Chapter 9:

On-policy Prediction with Approximation
$V(s_t) \leftarrow V(s_t) + \alpha \left[ \text{Target} - V(s_t) \right]$ 

Now $V(s_t) \propto \hat{V}(s_t, \theta) \quad \theta \in \mathbb{R}^n \quad n \ll |S|$

Value

Function $\theta \leftarrow \theta + \alpha \left[ \text{Target} - \hat{V}(s_t, \theta) \right] \nabla_\theta \hat{V}(s_t, \theta)$

Approximation (VFA)

Stochastic gradient descent (SGD)

$$\theta \leftarrow \theta + \alpha \nabla_\theta \text{Error} = \nabla_\theta f(\theta) = \left[ \frac{df_1(\theta)}{d\theta}, \ldots, \frac{df_n(\theta)}{d\theta} \right]^T$$

$$\theta \leftarrow \theta - \alpha \nabla_\theta \left[ \text{Target} - \hat{V}(s_t, \theta) \right]$$

$$\theta \leftarrow \theta - 2\alpha \left[ \text{Target} - \hat{V}(s_t, \theta) \right] \nabla_\theta \hat{V}(s_t, \theta)$$

QED

Linear VFA

$$\hat{V}(s, \theta) = \theta^T \phi(s) = \sum_{i=1}^n \theta_i \phi_i(s) \quad \phi(s) \in \mathbb{R}^n \quad \phi: S \rightarrow \mathbb{R}^n$$

$$\nabla_\theta \hat{V}(s, \theta) = \nabla_\theta \phi(s) = \sum_{i=1}^n \theta_i \phi_i(s) = \phi(s) \quad \text{QED}$$

Semigradient algorithms are when Target$_t$ depends on $\theta$,

But we ignore $\nabla_\theta \text{Target}_t$ (take it as 0) anyway.
3 waves of neural networks

• First explored in the 1950-60s: Perceptron, Adaline…
  • only one learnable layer

• Revived in the 1980-90s as Connectionism, Neural Networks
  • exciting multi-layer learning using backpropagation (SGD);
    many successful applications; remained popular in engineering

• Revived again in ~2010 as Deep Learning
  • dramatically improved over state-of-the-art in speech recognition
    and visual object recognition, transforming these fields
  • the best algorithms were essentially the same as in the 1980s,
    except with faster computers and larger training sets

i.e., NNs won (eventually) because their performance scaled with
Moore’s law, whereas competing methods did not
Deep learning
≡ multi-layer neural networks with many layers

- Each line has a learned connection weight
- Each node combines its weighted inputs, then applies a nonlinear transformation
- For each image, the network produces class labels as output, and true class labels are provided by people (supervised learning)
- Then each weight is incremented so as to reduce the squared error (stochastic gradient descent, backpropagation)
Value function approximation (VFA) replaces the table with a general parameterized form

\[ \hat{v}(S_t, \theta) \]
Stochastic Gradient Descent (SGD) is the idea behind most approximate learning.

General SGD: \( \theta \leftarrow \theta - \alpha \nabla_{\theta} \text{Error}^2 \)

For VFA: \( \leftarrow \theta - \alpha \nabla_{\theta} [\text{Target}_t - \hat{\upsilon}(S_t, \theta)]^2 \)

Chain rule: \( \leftarrow \theta - 2\alpha [\text{Target}_t - \hat{\upsilon}(S_t, \theta)] \nabla_{\theta} [\text{Target}_t - \hat{\upsilon}(S_t, \theta)] \)

Semi-gradient: \( \leftarrow \theta + \alpha [\text{Target}_t - \hat{\upsilon}(S_t, \theta)] \nabla_{\theta} \hat{\upsilon}(S_t, \theta) \)

Linear case: \( \leftarrow \theta + \alpha [\text{Target}_t - \hat{\upsilon}(S_t, \theta)] \phi(S_t) \)

Action-value form: \( \theta \leftarrow \theta + \alpha [\text{Target}_t - \hat{q}(S_t, A_t, \theta)] \phi(S_t, A_t) \)
A natural objective in VFA is to minimize the Mean Square Value Error

\[
\text{MSVE}(\theta) = \sum_{s \in S} d(s) \left[ v_\pi(s) - \hat{v}(s, \theta) \right]^2
\]

where \( d(s) \) is the fraction of time steps spent in state \( s \)

True SGD will converge to a local minimum of the error objective

In linear VFA, there is only one minimum: local=global
Gradient Monte Carlo Algorithm for Approximating $\hat{v} \approx v_\pi$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : S \times \mathbb{R}^n \to \mathbb{R}$

Initialize value-function weights $\theta$ as appropriate (e.g., $\theta = 0$)
Repeat forever:
   Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using $\pi$
   For $t = 0, 1, \ldots, T - 1$:
      $\theta \leftarrow \theta + \alpha [G_t - \hat{v}(S_t, \theta)] \nabla \hat{v}(S_t, \theta)$

If $U_t$ is an unbiased estimate, that is, if $\mathbb{E}[U_t] = v_\pi(S_t)$, for each $t$, then 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.

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 $G(n)$ or the DP target $P_{a,s_0,r}(\pi | S_t, a)$ all depend on the current value of the weight vector $\theta_t$, 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 $\theta_t$. This step would not be valid if a bootstrapping estimate was 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 $\theta_t$ 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 which makes them often clearly preferred. One reason for this is that they are typically significantly faster to learn, 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 $U_t = R_{t+1} + \hat{v}(S_{t+1}, \theta)$ as its target. Complete pseudocode for this method is given in the box at the top of the next page.
State aggregation is the simplest kind of VFA

- States are partitioned into disjoint subsets (groups)
- One component of $\theta$ is allocated to each group

$$\hat{v}(s, \theta) \doteq \theta_{\text{group}(s)}$$

$$\nabla_{\theta} \hat{v}(s, \theta) \doteq [0, 0, \ldots, 0, 1, 0, 0, \ldots, 0]$$

Recall:
$$\theta \leftarrow \theta + \alpha [Target_t - \hat{v}(S_t, \theta)] \nabla_{\theta} \hat{v}(S_t, \theta)$$
The 1000-state random walk example

- States are numbered 1 to 1000
- Walks start in the near middle, at state 500
- At each step, *jump* to one of the 100 states to the right, or to one of the 100 states to the left
- If the jump goes beyond 1 or 1000, terminates with a reward of \(-1\) or \(+1\)
  (otherwise \(R_t=0\))

\[S_0 = 500\]
\[S_1 \in \{400..499\} \cup \{501..600\}\]
The whole value function over 1000 states will be approximated with 10 numbers!
Gradient MC works well on the 1000-state random walk using *state aggregation*

- 10 groups of 100 states
- after 100,000 episodes
- $\alpha = 2 \times 10^{-5}$

- state distribution affects accuracy
Semi-gradient TD(0) for estimating $\hat{v} \approx v_{\pi}$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : S^+ \times \mathbb{R}^n \to \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$

Initialize value-function weights $\theta$ arbitrarily (e.g., $\theta = 0$)
Repeat (for each episode):
  Initialize $S$
  Repeat (for each step of episode):
    Choose $A \sim \pi(\cdot|S)$
    Take action $A$, observe $R, S'$
    $\theta \leftarrow \theta + \alpha[R + \gamma \hat{v}(S', \theta) - \hat{v}(S, \theta)] \nabla \hat{v}(S, \theta)$
    $S \leftarrow S'$
  until $S'$ is terminal

Example 9.1: State Aggregation on the 1000-state Random Walk
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 $\theta$) 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, $r \hat{v}(S_t, \theta)$, is 1 for $S_t$'s group's component and 0 for the other components.

Consider a 1000-state version of the random walk task (Examples 6.2 and 7.1). 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 $v_{\pi}$ for this task. It is nearly a straight line, but tilted slightly toward the horizontal and curving further in this direction 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 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
TD converges to the TD fixedpoint, $\theta_{TD}$, a biased but interesting answer

**TD(0) update:**

$$
\theta_{t+1} = \theta_t + \alpha \left( R_{t+1} + \gamma \theta_t \phi_{t+1} - \theta_t \phi_t \right) \phi_t \\
= \theta_t + \alpha \left( R_{t+1} \phi_t - \phi_t (\phi_t - \gamma \phi_{t+1})^T \theta_t \right)
$$

In expectation:

$$
E[\theta_{t+1}|\theta_t] = \theta_t + \alpha (b - A \theta_t),
$$

where

$$
b \approx E[R_{t+1} \phi_t] \in \mathbb{R}^n \quad \text{and} \quad A \approx E \left[ \phi_t (\phi_t - \gamma \phi_{t+1})^T \right] \in \mathbb{R}^n \times \mathbb{R}^n
$$

**Fixedpoint analysis:**

$$
b - A \theta_{TD} = 0 \\
\Rightarrow \quad b = A \theta_{TD} \\
\Rightarrow \quad \theta_{TD} = A^{-1} b
$$

**Guarantee:**

$$
\text{MSVE}(\theta_{TD}) \leq \frac{1}{1 - \gamma} \min_{\theta} \text{MSVE}(\theta)
$$
Gradient TD is less accurate than MC on the 1000-state random walk using state aggregation.

- 10 groups of 100 states
- after 100,000 episodes
- $\alpha = 2 \times 10^{-5}$

Relative values are still pretty accurate.
Bootstrapping still greatly speeds learning

### 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 195) 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 MC methods, as we investigated fully with the multi-step TD methods of Chapter 7. The right panel of Figure 9.2 shows results with an $n$-step semi-gradient TD method using state aggregation and the 1000-state random walk that are strikingly similar to those we obtained earlier with tabular methods and the 19-state random walk. To obtain such quantitatively similar results we switched the state aggregation to 20 groups of 50 states each. The 20 groups are then quantitatively close to the 19 states of the tabular problem. In particular, the state transitions

### Average RMS error over 1000 states and first 10 episodes

<table>
<thead>
<tr>
<th>Average RMS error</th>
<th>$n=1$</th>
<th>$n=2$</th>
<th>$n=4$</th>
<th>$n=8$</th>
<th>$n=16$</th>
<th>$n=32$</th>
<th>$n=64$</th>
<th>$n=128$</th>
<th>$n=256$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 states</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1000 states aggregated into 20 groups of 50
Bootstrapping still greatly speeds learning very much like the tabular 19-state walk.
With binary features, a continuous state space can be coarsely coded, adding generalization
The width of the receptive fields determines breadth of generalization.

1D example, supervised training

<table>
<thead>
<tr>
<th>#Examples</th>
<th>Narrow features</th>
<th>Medium features</th>
<th>Broad features</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td><img src="image1" alt="Narrow features" /></td>
<td><img src="image2" alt="Medium features" /></td>
<td><img src="image3" alt="Broad features" /></td>
</tr>
<tr>
<td>40</td>
<td><img src="image1" alt="Narrow features" /></td>
<td><img src="image2" alt="Medium features" /></td>
<td><img src="image3" alt="Broad features" /></td>
</tr>
<tr>
<td>160</td>
<td><img src="image1" alt="Narrow features" /></td>
<td><img src="image2" alt="Medium features" /></td>
<td><img src="image3" alt="Broad features" /></td>
</tr>
<tr>
<td>640</td>
<td><img src="image1" alt="Narrow features" /></td>
<td><img src="image2" alt="Medium features" /></td>
<td><img src="image3" alt="Broad features" /></td>
</tr>
<tr>
<td>2560</td>
<td><img src="image1" alt="Narrow features" /></td>
<td><img src="image2" alt="Medium features" /></td>
<td><img src="image3" alt="Broad features" /></td>
</tr>
<tr>
<td>10240</td>
<td><img src="image1" alt="Narrow features" /></td>
<td><img src="image2" alt="Medium features" /></td>
<td><img src="image3" alt="Broad features" /></td>
</tr>
</tbody>
</table>
Tile coding is coarse coding for digital computers, with rectangular receptive fields, controlled overlap.

2D example
Nevertheless, tile coding is very flexible.

Non-traditional tilings:

- a) Irregular
- b) Log stripes
- c) Diagonal stripes
- Hashing
Tile coding works better than state aggregation on the 1000-state random walk.

Figure 9.10: Why we use coarse coding. Shown are learning curves on the 1000-state random walk example for the gradient MC 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.

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 $n$ multiplications and additions, one simply computes the indices of the $m \times n$ active features and then adds up the corresponding components of the weight vector.

Generalization occurs to states other than the one trained if the 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 below 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.

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 $k$ the number of tilings, then $w/k$ is a fundamental unit. Within small squares $w/k$ 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 $w/k$ in any cartesian direction, the feature representation changes.

- groups/tiles of 200 states
- $\alpha$ set so that initial learning rate is the same for both methods
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. Good open-source implementations of tile coding, including hashing, are widely available.

**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, \(i\), has a Gaussian (bell-shaped) response 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, \(\sigma_i\):

\[
i(s) = \exp(-||s - c_i||^2 / 2\sigma_i^2).
\]

The norm or distance metric of course can be chosen in whatever way seems most appropriate to the states and task at hand. Figure 9.13 shows a one-dimensional example with a Euclidean distance metric.

The primary advantage of RBFs over binary features is that they produce approximative 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.

**Figure 9.13:**
One-dimensional radial basis functions.

1D radial basis functions

A 2D approx. value function learned with 2D radial basis functions

Smooth-edged receptive fields are little different but increase computational complexity
Conclusions

- Value-function approximation by stochastic gradient descent enables RL to be applied to arbitrarily large state spaces.
- Most algorithms just carry over the Targets from the tabular case.
- With bootstrapping (TD), we don’t get true gradient descent methods.
  - This complicates the analysis.
  - But the linear, on-policy case is still guaranteed convergent.
  - And learning is still much faster.
- For continuous state spaces, coarse/tile coding is a good strategy.
- For ambitious AI, artificial neural networks are an interesting strategy.