Reinforcement Learning: An Introduction

Second edition, in progress

****Draft****

Richard S. Sutton and Andrew G. Barto

A Bradford Book

The MIT Press
Cambridge, Massachusetts
London, England

Chapter 9

On-policy Prediction with Approximation

We have so far assumed that our estimates of value functions are represented as a table with one entry for each state or for each state–action pair. This is a particularly clear and instructive case, but of course it is limited to tasks with small numbers of states and actions. The problem is not just the memory needed for large tables, but the time and data needed to fill them accurately. 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?

This is a severe problem. In many tasks to which we would like to apply reinforcement learning, most states encountered will never have been experienced exactly before. This will almost always be the case when the state or action spaces include continuous variables or complex sensations, such as a visual image. The only way to learn anything at all on these tasks is to generalize from previously experienced states to ones that have never been seen.

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 a large 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 principle, any of the methods studied in these fields can be used in reinforcement learning as described in this chapter.

9.1 Value-Function Approximation

As usual, we begin with the prediction problem of estimating the state-value function v_π from experience generated using 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 parameter vector $\boldsymbol{\theta} \in \mathbb{R}^n$. We will write $\hat{v}(s,\boldsymbol{\theta}) \approx v_\pi(s)$ for the approximated value of state s given parameter vector $\boldsymbol{\theta}$. For example, $\hat{v}$ might be the function computed by an artificial neural network, with $\boldsymbol{\theta}$ the vector of connection weights. By adjusting the weights, any of a wide range of different functions $\hat{v}$ can be implemented by the network. Or $\hat{v}$ might be the function computed by a decision tree, where $\boldsymbol{\theta}$ is all the parameters defining the split points and leaf values of the tree. Typically, the number of parameters n (the number of components of $\boldsymbol{\theta}$) is much less than the number of states, and changing one parameter changes the estimated value of many states. Consequently, when a single state is backed up, the change generalizes from that state to affect the values of many other states.

All of the prediction methods covered in this book have been described as backups, that is, as updates to an estimated value function that shift its value at particular states toward a "backed-up value" for that state. Let us refer to an individual backup by the notation $s \mapsto g$, where s is the state backed up and g is the backed-up value, or target, that s's estimated value is shifted toward. For example, the Monte Carlo backup for value prediction is $S_t \mapsto G_t$, the TD(0) backup is $S_t \mapsto R_{t+1}+\gamma\hat{v}(S_{t+1},\boldsymbol{\theta}_t)$, and the general TD($\lambda$) backup is $S_t \mapsto G_t^\lambda$. In the DP policy evaluation backup $s \mapsto \mathbb{E}_\pi[R_{t+1} + \gamma\hat{v}(S_{t+1},\boldsymbol{\theta}_t) \mid S_t = s]$, an arbitrary state s is backed up, whereas in the other cases the state encountered in (possibly simulated) experience, S_t, is backed up.

It is natural to interpret each backup as specifying an example of the desired input–output behavior of the estimated value function. In a sense, the backup $s \mapsto g$ means that the estimated value for state s should be more like g. Up to now, the actual update implementing the backup has been trivial: the table entry for s's estimated value has simply been shifted a fraction of the way toward g. Now we permit arbitrarily complex and sophisticated function approximation methods to implement the backup. The normal inputs to these methods are 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 backup as a training example. We then interpret the approximate function they produce as an estimated value function.

Viewing each backup 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 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 interacting with the environment or with a model of the 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 GPI control methods 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). Methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning.

What performance measures are appropriate for evaluating function approximation methods? Most supervised learning methods seek to minimize the mean squared error (MSE) over some distribution over the inputs. In our value prediction problem, the inputs are states and the target function is the true value function v_π. Given a parameter vector, we seek to minimize the expected squared difference between the value estimates of the vector and the true values, which we call the *mean square value error* (MSVE):

$$\text{MSVE}(\boldsymbol{\theta}) = \sum_{s \in \mathcal{S}} d(s) \Big[v_\pi(s) - \hat{v}(s,\boldsymbol{\theta}) \Big]^2, \tag{9.1}$$

where $d : \mathcal{S} \to [0,1]$, such that $\sum_s d(s) = 1$, is a distribution over the states specifying the relative importance of errors in different states. This distribution is important because it is usually not possible to reduce the error to zero at all states. After all, there are generally far more states than there are components to $\boldsymbol{\theta}$. The flexibility of the function approximator is thus a scarce resource. Better approximation at some states can be gained, generally, only at the expense of worse approximation at other states. The distribution specifies how these trade-offs should be made.

The distribution d is also usually the distribution from which the states in the training examples are drawn, and thus the distribution of states at which backups are done. If we wish to minimize error over a certain distribution of states, then it makes sense to train the function approximator with examples from that same distribution. For example, if you want a uniform level of error over the entire state set, then it makes sense to train with backups distributed uniformly over the entire state set, such as in the exhaustive sweeps of some DP methods. Henceforth, let us assume that the distribution of states at which backups are done and the distribution that weights errors, d, are the same.

A distribution of particular interest is the one describing the frequency with which states are encountered while the agent is interacting with the environment and selecting actions according to π, the policy whose value function we are approximating. We call this the *on-policy distribution*, in part because it is the distribution of backups in on-policy control methods. Minimizing error over the on-policy distribution focuses function approximation resources on the states that actually occur while following the policy, ignoring those that never occur. The on-policy distribution is also the one for which it is easiest to get training examples using Monte Carlo or TD methods. These methods generate backups from sample experience using the policy π. Because a backup is generated for each state encountered in the experience, the training examples available are naturally distributed according to the on-policy distribution. Stronger convergence results are available for the on-policy distribution than for other distributions, as we discuss later.

It is not completely clear that we should care about minimizing the MSVE. Our goal in value prediction is potentially different because our ultimate purpose is to use the predictions to aid in finding a better policy. The best predictions for that purpose are not necessarily the best for minimizing MSVE. However, it is not yet clear what a more useful alternative goal for value prediction might be. For now, we continue to focus on MSVE.

An ideal goal in terms of MSVE would be to find a *global optimum*, a parameter vector $\boldsymbol{\theta}^*$ for which $\text{MSVE}(\boldsymbol{\theta}^*) \leq \text{MSVE}(\boldsymbol{\theta})$ for all possible $\boldsymbol{\theta}$. 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 parameter vector $\boldsymbol{\theta}^*$ for which $\text{MSVE}(\boldsymbol{\theta}^*) \leq \text{MSVE}(\boldsymbol{\theta})$ for all $\boldsymbol{\theta}$ in some neighborhood of $\boldsymbol{\theta}^*$. 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, convergence to an optimum, or even to within a bounded distance from an optimum cannot be assured. Some methods may in fact diverge, with their MSVE approaching infinity in the limit.

In this section we have 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 backups of the former to generate training examples for the latter. We have also outlined a range of MSVE performance measures to which these methods may aspire. The range of possible 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. If we had another chapter devoted to function approximation, we would also cover at least memory-based and decision-tree methods.

9.2 Gradient-Descent Methods

We now develop in detail one class of learning methods for function approximation in value prediction, those based on gradient descent. Gradient-descent 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 parameter vector is a column vector with a fixed number of real valued components, $\boldsymbol{\theta} \doteq (\theta_1, \theta_2, \ldots, \theta_n)^\top$,[1] and the approximate value function $\hat{v}(s,\boldsymbol{\theta})$ is a smooth differentiable function of $\boldsymbol{\theta}$ for all $s \in \mathcal{S}$. We will be

[1] The $^\top$ denotes transpose, needed here to turn the horizontal row vector into a vertical column vector; in this text all vectors are by default column vectors unless transposed.

updating $\boldsymbol{\theta}$ at each of a series of discrete time steps, $t = 1, 2, 3, \ldots$, so we will need a notation $\boldsymbol{\theta}_t$ for the 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 $\boldsymbol{\theta}$ 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, d, over which we are trying to minimize the MSVE 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 parameter vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t - \frac{1}{2}\alpha\nabla\Big[v_\pi(S_t) - \hat{v}(S_t,\boldsymbol{\theta}_t)\Big]^2 \\ &= \boldsymbol{\theta}_t + \alpha\Big[v_\pi(S_t) - \hat{v}(S_t,\boldsymbol{\theta}_t)\Big]\nabla\hat{v}(S_t,\boldsymbol{\theta}_t), \end{aligned} \tag{9.2}$$

where α is a positive step-size parameter, and $\nabla f(\boldsymbol{\theta}_t)$, for any expression $f(\boldsymbol{\theta}_t)$, denotes the vector of partial derivatives with respect to the components of the weight vector:

$$\left(\frac{\partial f(\boldsymbol{\theta}_t)}{\partial\theta_{t,1}}, \frac{\partial f(\boldsymbol{\theta}_t)}{\partial\theta_{t,2}}, \ldots, \frac{\partial f(\boldsymbol{\theta}_t)}{\partial\theta_{t,n}}\right)^\top.$$

This derivative vector is the *gradient* of f with respect to $\boldsymbol{\theta}_t$. This kind of method is called *gradient descent* because the overall step in $\boldsymbol{\theta}_t$ is proportional to the negative gradient of the example's squared error. This is the direction in which the error falls most rapidly. This method is called *stochastic* gradient descent when the update is done for only this one example, which might have been selected stochastically.

It may not be immediately apparent why only a small step is taken 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 on 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 the step-size parameter decreases over time. If it decreases in such a way as to satisfy the standard stochastic approximation conditions (2.7), then the SGD method (9.2) is guaranteed to converge to a local optimum.

We turn now to the case in which the target output, here denoted $V_t \in \mathbb{R}$, of the tth training example, $S_t \mapsto V_t$, is not the true value, $v_\pi(S_t)$, but some, possibly random, approximation of it. For example, V_t might be a noise-corrupted version of $v_\pi(S_t)$, or it might be one of the backed-up values using $\hat{v}$ mentioned in the previous

section. In such cases we cannot perform the exact update (9.2) because $v_\pi(S_t)$ is unknown, but we can approximate it by substituting V_t in place of $v_\pi(S_t)$. This yields the following general gradient-descent method for state-value prediction:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big[V_t - \hat{v}(S_t,\boldsymbol{\theta}_t)\Big]\nabla\hat{v}(S_t,\boldsymbol{\theta}_t). \tag{9.3}$$

If V_t is an *unbiased* estimate, that is, if $\mathbb{E}[V_t] = v_\pi(S_t)$, for each t, then $\boldsymbol{\theta}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (2.7) for decreasing the step-size parameter α.

For example, suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy π. Let G_t denote the return following each state, S_t. Because the true value of a state is the expected value of the return following it, the Monte Carlo target $V_t = G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$. With this choice, the general gradient-descent method (9.3) 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.

Similarly, we can use n-step TD returns and their averages for V_t. For example, the gradient-descent form of TD(λ) uses the λ-return, $V_t = G_t^\lambda$, as its approximation to $v_\pi(S_t)$, yielding the forward-view update:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big[G_t^\lambda - \hat{v}(S_t,\boldsymbol{\theta}_t)\Big]\nabla\hat{v}(S_t,\boldsymbol{\theta}_t). \tag{9.4}$$

Unfortunately, for $\lambda < 1$, G_t^λ is not an unbiased estimate of $v_\pi(S_t)$, and thus this method does not converge to a local optimum. The situation is the same when DP targets are used such as $V_t = \mathbb{E}_\pi[R_{t+1} + \gamma\hat{v}(S_{t+1},\boldsymbol{\theta}_t) \mid S_t]$. Nevertheless, such bootstrapping methods can be quite effective, and other performance guarantees are available for important special cases, as we discuss later in this chapter. For now we emphasize the relationship of these methods to the general gradient-descent form (9.3). Although increments as in (9.4) are not themselves gradients, it is useful to view this method as a gradient-descent method (9.3) with a bootstrapping approximation in place of the desired output, $v_\pi(S_t)$.

As (9.4) provides the forward view of gradient-descent TD(λ), so the backward view is provided by

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\,\delta_t\,\mathbf{e}_t, \tag{9.5}$$

where δ_t is the usual TD error, now using $\hat{v}$,

$$\delta_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\boldsymbol{\theta}_t) - \hat{v}(S_t,\boldsymbol{\theta}_t), \tag{9.6}$$

and $\mathbf{e}_t \doteq (e_{t,1}, e_{t,2}, \ldots, e_{t,n})^\top$ is a column vector of eligibility traces, one for each component of $\boldsymbol{\theta}_t$, updated by

$$\mathbf{e}_t \doteq \gamma\lambda\mathbf{e}_{t-1} + \nabla\hat{v}(S_t,\boldsymbol{\theta}_t), \tag{9.7}$$

Initialize $\boldsymbol{\theta}$ as appropriate for the problem, e.g., $\boldsymbol{\theta} = \mathbf{0}$
Repeat (for each episode):
  $\mathbf{e} = 0$
  $S \leftarrow$ initial state of episode
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe reward, R, and next state, S'
    $\delta \leftarrow R + \gamma\hat{v}(S',\boldsymbol{\theta}) - \hat{v}(S,\boldsymbol{\theta})$
    $\mathbf{e} \leftarrow \gamma\lambda\mathbf{e} + \nabla\hat{v}(S,\boldsymbol{\theta})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\,\delta\,\mathbf{e}$
    $S \leftarrow S'$
  until S' is terminal

Figure 9.1: On-line gradient-descent TD(λ) for estimating v_π.

with $\mathbf{e}_0 = \mathbf{0}$. A complete algorithm for on-line gradient-descent TD(λ) is given in Figure 9.1.

Two methods for gradient-based function approximation have been used widely in reinforcement learning. One is multilayer artificial neural networks using the error backpropagation algorithm. This maps immediately onto the equations and algorithms just given, where the backpropagation process is the way of computing the gradients. The second popular form is the linear form, which we discuss extensively in the next section.

Exercise 9.1 Show that table-lookup TD(λ) is a special case of general TD(λ) as given by equations (9.5–9.7).

Exercise 9.2 *State aggregation* is a simple form of generalizing function approximation in which states are grouped together, with one table entry (value estimate) used for each group. Whenever a state in a group is encountered, the group's entry is used to determine the state's value, and when the state is updated, the group's entry is updated. Show that this kind of state aggregation is a special case of a gradient method such as (9.4).

Exercise 9.3 The equations given in this section are for the on-line version of gradient-descent TD(λ). What are the equations for the *off-line* version? Give a complete description specifying the new weight vector at the end of an episode, $\boldsymbol{\theta}'$, in terms of the weight vector used during the episode, $\boldsymbol{\theta}$. Start by modifying a forward-view equation for TD(λ), such as (9.4).

9.3 Linear Methods

One of the most important special cases of gradient-descent function approximation is that in which the approximate function, $\hat{v}$, is a linear function of the parameter vector, $\boldsymbol{\theta}$. Corresponding to every state s, there is a vector of features $\boldsymbol{\phi}(s) \doteq (\phi_1(s), \phi_2(s), \ldots, \phi_n(s))^\top$, with the same number of components as $\boldsymbol{\theta}$. The features

may be constructed from the states in many different ways; we cover a few possibilities below. However the features are constructed, the approximate state-value function is given by

$$\hat{v}(s,\boldsymbol{\theta}) \doteq \boldsymbol{\theta}^\top \boldsymbol{\phi}(s) \doteq \sum_{i=1}^{n} \theta_i \phi_i(s). \tag{9.8}$$

In this case the approximate value function is said to be *linear in the parameters*, or simply *linear*.

It is natural to use gradient-descent updates with linear function approximation. The gradient of the approximate value function with respect to $\boldsymbol{\theta}$ in this case is

$$\nabla \hat{v}(s,\boldsymbol{\theta}) = \boldsymbol{\phi}(s).$$

Thus, the general gradient-descent update (9.3) reduces to a particularly simple form in the linear case. In addition, in the linear case there is only one optimum $\boldsymbol{\theta}^*$ (or, in degenerate cases, one set of equally good optima). Thus, any method guaranteed to converge to or near a local optimum is automatically guaranteed to converge to or near the global optimum. Because it is simple in these ways, the linear, gradient-descent 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, the gradient-descent TD(λ) algorithm discussed in the previous section (Figure 9.1) has been proved to converge in the linear case if the step-size parameter is reduced over time according to the usual conditions (2.7). Convergence is not to the minimum-error parameter vector, $\boldsymbol{\theta}^*$, but to a nearby parameter vector, $\boldsymbol{\theta}_\infty$, whose error is bounded according to

$$\text{MSVE}(\boldsymbol{\theta}_\infty) \quad \leq \quad \frac{1-\gamma\lambda}{1-\gamma}\,\text{MSVE}(\boldsymbol{\theta}^*). \tag{9.9}$$

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. An analogous bound applies to other on-policy bootstrapping methods. For example, linear gradient-descent DP backups (9.3), with the on-policy distribution, will converge to the same result as TD(0). Technically, this bound applies only to discounted continuing tasks, but a related result presumably holds for episodic tasks. There are also a few technical conditions on the rewards, features, and decrease in the step-size parameter, which we are omitting here. The full details can be found in the original paper (Tsitsiklis and Van Roy, 1997).

Critical to the above result is that states are backed up according to the on-policy distribution. For other backup 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 10.

Beyond these theoretical results, linear learning methods are also of interest 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 the features. 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 natural features of the task, those along which generalization is most 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.

In general, we also need features for combinations of these natural qualities. This is because the linear form prohibits the representation of 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), a high angular velocity may be either good or bad depending on the angular position. 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. In cases with such interactions one needs to introduce features for conjunctions of feature values when using linear function approximation methods. We next consider some general ways of doing this.

Exercise 9.4 How could we reproduce the tabular case within the linear framework?

Exercise 9.5 How could we reproduce the state aggregation case (see Exercise 8.4) within the linear framework?

Coarse Coding

Consider a task in which the state set is continuous and two-dimensional. A state in this case is a point in 2-space, a vector with two real components. One kind of feature for this case is those corresponding to *circles* in state space, as shown in Figure 9.2. 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*.

Assuming linear gradient-descent function approximation, consider the effect of the size and density of the circles. Corresponding to each circle is a single parameter (a component of $\boldsymbol{\theta}$) that is affected by learning. If we train at one point (state) X, then the parameters of all circles intersecting X will be affected. Thus, by (9.8), the approximate value function will be affected at all points within the union of the circles, with a greater effect the more circles a point has "in common" with X, as shown in Figure 9.2. If the circles are small, then the generalization will be over a short distance, as in Figure 9.3a, whereas if they are large, it will be over a large distance, as in Figure 9.3b. Moreover, 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

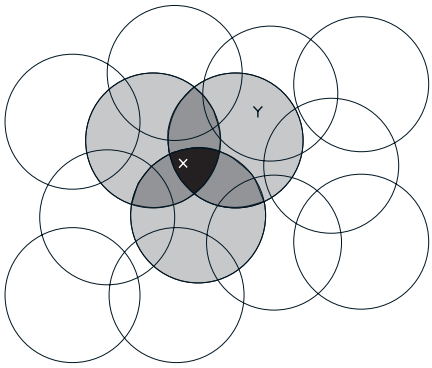


Figure 9.2: Coarse coding. Generalization from state X to state Y depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

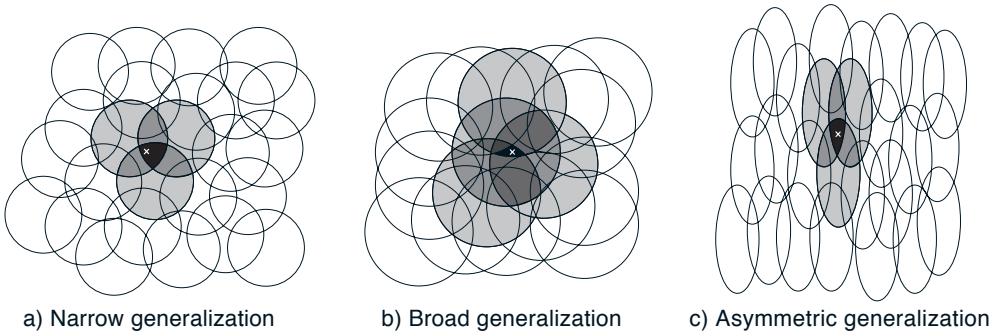


Figure 9.3: Generalization in linear function approximation methods is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.

Figure 9.3c.

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 9.1: 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 (9.3) was used to learn a one-dimensional square-wave function (shown at the top of Figure 9.4). The values of this function were used as the targets, V_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 $\alpha = \frac{0.2}{m}$, where m is the number of features that were present at one time. Figure 9.4 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.

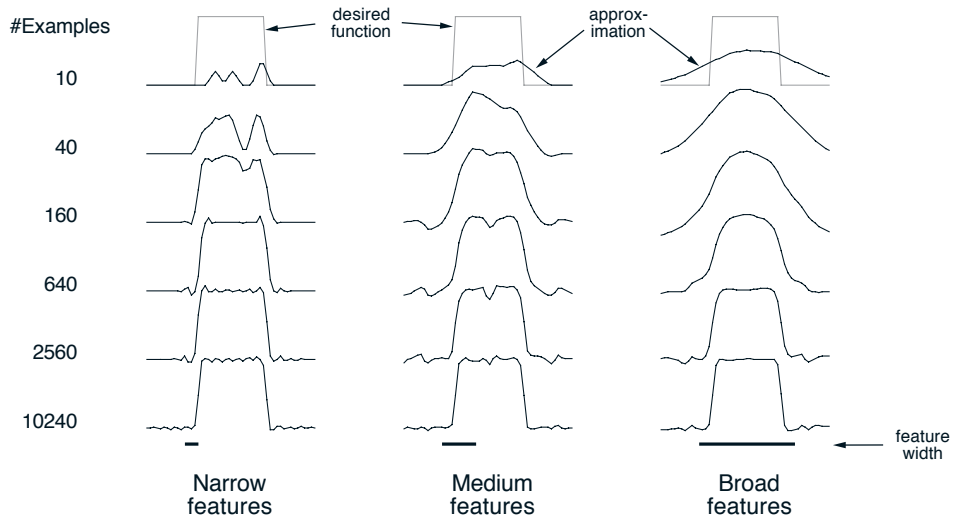


Figure 9.4: Example of feature width's strong effect on initial generalization (first row) and weak effect on asymptotic accuracy (last row).

■

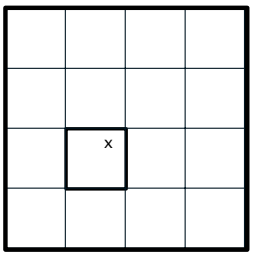
Tile Coding

Tile coding is a form of coarse coding that is particularly well suited for use on sequential digital computers and for efficient on-line learning. In tile coding the receptive fields of the features are grouped into exhaustive partitions of the input space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. Each tile is the receptive field for one binary feature.

An immediate advantage of tile coding is that the overall number of features that are present at one time is strictly controlled and independent of the input 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 $\alpha = \frac{1}{m}$, where m is the number of tilings, results in exact one-trial learning. If the example $s \mapsto v$ is received, then whatever the prior value, $\hat{v}(s,\boldsymbol{\theta})$, the new value will be $\hat{v}(s,\boldsymbol{\theta}) = v$. Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. For example, one might choose $\alpha = \frac{1}{10m}$, in

which case one would move one-tenth of the way to the target in one update.

Because tile coding uses exclusively binary (0–1-valued) features, the weighted sum making up the approximate value function (9.8) is almost trivial to compute. Rather than performing n multiplications and additions, one simply computes the indices of the $m \ll n$ present features and then adds up the m corresponding components of the parameter vector. The eligibility trace computation (9.7) is also simplified because the components of the gradient, $\nabla \hat{v}(s,\boldsymbol{\theta})$, are also usually 0, and otherwise 1.



The computation of the indices of the present features is particularly easy if gridlike tilings are used. The ideas and techniques here are best illustrated by examples. Suppose we address a task with two continuous state variables. Then the simplest way to tile the space is with a uniform two-dimensional grid such as that shown to the right. Given the x and y coordinates of a point in the space, it is computationally easy to determine the index of the tile it is in. When multiple tilings are used, each is offset by a different amount, so that each cuts the space in a different way. In the example shown in Figure 9.5, an extra row and an extra column of tiles have been added to the grid so that no points are left uncovered. The two tiles highlighted are those that are present in the state indicated by the X. The different tilings may be offset by random amounts, or by cleverly designed deterministic strategies (simply offsetting each dimension by the same increment is known not to be a good idea). The effects on generalization and asymptotic accuracy illustrated in Figures 9.3 and 9.4 apply here as well. The width and shape of the tiles should be chosen to match the width of generalization that one expects to be appropriate. The number of tilings should be chosen to influence the density of tiles. The denser the tiling, the finer and more accurately the desired function can be approximated, but the greater the computational costs.

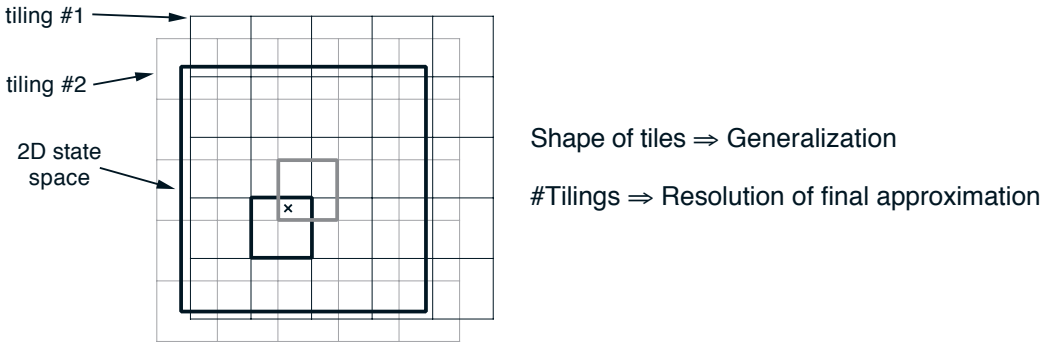


Figure 9.5: Multiple, overlapping gridtilings.

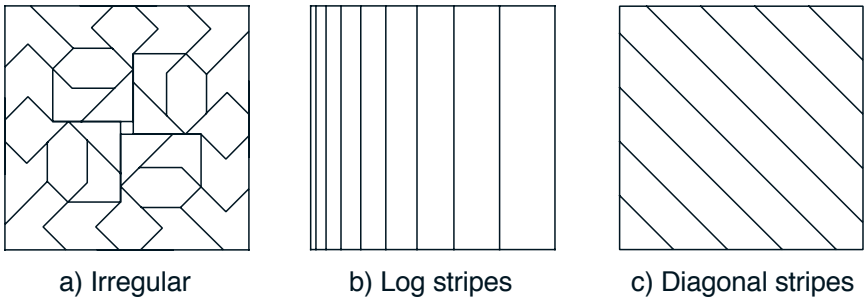
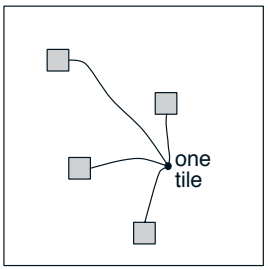


Figure 9.6: Tilings.

It is important to note that the tilings can be arbitrary and need not be uniform grids. Not only can the tiles be strangely shaped, as in Figure 9.6a, but they can be shaped and distributed to give particular kinds of generalization. For example, the stripe tiling in Figure 9.6b will promote generalization along the vertical dimension and discrimination along the horizontal dimension, particularly on the left. The diagonal stripe tiling in Figure 9.6c 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.



Another important 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 tiling. 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. Good public-domain implementations of tile coding, including hashing, are widely available.

Exercise 9.6 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?

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, i, has a Gaussian (bell-shaped) response $\phi_i(s)$

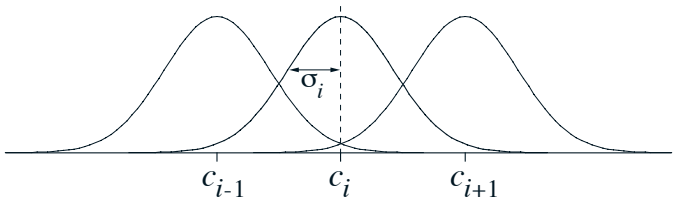


Figure 9.7: One-dimensional radial basis functions.

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:

$$\phi_i(s) \doteq \exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right).$$

The norm or distance metric of course can be chosen in whatever way seems most appropriate to the states and task at hand. Figure 9.7 shows a one-dimensional example with a Euclidean distance metric.

An *RBF network* is a linear function approximator using RBFs for its features. Learning is defined by equations (9.3) and (9.8), exactly as in other linear function approximators. The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. In addition, some learning methods for RBF networks change the centers and widths of the features as well. Such nonlinear methods may be able to fit the target function 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.

Kanerva Coding

On tasks with very high dimensionality, say *hundreds* of dimensions, tile coding and RBF networks become impractical. If we take either method at face value, its computational complexity increases exponentially with the number of dimensions. There are a number of tricks that can reduce this growth (such as hashing), but even these become impractical after a few tens of dimensions.

On the other hand, some of the general ideas underlying these methods can be practical for high-dimensional tasks. In particular, the idea of representing states by a list of the features present and then mapping those features linearly to an approximation may scale well to large tasks. The key is to keep the number of features from scaling explosively. Is there any reason to think this might be possible?

First we need to establish some realistic expectations. Roughly speaking, a function approximator of a given complexity can only accurately approximate target functions of comparable complexity. But as dimensionality increases, the size of the state space inherently increases exponentially. It is reasonable to assume that in the worst case the complexity of the target function scales like the size of the state

space. Thus, if we focus the worst case, then there is no solution, no way to get good approximations for high-dimensional tasks without using resources exponential in the dimension.

A more useful way to think about the problem is to focus on the complexity of the target function as separate and distinct from the size and dimensionality of the state space. The size of the state space may give an upper bound on complexity, but short of that high bound, complexity and dimension can be unrelated. For example, one might have a 1000-dimensional task where only one of the dimensions happens to matter. Given a certain level of complexity, we then seek to be able to accurately approximate any target function of that complexity or less. As the target level of complexity increases, we would like to get by with a proportionate increase in computational resources.

From this point of view, the real source of the problem is the complexity of the target function, or of a reasonable approximation of it, not the dimensionality of the state space. Thus, adding dimensions, such as new sensors or new features, to a task should be almost without consequence if the complexity of the needed approximations remains the same. The new dimensions may even make things easier if the target function can be simply expressed in terms of them. Unfortunately, methods like tile coding and RBF coding do not work this way. Their complexity increases exponentially with dimensionality even if the complexity of the target function does not. For these methods, dimensionality itself is still a problem. We need methods whose complexity is unaffected by dimensionality per se, methods that are limited only by, and scale well with, the complexity of what they approximate.

One simple approach that meets these criteria, which we call *Kanerva coding*, is to choose binary features that correspond to particular *prototype states*. For definiteness, let us say that the prototypes are randomly selected from the entire state space. The receptive field of such a feature is all states sufficiently close to the prototype. Kanerva coding uses a different kind of distance metric than in is used in tile coding and RBFs. For definiteness, consider a *binary* state space and the *hamming distance*, the number of bits at which two states differ. States are considered similar if they agree on enough dimensions, even if they are totally different on others.

The strength of Kanerva coding is that the complexity of the functions that can be learned depends entirely on the number of features, which bears no necessary relationship to the dimensionality of the task. The number of features can be more or less than the number of dimensions. Only in the worst case must it be exponential in the number of dimensions. Dimensionality itself is thus no longer a problem. Complex functions are still a problem, as they have to be. To handle more complex tasks, a Kanerva coding approach simply needs more features. There is not a great deal of experience with such systems, but what there is suggests that their abilities increase in proportion to their computational resources. This is an area of current research, and significant improvements in existing methods can still easily be found.

9.4 Control with Function Approximation

We now extend value prediction methods using function approximation to control methods, following the pattern of GPI. First we extend the state-value prediction methods to action-value prediction methods, then we combine them with policy improvement and action selection techniques. As usual, the problem of ensuring exploration is solved by pursuing either an on-policy or an off-policy approach.

The extension to action-value prediction 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 parameter vector $\boldsymbol{\theta}$. Whereas before we considered random training examples of the form $S_t \mapsto V_t$, now we consider examples of the form $S_t, A_t \mapsto Q_t$ (Q_t here is a scalar target, not the action-value array as in Part I of this book). The target output 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 the one-step Sarsa-style return, $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1},\boldsymbol{\theta}_t)$. The general gradient-descent update for action-value prediction is

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big[Q_t - \hat{q}(S_t, A_t,\boldsymbol{\theta}_t)\Big]\nabla\hat{q}(S_t, A_t,\boldsymbol{\theta}_t).$$

For example, the backward view of the action-value method analogous to TD(λ) is

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\,\delta_t\,\mathbf{e}_t,$$

where

$$\delta_t \doteq R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1},\boldsymbol{\theta}_t) - \hat{q}(S_t, A_t,\boldsymbol{\theta}_t),$$

and

$$\mathbf{e}_t \doteq \gamma\lambda\mathbf{e}_{t-1} + \nabla\hat{q}(S_t, A_t,\boldsymbol{\theta}_t),$$

with $\mathbf{e}_0 \doteq \mathbf{0}$. We call this method *gradient-descent Sarsa(λ)*, particularly when it is elaborated to form a full control method. For a constant policy, this method converges in the same way that TD(λ) does, with the same kind of error bound (9.9).

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 $\hat{q}(S_t, a,\boldsymbol{\theta}_t)$ and then find the greedy action $a^*_t = \arg\max_a \hat{q}(S_t, a,\boldsymbol{\theta}_t)$. Policy improvement is done by changing the estimation policy to the greedy policy (in off-policy methods) or to a soft approximation of the greedy policy such as the ε-greedy policy (in on-policy methods). Actions are selected according to this same policy in on-policy methods, or by an arbitrary policy in off-policy methods.

Figures 9.8 and 9.9 show examples of on-policy (Sarsa(λ)) and off-policy (Watkins's Q(λ)) control methods using function approximation. Both methods use linear,

Let $\boldsymbol{\theta}$ and $\mathbf{e}$ be vectors with one component for each possible feature
Let $\mathcal{F}_a$, for every possible action a, be a set of feature indices, initially empty
Initialize $\boldsymbol{\theta}$ as appropriate for the problem, e.g., $\boldsymbol{\theta} = \mathbf{0}$
Repeat (for each episode):
    $\mathbf{e} = \mathbf{0}$
    $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
    $\mathcal{F}_A \leftarrow$ set of features present in S, A
    Repeat (for each step of episode):
        For all $i \in \mathcal{F}_A$:
            $e_i \leftarrow e_i + 1$ (accumulating traces)
            or $e_i \leftarrow 1$ (replacing traces)
        Take action A, observe reward, R, and next state, S'
        $\delta \leftarrow R - \sum_{i \in \mathcal{F}_A} \theta_i$
        If S' is terminal, then $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\mathbf{e}$; go to next episode
        For all $a \in \mathcal{A}(S')$:
            $\mathcal{F}_a \leftarrow$ set of features present in S', a
            $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta_i$
        $A' \leftarrow$ new action in S' (e.g., ε-greedy)
        $\delta \leftarrow \delta + \gamma Q_{A'}$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\mathbf{e}$
        $\mathbf{e} \leftarrow \gamma\lambda\mathbf{e}$
        $S \leftarrow S'$
        $A \leftarrow A'$

Figure 9.8: Linear, gradient-descent Sarsa(λ) with binary features and ε-greedy policy. Updates for both accumulating and replacing traces are specified.

gradient-descent function approximation with binary features, such as in tile coding and Kanerva coding. Both methods use an ε-greedy policy for action selection, and the Sarsa method uses it for GPI as well. Both compute the sets of present features, $\mathcal{F}_a$, corresponding to the current state and all possible actions, a. If the value function for each action is a separate linear function of the same features (a common case), then the indices of the $\mathcal{F}_a$ for each action are essentially the same, simplifying the computation significantly.

All the methods we have discussed above have used *accumulating* eligibility traces. Although replacing traces are known to have advantages in tabular methods, replacing traces do not directly extend to the use of function approximation. Recall that the idea of replacing traces is to reset a state's trace to 1 each time it is visited instead of incrementing it by 1. But with function approximation there is no single trace corresponding to a state, just a trace for each component of $\boldsymbol{\theta}$, which corresponds to many states. One approach that seems to work well for linear, gradient-descent function approximation methods with binary features is to treat the features as if they were states for the purposes of replacing traces. That is, each time a state is encountered that has feature i, the trace for feature i is set to 1 rather than being incremented by 1, as it would be with accumulating traces.

When working with state–action traces, it may also be useful to clear (set to zero) the traces of all nonselected actions in the states encountered (see Section 7.8). This

Let $\boldsymbol{\theta}$ and $\mathbf{e}$ be vectors with one component for each possible feature
Let $\mathcal{F}_a$, for every possible action a, be a set of feature indices, initially empty
Initialize $\boldsymbol{\theta}$ as appropriate for the problem, e.g., $\boldsymbol{\theta} = \mathbf{0}$
Repeat (for each episode):
    $\mathbf{e} = \mathbf{0}$
    $S \leftarrow$ initial state of episode
    Repeat (for each step of episode):
        For all $a \in \mathcal{A}(S)$:
            $\mathcal{F}_a \leftarrow$ set of features present in S, a
            $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta_i$
        $A^* \leftarrow \arg\max_a Q_a$
        $A \leftarrow A^*$ with prob. $1 - \varepsilon$, else a random action $\in \mathcal{A}(S)$
        If $A \neq A^*$, then $\mathbf{e} = \mathbf{0}$
        Take action A, observe reward, R, and next state, S'
        $\delta \leftarrow R - Q_A$
        For all $i \in \mathcal{F}_A$:
            $e_i \leftarrow e_i + 1$ (accumulating traces)
            or $e_i \leftarrow 1$ (replacing traces)
        If S' is terminal, then $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\mathbf{e}$; go to next episode
        For all $a \in \mathcal{A}(S')$:
            $\mathcal{F}_a \leftarrow$ set of features present in S', a
            $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta_i$
        $\delta \leftarrow \delta + \gamma \max_{a \in \mathcal{A}(S')} Q_a$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\mathbf{e}$
        $\mathbf{e} \leftarrow \gamma\lambda\mathbf{e}$
        $S \leftarrow S'$

Figure 9.9: A linear, gradient-descent version of Watkins's Q(λ) with binary features and ε-greedy policy. Updates for both accumulating and replacing traces are specified.

idea can also be extended to the case of linear function approximation with binary features. For each state encountered, we first clear the traces of all features for the state and the actions not selected, then we set to 1 the traces of the features for the state and the action that was selected. As we noted for the tabular case, this may or may not be the best way to proceed when using replacing traces. A procedural specification of both kinds of traces, including the optional clearing for nonselected actions, is given for the Sarsa algorithm in Figure 9.8.

Example 9.2: 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 9.10. 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 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.

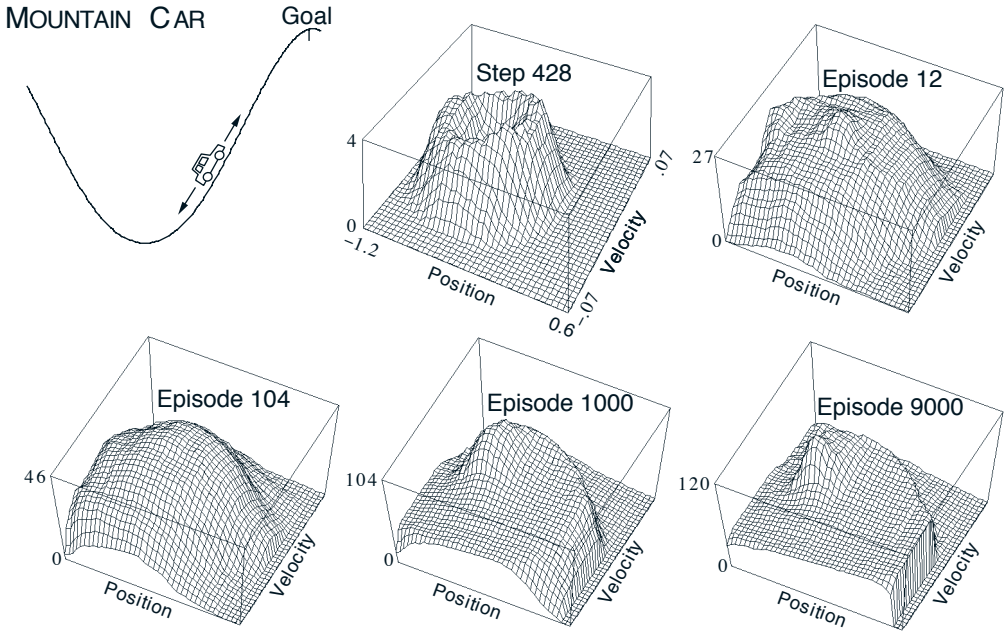


Figure 9.10: The mountain–car task (upper left panel) and the cost-to-go function ($-\max_a \hat{q}(s,a,\boldsymbol{\theta})$) learned during one run.

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, p_t, and velocity, $\dot{p}_t$, are updated by

$$p_{t+1} \doteq \mathit{bound}\big[p_t + \dot{p}_{t+1}\big]$$

$$\dot{p}_{t+1} \doteq \mathit{bound}\big[\dot{p}_t + 0.001A_t - 0.0025\cos(3p_t)\big],$$

where the *bound* operation enforces $-1.2 \leq p_{t+1} \leq 0.5$ and $-0.07 \leq \dot{p}_{t+1} \leq 0.07$. When p_{t+1} reached the left bound, $\dot{p}_{t+1}$ 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 and velocity uniformly chosen from these ranges. To convert the two continuous state variables to binary features, we used gridtilings as in Figure 9.5. We used ten 9×9 tilings, each offset by a random fraction of a tile width.

The Sarsa algorithm in Figure 9.8 (using replace traces and the optional clearing) readily solved this task, learning a near optimal policy within 100 episodes. Figure 9.10 shows the negative of the value function (the *cost-to-go* function) learned on one run, using the parameters $\lambda = 0.9$, $\varepsilon = 0$, and $\alpha = 0.05$ (e.g., $\frac{0.5}{m}$). 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

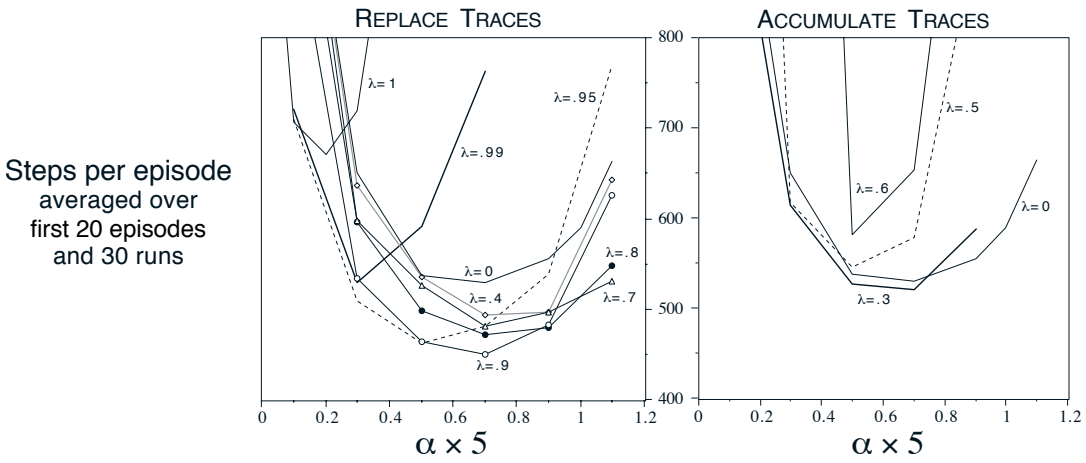


Figure 9.11: The effect of α, λ, and the kind of traces on early performance on the mountain–car task. This study used five 9×9 tilings.

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 9.11 shows the results of a detailed study of the effect of the parameters α and λ, and of the kind of traces, on the rate of learning on this task. ■

9.5 Should We Bootstrap?

At this point you may be wondering why we bother with bootstrapping methods at all. Nonbootstrapping methods can be used with function approximation more reliably and over a broader range of conditions than bootstrapping methods. Nonbootstrapping methods achieve a lower asymptotic error than bootstrapping methods, even when backups are done according to the on-policy distribution. By using eligibility traces and $\lambda = 1$, it is even possible to implement nonbootstrapping methods on-line, in a step-by-step incremental manner. Despite all this, in practice bootstrapping methods are usually the methods of choice.

In empirical comparisons, bootstrapping methods usually perform much better than nonbootstrapping methods. A convenient way to make such comparisons is to use a TD method with eligibility traces and vary λ from 0 (pure bootstrapping) to 1 (pure nonbootstrapping). Figure 9.12 summarizes a collection of such results. In all cases, performance became much worse as λ approached 1, the nonbootstrapping case. The example in the upper right of the figure is particularly significant in this regard. This is a policy evaluation (prediction) task and the performance measure used is MSVE (at the end of each episode, averaged over the first 20 episodes). Asymptotically, the $\lambda = 1$ case must be best according to this measure, but here,

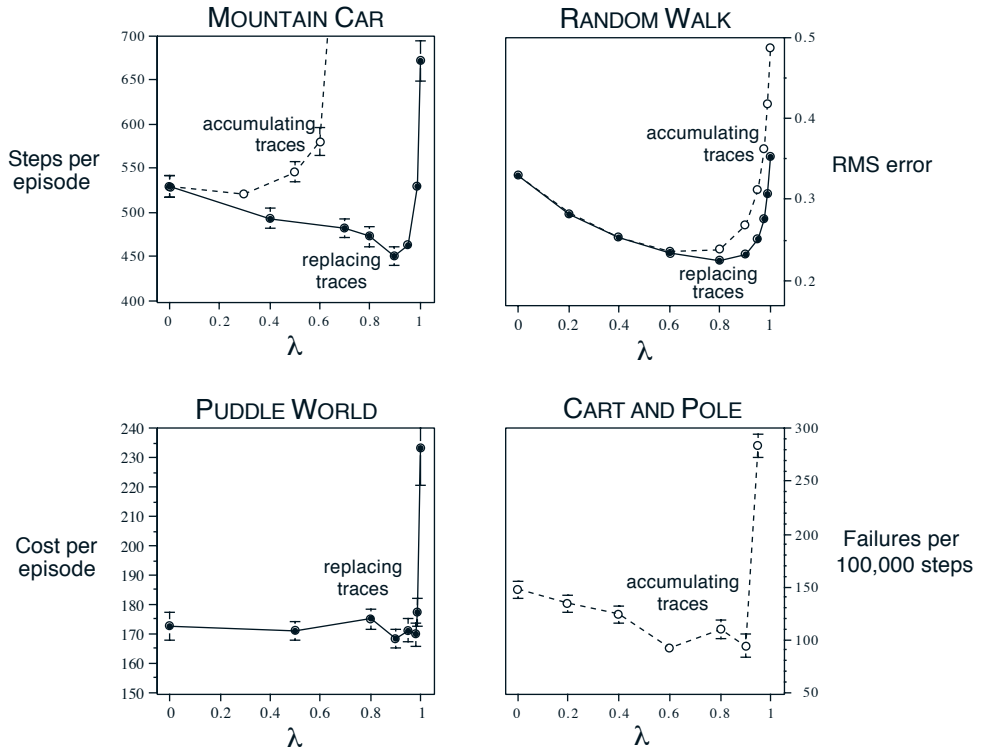


Figure 9.12: The effect of λ on reinforcement learning performance. In all cases, the better the performance, the *lower* the curve. The two left panels are applications to simple continuous-state control tasks using the Sarsa(λ) algorithm and tile coding, with either replacing or accumulating traces (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).

short of the asymptote, we see it performing much worse.

At this time it is unclear why methods that involve some bootstrapping perform so much better than pure nonbootstrapping methods. It could be that bootstrapping methods learn faster, or it could be that they actually learn something better than nonbootstrapping methods. The available results indicate that nonbootstrapping methods are better than bootstrapping methods at reducing MSVE from the true value function, but reducing MSVE is not necessarily the most important goal. For example, if you add 1000 to the true action-value function at all state–action pairs, then it will have very poor MSVE, but you will still get the optimal policy. Nothing quite that simple is going on with bootstrapping methods, but they do seem to do something right. We expect the understanding of these issues to improve as research continues.

9.6 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 backup as a training example. *Gradient-descent methods*, in particular, allow a natural extension to function approximation of all the techniques developed in previous chapters, including eligibility traces. *Linear* gradient-descent methods are particularly appealing theoretically and work 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. Linear methods include radial basis functions, tile coding, and Kanerva coding. Backpropagation methods for multilayer neural networks are methods for *nonlinear* gradient-descent function approximation.

For the most part, the extension of reinforcement learning prediction and control methods to gradient-descent forms is straightforward for the on-policy case. On-policy bootstrapping methods converge reliably with linear gradient-descent function approximation to a solution with mean-squared error bounded by $\frac{1-\gamma\lambda}{1-\gamma}$ times the minimum possible error. Bootstrapping methods are of persistent interest in reinforcement learning, despite their limited theoretical guarantees, because in practice they usually work significantly better than nonbootstrapping methods. The off-policy case involves considerably greater subtlety and is postponed to a later (future) chapter.

Bibliographical and Historical Remarks

Despite our treatment of generalization and function approximation late in the book, they have always been an integral part of reinforcement learning. It is only in the last decade or less that the field has focused on the tabular case, as we have here for the first eight chapters. 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.

9.2 Gradient-descent methods for the 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).

Gradient-descent analyses of TD learning date back at least to Sutton (1988). Methods more sophisticated than the simple gradient-descent methods covered in this section have also been studied in the context of reinforcement

learning, such as quasi-Newton methods (Werbos, 1990) and recursive-least-squares methods (Bradtke, 1993, 1994; Bradtke and Barto, 1996; Bradtke, Ydstie, and Barto, 1994). Bertsekas and Tsitsiklis (1996) provide a good discussion of these methods.

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).

9.3 TD(λ) with linear gradient-descent function approximation was first explored by Sutton (1984, 1988), who proved convergence of TD(0) in the mean to the minimal MSVE solution for the case in which the feature vectors, $\{\boldsymbol{\phi}(s) : s \in \mathcal{S}\}$, are linearly independent. Convergence with probability 1 for general λ 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 on-line updating. All of these results assumed linearly independent feature vectors, which implies at least as many components to $\boldsymbol{\theta}_t$ as there are states. Convergence of linear TD(λ) for the more interesting 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 Section 9.2, the bound on the asymptotic error of TD(λ) and other bootstrapping methods. Recently they extended their analysis to the undiscounted continuing case (Tsitsiklis and Van Roy, 1999).

Our presentation of the range of possibilities for linear function approximation is based on that by Barto (1990). The term *coarse coding* is due to Hinton (1984), and our Figure 9.2 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.

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 known in the literature. The term "tile coding" is new to 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).

Function approximation using radial basis functions (RBFs) has received wide attention ever since being related to neural networks by Broomhead and Lowe (1988). Powell (1987) reviewed earlier uses of RBFs, and Poggio and Girosi (1989, 1990) extensively developed and applied this approach.

What we call "Kanerva coding" was introduced by Kanerva (1988) as part of his more general idea of *sparse distributed memory*. A good review of this and related memory models is provided by Kanerva (1993). This approach has been pursued by Gallant (1993) and by Sutton and Whitehead (1993), among others.

9.4 Q(λ) with function approximation was first explored by Watkins (1989). Sarsa(λ) with function approximation was first explored by Rummery and Niranjan (1994). The mountain–car example is based on a similar task studied by Moore (1990). The results on it presented here are from Sutton (1996) and Singh and Sutton (1996).

Convergence of the Sarsa control method presented in this section has not been proved. The Q-learning control method is now known not to be sound and will diverge for some problems. Convergence results for control methods with state aggregation and other special kinds of function approximation are proved by Tsitsiklis and Van Roy (1996), Singh, Jaakkola, and Jordan (1995), and Gordon (1995).

The use of function approximation in reinforcement learning goes back to the early neural networks of Farley and Clark (1954; Clark and Farley, 1955), who used reinforcement learning to adjust the parameters of linear threshold functions representing policies. 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 neural network methods. These differences between Holland's approach and ours are not surprising because Holland's ideas were developed during a period when neural networks 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.

A number of reinforcement learning studies using function approximation methods that we have not covered previously should be mentioned. 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. Hampson (1983, 1989) was an early proponent of multilayer neural networks for learning value functions. Anderson (1986, 1987) coupled a TD algorithm with the error backpropagation algorithm to learn a value function. Barto and Anandan (1985) introduced a stochastic version of Widrow, Gupta, and Maitra's (1973) *selective bootstrap algorithm*, which they called the *associative reward-penalty* (A_{R-P}) *algorithm*. Williams (1986, 1987, 1988, 1992) extended this type of algorithm to a general class of REINFORCE algorithms, showing that they perform stochastic gradient ascent on the expected reinforcement. Gullapalli (1990) and Williams devised algorithms for learning generalizing policies for the case of continuous actions. Phansalkar and Thathachar (1995) proved both local and global convergence theorems for modified versions of REINFORCE algorithms. 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).