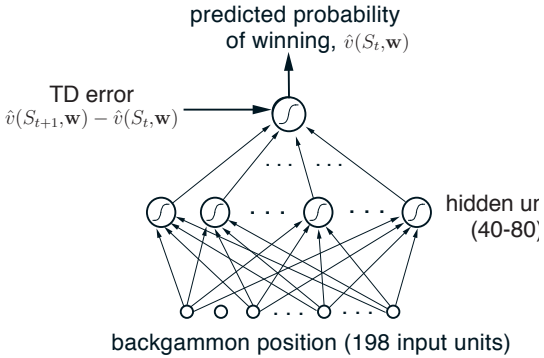


Figure 16.1: The TD-Gammon ANN

In the first version of TD-Gammon, TD-Gammon 0.0, backgammon positions were represented to the network in a relatively direct way that involved little backgammon knowledge. It did, however, involve substantial knowledge of how ANNs work and how information is best presented to them. It is instructive to note the exact representation Tesauro chose. There were a total of 198 input units to the network. For each point on the backgammon board, four units indicated the number of white pieces on the point. If there were no white pieces, then all four units took on the value zero. If there was one piece, then the first unit took on the value 1. This encoded the elementary concept of a "blot," i.e., a piece that can be hit by the opponent. If there were two or more pieces, then the second unit was set to 1. This encoded the basic concept of a "made point" on which the opponent cannot land. If there were exactly three pieces on the point, then the third unit was set to 1. This encoded the basic concept of a "single spare," i.e., an extra piece in addition to the two pieces that made the point. Finally, if there were more than three pieces, the fourth unit was set to a value proportionate to the number of additional pieces beyond three. Letting n denote the total number of pieces on the point, if $n > 3$, then the fourth unit took on the value $(n-3)/2$. This encoded a linear representation of "multiple spares" at the given point.

With four units for white and four for black at each of the 24 points, that made a total of 192 units. Two additional units encoded the number of white and black pieces on the bar (each took the value $n/2$, where n is the number of pieces on the bar), and two more encoded the number of black and white pieces already successfully removed from the board (these took the value $n/15$, where n is the number of pieces already borne off). Finally, two units indicated in a binary fashion whether it was white's or black's turn to move. The general logic behind these choices should be clear. Basically, Tesauro tried to represent the position in a straightforward way, while keeping the number of units relatively small. He provided one unit for each conceptually distinct possibility that seemed likely to be relevant, and he scaled them to roughly the same range, in this case between 0 and 1.

Given a representation of a backgammon position, the network computed its estimated value in the standard way. Corresponding to each connection from an input unit to a hidden unit was a real-valued weight. Signals from each input unit were multiplied by

their corresponding weights and summed at the hidden unit. The output, $h(j)$, of hidden unit j was a nonlinear sigmoid function of the weighted sum:

$$h(j) = \sigma\left(\sum_i w_{ij}x_i\right) = \frac{1}{1+e^{-\sum_i w_{ij}x_i}},$$

where x_i is the value of the ith input unit and w_{ij} is the weight of its connection to the jth hidden unit (all the weights in the network together make up the parameter vector $\mathbf{w}$). The output of the sigmoid is always between 0 and 1, and has a natural interpretation as a probability based on a summation of evidence. The computation from hidden units to the output unit was entirely analogous. Each connection from a hidden unit to the output unit had a separate weight. The output unit formed the weighted sum and then passed it through the same sigmoid nonlinearity.

TD-Gammon used the semi-gradient form of the TD(λ) algorithm described in Section 12.2, with the gradients computed by the error backpropagation algorithm (Rumelhart, Hinton, and Williams, 1986). Recall that the general update rule for this case is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]\mathbf{z}_t, \tag{16.1}$$

where $\mathbf{w}_t$ is the vector of all modifiable parameters (in this case, the weights of the network) and $\mathbf{z}_t$ is a vector of eligibility traces, one for each component of $\mathbf{w}_t$, updated by

$$\mathbf{z}_t \doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t,\mathbf{w}_t),$$

with $\mathbf{z}_0 \doteq \mathbf{0}$. The gradient in this equation can be computed efficiently by the backpropagation procedure. For the backgammon application, in which $\gamma = 1$ and the reward is always zero except upon winning, the TD error portion of the learning rule is usually just $\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})$, as suggested in Figure 16.1.

To apply the learning rule we need a source of backgammon games. Tesauro obtained an unending sequence of games by playing his learning backgammon player against itself. To choose its moves, TD-Gammon considered each of the 20 or so ways it could play its dice roll and the corresponding positions that would result. The resulting positions are *afterstates* as discussed in Section 6.8. The network was consulted to estimate each of their values. The move was then selected that would lead to the position with the highest estimated value. Continuing in this way, with TD-Gammon making the moves for both sides, it was possible to easily generate large numbers of backgammon games. Each game was treated as an episode, with the sequence of positions acting as the states, $S_0, S_1, S_2, \ldots$. Tesauro applied the nonlinear TD rule (16.1) fully incrementally, that is, after each individual move.

The weights of the network were set initially to small random values. The initial evaluations were thus entirely arbitrary. Because the moves were selected on the basis of these evaluations, the initial moves were inevitably poor, and the initial games often lasted hundreds or thousands of moves before one side or the other won, almost by accident. After a few dozen games however, performance improved rapidly.

After playing about 300,000 games against itself, TD-Gammon 0.0 as described above learned to play approximately as well as the best previous backgammon computer

programs. This was a striking result because all the previous high-performance computer programs had used extensive backgammon knowledge. For example, the reigning champion program at the time was, arguably, *Neurogammon*, another program written by Tesauro that used an ANN but not TD learning. Neurogammon's network was trained on a large training corpus of exemplary moves provided by backgammon experts, and, in addition, started with a set of features specially crafted for backgammon. Neurogammon was a highly tuned, highly effective backgammon program that decisively won the World Backgammon Olympiad in 1989. TD-Gammon 0.0, on the other hand, was constructed with essentially zero backgammon knowledge. That it was able to do as well as Neurogammon and all other approaches is striking testimony to the potential of self-play learning methods.

The tournament success of TD-Gammon 0.0 with zero expert backgammon knowledge suggested an obvious modification: add the specialized backgammon features but keep the self-play TD learning method. This produced TD-Gammon 1.0. TD-Gammon 1.0 was clearly substantially better than all previous backgammon programs and found serious competition only among human experts. Later versions of the program, TD-Gammon 2.0 (40 hidden units) and TD-Gammon 2.1 (80 hidden units), were augmented with a selective two-ply search procedure. To select moves, these programs looked ahead not just to the positions that would immediately result, but also to the opponent's possible dice rolls and moves. Assuming the opponent always took the move that appeared immediately best for him, the expected value of each candidate move was computed and the best was selected. To save computer time, the second ply of search was conducted only for candidate moves that were ranked highly after the first ply, about four or five moves on average. Two-ply search affected only the moves selected; the learning process proceeded exactly as before. The final versions of the program, TD-Gammon 3.0 and 3.1, used 160 hidden units and a selective three-ply search. TD-Gammon illustrates the combination of learned value functions and decision-time search as in heuristic search and MCTS methods. In follow-on work, Tesauro and Galperin (1997) explored trajectory sampling methods as an alternative to full-width search, which reduced the error rate of live play by large numerical factors (4x–6x) while keeping the think time reasonable at ∼5–10 seconds per move.

During the 1990s, Tesauro was able to play his programs in a significant number of games against world-class human players. A summary of the results is given in Table 16.1.

Program	Hidden Units	Training Games	Opponents	Results
TD-Gammon 0.0	40	300,000	other programs	tied for best
TD-Gammon 1.0	80	300,000	Robertie, Magriel, ...	−13 pts / 51 games
TD-Gammon 2.0	40	800,000	various Grandmasters	−7 pts / 38 games
TD-Gammon 2.1	80	1,500,000	Robertie	−1 pt / 40 games
TD-Gammon 3.0	80	1,500,000	Kazaros	+6 pts / 20 games

Table 16.1: Summary of TD-Gammon Results

Based on these results and analyses by backgammon grandmasters (Robertie, 1992; see Tesauro, 1995), TD-Gammon 3.0 appeared to play at close to, or possibly better than, the playing strength of the best human players in the world. Tesauro reported in a subsequent article (Tesauro, 2002) the results of an extensive rollout analysis of the move decisions and doubling decisions of TD-Gammon relative to top human players. The conclusion was that TD-Gammon 3.1 had a "lopsided advantage" in piece-movement decisions, and a "slight edge" in doubling decisions, over top humans.

TD-Gammon had a significant impact on the way the best human players play the game. For example, it learned to play certain opening positions differently than was the convention among the best human players. Based on TD-Gammon's success and further analysis, the best human players now play these positions as TD-Gammon does (Tesauro, 1995). The impact on human play was greatly accelerated when several other self-teaching ANN backgammon programs inspired by TD-Gammon, such as Jellyfish, Snowie, and GNUBackgammon, became widely available. These programs enabled wide dissemination of new knowledge generated by the ANNs, resulting in great improvements in the overall caliber of human tournament play (Tesauro, 2002).

16.2 Samuel's Checkers Player

An important precursor to Tesauro's TD-Gammon was the seminal work of Arthur Samuel (1959, 1967) in constructing programs for learning to play checkers. Samuel was one of the first to make effective use of heuristic search methods and of what we would now call temporal-difference learning. His checkers players are instructive case studies in addition to being of historical interest. We emphasize the relationship of Samuel's methods to modern reinforcement learning methods and try to convey some of Samuel's motivation for using them.

Samuel first wrote a checkers-playing program for the IBM 701 in 1952. His first *learning* program was completed in 1955 and was demonstrated on television in 1956. Later versions of the program achieved good, though not expert, playing skill. Samuel was attracted to game-playing as a domain for studying machine learning because games are less complicated than problems "taken from life" while still allowing fruitful study of how heuristic procedures and learning can be used together. He chose to study checkers instead of chess because its relative simplicity made it possible to focus more strongly on learning.

Samuel's programs played by performing a lookahead search from each current position. They used what we now call heuristic search methods to determine how to expand the search tree and when to stop searching. The terminal board positions of each search were evaluated, or "scored," by a value function, or "scoring polynomial," using linear function approximation. In this and other respects Samuel's work seems to have been inspired by the suggestions of Shannon (1950). In particular, Samuel's program was based on Shannon's minimax procedure to find the best move from the current position. Working backward through the search tree from the scored terminal positions, each position was given the score of the position that would result from the best move, assuming that the machine would always try to maximize the score, while the opponent would always try to

minimize it. Samuel called this the "backed-up score" of the position. When the minimax procedure reached the search tree's root—the current position—it yielded the best move under the assumption that the opponent would be using the same evaluation criterion, shifted to its point of view. Some versions of Samuel's programs used sophisticated search control methods analogous to what are known as "alpha-beta" cutoffs (e.g., see Pearl, 1984).

Samuel used two main learning methods, the simplest of which he called *rote learning*. It consisted simply of saving a description of each board position encountered during play together with its backed-up value determined by the minimax procedure. The result was that if a position that had already been encountered were to occur again as a terminal position of a search tree, the depth of the search was effectively amplified because this position's stored value cached the results of one or more searches conducted earlier. One initial problem was that the program was not encouraged to move along the most direct path to a win. Samuel gave it a "a sense of direction" by decreasing a position's value a small amount each time it was backed up a level (called a ply) during the minimax analysis. "If the program is now faced with a choice of board positions whose scores differ only by the ply number, it will automatically make the most advantageous choice, choosing a low-ply alternative if winning and a high-ply alternative if losing" (Samuel, 1959, p. 80). Samuel found this discounting-like technique essential to successful learning. Rote learning produced slow but continual improvement that was most effective for opening and endgame play. His program became a "better-than-average novice" after learning from many games against itself, a variety of human opponents, and from book games in a supervised learning mode.

Rote learning and other aspects of Samuel's work strongly suggest the essential idea of temporal-difference learning—that the value of a state should equal the value of likely following states. Samuel came closest to this idea in his second learning method, his "learning by generalization" procedure for modifying the parameters of the value function. Samuel's method was the same in concept as that used much later by Tesauro in TD-Gammon. He played his program many games against another version of itself and performed an update after each move. The idea of Samuel's update is suggested by the backup diagram in Figure 16.2. Each open circle represents a position where the program moves next, an *on-move* position, and each solid circle represents a position where the opponent moves next. An update was made to the value of each on-move position after a move by each side, resulting in a second on-move position. The update was toward the minimax value of a search launched from the second on-move position. Thus, the overall effect was that of a backing-up over one full move of real events and then a search over possible events, as suggested by Figure 16.2. Samuel's actual algorithm was significantly more complex than this for computational reasons, but this was the basic idea.

Samuel did not include explicit rewards. Instead, he fixed the weight of the most important feature, the *piece advantage* feature, which measured the number of pieces the program had relative to how many its opponent had, giving higher weight to kings, and including refinements so that it was better to trade pieces when winning than when losing. Thus, the goal of Samuel's program was to improve its piece advantage, which in checkers is highly correlated with winning.

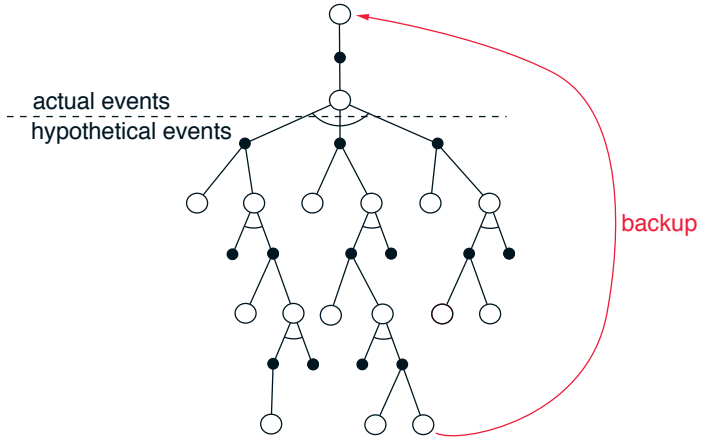


Figure 16.2: The backup diagram for Samuel's checkers player.

However, Samuel's learning method may have been missing an essential part of a sound temporal-difference algorithm. Temporal-difference learning can be viewed as a way of making a value function consistent with itself, and this we can clearly see in Samuel's method. But also needed is a way of tying the value function to the true value of the states. We have enforced this via rewards and by discounting or giving a fixed value to the terminal state. But Samuel's method included no rewards and no special treatment of the terminal positions of games. As Samuel himself pointed out, his value function could have become consistent merely by giving a constant value to all positions. He hoped to discourage such solutions by giving his piece-advantage term a large, nonmodifiable weight. But although this may decrease the likelihood of finding useless evaluation functions, it does not prohibit them. For example, a constant function could still be attained by setting the modifiable weights so as to cancel the effect of the nonmodifiable one.

Because Samuel's learning procedure was not constrained to find useful evaluation functions, it should have been possible for it to become worse with experience. In fact, Samuel reported observing this during extensive self-play training sessions. To get the program improving again, Samuel had to intervene and set the weight with the largest absolute value back to zero. His interpretation was that this drastic intervention jarred the program out of local optima, but another possibility is that it jarred the program out of evaluation functions that were consistent but had little to do with winning or losing the game.

Despite these potential problems, Samuel's checkers player using the generalization learning method approached "better-than-average" play. Fairly good amateur opponents characterized it as "tricky but beatable" (Samuel, 1959). In contrast to the rote-learning version, this version was able to develop a good middle game but remained weak in opening and endgame play. This program also included an ability to search through sets of features to find those that were most useful in forming the value function. A later version (Samuel, 1967) included refinements in its search procedure, such as alpha-beta pruning,

extensive use of a supervised learning mode called "book learning," and hierarchical lookup tables called signature tables (Griffith, 1966) to represent the value function instead of linear function approximation. This version learned to play much better than the 1959 program, though still not at a master level. Samuel's checkers-playing program was widely recognized as a significant achievement in artificial intelligence and machine learning.

16.3 Watson's Daily-Double Wagering

IBM Watson[1] is the system developed by a team of IBM researchers to play the popular TV quiz show *Jeopardy!*.[2] It gained fame in 2011 by winning first prize in an exhibition match against human champions. Although the main technical achievement demonstrated by Watson was its ability to quickly and accurately answer natural language questions over broad areas of general knowledge, its winning *Jeopardy!* performance also relied on sophisticated decision-making strategies for critical parts of the game. Tesauro, Gondek, Lechner, Fan, and Prager (2012, 2013) adapted Tesauro's TD-Gammon system described above to create the strategy used by Watson in "Daily-Double" (DD) wagering in its celebrated winning performance against human champions. These authors report that the effectiveness of this wagering strategy went well beyond what human players are able to do in live game play, and that it, along with other advanced strategies, was an important contributor to Watson's impressive winning performance. Here we focus only on DD wagering because it is the component of Watson that owes the most to reinforcement learning.

Jeopardy! is played by three contestants who face a board showing 30 squares, each of which hides a clue and has a dollar value. The squares are arranged in six columns, each corresponding to a different category. A contestant selects a square, the host reads the square's clue, and each contestant may choose to respond to the clue by sounding a buzzer ("buzzing in"). The first contestant to buzz in gets to try responding to the clue. If this contestant's response is correct, their score increases by the dollar value of the square; if their response is not correct, or if they do not respond within five seconds, their score decreases by that amount, and the other contestants get a chance to buzz in to respond to the same clue. One or two squares (depending on the game's current round) are special DD squares. A contestant who selects one of these gets an exclusive opportunity to respond to the square's clue and has to decide—before the clue is revealed—on how much to wager, or bet. The bet has to be greater than five dollars but not greater than the contestant's current score. If the contestant responds correctly to the DD clue, their score increases by the bet amount; otherwise it decreases by the bet amount. At the end of each game is a "Final Jeopardy" (FJ) round in which each contestant writes down a sealed bet and then writes an answer after the clue is read. The contestant with the highest score after three rounds of play (where a round consists of revealing all 30 clues) is the winner. The game has many other details, but these are enough to appreciate

[1] Registered trademark of IBM Corp.

[2] Registered trademark of Jeopardy Productions Inc.

the importance of DD wagering. Winning or losing often depends on a contestant's DD wagering strategy.

Whenever Watson selected a DD square, it chose its bet by comparing action values, $\hat{q}(s, bet)$, that estimated the probability of a win from the current game state, s, for each round-dollar legal bet. Except for some risk-abatement measures described below, Watson selected the bet with the maximum action value. Action values were computed whenever a betting decision was needed by using two types of estimates that were learned before any live game play took place. The first were estimated values of the afterstates (Section 6.8) that would result from selecting each legal bet. These estimates were obtained from a state-value function, $\hat{v}(\cdot,\mathbf{w})$, defined by parameters $\mathbf{w}$, that gave estimates of the probability of a win for Watson from any game state. The second estimates used to compute action values gave the "in-category DD confidence," p_{DD}, which estimated the likelihood that Watson would respond correctly to the as-yet unrevealed DD clue.

Tesauro et al. used the reinforcement learning approach of TD-Gammon described above to learn $\hat{v}(\cdot,\mathbf{w})$: a straightforward combination of nonlinear TD(λ) using a multilayer ANN with weights $\mathbf{w}$ trained by backpropagating TD errors during many simulated games. States were represented to the network by feature vectors specifically designed for *Jeopardy!*. Features included the current scores of the three players, how many DDs remained, the total dollar value of the remaining clues, and other information related to the amount of play left in the game. Unlike TD-Gammon, which learned by self-play, Watson's $\hat{v}$ was learned over millions of simulated games against carefully-crafted models of human players. In-category confidence estimates were conditioned on the number of right responses r and wrong responses w that Watson gave in previously-played clues in the current category. The dependencies on (r, w) were estimated from Watson's actual accuracies over many thousands of historical categories.

With the previously learned value function $\hat{v}$ and in-category DD confidence p_{DD}, Watson computed $\hat{q}(s, bet)$ for each legal round-dollar bet as follows:

$$\hat{q}(s, bet) = p_{DD} \times \hat{v}(S_W + bet, \ldots) + (1 - p_{DD}) \times \hat{v}(S_W - bet, \ldots), \tag{16.2}$$

where S_W is Watson's current score, and $\hat{v}$ gives the estimated value for the game state after Watson's response to the DD clue, which is either correct or incorrect. Computing an action value this way corresponds to the insight from Exercise 3.19 that an action value is the expected next state value given the action (except that here it is the expected next *afterstate* value because the full next state of the entire game depends on the next square selection).

Tesauro et al. found that selecting bets by maximizing action values incurred "a frightening amount of risk," meaning that if Watson's response to the clue happened to be wrong, the loss could be disastrous for its chances of winning. To decrease the downside risk of a wrong answer, Tesauro et al. adjusted (16.2) by subtracting a small fraction of the standard deviation over Watson's correct/incorrect afterstate evaluations. They further reduced risk by prohibiting bets that would cause the wrong-answer afterstate value to decrease below a certain limit. These measures slightly reduced Watson's expectation of winning, but they significantly reduced downside risk, not only in terms of average risk per DD bet, but even more so in extreme-risk scenarios where a risk-neutral Watson would bet most or all of its bankroll.

Why was the TD-Gammon method of self-play not used to learn the critical value function $\hat{v}$? Learning from self-play in *Jeopardy!* would not have worked very well because WATSON was so different from any human contestant. Self-play would have led to exploration of state space regions that are not typical for play against human opponents, particularly human champions. In addition, unlike backgammon, *Jeopardy!* is a game of imperfect information because contestants do not have access to all the information influencing their opponents' play. In particular, *Jeopardy!* contestants do not know how much confidence their opponents have for responding to clues in the various categories. Self-play would have been something like playing poker with someone who is holding the same cards that you hold.

As a result of these complications, much of the effort in developing WATSON's DD-wagering strategy was devoted to creating good models of human opponents. The models did not address the natural language aspect of the game, but were instead stochastic process models of events that can occur during play. Statistics were extracted from an extensive fan-created archive of game information from the beginning of the show to the present day. The archive includes information such as the ordering of the clues, right and wrong contestant answers, DD locations, and DD and FJ bets for nearly 300,000 clues. Three models were constructed: an Average Contestant model (based on all the data), a Champion model (based on statistics from games with the 100 best players), and a Grand Champion model (based on statistics from games with the 10 best players). In addition to serving as opponents during learning, the models were used to asses the benefits produced by the learned DD-wagering strategy. WATSON's win rate in simulation when it used a baseline heuristic DD-wagering strategy was 61%; when it used the learned values and a default confidence value, its win rate increased to 64%; and with live in-category confidence, it was 67%. Tesauro et al. regarded this as a significant improvement, given that the DD wagering was needed only about 1.5 to 2 times in each game.

Because WATSON had only a few seconds to bet, as well as to select squares and decide whether or not to buzz in, the computation time needed to make these decisions was a critical factor. The ANN implementation of $\hat{v}$ allowed DD bets to be made quickly enough to meet the time constraints of live play. However, once games could be simulated fast enough through improvements in the simulation software, near the end of a game it was feasible to estimate the value of bets by averaging over many Monte-Carlo trials in which the consequence of each bet was determined by simulating play to the game's end. Selecting endgame DD bets in live play based on Monte-Carlo trials instead of the ANN significantly improved WATSON's performance because errors in value estimates in endgames could seriously affect its chances of winning. Making all the decisions via Monte-Carlo trials might have led to better wagering decisions, but this was simply impossible given the complexity of the game and the time constraints of live play.

Although its ability to quickly and accurately answer natural language questions stands out as WATSON's major achievement, all of its sophisticated decision strategies contributed to its impressive defeat of human champions. According to Tesauro et al. (2012):

> ... it is plainly evident that our strategy algorithms achieve a level of quantitative precision and real-time performance that exceeds human capabilities. This is particularly true in the cases of DD wagering and endgame buzzing, where humans simply cannot come close to matching the precise equity and confidence estimates and complex decision calculations performed by Watson.

16.4 Optimizing Memory Control

Most computers use dynamic random access memory (DRAM) as their main memory because of its low cost and high capacity. The job of a DRAM memory controller is to efficiently use the interface between the processor chip and an off-chip DRAM system to provide the high-bandwidth and low-latency data transfer necessary for high-speed program execution. A memory controller needs to deal with dynamically changing patterns of read/write requests while adhering to a large number of timing and resource constraints required by the hardware. This is a formidable scheduling problem, especially with modern processors with multiple cores sharing the same DRAM.

İpek, Mutlu, Martínez, and Caruana (2008) (also Martínez and İpek, 2009) designed a reinforcement learning memory controller and demonstrated that it can significantly improve the speed of program execution over what was possible with conventional controllers at the time of their research. They were motivated by limitations of existing state-of-the-art controllers that used policies that did not take advantage of past scheduling experience and did not account for long-term consequences of scheduling decisions. İpek et al.'s project was carried out by means of simulation, but they designed the controller at the detailed level of the hardware needed to implement it—including the learning algorithm—directly on a processor chip.

Accessing DRAM involves a number of steps that have to be done according to strict time constraints. DRAM systems consist of multiple DRAM chips, each containing multiple rectangular arrays of storage cells arranged in rows and columns. Each cell stores a bit as the charge on a capacitor. Because the charge decreases over time, each DRAM cell needs to be recharged—refreshed—every few milliseconds to prevent memory content from being lost. This need to refresh the cells is why DRAM is called "dynamic."

Each cell array has a row buffer that holds a row of bits that can be transferred into or out of one of the array's rows. An *activate* command "opens a row," which means moving the contents of the row whose address is indicated by the command into the row buffer. With a row open, the controller can issue *read* and *write* commands to the cell array. Each read command transfers a word (a short sequence of consecutive bits) in the row buffer to the external data bus, and each write command transfers a word in the external data bus to the row buffer. Before a different row can be opened, a *precharge* command must be issued which transfers the (possibly updated) data in the row buffer back into the addressed row of the cell array. After this, another activate command can open a new row to be accessed. Read and write commands are *column commands* because they sequentially transfer bits into or out of columns of the row buffer; multiple bits can be transferred without re-opening the row. Read and write commands to the currently-open row can be carried out more quickly than accessing a different row, which would involve additional *row commands*: precharge and activate; this is sometimes

referred to as "row locality." A memory controller maintains a *memory transaction queue* that stores memory-access requests from the processors sharing the memory system. The controller has to process requests by issuing commands to the memory system while adhering to a large number of timing constraints.

A controller's policy for scheduling access requests can have a large effect on the performance of the memory system, such as the average latency with which requests can be satisfied and the throughput the system is capable of achieving. The simplest scheduling strategy handles access requests in the order in which they arrive by issuing all the commands required by the request before beginning to service the next one. But if the system is not ready for one of these commands, or executing a command would result in resources being underutilized (e.g., due to timing constraints arising from servicing that one command), it makes sense to begin servicing a newer request before finishing the older one. Policies can gain efficiency by reordering requests, for example, by giving priority to read requests over write requests, or by giving priority to read/write commands to already open rows. The policy called First-Ready, First-Come-First-Serve (FR-FCFS), gives priority to column commands (read and write) over row commands (activate and precharge), and in case of a tie gives priority to the oldest command. FR-FCFS was shown to outperform other scheduling policies in terms of average memory-access latency under conditions commonly encountered (Rixner, 2004).

Figure 16.3 is a high-level view of İpek et al.'s reinforcement learning memory controller. They modeled the DRAM access process as an MDP whose states are the contents of the transaction queue and whose actions are commands to the DRAM system: *precharge*, *activate*, *read*, *write*, and *NoOp*. The reward signal is 1 whenever the action is *read* or *write*, and otherwise it is 0. State transitions were considered to be stochastic because the next state of the system not only depends on the scheduler's command, but also on aspects of the system's behavior that the scheduler cannot control, such as the workloads of the processor cores accessing the DRAM system.

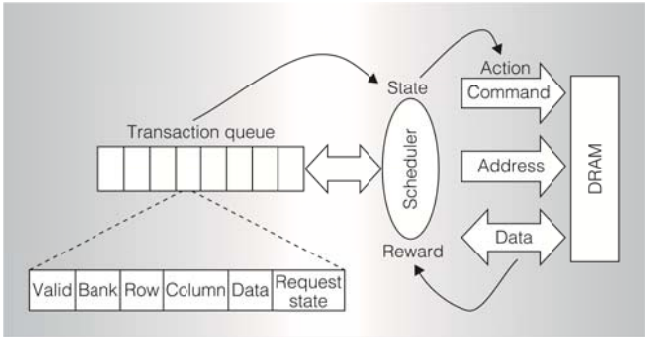


Figure 16.3: High-level view of the reinforcement learning DRAM controller. The scheduler is the reinforcement learning agent. Its environment is represented by features of the transaction queue, and its actions are commands to the DRAM system. ©2009 IEEE. Reprinted, with permission, from J. F. Martínez and E. İpek, Dynamic multicore resource management: A machine learning approach, *Micro, IEEE, 29*(5), p. 12.

Critical to this MDP are constraints on the actions available in each state. Recall from Chapter 3 that the set of available actions can depend on the state: $A_t \in \mathcal{A}(S_t)$, where A_t is the action at time step t and $\mathcal{A}(S_t)$ is the set of actions available in state S_t. In this application, the integrity of the DRAM system was assured by not allowing actions that would violate timing or resource constraints. Although İpek et al. did not make it explicit, they effectively accomplished this by pre-defining the sets $\mathcal{A}(S_t)$ for all possible states S_t.

These constraints explain why the MDP has a *NoOp* action and why the reward signal is 0 except when a *read* or *write* command is issued. *NoOp* is issued when it is the sole legal action in a state. To maximize utilization of the memory system, the controller's task is to drive the system to states in which either a *read* or a *write* action can be selected: only these actions result in sending data over the external data bus, so it is only these that contribute to the throughput of the system. Although *precharge* and *activate* produce no immediate reward, the agent needs to select these actions to make it possible to later select the rewarded *read* and *write* actions.

The scheduling agent used Sarsa (Section 6.4) to learn an action-value function. States were represented by six integer-valued features. To approximate the action-value function, the algorithm used linear function approximation implemented by tile coding with hashing (Section 9.5.4). The tile coding had 32 tilings, each storing 256 action values as 16-bit fixed point numbers. Exploration was ε-greedy with $\varepsilon = 0.05$.

State features included the number of read requests in the transaction queue, the number of write requests in the transaction queue, the number of write requests in the transaction queue waiting for their row to be opened, and the number of read requests in the transaction queue waiting for their row to be opened that are the oldest issued by their requesting processors. (The other features depended on how the DRAM interacts with cache memory, details we omit here.) The selection of the state features was based on İpek et al.'s understanding of factors that impact DRAM performance. For example, balancing the rate of servicing reads and writes based on how many of each are in the transaction queue can help avoid stalling the DRAM system's interaction with cache memory. The authors in fact generated a relatively long list of potential features, and then pared them down to a handful using simulations guided by stepwise feature selection.

An interesting aspect of this formulation of the scheduling problem as an MDP is that the features input to the tile coding for defining the action-value function were different from the features used to specify the action-constraint sets $\mathcal{A}(S_t)$. Whereas the tile coding input was derived from the contents of the transaction queue, the constraint sets depended on a host of other features related to timing and resource constraints that had to be satisfied by the hardware implementation of the entire system. In this way, the action constraints ensured that the learning algorithm's exploration could not endanger the integrity of the physical system, while learning was effectively limited to a "safe" region of the much larger state space of the hardware implementation.

Because an objective of this work was that the learning controller could be implemented on a chip so that learning could occur online while a computer is running, hardware implementation details were important considerations. The design included two five-stage pipelines to calculate and compare two action values at every processor clock cycle, and

to update the appropriate action value. This included accessing the tile coding which was stored on-chip in static RAM. For the configuration İpek et al. simulated, which was a 4GHz 4-core chip typical of high-end workstations at the time of their research, there were 10 processor cycles for every DRAM cycle. Considering the cycles needed to fill the pipes, up to 12 actions could be evaluated in each DRAM cycle. İpek et al. found that the number of legal commands for any state was rarely greater than this, and that performance loss was negligible if enough time was not always available to consider all legal commands. These and other clever design details made it feasible to implement the complete controller and learning algorithm on a multi-processor chip.

İpek et al. evaluated their learning controller in simulation by comparing it with three other controllers: 1) the FR-FCFS controller mentioned above that produces the best on-average performance, 2) a conventional controller that processes each request in order, and 3) an unrealizable ideal controller, called the Optimistic controller, able to sustain 100% DRAM throughput if given enough demand by ignoring all timing and resource constraints, but otherwise modeling DRAM latency (as row buffer hits) and bandwidth. They simulated nine memory-intensive parallel workloads consisting of scientific and data-mining applications. Figure 16.4 shows the performance (the inverse of execution time normalized to the performance of FR-FCFS) of each controller for the nine applications, together with the geometric mean of their performances over the applications. The learning controller, labeled RL in the figure, improved over that of FR-FCFS by from 7% to 33% over the nine applications, with an average improvement of 19%. Of course, no realizable controller can match the performance of Optimistic, which ignores all timing and resource constraints, but the learning controller's performance closed the gap with Optimistic's upper bound by an impressive 27%.

Because the rationale for on-chip implementation of the learning algorithm was to allow the scheduling policy to adapt online to changing workloads, İpek et al. analyzed the impact of online learning compared to a previously-learned fixed policy. They trained

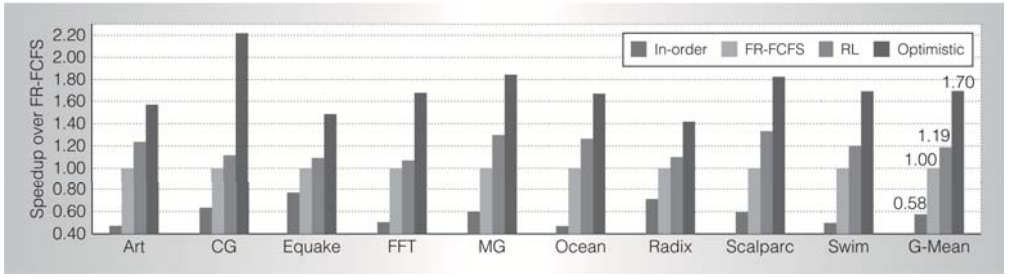


Figure 16.4: Performances of four controllers over a suite of 9 simulated benchmark applications. The controllers are: the simplest 'in-order' controller, FR-FCFS, the learning controller RL, and the unrealizable Optimistic controller which ignores all timing and resource constraints to provide a performance upper bound. Performance, normalized to that of FR-FCFS, is the inverse of execution time. At far right is the geometric mean of performances over the 9 benchmark applications for each controller. Controller RL comes closest to the ideal performance. ©2009 IEEE. Reprinted, with permission, from J. F. Martínez and E. İpek, Dynamic multicore resource management: A machine learning approach, *Micro, IEEE, 29*(5), p. 13.

their controller with data from all nine benchmark applications and then held the resulting action values fixed throughout the simulated execution of the applications. They found that the average performance of the controller that learned online was 8% better than that of the controller using the fixed policy, leading them to conclude that online learning is an important feature of their approach.

This learning memory controller was never committed to physical hardware because of the large cost of fabrication. Nevertheless, İpek et al. could convincingly argue on the basis of their simulation results that a memory controller that learns online via reinforcement learning has the potential to improve performance to levels that would otherwise require more complex and more expensive memory systems, while removing from human designers some of the burden required to manually design efficient scheduling policies. Mukundan and Martínez (2012) took this project forward by investigating learning controllers with additional actions, other performance criteria, and more complex reward functions derived using genetic algorithms. They considered additional performance criteria related to energy efficiency. The results of these studies surpassed the earlier results described above and significantly surpassed the 2012 state-of-the-art for all of the performance criteria they considered. The approach is especially promising for developing sophisticated power-aware DRAM interfaces.

16.5 Human-level Video Game Play

One of the greatest challenges in applying reinforcement learning to real-world problems is deciding how to represent and store value functions and/or policies. Unless the state set is finite and small enough to allow exhaustive representation by a lookup table—as in many of our illustrative examples—one must use a parameterized function approximation scheme. Whether linear or nonlinear, function approximation relies on features that have to be readily accessible to the learning system and able to convey the information necessary for skilled performance. Most successful applications of reinforcement learning owe much to sets of features carefully handcrafted based on human knowledge and intuition about the specific problem to be tackled.

A team of researchers at Google DeepMind developed an impressive demonstration that a deep multi-layer ANN can automate the feature design process (Mnih et al., 2013, 2015). Multi-layer ANNs have been used for function approximation in reinforcement learning ever since the 1986 popularization of the backpropagation algorithm as a method for learning internal representations (Rumelhart, Hinton, and Williams, 1986; see Section 9.6). Striking results have been obtained by coupling reinforcement learning with backpropagation. The results obtained by Tesauro and colleages with TD-Gammon and Watson discussed above are notable examples. These and other applications benefited from the ability of multi-layer ANNs to learn task-relevant features. However, in all the examples of which we are aware, the most impressive demonstrations required the network's input to be represented in terms of specialized features handcrafted for the given problem. This is vividly apparent in the TD-Gammon results. TD-Gammon 0.0, whose network input was essentially a "raw" representation of the backgammon board, meaning that it involved very little knowledge of backgammon, learned to play approximately as well as

the best previous backgammon computer programs. Adding specialized backgammon features produced TD-Gammon 1.0 which was substantially better than all previous backgammon programs and competed well against human experts.

Mnih et al. developed a reinforcement learning agent called *deep Q-network* (DQN) that combined Q-learning with a *deep convolutional* ANN, a many-layered, or deep, ANN specialized for processing spatial arrays of data such as images. We describe deep convolutional ANNs in Section 9.6. By the time of Mnih et al.'s work with DQN, deep ANNs, including deep convolutional ANNs, had produced impressive results in many applications, but they had not been widely used in reinforcement learning.

Mnih et al. used DQN to show how a reinforcement learning agent can achieve a high level of performance on any of a collection of different problems without having to use different problem-specific feature sets. To demonstrate this, they let DQN learn to play 49 different Atari 2600 video games by interacting with a game emulator. DQN learned a different policy for each of the 49 games (because the weights of its ANN were reset to random values before learning on each game), but it used the same raw input, network architecture, and parameter values (e.g., step size, discount rate, exploration parameters, and many more specific to the implementation) for all the games. DQN achieved levels of play at or beyond human level on a large fraction of these games. Although the games were alike in being played by watching streams of video images, they varied widely in other respects. Their actions had different effects, they had different state-transition dynamics, and they needed different policies for learning high scores. The deep convolutional ANN learned to transform the raw input common to all the games into features specialized for representing the action values required for playing at the high level DQN achieved for most of the games.

The Atari 2600 is a home video game console that was sold in various versions by Atari Inc. from 1977 to 1992. It introduced or popularized many arcade video games that are now considered classics, such as Pong, Breakout, Space Invaders, and Asteroids. Although much simpler than modern video games, Atari 2600 games are still entertaining and challenging for human players, and they have been attractive as testbeds for developing and evaluating reinforcement learning methods (Diuk, Cohen, Littman, 2008; Naddaf, 2010; Cobo, Zang, Isbell, and Thomaz, 2011; Bellemare, Veness, and Bowling, 2013). Bellemare, Naddaf, Veness, and Bowling (2012) developed the publicly available Arcade Learning Environment (ALE) to encourage and simplify using Atari 2600 games to study learning and planning algorithms.

These previous studies and the availability of ALE made the Atari 2600 game collection a good choice for Mnih et al.'s demonstration, which was also influenced by the impressive human-level performance that TD-Gammon was able to achieve in backgammon. DQN is similar to TD-Gammon in using a multi-layer ANN as the function approximation method for a semi-gradient form of a TD algorithm, with the gradients computed by the backpropagation algorithm. However, instead of using TD(λ) as TD-Gammon did, DQN used the semi-gradient form of Q-learning. TD-Gammon estimated the values of afterstates, which were easily obtained from the rules for making backgammon moves. To use the same algorithm for the Atari games would have required generating the next states for each possible action (which would not have been afterstates in that case).

This could have been done by using the game emulator to run single-step simulations for all the possible actions (which ALE makes possible). Or a model of each game's state-transition function could have been learned and used to predict next states (Oh, Guo, Lee, Lewis, and Singh, 2015). While these methods might have produced results comparable to DQN's, they would have been more complicated to implement and would have significantly increased the time needed for learning. Another motivation for using Q-learning was that DQN used the *experience replay* method, described below, which requires an off-policy algorithm. Being model-free and off-policy made Q-learning a natural choice.

Before describing the details of DQN and how the experiments were conducted, we look at the skill levels DQN was able to achieve. Mnih et al. compared the scores of DQN with the scores of the best performing learning system in the literature at the time, the scores of a professional human games tester, and the scores of an agent that selected actions at random. The best system from the literature used linear function approximation with features designed using some knowledge about Atari 2600 games (Bellemare, Naddaf, Veness, and Bowling, 2013). DQN learned on each game by interacting with the game emulator for 50 million frames, which corresponds to about 38 days of experience with the game. At the start of learning on each game, the weights of DQN's network were reset to random values. To evaluate DQN's skill level after learning, its score was averaged over 30 sessions on each game, each lasting up to 5 minutes and beginning with a random initial game state. The professional human tester played using the same emulator (with the sound turned off to remove any possible advantage over DQN which did not process audio). After 2 hours of practice, the human played about 20 episodes of each game for up to 5 minutes each and was not allowed to take any break during this time. DQN learned to play better than the best previous reinforcement learning systems on all but 6 of the games, and played better than the human player on 22 of the games. By considering any performance that scored at or above 75% of the human score to be comparable to, or better than, human-level play, Mnih et al. concluded that the levels of play DQN learned reached or exceeded human level on 29 of the 46 games. See Mnih et al. (2015) for a more detailed account of these results.

For an artificial learning system to achieve these levels of play would be impressive enough, but what makes these results remarkable—and what many at the time considered to be breakthrough results for artificial intelligence—is that the very same learning system achieved these levels of play on widely varying games without relying on any game-specific modifications.

A human playing any of these 49 Atari games sees 210×160 pixel image frames with 128 colors at 60Hz. In principle, exactly these images could have formed the raw input to DQN, but to reduce memory and processing requirements, Mnih et al. preprocessed each frame to produce an 84×84 array of luminance values. Because the full states of many of the Atari games are not completely observable from the image frames, Mnih et al. "stacked" the four most recent frames so that the inputs to the network had dimension $84 \times 84 \times 4$. This did not eliminate partial observability for all of the games, but it was helpful in making many of them more Markovian.

An essential point here is that these preprocessing steps were exactly the same for all 46

games. No game-specific prior knowledge was involved beyond the general understanding that it should still be possible to learn good policies with this reduced dimension and that stacking adjacent frames should help with the partial observability of some of the games. Because no game-specific prior knowledge beyond this minimal amount was used in preprocessing the image frames, we can think of the $84\times84\times4$ input vectors as being "raw" input to DQN.

The basic architecture of DQN is similar to the deep convolutional ANN illustrated in Figure 9.15 (though unlike that network, subsampling in DQN is treated as part of each convolutional layer, with feature maps consisting of units having only a selection of the possible receptive fields). DQN has three hidden convolutional layers, followed by one fully connected hidden layer, followed by the output layer. The three successive hidden convolutional layers of DQN produce 32 20×20 feature maps, 64 9×9 feature maps, and 64 7×7 feature maps. The activation function of the units of each feature map is a rectifier nonlinearity ($\max(0, x)$). The 3,136 ($64\times7\times7$) units in this third convolutional layer all connect to each of 512 units in the fully connected hidden layer, which then each connect to all 18 units in the output layer, one for each possible action in an Atari game.

The activation levels of DQN's output units were the estimated optimal action values of the corresponding state–action pairs, for the state represented by the network's input. The assignment of output units to a game's actions varied from game to game, and because the number of valid actions varied between 4 and 18 for the games, not all output units had functional roles in all of the games. It helps to think of the network as if it were 18 separate networks, one for estimating the optimal action value of each possible action. In reality, these networks shared their initial layers, but the output units learned to use the features extracted by these layers in different ways.

DQN's reward signal indicated how a games's score changed from one time step to the next: $+1$ whenever it increased, -1 whenever it decreased, and 0 otherwise. This standardized the reward signal across the games and made a single step-size parameter work well for all the games despite their varying ranges of scores. DQN used an ε-greedy policy, with ε decreasing linearly over the first million frames and remaining at a low value for the rest of the learning session. The values of the various other parameters, such as the learning step size, discount rate, and others specific to the implementation, were selected by performing informal searches to see which values worked best for a small selection of the games. These values were then held fixed for all of the games.

After DQN selected an action, the action was executed by the game emulator, which returned a reward and the next video frame. The frame was preprocessed and added to the four-frame stack that became the next input to the network. Skipping for the moment the changes to the basic Q-learning procedure made by Mnih et al., DQN used the following semi-gradient form of Q-learning to update the network's weights:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t), \quad (16.3)$$

where $\mathbf{w}_t$ is the vector of the network's weights, A_t is the action selected at time step t, and S_t and S_{t+1} are respectively the preprocessed image stacks input to the network at time steps t and $t+1$.

The gradient in (16.3) was computed by backpropagation. Imagining again that there

was a separate network for each action, for the update at time step t, backpropagation was applied only to the network corresponding to A_t. Mnih et al. took advantage of techniques shown to improve the basic backpropagation algorithm when applied to large networks. They used a *mini-batch method* that updated weights only after accumulating gradient information over a small batch of images (here after 32 images). This yielded smoother sample gradients compared to the usual procedure that updates weights after each action. They also used a gradient-ascent algorithm called RMSProp (Tieleman and Hinton, 2012) that accelerates learning by adjusting the step-size parameter for each weight based on a running average of the magnitudes of recent gradients for that weight.

Mnih et al. modified the basic Q-learning procedure in three ways. First, they used a method called *experience replay* first studied by Lin (1992). This method stores the agent's experience at each time step in a replay memory that is accessed to perform the weight updates. It worked like this in DQN. After the game emulator executed action A_t in a state represented by the image stack S_t, and returned reward R_{t+1} and image stack S_{t+1}, it added the tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ to the replay memory. This memory accumulated experiences over many plays of the same game. At each time step multiple Q-learning updates—a mini-batch—were performed based on experiences sampled uniformly at random from the replay memory. Instead of S_{t+1} becoming the new S_t for the next update as it would in the usual form of Q-learning, a new unconnected experience was drawn from the replay memory to supply data for the next update. Because Q-learning is an off-policy algorithm, it does not need to be applied along connected trajectories.

Q-learning with experience replay provided several advantages over the usual form of Q-learning. The ability to use each stored experience for many updates allowed DQN to learn more efficiently from its experiences. Experience replay reduced the variance of the updates because successive updates were not correlated with one another as they would be with standard Q-learning. And by removing the dependence of successive experiences on the current weights, experience replay eliminated one source of instability.

Mnih et al. modified standard Q-learning in a second way to improve its stability. As in other methods that bootstrap, the target for a Q-learning update depends on the current action-value function estimate. When a parameterized function approximation method is used to represent action values, the target is a function of the same parameters that are being updated. For example, the target in the update given by (16.3) is $\gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$. Its dependence on $\mathbf{w}_t$ complicates the process compared to the simpler supervised-learning situation in which the targets do not depend on the parameters being updated. As discussed in Chapter 11 this can lead to oscillations and/or divergence.

To address this problem Mnih et al. used a technique that brought Q-learning closer to the simpler supervised-learning case while still allowing it to bootstrap. Whenever a certain number, C, of updates had been done to the weights $\mathbf{w}$ of the action-value network, they inserted the network's current weights into another network and held these duplicate weights fixed for the next C updates of $\mathbf{w}$. The outputs of this duplicate network over the next C updates of $\mathbf{w}$ were used as the Q-learning targets. Letting $\tilde{q}$ denote the output of this duplicate network, then instead of (16.3) the update rule was:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t).$$

A final modification of standard Q-learning was also found to improve stability. They clipped the error term $R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)$ so that it remained in the interval $[-1, 1]$.

Mnih et al. conducted a large number of learning runs on 5 of the games to gain insight into the effect that various of DQN's design features had on its performance. They ran DQN with the four combinations of experience replay and the duplicate target network being included or not included. Although the results varied from game to game, each of these features alone significantly improved performance, and very dramatically improved performance when used together. Mnih et al. also studied the role played by the deep convolutional ANN in DQN's learning ability by comparing the deep convolutional version of DQN with a version having a network of just one linear layer, both receiving the same stacked preprocessed video frames. Here, the improvement of the deep convolutional version over the linear version was particularly striking across all 5 of the test games.

Creating artificial agents that excel over a diverse collection of challenging tasks has been an enduring goal of artificial intelligence. The promise of machine learning as a means for achieving this has been frustrated by the need to craft problem-specific representations. DeepMind's DQN stands as a major step forward by demonstrating that a single agent can learn problem-specific features enabling it to acquire human-competitive skills over a range of tasks. This demonstration did not produce one agent that simultaneously excelled at all the tasks (because learning occurred separately for each task), but it showed that deep learning can reduce, and possibly eliminate, the need for problem-specific design and tuning. As Mnih et al. point out, however, DQN is not a complete solution to the problem of task-independent learning. Although the skills needed to excel on the Atari games were markedly diverse, all the games were played by observing video images, which made a deep convolutional ANN a natural choice for this collection of tasks. In addition, DQN's performance on some of the Atari 2600 games fell considerably short of human skill levels on these games. The games most difficult for DQN—especially Montezuma's Revenge on which DQN learned to perform about as well as the random player—require deep planning beyond what DQN was designed to do. Further, learning control skills through extensive practice, like DQN learned how to play the Atari games, is just one of the types of learning humans routinely accomplish. Despite these limitations, DQN advanced the state-of-the-art in machine learning by impressively demonstrating the promise of combining reinforcement learning with modern methods of deep learning.

16.6 Mastering the Game of Go

The ancient Chinese game of Go has challenged artificial intelligence researchers for many decades. Methods that achieve human-level skill, or even superhuman-level skill, in other games have not been successful in producing strong Go programs. Thanks to a very active community of Go programmers and international competitions, the level of Go program play has improved significantly over the years. Until recently, however, no Go program had been able to play anywhere near the level of a human Go master.

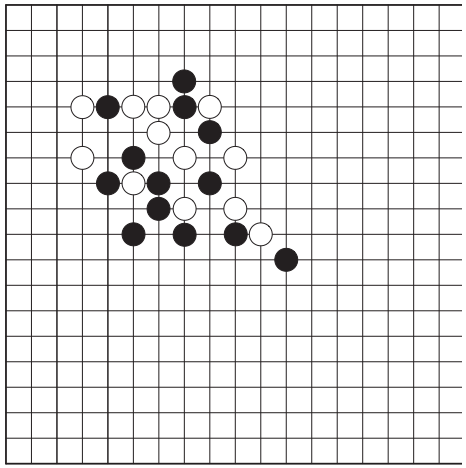
A team at DeepMind (Silver et al., 2016) developed the program *AlphaGo* that broke

this barrier by combining deep ANNs (Section 9.6), supervised learning, Monte Carlo tree search (MCTS, Section 8.11), and reinforcement learning. By the time of Silver et al.'s 2016 publication, *AlphaGo* had been shown to be decisively stronger than other current Go programs, and it had defeated the European Go champion Fan Hui 5 games to 0. These were the first victories of a Go program over a professional human Go player without handicap in full Go games. Shortly thereafter, a similar version of *AlphaGo* won stunning victories over the 18-time world champion Lee Sedol, winning 4 out of a 5 games in a challenge match, making worldwide headline news. Artificial intelligence researchers thought that it would be many more years, perhaps decades, before a program reached this level of play.

Here we describe *AlphaGo* and a successor program called *AlphaGo Zero* (Silver et al. 2017a). Where in addition to reinforcement learning, *AlphaGo* relied on supervised learning from a large database of expert human moves, *AlphaGo Zero* used only reinforcement learning and no human data or guidance beyond the basic rules of the game (hence the *Zero* in its name). We first describe *AlphaGo* in some detail in order to highlight the relative simplicity of *AlphaGo Zero*, which is both higher-performing and more of a pure reinforcement learning program.

In many ways, both *AlphaGo* and *AlphaGo Zero* are descendants of Tesauro's TD-Gammon (Section 16.1), itself a descendant of Samuel's checkers player (Section 16.2). All these programs included reinforcement learning over simulated games of self-play. *AlphaGo* and *AlphaGo Zero* also built upon the progress made by DeepMind on playing Atari games with the program DQN (Section 16.5) that used deep convolutional ANNs to approximate optimal value functions.

Go is a game between two players who alternately place black and white 'stones' on unoccupied intersections, or 'points,' on a board with a grid of 19 horizontal and 19 vertical lines to produce positions like that shown to the right. The game's goal is to capture an area of the board larger than that captured by the opponent. Stones are captured according to simple rules. A player's stones are captured if they are completely surrounded by the other player's stones, meaning that there is no horizontally or vertically adjacent point that is unoccupied. For example, Figure 16.5 shows on the left three white stones with an unoccupied adjacent point (labeled X). If player black places a stone on X, the three white stones are captured and taken off the board (Figure 16.5 middle). However, if player white were to place a stone on point X first, then the possibility of this capture would be blocked (Figure 16.5 right). Other rules are needed to prevent infinite capturing/re-capturing loops. The game ends when neither player wishes to place another stone. These rules are simple, but they produce a very complex game that has had wide



A Go board configuration

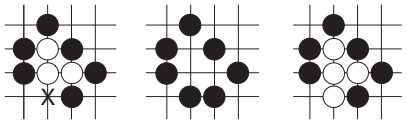


Figure 16.5: Go capturing rule. Left: the three white stones are not surrounded because point X is unoccupied. Middle: if black places a stone on X, the three white stones are captured and removed from the board. Right: if white places a stone on point X first, the capture is blocked.

appeal for thousands of years.

Methods that produce strong play for other games, such as chess, have not worked as well for Go. The search space for Go is significantly larger than that of chess because Go has a larger number of legal moves per position than chess (≈ 250 versus ≈ 35) and Go games tend to involve more moves than chess games (≈ 150 versus ≈ 80). But the size of the search space is not the major factor that makes Go so difficult. Exhaustive search is infeasible for both chess and Go, and Go on smaller boards (e.g., 9×9) has proven to be exceedingly difficult as well. Experts agree that the major stumbling block to creating stronger-than-amateur Go programs is the difficulty of defining an adequate position evaluation function. A good evaluation function allows search to be truncated at a feasible depth by providing relatively easy-to-compute predictions of what deeper search would likely yield. According to Müller (2002): "No simple yet reasonable evaluation function will ever be found for Go." A major step forward was the introduction of MCTS to Go programs. The strongest programs at the time of *AlphaGo*'s development all included MCTS, but master-level skill remained elusive.

Recall from Section 8.11 that MCTS is a decision-time planning procedure that does not attempt to learn and store a global evaluation function. Like a rollout algorithm (Section 8.10), it runs many Monte Carlo simulations of entire episodes (here, entire Go games) to select each action (here, each Go move: where to place a stone or to resign). Unlike a simple rollout algorithm, however, MCTS is an iterative procedure that incrementally extends a search tree whose root node represents the current environment state. As illustrated in Figure 8.10, each iteration traverses the tree by simulating actions guided by statistics associated with the tree's edges. In its basic version, when a simulation reaches a leaf node of the search tree, MCTS expands the tree by adding some, or all, of the leaf node's children to the tree. From the leaf node, or one of its newly added child nodes, a rollout is executed: a simulation that typically proceeds all the way to a terminal state, with actions selected by a rollout policy. When the rollout completes, the statistics associated with the search tree's edges that were traversed in this iteration are updated by backing up the return produced by the rollout. MCTS continues this process, starting each time at the search tree's root at the current state, for as many iterations as possible given the time constraints. Then, finally, an action from the root node (which still represents the current environment state) is selected according to statistics accumulated in the root node's outgoing edges. This is the action the agent takes. After the environment transitions to its next state, MCTS is executed again with the root node set to represent the new current state. The search tree at the start of this next execution might be just this new root node, or it might include descendants of this node left over from MCTS's previous execution. The remainder of the tree is discarded.

16.6.1 AlphaGo

The main innovation that made *AlphaGo* such a strong player is that it selected moves by a novel version of MCTS that was guided by both a policy and a value function learned by reinforcement learning with function approximation provided by deep convolutional ANNs. Another key feature is that instead of reinforcement learning starting from random network weights, it started from weights that were the result of previous supervised learning from a large collection of human expert moves.

The DeepMind team called *AlphaGo*'s modification of basic MCTS "asynchronous policy and value MCTS," or APV-MCTS. It selected actions via basic MCTS as described above but with some twists in how it extended its search tree and how it evaluated action edges. In contrast to basic MCTS, which expands its current search tree by using stored action values to select an unexplored edge from a leaf node, APV-MCTS, as implemented in *AlphaGo*, expanded its tree by choosing an edge according to probabilities supplied by a 13-layer deep convolutional ANN, called the *SL-policy network*, trained previously by supervised learning to predict moves contained in a database of nearly 30 million human expert moves.

Then, also in contrast to basic MCTS, which evaluates the newly-added state node solely by the return of a rollout initiated from it, APV-MCTS evaluated the node in two ways: by this return of the rollout, but also by a value function, v_θ, learned previously by a reinforcement learning method. If s was the newly-added node, its value became

$$v(s) = (1 - \eta)v_\theta(s) + \eta G, \tag{16.4}$$

where G was the return of the rollout and η controlled the mixing of the values resulting from these two evaluation methods. In *AlphaGo*, these values were supplied by the *value network*, another 13-layer deep convolutional ANN that was trained as we describe below to output estimated values of board positions. APV-MCTS's rollouts in *AlphaGo* were simulated games with both players using a fast *rollout policy* provided by a simple linear network, also trained by supervised learning before play. Throughout its execution, APV-MCTS kept track of how many simulations passed through each edge of the search tree, and when its execution completed, the most-visited edge from the root node was selected as the action to take, here the move *AlphaGo* actually made in a game.

The value network had the same structure as the deep convolutional SL policy network except that it had a single output unit that gave estimated values of game positions instead of the SL policy network's probability distributions over legal actions. Ideally, the value network would output optimal state values, and it might have been possible to approximate the optimal value function along the lines of TD-Gammon described above: self-play with nonlinear TD(λ) coupled to a deep convolutional ANN. But the DeepMind team took a different approach that held more promise for a game as complex as Go. They divided the process of training the value network into two stages. In the first stage, they created the best policy they could by using reinforcement learning to train an *RL*

policy network. This was a deep convolutional ANN with the same structure as the SL policy network. It was initialized with the final weights of the SL policy network that were learned via supervised learning, and then policy-gradient reinforcement learning was used to improve upon the SL policy. In the second stage of training the value network, the team used Monte Carlo policy evaluation on data obtained from a large number of simulated self-play games with moves selected by the RL policy network.

Figure 16.6 illustrates the networks used by *AlphaGo* and the steps taken to train them in what the DeepMind team called the "*AlphaGo* pipeline." All these networks were trained before any live game play took place, and their weights remained fixed throughout live play.

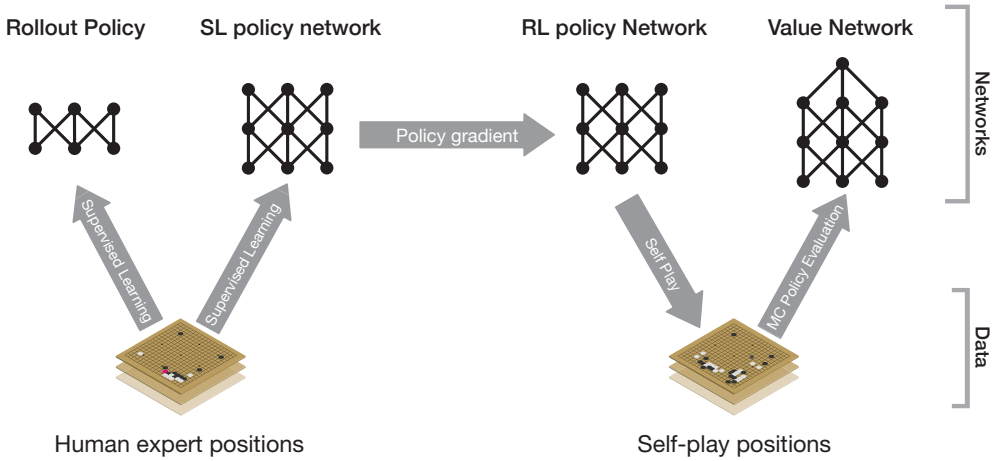


Figure 16.6: *AlphaGo* pipeline. Adapted with permission from Macmillan Publishers Ltd: *Nature*, vol. 529(7587), p. 485, ©2016.

Here is some more detail about *AlphaGo*'s ANNs and their training. The identically-structured SL and RL policy networks were similar to DQN's deep convolutional network described in Section 16.5 for playing Atari games, except that they had 13 convolutional layers with the final layer consisting of a soft-max unit for each point on the 19×19 Go board. The networks' input was a $19 \times 19 \times 48$ image stack in which each point on the Go board was represented by the values of 48 binary or integer-valued features. For example, for each point, one feature indicated if the point was occupied by one of *AlphaGo*'s stones, one of its opponent's stones, or was unoccupied, thus providing the "raw" representation of the board configuration. Other features were based on the rules of Go, such as the number of adjacent points that were empty, the number of opponent stones that would be captured by placing a stone there, the number of turns since a stone was placed there, and other features that the design team considered to be important.

Training the SL policy network took approximately 3 weeks using a distributed implementation of stochastic gradient ascent on 50 processors. The network achieved 57% accuracy, where the best accuracy achieved by other groups at the time of publication

was 44.4%. Training the RL policy network was done by policy gradient reinforcement learning over simulated games between the RL policy network's current policy and opponents using policies randomly selected from policies produced by earlier iterations of the learning algorithm. Playing against a randomly selected collection of opponents prevented overfitting to the current policy. The reward signal was +1 if the current policy won, −1 if it lost, and zero otherwise. These games directly pitted the two policies against one another without involving MCTS. By simulating many games in parallel on 50 processors, the DeepMind team trained the RL policy network on a million games in a single day. In testing the final RL policy, they found that it won more than 80% of games played against the SL policy, and it won 85% of games played against a Go program using MCTS that simulated 100,000 games per move.

The value network, whose structure was similar to that of the SL and RL policy networks except for its single output unit, received the same input as the SL and RL policy networks with the exception that there was an additional binary feature giving the current color to play. Monte Carlo policy evaluation was used to train the network from data obtained from a large number of self-play games played using the RL policy. To avoid overfitting and instability due to the strong correlations between positions encountered in self-play, the DeepMind team constructed a data set of 30 million positions each chosen randomly from a unique self-play game. Then training was done using 50 million mini-batches each of 32 positions drawn from this data set. Training took one week on 50 GPUs.

The rollout policy was learned prior to play by a simple linear network trained by supervised learning from a corpus of 8 million human moves. The rollout policy network had to output actions quickly while still being reasonably accurate. In principle, the SL or RL policy networks could have been used in the rollouts, but the forward propagation through these deep networks took too much time for either of them to be used in rollout simulations, a great many of which had to be carried out for each move decision during live play. For this reason, the rollout policy network was less complex than the other policy networks, and its input features could be computed more quickly than the features used for the policy networks. The rollout policy network allowed approximately 1,000 complete game simulations per second to be run on each of the processing threads that *AlphaGo* used.

One may wonder why the SL policy was used instead of the better RL policy to select actions in the expansion phase of APV-MCTS. These policies took the same amount of time to compute because they used the same network architecture. The team actually found that *AlphaGo* played better against human opponents when APV-MCTS used as the SL policy instead of the RL policy. They conjectured that the reason for this was that the latter was tuned to respond to optimal moves rather than to the broader set of moves characteristic of human play. Interestingly, the situation was reversed for the value function used by APV-MCTS. They found that when APV-MCTS used the value function derived from the RL policy, it performed better than if it used the value function derived from the SL policy.

Several methods worked together to produce *AlphaGo*'s impressive playing skill. The DeepMind team evaluated different versions of *AlphaGo* in order to assess the contributions

made by these various components. The parameter η in (16.4) controlled the mixing of game state evaluations produced by the value network and by rollouts. With $\eta = 0$, *AlphaGo* used just the value network without rollouts, and with $\eta = 1$, evaluation relied just on rollouts. They found that *AlphaGo* using just the value network played better than the rollout-only *AlphaGo*, and in fact played better than the strongest of all other Go programs existing at the time. The best play resulted from setting $\eta = 0.5$, indicating that combining the value network with rollouts was particularly important to *AlphaGo*'s success. These evaluation methods complemented one another: the value network evaluated the high-performance RL policy that was too slow to be used in live play, while rollouts using the weaker but much faster rollout policy were able to add precision to the value network's evaluations for specific states that occurred during games.

Overall, *AlphaGo*'s remarkable success fueled a new round of enthusiasm for the promise of artificial intelligence, specifically for systems combining reinforcement learning with deep ANNs, to address problems in other challenging domains.

16.6.2 AlphaGo Zero

Building upon the experience with *AlphaGo*, a DeepMind team developed *AlphaGo Zero* (Silver et al. 2017a). In contrast to *AlphaGo*, this program used *no human data or guidance beyond the basic rules of the game* (hence the *Zero* in its name). It learned exclusively from self-play reinforcement learning, with input giving just "raw" descriptions of the placements of stones on the Go board. *AlphaGo Zero* implemented a form of policy iteration (Section 4.3), interleaving policy evaluation with policy improvement. Figure 16.7 is an overview of *AlphaGo Zero*'s algorithm. A significant difference between *AlphaGo Zero* and *AlphaGo* is that *AlphaGo Zero* used MCTS to select moves throughout self-play reinforcement learning, whereas *AlphaGo* used MCTS for live play after—but not during—learning. Other differences besides not using any human data or human-crafted features are that *AlphaGo Zero* used only one deep convolutional ANN and used a simpler version of MCTS.

AlphaGo Zero's MCTS was simpler than the version used by *AlphaGo* in that it did not include rollouts of complete games, and therefore did not need a rollout policy. Each iteration of *AlphaGo Zero*'s MCTS ran a simulation that ended at a leaf node of the current search tree instead of at the terminal position of a complete game simulation. But as in *AlphaGo*, each iteration of MCTS in *AlphaGo Zero* was guided by the output of a deep convolutional network, labeled f_θ in Figure 16.7, were θ is the network's weight vector. The input to the network, whose architecture we describe below, consisted of raw representations of board positions, and its output had two parts: a scalar value, v, an estimate of the probability that the current player will win from from the current board position, and a vector, $\mathbf{p}$, of move probabilities, one for each possible stone placement on the current board, plus the pass, or resign, move.

Instead of selecting self-play actions according to the probabilities $\mathbf{p}$, however, *AlphaGo Zero* used these probabilities, together with the network's value output, to direct each execution of MCTS, which returned new move probabilities, shown in Figure 16.7 as the policies π_i. These policies benefitted from the many simulations that MCTS conducted

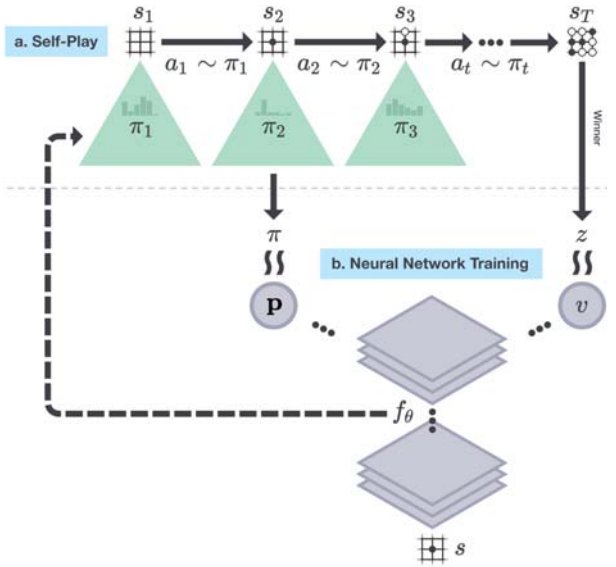


Figure 16.7: *AlphaGo Zero* self-play reinforcement learning. a) The program played many games against itself, one shown here as a sequence of board positions s_i, $i = 1, 2, \ldots, T$, with moves a_i, $i = 1, 2, \ldots, T$, and winner z. Each move a_i was determined by action probabilities π_i returned by MCTS executed from root node s_i and guided by a deep convolutional network, here labeled f_θ, with latest weights θ. Shown here for just one position s but repeated for all s_i, the network's inputs were raw representations of board positions s_i (together with several past positions, though not shown here), and its outputs were vectors $\mathbf{p}$ of move probabilities that guided MCTS's forward searches, and scalar values v that estimated the probability of the current player winning from each position s_i. b) Deep convolutional network training. Training examples were randomly sampled steps from recent self-play games. Weights θ were updated to move the policy vector $\mathbf{p}$ toward the probabilities π returned by MCTS, and to include the winners z in the estimated win probability v. Reprinted from draft of Silver et al. (2017a) with permission of the authors and DeepMind.

each time it executed. The result was that the policy actually followed by *AlphaGo Zero* was an improvement over the policy given by the network's outputs $\mathbf{p}$. Silver et al. (2017a) wrote that "MCTS may therefore be viewed as a powerful *policy improvement* operator."

Here is more detail about *AlphaGo Zero*'s ANN and how it was trained. The network took as input a $19 \times 19 \times 17$ image stack consisting of 17 binary feature planes. The first 8 feature planes were raw representations of the positions of the current player's stones in the current and seven past board configurations: a feature value was 1 if a player's stone was on the corresponding point, and was 0 otherwise. The next 8 feature planes similarly coded the positions of the opponent's stones. A final input feature plane had a constant value indicating the color of the current play: 1 for black; 0 for white. Because repetition is not allowed in Go and one player is given some number of "compensation points" for not getting the first move, the current board position is not a Markov state of Go. This

is why features describing past board positions and the color feature were needed.

The network was "two-headed," meaning that after a number of initial layers, the network split into two separate "heads" of additional layers that separately fed into two sets of output units. In this case, one head fed 362 output units producing $19^2 + 1$ move probabilities $\mathbf{p}$, one for each possible stone placement plus pass; the other head fed just one output unit producing the scalar v, an estimate of the probability that the current player will win from the current board position. The network before the split consisted of 41 convolutional layers, each followed by batch normalization, and with skip connections added to implement residual learning by pairs of layers (see Section 9.6). Overall, move probabilities and values were computed by 43 and 44 layers respectively.

Starting with random weights, the network was trained by stochastic gradient descent (with momentum, regularization, and step-size parameter decreasing as training continues) using batches of examples sampled uniformly at random from all the steps of the most recent 500,000 games of self-play with the current best policy. Extra noise was added to the network's output $\mathbf{p}$ to encourage exploration of all possible moves. At periodic checkpoints during training, which Silver et al. (2017a) chose to be at every 1,000 training steps, the policy output by the ANN with the latest weights was evaluated by simulating 400 games (using MCTS with 1,600 iterations to select each move) against the current best policy. If the new policy won (by a margin set to reduce noise in the outcome), then it became the best policy to be used in subsequent self-play. The network's weights were updated to make the network's policy output $\mathbf{p}$ more closely match the policy returned by MCTS, and to make its value output, v, more closely match the probability that the current best policy wins from the board position represented by the network's input.

The DeepMind team trained *AlphaGo Zero* over 4.9 million games of self-play, which took about 3 days. Each move of each game was selected by running MCTS for 1,600 iterations, taking approximately 0.4 second per move. Network weights were updated over 700,000 batches each consisting of 2,048 board configurations. They then ran tournaments with the trained *AlphaGo Zero* playing against the version of *AlphaGo* that defeated Fan Hui by 5 games to 0, and against the version that defeated Lee Sedol by 4 games to 1. They used the Elo rating system to evaluate the relative performances of the programs. The difference between two Elo ratings is meant to predict the outcome of games between the players. The Elo ratings of *AlphaGo Zero*, the version of *AlphaGo* that played against Fan Hui, and the version that played against Lee Sedol were respectively 4,308, 3,144, and 3,739. The gaps in these Elo ratings translate into predictions that *AlphaGo Zero* would defeat these other programs with probabilities very close to one. In a match of 100 games between *AlphaGo Zero*, trained as described, and the exact version of *AlphaGo* that defeated Lee Sedol held under the same conditions that were used in that match, *AlphaGo Zero* defeated *AlphaGo* in all 100 games.

The DeepMind team also compared *AlphaGo Zero* with a program using an ANN with the same architecture but trained by supervised learning to predict human moves in a data set containing nearly 30 million positions from 160,000 games. They found that the supervised-learning player initially played better than *AlphaGo Zero*, and was better at predicting human expert moves, but played less well after *AlphaGo Zero* was trained for a day. This suggested that *AlphaGo Zero* had discovered a strategy for playing that was

different from how humans play. In fact, *AlphaGo Zero* discovered, and came to prefer, some novel variations of classical move sequences.

Final tests of *AlphaGo Zero*'s algorithm were conducted with a version having a larger ANN and trained over 29 million self-play games, which took about 40 days, again starting with random weights. This version achieved an Elo rating of 5,185. The team pitted this version of *AlphaGo Zero* against a program called *AlphaGo Master*, the strongest program at the time, that was identical to *AlphaGo Zero* but, like *AlphaGo*, used human data and features. *AlphaGo Master*'s Elo rating was 4,858, and it had defeated the strongest human professional players 60 to 0 in online games. In a 100 game match, *AlphaGo Zero* with the larger network and more extensive learning defeated *AlphaGo Master* 89 games to 11, thus providing a convincing demonstration of the problem-solving power of *AlphaGo Zero*'s algorithm.

AlphaGo Zero soundly demonstrated that superhuman performance can be achieved by pure reinforcement learning, augmented by a simple version of MCTS, and deep ANNs with very minimal knowledge of the domain and no reliance on human data or guidance. We will surely see systems inspired by the DeepMind accomplishments of both *AlphaGo* and *AlphaGo Zero* applied to challenging problems in other domains.

Recently, yet a better program, *AlphaZero*, was described by Silver et al. (2017b) that does not even incorporate knowledge of Go. *AlphaZero* is a general reinforcement learning algorithm that improves over the world's hitherto best programs in the diverse games of Go, chess, and shogi.

16.7 Personalized Web Services

Personalizing web services such as the delivery of news articles or advertisements is one approach to increasing users' satisfaction with a website or to increase the yield of a marketing campaign. A policy can recommend content considered to be the best for each particular user based on a profile of that user's interests and preferences inferred from their history of online activity. This is a natural domain for machine learning, and in particular, for reinforcement learning. A reinforcement learning system can improve a recommendation policy by making adjustments in response to user feedback. One way to obtain user feedback is by means of website satisfaction surveys, but for acquiring feedback in real time it is common to monitor user clicks as indicators of interest in a link.

A method long used in marketing called *A/B testing* is a simple type of reinforcement learning used to decide which of two versions, A or B, of a website users prefer. Because it is non-associative, like a two-armed bandit problem, this approach does not personalize content delivery. Adding context consisting of features describing individual users and the content to be delivered allows personalizing service. This has been formalized as a contextual bandit problem (or an associative reinforcement learning problem, Section 2.9) with the objective of maximizing the total number of user clicks. Li, Chu, Langford, and Schapire (2010) applied a contextual bandit algorithm to the problem of personalizing the Yahoo! Front Page Today webpage (one of the most visited pages on the internet at the time of their research) by selecting the news story to feature. Their objective was to

maximize the *click-through rate* (CTR), which is the ratio of the total number of clicks all users make on a webpage to the total number of visits to the page. Their contextual bandit algorithm improved over a standard non-associative bandit algorithm by 12.5%.

Theocharous, Thomas, and Ghavamzadeh (2015) argued that better results are possible by formulating personalized recommendation as a Markov decision problem (MDP) with the objective of maximizing the total number of clicks users make over repeated visits to a website. Policies derived from the contextual bandit formulation are greedy in the sense that they do not take long-term effects of actions into account. These policies effectively treat each visit to a website as if it were made by a new visitor uniformly sampled from the population of the website's visitors. By not using the fact that many users repeatedly visit the same websites, greedy policies do not take advantage of possibilities provided by long-term interactions with individual users.

As an example of how a marketing strategy might take advantage of long-term user interaction, Theocharous et al. contrasted a greedy policy with a longer-term policy for displaying ads for buying a product, say a car. The ad displayed by the greedy policy might offer a discount if the user buys the car immediately. A user either takes the offer or leaves the website, and if they ever return to the site, they would likely see the same offer. A longer-term policy, on the other hand, can transition the user "down a sales funnel" before presenting the final deal. It might start by describing the availability of favorable financing terms, then praise an excellent service department, and then, on the next visit, offer the final discount. This type of policy can result in more clicks by a user over repeated visits to the site, and if the policy is suitably designed, more eventual sales.

Working at Adobe Systems Incorporated, Theocharous et al. conducted experiments to see if policies designed to maximize clicks over the long term could in fact improve over short-term greedy policies. The Adobe Marketing Cloud, a set of tools that many companies use to run digital marketing campaigns, provides infrastructure for automating user-targed advertising and fund-raising campaigns. Actually deploying novel policies using these tools entails significant risk because a new policy may end up performing poorly. For this reason, the research team needed to assess what a policy's performance would be if it were to be actually deployed, but to do so on the basis of data collected under the execution of other policies. A critical aspect of this research, then, was off-policy evaluation. Further, the team wanted to do this with high confidence to reduce the risk of deploying a new policy. Although high confidence off-policy evaluation was a central component of this research (see also Thomas, 2015; Thomas, Theocharous, and Ghavamzadeh, 2015), here we focus only on the algorithms and their results.

Theocharous et al. compared the results of two algorithms for learning ad recommendation policies. The first algorithm, which they called *greedy optimization*, had the goal of maximizing only the probability of immediate clicks. As in the standard contextual bandit formulation, this algorithm did not take the long-term effects of recommendations into account. The other algorithm, a reinforcement learning algorithm based on an MDP formulation, aimed at improving the number of clicks users made over multiple visits to a website. They called this latter algorithm *life-time value* (LTV) optimization. Both algorithms faced challenging problems because the reward signal in this domain is very sparse because users usually do not click on ads, and user clicking is very random so that returns have high variance.

Data sets from the banking industry were used for training and testing these algorithms. The data sets consisted of many complete trajectories of customer interaction with a bank's website that showed each customer one out of a collection of possible offers. If a customer clicked, the reward was 1, and otherwise it was 0. One data set contained approximately 200,000 interactions from a month of a bank's campaign that randomly offered one of 7 offers. The other data set from another bank's campaign contained 4,000,000 interactions involving 12 possible offers. All interactions included customer features such as the time since the customer's last visit to the website, the number of their visits so far, the last time the customer clicked, geographic location, one of a collection of interests, and features giving demographic information.

Greedy optimization was based on a mapping estimating the probability of a click as a function of user features. The mapping was learned via supervised learning from one of the data sets by means of a random forest (RF) algorithm (Breiman, 2001). RF algorithms have been widely used for large-scale applications in industry because they are effective predictive tools that tend not to overfit and are relatively insensitive to outliers and noise. Theocharous et al. then used the mapping to define an ε-greedy policy that selected with probability 1-ε the offer predicted by the RF algorithm to have the highest probability of producing a click, and otherwise selected from the other offers uniformly at random.

LTV optimization used a batch-mode reinforcement learning algorithm called *fitted Q iteration* (FQI). It is a variant of *fitted value iteration* (Gordon, 1999) adapted to Q-learning. Batch mode means that the entire data set for learning is available from the start, as opposed to the online mode of the algorithms we focus on in this book in which data are acquired sequentially while the learning algorithm executes. Batch-mode reinforcement learning algorithms are sometimes necessary when online learning is not practical, and they can use any batch-mode supervised learning regression algorithm, including algorithms known to scale well to high-dimensional spaces. The convergence of FQI depends on properties of the function approximation algorithm (Gordon, 1999). For their application to LTV optimization, Theocharous et al. used the same RF algorithm they used for the greedy optimization approach. Because in this case FQI convergence is not monotonic, Theocharous et al. kept track of the best FQI policy by off-policy evaluation using a validation training set. The final policy for testing the LTV approach was the ε-greedy policy based on the best policy produced by FQI with the initial action-value function set to the mapping produced by the RF for the greedy optimization approach.

To measure the performance of the policies produced by the greedy and LTV approaches, Theocharous et al. used the CTR metric and a metric they called the LTV metric. These metrics are similar, except that the LTV metric critically distinguishes between individual website visitors:

$$\text{CTR} = \frac{\text{Total \# of Clicks}}{\text{Total \# of Visits}},$$

$$\text{LTV} = \frac{\text{Total \# of Clicks}}{\text{Total \# of Visitors}}.$$

Figure 16.8 illustrates how these metrics differ. Each circle represents a user visit to the site; black circles are visits at which the user clicks. Each row represents visits by a particular user. By not distinguishing between visitors, the CTR for these sequences is 0.35, whereas the LTV is 1.5. Because LTV is larger than CTR to the extent that individual users revisit the site, it is an indicator of how successful a policy is in encouraging users to engage in extended interactions with the site.

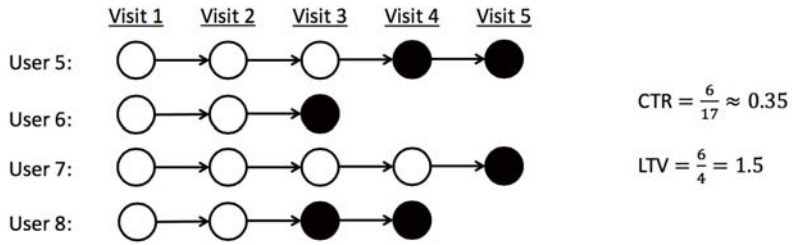


Figure 16.8: Click through rate (CTR) versus life-time value (LTV). Each circle represents a user visit; black circles are visits at which the user clicks. Adapted from Theocharous et al. (2015).

Testing the policies produced by the greedy and LTV approaches was done using a high confidence off-policy evaluation method on a test data set consisting of real-world interactions with a bank website served by a random policy. As expected, results showed that greedy optimization performed best as measured by the CTR metric, while LTV optimization performed best as measured by the LTV metric. Furthermore—although we have omitted its details—the high confidence off-policy evaluation method provided probabilistic guarantees that the LTV optimization method would, with high probability, produce policies that improve upon policies currently deployed. Assured by these probabilistic guarantees, Adobe announced in 2016 that the new LTV algorithm would be a standard component of the Adobe Marketing Cloud so that a retailer could issue a sequence of offers following a policy likely to yield higher return than a policy that is insensitive to long-term results.

16.8 Thermal Soaring

Birds and gliders take advantage of upward air currents—thermals—to gain altitude in order to maintain flight while expending little, or no, energy. Thermal soaring, as this behavior is called, is a complex skill requiring responding to subtle environmental cues to increase altitude by exploiting a rising column of air for as long as possible. Reddy, Celani, Sejnowski, and Vergassola (2016) used reinforcement learning to investigate thermal soaring policies that are effective in the strong atmospheric turbulence usually accompanying rising air currents. Their primary goal was to provide insight into the cues birds sense and how they use them to achieve their impressive thermal soaring performance, but the results also contribute to technology relevant to autonomous gliders. Reinforcement learning had previously been applied to the problem of navigating efficiently

to the vicinity of a thermal updraft (Woodbury, Dunn, and Valasek, 2014) but not to the more challenging problem of soaring within the turbulence of the updraft itself.

Reddy et al. modeled the soaring problem as a continuing MDP with discounting. The agent interacted with a detailed model of a glider flying in turbulent air. They devoted significant effort toward making the model generate realistic thermal soaring conditions, including investigating several different approaches to atmospheric modeling. For the learning experiments, air flow in a three-dimensional box with one kilometer sides, one of which was at ground level, was modeled by a sophisticated physics-based set of partial differential equations involving air velocity, temperature, and pressure. Introducing small random perturbations into the numerical simulation caused the model to produce analogs of thermal updrafts and accompanying turbulence (Figure 16.9 Left) Glider flight was modeled by aerodynamic equations involving velocity, lift, drag, and other factors governing powerless flight of a fixed-wing aircraft. Maneuvering the glider involved changing its angle of attack (the angle between the glider's wing and the direction of air flow) and its bank angle (Figure 16.9 Right).

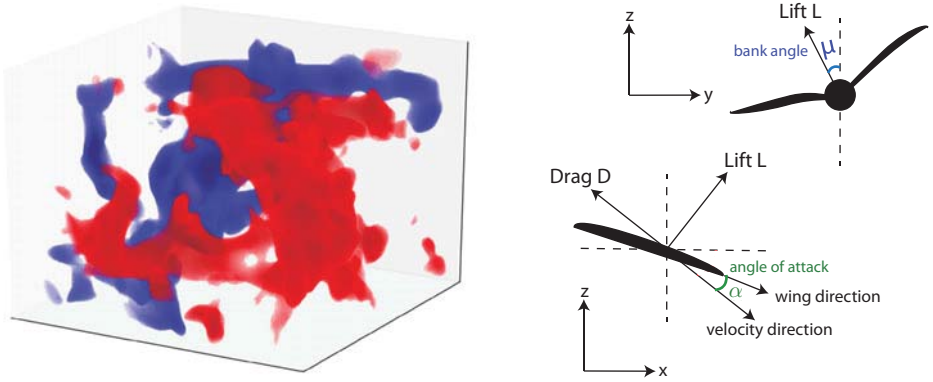


Figure 16.9: Thermal soaring model: Left: snapshot of the vertical velocity field of the simulated cube of air: in red (blue) is a region of large upward (downward) flow. Right: diagram of powerless flight showing bank angle μ and angle of attack α. Adapted with permission From PNAS vol. 113(22), p. E4879, 2016, Reddy, Celani, Sejnowski, and Vergassola, Learning to Soar in Turbulent Environments.

The interface between the agent and the environment required defining the agent's actions, the state information the agent receives from the environment, and the reward signal. By experimenting with various possibilities, Reddy et al. decided that three actions each for the angle of attack and the bank angle were enough for their purposes: increment or decrement the current bank angle and angle of attack by 5° and 2.5°, respectively, or leave them unchanged. This resulted in 3^2 possible actions. The bank angle was bounded to remain between $-15°$ and $+15°$.

Because a goal of their study was to try to determine what minimal set of sensory cues are necessary for effective soaring, both to shed light on the cues birds might use for soaring and to minimize the sensing complexity required for automated glider soaring,

the authors tried various sets of signals as input to the reinforcement learning agent. They started by using state aggregation (Section 9.3) of a four-dimensional state space with dimensions giving local vertical wind speed, local vertical wind acceleration, torque depending on the difference between the vertical wind velocities at the left and right wing tips, and the local temperature. Each dimension was discretized into three bins: positive high, negative high, and small. Results, described below, showed that only two of these dimensions were critical for effective soaring behavior.

The overall objective of thermal soaring is to gain as much altitude as possible from each rising column of air. Reddy et al. tried a straightforward reward signal that rewarded the agent at the end of each episode based on the altitude gained over the episode, a large negative reward signal if the glider touched the ground, and zero otherwise. They found that learning was not successful with this reward signal for episodes of realistic duration and that eligibility traces did not help. By experimenting with various reward signals, they found that learning was best with a reward signal that at each time step linearly combined the vertical wind velocity and vertical wind acceleration observed on the previous time step.

Learning was by one-step Sarsa, with actions selected according to a soft-max distribution based on normalized action values. Specifically, the action probabilities were computed according to (13.2) with action preferences:

$$h(s,a,\boldsymbol{\theta}) = \frac{\hat{q}(s,a,\boldsymbol{\theta}) - \min_b \hat{q}(s,b,\boldsymbol{\theta})}{\tau\big(\max_b \hat{q}(s,b,\boldsymbol{\theta}) - \min_b \hat{q}(s,b,\boldsymbol{\theta})\big)},$$

where $\boldsymbol{\theta}$ is a parameter vector with one component for each action and aggregated group of states, and $\hat{q}(s,a,\boldsymbol{\theta})$ merely returned the component corresponding to s,a in the usual way for state aggregation methods. The above equation forms the action preferences by normalizing the approximate action values to the interval $[0,1]$ then dividing by τ, a positive "temperature parameter."[3] As τ increases, the probability of selecting an action becomes less dependent on its preference; as τ decreases toward zero, the probability of selecting the most highly-preferred action approaches one, making the policy approach the greedy policy. The temperature parameter τ was initialized to 2.0 and incrementally decreased to 0.2 during learning. Action preferences were computed from the current estimates of the action values: the action with the maximum estimated action value was given preference $1/\tau$, the action with the minimum estimated action value was given preference 0, and the preferences of the other actions were scaled between these extremes. The step-size and discount-rate parameters were fixed at 0.1 and 0.98 respectively.

Each learning episode took place with the agent controlling simulated flight in an independently generated period of simulated turbulent air currents. Each episode lasted 2.5 minutes simulated with a 1 second time step. Learning effectively converged after a few hundred episodes. The left panel of Figure 16.10 shows a sample trajectory before learning where the agent selects actions randomly. Starting at the top of the volume shown, the glider's trajectory is in the direction indicated by the arrow and quickly loses altitude. Figure 16.10's right panel is a trajectory after learning. The glider starts at the same place (here appearing at the bottom of the volume) and gains altitude by spiraling

[3] Reddy et al. described this slightly differently, but our version is equivalent to theirs.

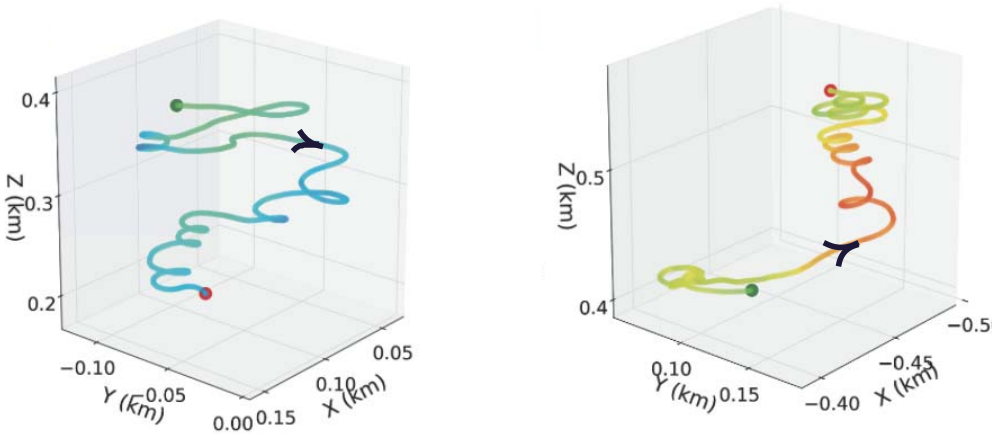


Figure 16.10: Sample thermal soaring trajectories, with arrows showing the direction of flight from the same starting point (note that the altitude scales are shifted). Left: before learning: the agent selects actions randomly and the glider descends. Right: after learning: the glider gains altitude by following a spiral trajectory. Adapted with permission from PNAS vol. 113(22), p. E4879, 2016, Reddy, Celani, Sejnowski, and Vergassola, Learning to Soar in Turbulent Environments.

within the rising column of air. Although Reddy et al. found that performance varied widely over different simulated periods of air flow, the number of times the glider touched the ground consistently decreased to nearly zero as learning progressed.

After experimenting with different sets of features available to the learning agent, it turned out that the combination of just vertical wind acceleration and torques worked best. The authors conjectured that because these features give information about the gradient of vertical wind velocity in two different directions, they allow the controller to select between turning by changing the bank angle or continuing along the same course by leaving the bank angle alone. This allows the glider to stay within a rising column of air. Vertical wind velocity is indicative of the strength of the thermal but does not help in staying within the flow. They found that sensitivity to temperature was of little help. They also found that controlling the angle of attack is not helpful in staying within a particular thermal, being useful instead for traveling between thermals when covering large distances, as in cross-country gliding and bird migration.

Due to the fact that soaring in different levels of turbulence requires different policies, training was done in conditions ranging from weak to strong turbulence. In strong turbulence the rapidly changing wind and glider velocities allowed less time for the controller to react. This reduced the amount of control possible compared to what was possible for maneuvering when fluctuations were weak. Reddy et al. examined the policies Sarsa learned under these different conditions. Common to policies learned in all regimes were these features: when sensing negative wind acceleration, bank sharply in the direction of the wing with the higher lift; when sensing large positive wind acceleration and no torque, do nothing. However, different levels of turbulence led to policy differences.

Policies learned in strong turbulence were more conservative in that they preferred small bank angles, whereas in weak turbulence, the best action was to turn as much as possible by banking sharply. Systematic study of the bank angles preferred by the policies learned under the different conditions led the authors to suggest that by detecting when vertical wind acceleration crosses a certain threshold the controller can adjust its policy to cope with different turbulence regimes.

Reddy et al. also conducted experiments to investigate the effect of the discount-rate parameter γ on the performance of the learned policies. They found that the altitude gained in an episode increased as γ increased, reaching a maximum for $\gamma = .99$, suggesting that effective thermal soaring requires taking into account long-term effects of control decisions.

This computational study of thermal soaring illustrates how reinforcement learning can further progress toward different kinds of objectives. Learning policies having access to different sets of environmental cues and control actions contributes to both the engineering objective of designing autonomous gliders and the scientific objective of improving understanding of the soaring skills of birds. In both cases, hypotheses resulting from the learning experiments can be tested in the field by instrumenting real gliders and by comparing predictions with observed bird soaring behavior.

Chapter 17

Frontiers

In this final chapter we touch on some topics that are beyond the scope of this book but that we see as particularly important for the future of reinforcement learning. Many of these topics bring us beyond what is reliably known, and some bring us beyond the MDP framework.

17.1 General Value Functions and Auxiliary Tasks

Over the course of this book, our notion of value function has become quite general. With off-policy learning we allowed a value function to be conditional on an arbitrary target policy. Then in Section 12.8 we generalized discounting to a *termination function* $\gamma : \mathcal{S} \mapsto [0,1]$, so that a different discount rate could be applied at each time step in determining the return (12.17). This allowed us to express predictions about how much reward we will get over an arbitrary, state-dependent horizon. The next, and perhaps final, step is to generalize beyond rewards to permit predictions about arbitrary signals. Rather than predicting the sum of future rewards, we might predict the sum of the future values of a sound or color sensation, or of an internal, highly processed signal such as another prediction. Whatever signal is added up in this way in a value-function-like prediction, we call it the *cumulant* of that prediction. We formalize it in a *cumulant signal* $C_t \in \mathbb{R}$. Using this, a *general value function*, or GVF, is written

$$v_{\pi,\gamma,C}(s) = \mathbb{E}\left[\sum_{k=t}^{\infty}\left(\prod_{i=t+1}^{k}\gamma(S_i)\right)C_{k+1} \;\middle|\; S_t{=}s, A_{t:\infty}\sim\pi\right]. \tag{17.1}$$

As with conventional value functions (such as v_π or q_*) this is an ideal function that we seek to approximate with a parameterized form, which we might continue to denote $\hat{v}(s,\mathbf{w})$, although of course there would have to be a different $\mathbf{w}$ for each prediction, that is, for each choice of π, γ, and C. Because a GVF has no necessary connection to reward, it is perhaps a misnomer to call it a *value* function. One could simply call it a prediction or, to make it more distinctive, a *forecast* (Ring, in preparation). Whatever it is called, it is in the form of a value function and thus can be learned in the usual ways using the methods developed in this book for learning approximate value functions. Along with the learned predictions, we might also learn policies to maximize the predictions in the usual ways by Generalized Policy Iteration (Section 4.6) or by actor–critic methods. In this way an agent could learn to predict and control great numbers of signals, not just long-term reward.

Why might it be useful to predict and control signals other than long-term reward? These are *auxiliary* tasks in that they are extra, in-addition-to, the main task of maximizing reward. One answer is that the ability to predict and control a diverse multitude of signals can constitute a powerful kind of environmental model. As we saw in Chapter 8, a good model can enable the agent to get reward more efficiently. It takes a couple of further concepts to develop this answer clearly, so we postpone it to the next section. First let's consider two simpler ways in which a multitude of diverse predictions can be helpful to a reinforcement learning agent.

One simple way in which auxiliary tasks can help on the main task is that they may require some of the same representations as are needed on the main task. Some of the auxiliary tasks may be easier, with less delay and a clearer connection between actions and outcomes. If good features can be found early on easy auxilary tasks, then those features may significantly speed learning on the main task. There is no necessary reason why this has to be true, but in many cases it seems plausible. For example, if you learn to predict and control your sensors over short time scales, say seconds, then you might plausibly come up with part of the idea of objects, which would then greatly help with the prediction and control of long-term reward.

We might imagine an artificial neural network (ANN) in which the last layer is split into multiple parts, or *heads*, each working on a different task. One head might produce the approximate value function for the main task (with reward as its cumulant) whereas the others would produce solutions to various auxilary tasks. All heads could propagate errors by stochastic gradient descent into the same body—the shared preceding part of the network—which would then try to form representations, in its next-to-last layer, to support all the heads. Researchers have experimented with auxiliary tasks such as predicting change in pixels, predicting the next time step's reward, and predicting the distribution of the return. In many cases this approach has been shown to greatly accelerate learning on the main task (Jaderberg et al., 2017). Multiple predictions have similarly been repeatedly proposed as a way of directing the construction of state estimates (see Section 17.3).

Another simple way in which the learning of auxiliary tasks can improve performance is best explained by analogy to the psychological phenomena of classical conditioning (Section 14.2). One way of understanding classical conditioning is that evolution has built in a reflexive (non-learned) association to a particular action from the prediction of a particular signal. For example, humans and many other animals appear to have a built-in reflex to blink whenever their prediction of being poked in the eye exceeds some threshold. The prediction is learned, but the association from prediction to eye closure is built in, and thus the animal is saved many unprotected pokes in its eye. Similarly, the association from fear to increased heart rate, or to freezing, can be built in. Agent

designers can do something similar, connecting by design (without learning) predictions of specific events to predetermined actions. For example, a self-driving car that learns to predict whether going forward will produce a collision could be given a built-in reflex to stop, or to turn away, whenever the prediction is above some threshold. Or consider a vacuum-cleaning robot that learned to predict whether it might run out of battery power before returning to the charger and that reflexively headed back to the charger whenever the prediction became non-zero. The correct prediction would depend on the size of the house, the room the robot was in, and the age of the battery, all of which would be hard for the robot designer to know. It would be difficult for the designer to build in a reliable algorithm for deciding whether to head back to the charger in sensory terms, but it might be easy to do this in terms of the learned prediction. We foresee many possible ways like this in which learned predictions might combine usefully with built-in algorithms for controlling behavior.

Finally, perhaps the most important role for auxiliary tasks is in moving beyond the assumption we have made throughout this book that the state representation is fixed and given to the agent. To explain this role, we first have to take a few steps back to appreciate the magnitude of this assumption and the implications of removing it. We do that in Section 17.3.

17.2 Temporal Abstraction via Options

An appealing aspect of the MDP formalism is that it can be applied usefully to tasks at many different time scales. One can use it to formalize the task of deciding which muscles to twitch to grasp an object, which airplane flight to take to arrive conveniently at a distant city, and which job to take to lead a satisfying life. These tasks differ greatly in their time scales, yet each can be usefully formulated as an MDP that can be solved by planning or learning processes as described in this book. All involve interaction with the world, sequential decision making, and a goal usefully conceived of as accumulating rewards over time, and so all can be formulated as MDPs.

Although all these tasks can be formulated as MDPs, one might think that they cannot be formulated as a *single* MDP. They involve such different time scales, such different notions of choice and action! It would be no good, for example, to plan a flight across a continent at the level of muscle twitches. Yet for other tasks, grasping, throwing darts, or hitting a baseball, low-level muscle twitches may be just the right level. People do all these things seamlessly without appearing to switch between levels. Can the MDP framework be stretched to cover all the levels simultaneously?

Perhaps it can. One popular idea is to formalize an MDP at a detailed level, with a small time step, yet enable planning at higher levels using extended courses of action that correspond to many base-level time steps. To do this we need a notion of course of action that extends over many time steps and includes a notion of termination. A general way to formulate these two ideas is as a policy, π, and a state-dependent termination function, γ, as in GVFs. We define a pair of these as a generalized notion of action termed an *option*. To execute an option $\omega = \langle \pi_\omega, \gamma_\omega \rangle$ at time t is to obtain the action to take, A_t, from $\pi_\omega(\cdot|S_t)$, then terminate at time $t+1$ with probability $\gamma_\omega(S_{t+1})$. If the option does

not terminate at $t+1$, then A_{t+1} is selected from $\pi_\omega(\cdot|S_{t+1})$, and the option terminates at $t+2$ with probability $\gamma_\omega(S_{t+2})$, and so on until eventual termination. It is convenient to consider low-level actions to be special cases of options—each action a corresponds to an option $\langle \pi_\omega, \gamma_\omega \rangle$ whose policy picks the action ($\pi_\omega(s) = a$ for all $s \in \mathcal{S}$) and whose termination function is zero ($\gamma_\omega(s) = 0$ for all $s \in \mathcal{S}^+$). Options effectively extend the action space. The agent can either select a low-level action/option, terminating after one time step, or select an extended option that might execute for many time steps before terminating.

Options are designed so that they are interchangable with low-level actions. For example, the notion of an action-value function q_π naturally generalizes to an *option-value function* that takes a state and option as input and returns the expected return starting from that state, executing that option to termination, and thereafter following the policy, π. We can also generalize the notion of policy to a *hierarchical policy* that selects from options rather than actions, where options, when selected, execute until termination. With these ideas, many of the algorithms in this book can be generalized to learn approximate option-value functions and hierarchical policies. In the simplest case, the learning process 'jumps' from option initiation to option termination, with an update only occurring when an option terminates. More subtly, updates can be made on each time step, using "intra-option" learning algorithms, which in general require off-policy learning.

Perhaps the most important generalization made possible by option ideas is that of the environmental model as developed in Chapters 3, 4, and 8. The conventional model of an action is the state-transition probabilities and the expected immediate reward for taking the action in each state. How do conventional action models generalize to *option models*? For options, the appropriate model is again of two parts, one corresponding to the state transition resulting from executing the option and one corresponding to the expected cumulative reward along the way. The reward part of an option model, analogous to the expected reward for state–action pairs (3.5), is

$$r(s,\omega) \doteq \mathbb{E}\big[R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots + \gamma^{\tau-1} R_\tau \;\big|\; S_0 = s, A_{0:\tau-1} \sim \pi_\omega, \tau \sim \gamma_\omega\big], \quad (17.2)$$

for all options ω and all states $s \in \mathcal{S}$, where τ is the random time step at which the option terminates according to γ_ω. Note the role of the overall discounting parameter γ in this equation—discounting is according to γ, but termination of the option is according to γ_ω. The state-transition part of an option model is a little more subtle. This part of the model characterizes the probability of each possible resulting state (as in (3.4)), but now this state may result after various numbers of time steps, each of which must be discounted differently. The model for option ω specifies, for each state s that ω might start executing in, and for each state s' that ω might terminate in,

$$p(s'|s,\omega) \doteq \sum_{k=1}^{\infty} \gamma^k \Pr\{S_k = s', \tau = k \mid S_0 = s, A_{0:k-1} \sim \pi_\omega, \tau \sim \gamma_\omega\}. \quad (17.3)$$

Note that, because of the factor of γ^k, this $p(s'|s,\omega)$ is no longer a transition probability and no longer sums to one over all values of s'. (Nevertheless, we continue to use the '|' notation in p.)

The above definition of the transition part of an option model allows us to formulate Bellman equations and dynamic programming algorithms that apply to all options, including low-level actions as a special case. For example, the general Bellman equation for the state values of a hierarchical policy π is

$$v_\pi(s) = \sum_{\omega \in \Omega(s)} \pi(\omega|s) \left[r(s,\omega) + \sum_{s'} p(s'\,|\,s,\omega) v_\pi(s') \right], \tag{17.4}$$

where $\Omega(s)$ denotes the set of options available in state s. If $\Omega(s)$ includes only the low-level actions, then this equation reduces to a version of the usual Bellman equation (3.14), except of course γ is included in the new p (17.3) and thus does not appear. Similarly, the corresponding planning algorithms also have no γ. For example, the value iteration algorithm with options, analogous to (4.10), is

$$v_{k+1}(s) \doteq \max_{\omega \in \Omega(s)} \left[r(s,\omega) + \sum_{s'} p(s'\,|\,s,\omega) v_k(s') \right], \text{ for all } s \in \mathcal{S}.$$

If $\Omega(s)$ includes all the low-level actions available in each s, then this algorithm converges to the conventional v_*, from which the optimal policy can be computed. However, it is particularly useful to plan with options when only a subset of the possible options are considered (in $\Omega(s)$) in each state. Value iteration will then converge to the best hierarchical policy limited to the restricted set of options. Although this policy may be sub-optimal, convergence can be much faster because fewer options are considered and because each option can jump over many time steps.

To plan with options, one must either be given the option models, or learn them. One natural way to learn an option model is to formulate it as a collection of GVFs (as defined in the preceding section) and then learn the GVFs using the methods presented in this book. It is not difficult to see how this could be done for the reward part of the option model. One merely chooses one GVF's cumulant to be the reward ($C_t = R_t$), its policy to be the the option's policy ($\pi = \pi_\omega$), and its termination function to be the discount rate times the option's termination function ($\gamma(s) = \gamma \cdot \gamma_\omega(s)$). The true GVF then equals the reward part of the option model, $v_{\pi,\gamma,C}(s) = r(s,\omega)$, and the learning methods described in this book can be used to approximate it. The state-transition part of the option model is only a little more complicated. One needs to allocate one GVF for each state that the option might terminate in. We don't want these GVFs to accumulate anything except when the option terminates, and then only when the termination is in the appropriate state. This can be achieved by choosing the cumulant of the GVF that predicts transition to state s' to be $C_t = \gamma(S_t) \cdot \mathbb{1}_{S_t = s'}$. The GVF's policy and termination functions are chosen the same as for the reward part of the option model. The true GVF then equals the s' portion of the option's state-transition model, $v_{\pi,\gamma,C}(s) = p(s'\,|\,s,\omega)$, and again this book's methods could be employed to learn it. Although each of these steps is seemingly natural, putting them all together (including function approximation and other essential components) is quite challenging and beyond the current state of the art.

Exercise 17.1 This section has presented options for the discounted case, but discounting is arguably inappropriate for control when using function approximation (Section 10.4). What is the natural Bellman equation for a hierarchical policy, analogous to (17.4), but for the average reward setting (Section 10.3)? What are the two parts of the option model, analogous to (17.2) and (17.3), for the average reward setting? □

17.3 Observations and State

Throughout this book we have written the learned approximate value functions (and the policies in Chapter 13) as functions of the environment's state. This is a significant limitation of the methods presented in Part I, in which the learned value function was implemented as a table such that any value function could be exactly approximated; that case is tantamount to assuming that the state of the environment is completely observed by the agent. But in many cases of interest, and certainly in the lives of all natural intelligences, the sensory input gives only partial information about the state of the world. Some objects may be occluded by others, or behind the agent, or miles away. In these cases, potentially important aspects of the environment's state are not directly observable, and it is a strong, unrealistic, and limiting assumption to assume that the learned value function is implemented as a table over the environment's state space.

The framework of parametric function approximation that we developed in Part II is far less restrictive and, arguably, no limitation at all. In Part II we retained the assumption that the learned value functions (and policies) are functions of the environment's state, but allowed these functions to be arbitrarily restricted by the parameterization. It is somewhat surprising and not widely recognized that function approximation includes important aspects of partial observability. For example, if there is a state variable that is not observable, then the parameterization can be chosen such that the approximate value does not depend on that state variable. The effect is just as if the state variable were not observable. Because of this, all the results obtained for the parameterized case apply to partial observability without change. In this sense, the case of parameterized function approximation includes the case of partial observability.

Nevertheless, there are many issues that cannot be investigated without a more explicit treatment of partial observability. Although we cannot give them a full treatment here, we can outline the changes that would be needed to do so. There are four steps.

First, we would change the problem. The environment would emit not its states, but only *observations*—signals that depend on its state but, like a robot's sensors, provide only partial information about it. For convenience, without loss of generality, we assume that the reward is a direct, known function of the observation (perhaps the observation is a vector, and the reward is one of is components). The environmental interaction would then have no explicit states or rewards, but would simply be an alternating sequence of actions $A_t \in \mathcal{A}$ and observations $O_t \in \mathcal{O}$:

$$A_0, O_1, A_1, O_2, A_2, O_3, A_3, O_4, \ldots,$$

going on forever (cf. Equation 3.1) or forming episodes each ending with a special terminal observation.

Second, we can recover the idea of state as used in this book from the sequence of observations and actions. Let us use the word *history*, and the notation H_t, for an initial portion of the trajectory up to an observation: $H_t \doteq A_0, O_1, \ldots, A_{t-1}, O_t$. The history represents the most that we can know about the past without looking outside of the data stream (because the history is the whole past data stream). Of course, the history grows with t and can become large and unwieldy. The idea of state is that of some compact summary of the history that is as useful as the actual history for predicting the future. Let us be clear about exactly what this means. To be a summary of the history, the state must be a function of history, $S_t = f(H_t)$, and to be as useful for predicting the future as the whole history, it must have what is known as the *Markov property*. Formally, this is a property of the function f. A function f has the Markov property if and only if any two histories h and h' that are mapped by f to the same state ($f(h)=f(h')$) also have the same probabilities for their next observation,

$$f(h) = f(h') \;\Rightarrow\; \Pr\{O_{t+1}=o|H_t=h, A_t=a\} = \Pr\{O_{t+1}=o|H_t=h', A_t=a\}, \quad (17.5)$$

for all $o \in \mathcal{O}$ and $a \in \mathcal{A}$. If f is Markov, then $S_t = f(H_t)$ is a state as we have used the term in this book. Let us henceforth call it a *Markov state* to distinguish it from states that are summaries of the history but fall short of the Markov property (which we will consider shortly).

A Markov state is a good basis for predicting the next observation (17.5) but, more importantly, it is also a good basis for predicting or controlling *anything*. For example, let a *test* be any specific sequence of alternating actions and observations that might occur in the future. For example, a three-step test is denoted $\tau = a_1 o_1 a_2, o_2, a_3, o_3$. The probability of this test given a specific history h is defined as

$$p(\tau|h) \doteq \Pr\{O_{t+1}=o_1, O_{t+2}=o_2, O_{t+3}=o_3 \mid H_t=h, A_t=a_1, A_{t+1}=a_2, A_{t+2}=a_3\}. \quad (17.6)$$

If f is Markov and h and h' are any two histories that map to the same state under f, then for any test τ of any length, its probabilities given the two histories must also be the same:

$$f(h) = f(h') \;\Rightarrow\; p(\tau|h) = p(\tau|h'). \quad (17.7)$$

In other words, a Markov state summarizes all the information in the history necessary for determining any test's probability. In fact, it summarizes all that is necessary for making *any prediction*, including any GVF, and for behaving optimally (if f is Markov, then there is always a deterministic function π such that choosing $A_t \doteq \pi(f(H_t))$ is optimal).

The third step in extending reinforcement learning to partial observability is to deal with certain computational considerations. In particular, we want the state to be a *compact* summary of the history. For example, the identity function completely satisfies the conditions for a Markov f, but would nevertheless be of little use because the corresponding state $S_t = H_t$ would grow with time and become unwieldy, as mentioned earlier, but more fundamentally because it would never recur; the agent would never

encounter the same state twice (in a continuing task) and thus could never benefit from a tabular learning method. We want our states to be compact as well as Markov. There is a similar issue regarding how state is obtained and updated. We don't really want a function f that takes whole histories. Instead, for computational reasons we prefer to obtain the same effect as f with an incremental, recursive update that computes S_{t+1} from S_t, incorporating the next increment of data, A_t and O_{t+1}:

$$S_{t+1} \doteq u(S_t, A_t, O_{t+1}), \text{ for all } t \geq 0, \quad (17.8)$$

with the first state S_0 given. The function u is called the *state-update* function. For example, if f were the identity ($S_t = H_t$), then u would merely extend S_t by appending A_t and O_{t+1} to it. Given f, it is always possible to construct a corresponding u, but it may not be computationally convenient and, as in the identity example, it may not produce a compact state. The state-update function is a central part of any agent architecture that handles partial observability. It must be efficiently computable, as no actions or predictions can be made until the state is available. An overall diagram of such an agent architecture is given in Figure 17.1.

An example of obtaining Markov states through a state-update function is provided by the popular Bayesian approach known as *Partially Observable MDPs*, or *POMDPs*. In this approach the environment is assumed to have a well defined *latent state* X_t that underlies and produces the environment's observations, but is never available to the agent (and is not to be confused with the state S_t used by the agent to make predictions and decisions). The natural Markov state, S_t, for a POMDP is the *distribution* over the latent states given the history, called the *belief state*. For concreteness, assume the usual case in which there are a finite number of hidden states, $X_t \in \{1, 2, \ldots, d\}$. Then the belief state is the vector $S_t \doteq \mathbf{s}_t \in \mathbb{R}^d$ with components

$$\mathbf{s}_t[i] \doteq \Pr\{X_t = i \mid H_t\}, \text{ for all possible latent states } i \in \{1, 2, \ldots, d\}.$$

The belief state remains the same size (same number of components) however t grows. It can also be incrementally updated by Bayes' rule, assuming one has complete knowledge of the internal workings of the environment. Specifically, the ith component of the belief-state update function is

$$u(\mathbf{s}, a, o)[i] \doteq \frac{\sum_{x=1}^{d} \mathbf{s}[x] p(i, o|x, a)}{\sum_{x=1}^{d} \sum_{x'=1}^{d} \mathbf{s}[x] p(x', o|x, a)}, \quad (17.9)$$

for all $a \in \mathcal{A}$, $o \in \mathcal{O}$, and belief states $\mathbf{s} \in \mathbb{R}^d$ with components $\mathbf{s}[x]$, where the four-argument p function here is not the usual one for MDPs (as in Chapter 3), but the analogous one for POMDPs, in terms of the *latent* state: $p(x', o|x, a) \doteq \Pr\{X_t = x', O_t = o \mid X_{t-1} = x, A_{t-1} = a\}$. This approach is popular in theoretical work and has many significant applications, but its assumptions and computational complexity scale poorly and we do not recommend it as an approach to artificial intelligence.

Another example of Markov states is provided by *Predictive State Representations*, or *PSRs*. PSRs address the weakness of the POMDP approach that the semantics of its agent state S_t are grounded in the environment state, X_t, which is never observed and

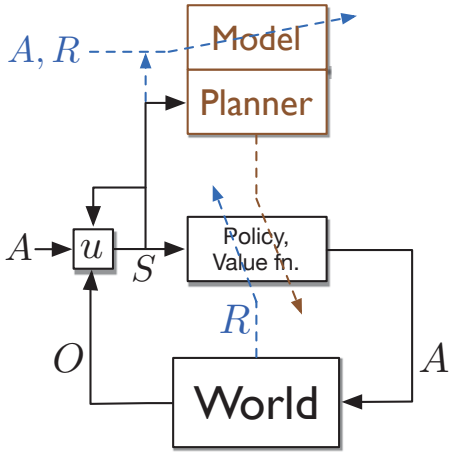


Figure 17.1: A conceptual agent architecture including a model, a planner, and a state-update function. The world in this case receives actions A and emits observations O. The observations and a copy of the action are used by the state-update function u to produce the new state. The new state is input to the policy and value function, producing the next action, and is also input to the planner (and to u). The information flows most responsible for learning are shown by dashed lines that pass diagonally across the boxes that they change. The reward R directly changes the policy and value function. The action, reward, and state change the model, which works closely with the planner to also change the policy and value function. Note that the operation of the planner can be decoupled from the agent–environment interaction, whereas the other processes should operate in lock step with this interaction to keep up with the arrival of new data. Also note that the model and planner do not deal with observations directly, but only with the states produced by u, which can act as targets for model learning.

thus is difficult to learn about. In PSRs and related approaches, the semantics of the agent state is instead grounded in predictions about future observations and actions, which are readily observable. In PSRs, a Markov state is defined as a d-vector of the probabilities of d specially chosen "core" tests as defined above (17.6). The vector is then updated by a state-update function u that is analogous to Bayes rule, but with a semantics grounded in observable data, which arguably makes it easier to learn. This approach has been extended in many ways, including end-tests, compositional tests, powerful "spectral" methods, and closed-loop and temporally abstract tests learned by TD methods. Some of the best theoretical developments are for systems known as *Observable Operator Models* (OOMs) and Sequential Systems (Thon, 2017).

The fourth and final step in our brief outline of how to handle partial observability in reinforcement learning is to re-introduce approximation. As discussed in the introduction to Part II, to approach artificial intelligence ambitiously one must embrace approximation. This is just as true for states as it is for value functions. We must accept and work with an approximate notion of state. The approximate state will play the same role in our algorithms as before, so we continue to use the notation S_t for the state used by the agent, even though it may not be Markov.

Perhaps the simplest example of an approximate state is just the latest observation, $S_t \doteq O_t$. Of course this approach cannot handle any hidden state information. It would be better to use the last k observations and actions, $S_t \doteq O_t, A_{t-1}, O_{t-1}, \ldots, A_{t-k}$, for some $k \geq 1$, which can be achieved by a state-update function that just shifts the new data in and the oldest data out. This *kth-order history* approach is still very simple, but can greatly increase the agent's capabilities compared to trying to use the single immediate observation directly as the state.

What happens when the Markov property (17.5) is only approximately satisfied? Unfortunately, long-term prediction performance can degrade dramatically when the one-step predictions defining the Markov property become even slightly inaccurate. Longer-term tests, GVFs, and state-update functions may all approximate poorly. The short-term and long-term approximation objectives are just different, and there are no useful theoretical guarantees at present.

Nevertheless, there are still reasons to think that the general idea outlined in this section applies to the approximate case. The general idea is that a state that is good for some predictions is also good for others (in particular, that a Markov state, sufficient for one-step predictions, is also sufficient for all others). If we step back from that specific result for the Markov case, the general idea is similar to what we discussed in Section 17.1 with multi-headed learning and auxiliary tasks. We discussed how representations that were good for the auxiliary tasks were often also good for the main task. Taken together, these suggest an approach to both partial observability and representation learning in which multiple predictions are pursued and used to direct the construction of state features. The guarantee provided by the perfect-but-impractical Markov property is replaced by the heuristic that what's good for some predictions may be good for others. This approach scales well with computational resources. With a large machine one could experiment with large numbers of predictions, perhaps favoring those that are most similar to the ones of ultimate interest or that are easiest to learn reliably, or according to some other criteria. It is important here to move beyond selecting the predictions manually. The agent should do it. This would require a general language for predictions, so that the agent can systematically explore a large space of possible predictions, sifting through them for the ones that are most useful.

In particular, both POMDP and PSR approaches can be applied with approximate states. The semantics of the state is useful in forming the state-update function, as it is in these two approaches and in the k-order approach. There is not a strong need for the semantics to be correct in order to retain useful information in the state. Some approaches to state augmentation, such as Echo state networks (Jaeger, 2002), keep almost arbitrary information about the history and can nevertheless perform well. There are many possibilities and we expect more work and ideas in this area. Learning the state-update function for an approximate state is a major part of the representation learning problem as it arises in reinforcement learning.

17.4 Designing Reward Signals

A major advantage of reinforcement learning over supervised learning is that reinforcement learning does not rely on detailed instructional information: generating a reward signal does not depend on knowledge of what the agent's correct actions should be. But the success of a reinforcement learning application strongly depends on how well the reward signal frames the goal of the application's designer and how well the signal assesses progress in reaching that goal. For these reasons, designing a reward signal is a critical part of any application of reinforcement learning.

By designing a reward signal we mean designing the part of an agent's environment that is responsible for computing each scalar reward R_t and sending it to the agent at each time t. In our discussion of terminology at the end of Chapter 14, we said that R_t is more like a signal generated inside an animal's brain than it is like an object or event in the animal's external environment. The parts of our brains that generate these signals for us evolved over millions of years to be well suited to the challenges our ancestors had to face in their struggles to propagate their genes to future generations. We should therefore not think that designing a good reward signal is always an easy thing to do!

One challenge is to design a reward signal so that as an agent learns, its behavior approaches, and ideally eventually achieves, what the application's designer actually desires. This can be easy if the designer's goal is simple and easy to identify, such as finding the solution to a well-defined problem or earning a high score in a well-defined game. In cases like these, it is usual to reward the agent according to its success in solving the problem or its success in improving its score. But some problems involve goals that are difficult to translate into reward signals. This is especially true when the problem requires the agent to skillfully perform a complex task or set of tasks, such as would be required of a useful household robotic assistant. Further, reinforcement learning agents can discover unexpected ways to make their environments deliver reward, some of which might be undesirable, or even dangerous. This is a longstanding and critical challenge for any method, like reinforcement learning, that is based on optimization. We discuss this issue more in Section 17.6, the final section of this book.

Even when there is a simple and easily identifiable goal, the problem of *sparse reward* often arises. Delivering non-zero reward frequently enough to allow the agent to achieve the goal once, let alone to learn to achieve it efficiently from multiple initial conditions, can be a daunting challenge. State–action pairs that clearly deserve to trigger reward may be few and far between, and rewards that mark progress toward a goal can be infrequent because progress is difficult or even impossible to detect. The agent may wander aimlessly for long periods of time (what Minsky, 1961, called the "plateau problem").

In practice, designing a reward signal is often left to an informal trial-and-error search for a signal that produces acceptable results. If the agent fails to learn, learns too slowly, or learns the wrong thing, then the designer tweaks the reward signal and tries again. To do this, the designer judges the agent's performance by criteria that he or she is attempting to translate into a reward signal so that the agent's goal matches his or her own. And if learning is too slow, the designer may try to design a non-sparse reward signal that effectively guides learning throughout the agent's interaction with its environment.

It is tempting to address the sparse reward problem by rewarding the agent for achieving subgoals that the designer thinks are important way stations to the overall goal. But augmenting the reward signal with well-intentioned supplemental rewards may lead the agent to behave very differently from what is intended; the agent may end up not achieving the overall goal at all. A better way to provide such guidance is to leave the reward signal alone and instead augment the value-function approximation with an initial guess of what it should ultimately be, or augment it with initial guesses as to what certain parts of it should be. For example, suppose one wants to offer $v_0 : \mathcal{S} \to \mathbb{R}$ as an initial guess at the true optimal value function v_*, and that one is using linear function approximation with features $\mathbf{x} : \mathcal{S} \to \mathbb{R}^d$. Then one would define the initial value function approximation to be

$$\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) + v_0(s), \tag{17.10}$$

and update the weights $\mathbf{w}$ as usual. If the initial weight vector is $\mathbf{0}$, then the initial value function will be v_0, but the asymptotic solution quality will be determined by the feature vectors as usual. This initialization can be done for arbitrary nonlinear approximators and arbitrary forms of v_0, though it is not guaranteed to always accelerate learning.

A particularly effective approach to the sparse reward problem is the *shaping* technique introduced by the psychologist B. F. Skinner and described in Section 14.3. The effectiveness of this technique relies on the fact that sparse reward problems are not just problems with the reward signal; they are also problems with an agent's policy in preventing the agent from frequently encountering rewarding states. Shaping involves changing the reward signal as learning proceeds, starting from a reward signal that is not sparse given the agent's initial behavior, and gradually modifying it toward the reward signal suited to problem of original interest. Each modification is made so that the agent is frequently rewarded given its current behavior. The agent faces a sequence of increasingly-difficult reinforcement learning problems, where what is learned at each stage makes the next-harder problem relatively easy because the agent now encounters reward more frequently than it would if it did not have prior experience with easier problems. This kind of shaping is an essential technique in training animals, and it is effective in computational reinforcement learning as well.

What if one has no idea what the rewards should be but there is another agent, perhaps a person, who is already expert at the task and whose behavior can be observed? In this case one can use methods known variously as "imitation learning," "learning from demonstration," and "apprenticeship learning." The idea here is to benefit from the expert agent but leave open the possibility of eventually performing better. Learning from an expert's behavior can be done either by learning directly by supervised learning or by extracting a reward signal using what is known as "inverse reinforcement learning" and then using a reinforcement learning algorithm with that reward signal to learn a policy. The task of inverse reinforcement learning as explored by Ng and Russell (2000) is to try to recover the expert's reward signal from the expert's behavior alone. This cannot be done exactly because a policy can be optimal with respect to many different reward signals (for example, any reward signal that gives the same reward for all states and actions), but it is possible to find plausible reward signal candidates. Unfortunately, strong assumptions are required, including knowledge of the environment's dynamics and of the feature vectors

in which the reward signal is linear. The method also requires completely solving the problem (e.g., by dynamic programming methods) multiple times. These difficulties notwithstanding, Abbeel and Ng (2004) argue that the inverse reinforcement learning approach can sometimes be more effective than supervised learning for benefiting from the behavior of an expert.

Another approach to finding a good reward signal is to automate the trial-and-error search for a good signal that we mentioned above. From an application perspective, the reward signal is a parameter of the learning algorithm. As is true for other algorithm parameters, the search for a good reward signal can be automated by defining a space of feasible candidates and applying an optimization algorithm. The optimization algorithm evaluates each candidate reward signal by running the reinforcement learning system with that signal for some number of steps, and then scoring the overall result by a "high-level" objective function intended to encode the designer's true goal, ignoring the limitations of the agent. Reward signals can even be improved via online gradient ascent, where the gradient is that of the high-level objective function (Sorg, Lewis, and Singh, 2010). Relating this approach to the natural world, the algorithm for optimizing the high-level objective function is analogous to evolution, where the high-level objective function is an animal's evolutionary fitness determined by the number of its offspring that survive to reproductive age.

Computational experiments with this bilevel optimization approach—one level analogous to evolution, and the other due to reinforcement learning by individual agents—have confirmed that intuition alone is not always adequate to devise a good reward signal (Singh, Lewis, and Barto, 2009). The performance of a reinforcement learning agent as evaluated by the high-level objective function can be very sensitive to details of the agent's reward signal in subtle ways determined by the agent's limitations and the environments in which it acts and learns. These experiments also demonstrated that an agent's goal should not always be the same as the goal of the agent's designer.

At first this seems counterintuitive, but it may be impossible for the agent to achieve the designer's goal no matter what its reward signal is. The agent has to learn under various kinds of constraints, such as limited computational power, limited access to information about its environment, or limited time to learn. When there are constraints like these, learning to achieve a goal that is different from the designer's goal can sometimes end up getting closer to the designer's goal than if that goal were pursued directly (Sorg, Singh, and Lewis, 2010; Sorg, 2011). Examples of this in the natural world are easy to find. Because we cannot directly assess the nutritional value of most foods, evolution—the designer of our reward signal—gave us a reward signal that makes us seek certain tastes. Though certainly not infallible (indeed, possibly detrimental in environments that differ in certain ways from ancestral environments), this compensates for many of our limitations: our limited sensory abilities, the limited time over which we can learn, and the risks involved in finding a healthy diet through personal experimentation. Similarly, because an animal cannot observe its own evolutionary fitness, that objective function does not work as a reward signal for learning. Evolution instead provides reward signals that are sensitive to observable predictors of evolutionary fitness.

Finally, remember that a reinforcement learning agent is not necessarily like a complete

organism or robot; it can be a component of a larger behaving system. This means that reward signals may be influenced by things inside the larger behaving agent, such as motivational states, memories, ideas, or even hallucinations. Reward signals may also depend on properties of the learning process itself, such as measures of how much progress learning is making. Making reward signals sensitive to information about internal factors such as these makes it possible for an agent to learn how to control the "cognitive architecture" of which it is a part, as well as to acquire knowledge and skills that would be difficult to learn from a reward signal that depended only on external events. Possibilities like these led to the idea of "intrinsically-motivated reinforcement learning" that we briefly discuss further at the end of the following section.

17.5 Remaining Issues

In this book we have presented the foundations of a reinforcement learning approach to artificial intelligence. Roughly speaking, that approach is based on model-free and model-based methods working together, as in the Dyna architecture of Chapter 8, combined with function approximation as developed in Part II. The focus has been on online and incremental algorithms, which we see as fundamental even to model-based methods, and on how these can be applied in off-policy training situations. The full rationale for the latter has been presented only in this last chapter. That is, we have all along presented off-policy learning as an appealing way to deal with the explore/exploit dilemma, but only in this chapter have we discussed learning about many diverse auxiliary tasks simultaneously with GVFs and learning about the world hierarchically in terms of temporally-abstract option models, both of which involve off-policy learning. Much remains to be worked out, as we have indicated throughout the book and as evidenced by the directions for additional research discussed in this chapter. But suppose we are generous and grant the broad outlines of everything that we have done in the book *and* everything that has been outlined so far in this chapter. What would remain after that? Of course we can't know for sure what will be required, but we can make some guesses. In this section we highlight six further issues which it seems to us will still need to be addressed by future research.

First, we still need powerful parametric function approximation methods that work well in fully incremental and online settings. Methods based on deep learning and ANNs are a major step in this direction but, still, only work well with batch training on large data sets, with training from extensive offline self play, or with learning from the interleaved experience of multiple agents on the same task. These and other settings are ways of working around a basic limitation of today's deep learning methods, which struggle to learn rapidly in the incremental, online settings that are most natural for the reinforcement learning algorithms emphasized in this book. The problem is sometimes described as one of "catastrophic interference" or "correlated data." When something new is learned it tends to replace what has previously been learned rather than adding to it, with the result that the benefit of the older learning is lost. Techniques such as "replay buffers" are often used to retain and replay old data so that its benefits are not permanently lost. An honest assessment has to be that current deep learning methods are not well suited to online learning. We see no reason that this limitation is insurmountable, but algorithms

that address it, while at the same time retaining the advantages of deep learning, have not yet been devised. Most current deep learning research is directed toward working around this limitation rather than removing it.

Second (and perhaps closely related), we still need methods for learning features such that subsequent learning generalizes well. This issue is an instance of a general problem variously called "representation learning," "constructive induction," and "meta-learning"—how can we use experience not just to learn a given desired function, but to learn inductive biases such that future learning generalizes better and is thus faster? This is an old problem, dating back to the origins of artificial intelligence and pattern recognition in the 1950s and 1960s.[1] Such age should give one pause. Perhaps there is no solution. But it is equally likely that the time for finding a solution and demonstrating its effectiveness has not yet arrived. Today machine learning is conducted at a far larger scale than it has been in the past, and the potential benefits of a good representation learning method have become much more apparent. We note that a new annual conference—the International Conference on Learning Representations—has been exploring this and related topics every year since 2013. It is also less common to explore representation learning within a reinforcement learning context. Reinforcement learning brings some new possibilities to this old issue, such as the auxiliary tasks discussed in Section 17.1. In reinforcement learning, the problem of representation learning can be identified with the problem of learning the state-update function discussed in Section 17.3.

Third, we still need scalable methods for planning with learned environment models. Planning methods have proven extremely effective in applications such as AlphaGo Zero and computer chess in which the model of the environment is known from the rules of the game or can otherwise be supplied by human designers. But cases of full model-based reinforcement learning, in which the environment model is learned from data and then used for planning, are rare. The Dyna system described in Chapter 8 is one example, but as described there and in most subsequent work it uses a tabular model without function approximation, which greatly limits its applicability. Only a few studies have included learned linear models, and even fewer have also explored including temporally-abstract models using options as discussed in Section 17.2.

More work is needed before planning with learned models can be effective. For example, the learning of the model needs to be selective because the scope of a model strongly affects planning efficiency. If a model focuses on the key consequences of the most important options, then planning can be efficient and rapid, but if a model includes details of unimportant consequences of options that are unlikely to be selected, then planning may be almost useless. Environment models should be constructed judiciously with regard to both their states and dynamics with the goal of optimizing the planning process. The various parts of the model should be continually monitored as to the degree to which they contribute to, or detract from, planning efficiency. The field has not yet addressed this complex of issues or designed model-learning methods that take into account their implications.

[1] Some would claim that deep learning solves this problem, for example, that DQN as described in Section 16.5 illustrates a solution, but we are unconvinced. There is as yet little evidence that deep learning alone solves the representation learning problem in a general and efficient way.

A fourth issue that needs to be addressed in future research is that of automating the choice of tasks on which an agent works and uses to structure its developing competence. It is usual in machine learning for human designers to set the tasks that the learning agent is expected to master. Because these tasks are known in advance and remain fixed, they can be built into the learning algorithm code. However, looking ahead, we will want the agent to make its own choices about what tasks it should try to master. These might be subtasks of a specific overall task that is already known, or they might be intended to create building blocks that permit more efficient learning of many different tasks that the agent is likely to face in the future but which are currently unknown.

These tasks may be like the auxiliary tasks or the GVFs discussed in Section 17.1, or tasks solved by options as discussed in Section 17.2. In forming a GVF, for example, what should the cumulant, the policy, and the termination function be? The current state of the art is to select these manually, but far greater power and generality would come from making these task choices automatically, particularly when they derive from what the agent has previously constructed as a result of representation learning or experience with previous subproblems. If GVF design is automated, then the design choices themselves will have to be explicitly represented. Rather than the task choices being in the mind of the designer and built into the code, they will have to be in the machine itself in such a way that they can be set and changed, monitored, filtered, and searched among automatically. Tasks could then be built hierarchically upon others much like features are in an ANN. The tasks are the questions, and the contents of the ANN are the answers to those questions. We expect there will need to be a full hierarchy of questions to match the hierarchy of answers provided by modern deep learning methods.

The fifth issue that we would like to highlight for future research is that of the interaction between behavior and learning via some computational analog of *curiosity*. In this chapter we have been imagining a setting in which many tasks are being learned simultaneously, using off-policy methods, from the same stream of experience. The actions taken will of course influence this stream of experience, which in turn will determine how much learning occurs and which tasks are learned. When reward is not available, or not strongly influenced by behavior, the agent is free to choose actions that maximize in some sense the learning on the tasks, that is, to use some measure of learning progress as an internal or "intrinsic" reward, implementing a computational form of curiosity. In addition to measuring learning progress, intrinsic reward can, among other possibilities, signal the receipt of unexpected, novel, or otherwise interesting input, or can assess the agent's ability to cause changes in its environment. Intrinsic reward signals generated in these ways can be used by an agent to pose tasks for itself by defining auxiliary tasks, GVFs, or options, as discussed above, so that skills learned in this way can contribute to the agent's ability to master future tasks. The result is a computational analog of something like *play*. Many preliminary studies of such uses of intrinsic reward signals have been conducted, and exciting topics for future research remain in this general area.

A final issue that demands attention in future research is that of developing methods to make it acceptably safe to embed reinforcement learning agents into physical environments. This is one of the most pressing areas for future research, and we discuss it further in the following section.

17.6 The Future of Artificial Intelligence

When we were writing the first edition of this book in the mid-1990s, artificial intelligence was making significant progress and was having an impact on society, though it was mostly still the *promise* of artificial intelligence that was inspiring developments. Machine learning was part of that outlook, but it had not yet become indispensable to artificial intelligence. By today that promise has transitioned to applications that are changing the lives of millions of people, and machine learning has come into its own as a key technology. As we write this second edition, some of the most remarkable developments in artificial intelligence have involved reinforcement learning, most notably "deep reinforcement learning"—reinforcement learning with function approximation by deep artificial neural networks. We are at the beginning of a wave of real-world applications of artificial intelligence, many of which will include reinforcement learning, deep and otherwise, that will impact our lives in ways that are hard to predict.

But an abundance of successful real-world applications does not mean that true artificial intelligence has arrived. Despite great progress in many areas, the gulf between artificial intelligence and the intelligence of humans, and even of other animals, remains great. Superhuman performance can be achieved in some domains, even formidable domains like Go, but it remains a significant challenge to develop systems that are like us in being complete, interactive agents having general adaptability and problem-solving skills, emotional sophistication, creativity, and the ability to learn quickly from experience. With its focus on learning by interacting with dynamic environments, reinforcement learning, as it develops over the future, will be a critical component of agents with these abilities.

Reinforcement learning's connections to psychology and neuroscience (Chapters 14 and 15) underscore its relevance to another longstanding goal of artificial intelligence: shedding light on fundamental questions about the mind and how it emerges from the brain. Reinforcement learning theory is already contributing to our understanding of the brain's reward, motivation, and decision-making processes, and there is good reason to believe that through its links to computational psychiatry, reinforcement learning theory will contribute to methods for treating mental disorders, including drug abuse and addiction.

Another contribution that reinforcement learning can make over the future is as an aid to human decision making. Policies derived by reinforcement learning in simulated environments can advise human decision makers in such areas as education, healthcare, transportation, energy, and public-sector resource allocation. Particularly relevant is the key feature of reinforcement learning that it takes long-term consequences of decisions into account. This is very clear in games like backgammon and Go, where some of the most impressive results of reinforcement learning have been demonstrated, but it is also a property of many high-stakes decisions that affect our lives and our planet. Reinforcement learning follows related methods for advising human decision making that have been developed in the past by decision analysts in many disciplines. With advanced function approximation methods and massive computational power, reinforcement learning methods have the potential to overcome some of the difficulties of scaling up traditional decision-support methods to larger and more complex problems.

The rapid pace of advances in artificial intelligence has led to warnings that artificial intelligence poses serious threats to our societies, even to humanity itself. The renowned scientist and artificial intelligence pioneer Herbert Simon anticipated the warnings we are hearing today in a presentation at the Earthware Symposium at CMU in 2000 (Simon, 2000). He spoke of the eternal conflict between the promise and perils of any new knowledge, reminding us of the Greek myths of Prometheus, the hero of modern science, who stole fire from the gods for the benefit of mankind, and Pandora, whose box could be opened by a small and innocent action to release untold perils on the world. While accepting that this conflict is inevitable, Simon urged us to recognize that as designers of our future and not mere spectators, the decisions *we* make can tilt the scale in Prometheus' favor. This is certainly true for reinforcement learning, which can benefit society but can also produce undesirable outcomes if it is carelessly deployed. Thus, the *safety* of artificial intelligence applications involving reinforcement learning is a topic that deserves careful attention.

A reinforcement learning agent can learn by interacting with either the real world or with a simulation of some piece of the real world, or by a mixture of these two sources of experience. Simulators provide safe environments in which an agent can explore and learn without risking real damage to itself or to its environment. In most current applications, policies are learned from simulated experience instead of direct interaction with the real world. In addition to avoiding undesirable real-world consequences, learning from simulated experience can make virtually unlimited data available for learning, generally at less cost than needed to obtain real experience, and because simulations typically run much faster than real time, learning can often occur more quickly than if it relied on real experience.

Nevertheless, the full potential of reinforcement learning requires reinforcement learning agents to be embedded into the flow of real-world experience, where they act, explore, and learn in *our* world, and not just in *their* worlds. After all, reinforcement learning algorithms—at least those upon which we focus in this book—are designed to learn online, and they emulate many aspects of how animals are able to survive in nonstationary and hostile environments. Embedding reinforcement learning agents in the real world can be transformative in realizing the promises of artificial intelligence to amplify and extend human abilities.

A major reason for wanting a reinforcement learning agent to act and learn in the real world is that it is often difficult, sometimes impossible, to simulate real-world experience with enough fidelity to make the resulting policies, whether derived by reinforcement learning or by other methods, work well—and safely—when directing real actions. This is especially true for environments whose dynamics depend on the behavior of humans, such as in education, healthcare, transportation, and public policy, domains that can surely benefit from improved decision making. However, it is for real-world embedded agents that warnings about potential dangers of artificial intelligence need to be heeded.

Some of these warnings are particularly relevant to reinforcement learning. Because reinforcement learning is based on optimization, it inherits the pluses and minuses of all optimization methods. On the minus side is the problem of devising objective functions, or reward signals in the case of reinforcement learning, so that optimization produces

the desired results while avoiding undesirable results. We said in Section 17.4 that reinforcement learning agents can discover unexpected ways to make their environments deliver reward, some of which might be undesirable, or even dangerous. When we specify what we want a system to learn only indirectly, as we do in designing a reinforcement learning system's reward signal, we will not know how closely the agent will fulfill our desire until its learning is complete. This is hardly a new problem with reinforcement learning; recognition of it has a long history in both literature and engineering. For example, in Goethe's poem "The Sorcerer's Apprentice" (Goethe, 1878), the apprentice uses magic to enchant a broom to do his job of fetching water, but the result is an unintended flood due to the apprentice's inadequate knowledge of magic. In the engineering context, Norbert Wiener, the founder of cybernetics, warned of this problem more than half a century ago by relating the supernatural story of "The Monkey's Paw" (Wiener, 1964): "... it grants what you ask for, not what you should have asked for or what you intend" (p. 59). The problem has also been discussed at length in a modern context by Nick Bostrom (2014). Anyone having experience with reinforcement learning has likely seen their systems discover unexpected ways to obtain a lot of reward. Sometimes the unexpected behavior is good: it solves a problem in a nice new way. In other instances, what the agent learns violates considerations that the system designer may never have thought about. Careful design of reward signals is essential if an agent is to act in the real world with no opportunity for human vetting of its actions or means to easily interrupt its behavior.

Despite the possibility of unintended negative consequences, optimization has been used for hundreds of years by engineers, architects, and others whose designs have positively impacted the world. We owe much that is good in our environment to the application of optimization methods. Many approaches have been developed to mitigate the risk of optimization, such as adding hard and soft constraints, restricting optimization to robust and risk-sensitive policies, and optimizing with multiple objective functions. Some of these approaches have been adapted to reinforcement learning, and more research is needed to address these concerns. The problem of ensuring that a reinforcement learning agent's goal is attuned to our own remains a challenge.

Another challenge if reinforcement learning agents are to act and learn in the real world is not just about what they might learn *eventually*, but about how they will behave while they are learning. How do you make sure that an agent gets enough experience to learn a high-performing policy, all the while not harming its environment, other agents, or itself (or more realistically, while keeping the probability of harm acceptably low)? This problem is also not novel or unique to reinforcement learning. Risk management and mitigation for embedded reinforcement learning is similar to what control engineers have had to confront from the beginning of using automatic control in situations where a controller's behavior can have unacceptable, possibly catastrophic, consequences, as in the control of an aircraft or a delicate chemical process. Control applications rely on careful system modeling, model validation, and extensive testing, and there is a highly-developed body of theory aimed at ensuring convergence and stability of adaptive controllers designed for use when the dynamics of the system to be controlled are not fully known. Theoretical guarantees are never iron-clad because they depend on the validity of the assumptions underlying the mathematics, but without this theory, combined with risk-management and mitigation practices, automatic control—adaptive and otherwise—would not be as

beneficial as it is today in improving the quality, efficiency, and cost-effectiveness of processes on which we have come to rely. One of the most pressing areas for future reinforcement learning research is to adapt and extend methods developed in control engineering with the goal of making it acceptably safe to fully embed reinforcement learning agents into physical environments.

In closing, we return to Simon's call for us to recognize that we are designers of our future and not simply spectators. By decisions we make as individuals, and by the influence we can exert on how our societies are governed, we can work toward ensuring that the benefits made possible by a new technology outweigh the harm it can cause. There is ample opportunity to do this in the case of reinforcement learning, which can help improve the quality, fairness, and sustainability of life on our planet, but which can also release new perils. A threat already here is the displacement of jobs caused by applications of artificial intelligence. Still there are good reasons to believe that the benefits of artificial intelligence can outweigh the disruption it causes. As to safety, hazards possible with reinforcement learning are not completely different from those that have been managed successfully for related applications of optimization and control methods. As reinforcement learning moves out into the real world in future applications, developers have an obligation to follow best practices that have evolved for similar technologies, while at the same time extending them to make sure that Prometheus keeps the upper hand.