# RL Ch9 On-policy Prediction with Approximation

Le-Rong Hsu

July 2023

## 1 Introduction

- RL is a science of decision making
- agent → **policy**, **value function**, **model**
- Decisions (actions) affects **agent state** and **environment state**

### Why Function Approximation

- The policy, value function, model are all function
- If the state set is too large, we need to approximate
  - Go: $10^{1}70$ states
  - Helicopter: continuous state space
  - self-driving car: real world
- This is called **deep reinforcement learning**, which uses **neural network** to represent these functions
- e.g. Alpha Go → value network, policy network

### About This Chapter

I will skip 9.7, 9.9 and 9.12 in this note because they are less important than other sections. Readers can check the textbook if needed.

## 2 Value-function Approximation

For tabular methods, we update the **lookup tables** of value functions, in which every state has an entry $v_\pi(s)$ or $q_\pi(s, a)$. To store a lookup table, we usually use a numpy array to achieve this. However, this only works for appropriate size of the state set due to large memory cost and computation.

For large state set, we approximate the value function with **function approximation**. Let $\mathbf{w} \in \mathbb{R}^n$ be a **feature vector** whose components are **feature weights**. In value approximation, we approximate $v_\pi(s)$ with $\hat{v}(s, \mathbf{w})$ by updating the feature weights of $\mathbf{w}$.

There's a notation introduced in this section: $s \mapsto u$, which simply means estimate $\mapsto$ target. For example, the Monte Carlo update for value prediction is $S_t \mapsto G_t$, the TD(0) update us $S_t \mapsto R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$, and the n-step TD update is $S_t \mapsto G_{t:t+n}$. For the DP policy-evaluation update, it is $s \mapsto \mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) | S_t = s]$.

# 3 The Prediction Objective $\overline{\text{VE}}$ (Loss)

Define the **state distribution** $\mu(s) \geq 0, \sum_s \mu(s) = 1$ to represent how much we care about this state. The **error** of a state means the difference between $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$. The **mean square error** is defined as

$$\overline{\text{VE}}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[ v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 . \tag{9.1}$$

Often $\mu(s)$ is chosen to be the fraction of time spent in $s$. Under on-policy training this is called the **on-policy distribution**; we focus entirely on this case in this chapter. In continuing tasks, the on-policy distribution is the stationary distribution under $\pi$.

## On-policy distribution in episodic tasks

- $h(s)$: the probability that an episode begins in state $s$
- $\eta(s)$: the number of time steps spent in state s in a single episode (on average)

If transitions are made into $s$ from a preceding state $\bar{s}$, the time spent in $s$ is

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s}, a). \tag{9.2}$$

The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}. \tag{9.3}$$

This is the natural choice without discounting. If there is discounting ($\gamma < 1$) it should be treated as a form of termination, which can be done simply by including a factor of $\gamma$ in the second term of (9.2).

The best value function for this purpose is not necessarily the best for minimizing $\overline{\text{VE}}$. Nevertheless, it is not yet clear what a more useful alternative goal for value prediction might be. For now, we will focus on $\overline{\text{VE}}$.

# 4 Stochastic-gradient and Semi-gradient Methods

Assume that on each step, we obsereve a new example $S_t \mapsto v_\pi(S_t)$. The **Stochastic gradient descent** is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \tag{9.4}$$

$$= \mathbf{w}_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{9.5}$$

where $\alpha$ is the step size and

$$\nabla f(\mathbf{w}) \doteq \left( \frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \cdots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^T. \tag{9.6}$$

$\alpha$ should decreases over time to ensure the convergence of SGD.

Sometimes, $v_\pi(S_t)$ is unknown, so the original SGD does not work. Let the target output to be $U_t$ which is some, possibly random, approximation to $v_\pi(S_t)$. For example, $U_t$ might be a noise-corrupted version of $v_\pi(S_t)$. This yields general SGD method:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[ U_t - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t). \tag{9.7}$$

If $U_t$ is unbiased, i.e., $\pi[U_t | S_t = s] = v_\pi(s)$, and the usual stochastic approximation conditions (2.7) are satisfied, then $\mathbf{w}_t$ is guaranteed to converge to a local minimum.

**An Important Idea about G.D.**

It may not be immediately apparent why SGD takes only a small step in the direction of the gradient. Could we not move all the way in this direction and completely eliminate the error on the example?

- we do not seek or expect to find a value function that has zero error for all states
- only an approximation that **balances the errors in different states**

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 -step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r} \pi(a|S_t) p(s', r|S_t, a)[r + \gamma \hat{v}(s', \mathbf{w_t})]$ all depend on $\mathbf{w_t}$ which implies that they are all biased and will not produce a true gradient descent method.

## Semi-gradient Methods

Bootstrapping methods are not in fact instances of true gradient descent (Barnard, 1993).

**Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$**

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using $\pi$
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

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**. One reason for this is that they typically enable significantly **faster learning**, as we have seen in Chapters 6 and 7. Another is that they enable learning to be continual and online, **without waiting for the end of an episode**.

**Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$**

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action $A$, observe $R, S'$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until $S$ is terminal

# 5 Linear Methods

$\hat{v}(\cdot, \mathbf{w})$ can be **a linear function of.the weight vector**, $\mathbf{w}$. Corresponding to every state $s$, there is a real-valued vector $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \cdots, x_d(s))^T$ with the same number of components as $\mathbf{w}$. Linear methods approximate the state-value function by the inner product of $\mathbf{w}$ and $\mathbf{x}$:

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T\mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s). \tag{9.8}$$

In this case, we say the approximate value function is **linear in weights**, or simply **linear**.

$\mathbf{x}(s)$ is called the **feature vector representing** $s$, and each component $x_i(s) : \mathcal{S} \to \mathbb{R}$ is called a **feature** of $s$. For linear methods, these features are **basis functions** because they form a linear basis for the set of approximate functions.

The gradient of the approximate value function with respect to w in this case is

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s).$$

Thus, in the linear case the **general SGD** update (9.7) reduces to a particularly simple form:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[U_t - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(S_t).$$

## Advantages of Linear Methods

- **guaranteed to converge** (to the global minimum)
- very efficient in terms of both data and computation

## Linear TD(0)

The semi-gradient TD(0) algorithm also converges under linear function approximation, but this does not follow from general results on SGD; a separate theorem is necessary.

### TD fixed point

The weight vector converged to is also not the global optimum, but rather a point near the local optimum.

The update at each time t is

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha(R_{t+1} + \gamma \mathbf{w}_t^T \mathbf{x}_{t+1} - \mathbf{w}_t^T \mathbf{x}_t)\mathbf{x}_t \\ &= \mathbf{w}_t + \alpha(R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T \mathbf{w}_t) \end{aligned} \tag{9.9}$$

where $\mathbf{x_t} = \mathbf{x}(\mathbf{S_t})$. Once the system has reached steady state, for any given $\mathbf{w_t}$, the expected next weight vector can be written

$$\mathbb{E}[\mathbf{w}_{t+1} | \mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t) \tag{9.10}$$

with

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \text{ and } \mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T] \in \mathbb{R}^{d \times d}. \tag{9.11}$$

If the system converges, it must converge to the weight vector $\mathbf{w_{TD}}$ at which

$$\begin{aligned} &\mathbf{b} - \mathbf{A}\mathbf{w}_{TD} = 0 \\ \implies &\mathbf{b} = \mathbf{A}\mathbf{w}_{TD} \\ \implies &\mathbf{w}_{TD} \doteq \mathbf{A}^{-1}\mathbf{b} \end{aligned}$$

## Proof of Convergence of Linear TD(0)

Write (9.10) as

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = (\mathbf{I} - \alpha\mathbf{A})\mathbf{w}_t + \alpha\mathbf{b} \tag{9.13}$$

### Insight

Suppose $\mathbf{A}$ is a diagonal matrix. If $\mathbf{A}$ is negative, then $\mathbf{I} - \alpha\mathbf{A}$ will be greater than 1, and each component of $\mathbf{w}_t$ will be amplified, which lead to divergence (here, assume that $\mathbf{w}$ and $\mathbf{b}$ are positive). If $\alpha$ is small enough, then all diagonal elements of $\mathbf{I}-\alpha\mathbf{A}$ will be between 0 and 1. $\mathbf{w}_t$ shrinks so the stability is assured.

### Proof (actually this is a special case)

We still consider diagonal $\mathbf{A}$. In general, $\mathbf{w}_t$ will be reduced toward zero whenever $\mathbf{A}$ is positive definite. Our goal is thus to show that $\mathbf{A}$ is positive definite.

*Remark.* Positive definiteness also ensures that $\mathbf{A}^{-1}$ exists (pf. by contradiction).

For linear TD(0), in the continuing case with $\gamma < 1$, the $\mathbf{A}$ (9.11) can be written as

$$\begin{aligned}
\mathbf{A} &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{r,s'} p(r, s'|s, a)\mathbf{x}(s)(\mathbf{x}(s) - \gamma\mathbf{x}(s'))^T \\
&= \sum_s \mu(s) \sum_{s'} p(s'|s)\mathbf{x}(s)(\mathbf{x}(s) - \gamma\mathbf{x}(s'))^T \\
&= \sum_s \mu(s)\mathbf{x}(s) \left(\mathbf{x}(s) - \gamma \sum_{s'} p(s'|s)\mathbf{x}(s')\right)^T \\
&= \mathbf{X}^T\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})\mathbf{X}
\end{aligned}$$

- $\mu(s)$: stationary distribution under $\pi$
- $p(s'|s)$: probability of transition from $s$ to $s'$ under $\pi$
- $\mathbf{P}$: matrix of $p(s'|s)$
- $\mathbf{D}$: $|\mathcal{S}| \times |\mathbf{S}|$ diagonal matrix with $\mu(s)$ on its diagonal
- $\mathbf{X}$: $|\mathcal{S}| \times d$ matrix with $\mathbf{x}(s)$ as its row

Now it suffices to show that $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$ is positive definite. The theorems are given by Sutton (1988, p. 27) and Varga (1962, p. 23).

**Theorem.** *Any matrix $M$ is positive definite if and only if the symmetric matrix $S = M + M^T$ is positive definite.*

**Theorem.** *Any symmetric real matrix $S$ is positive definite if all of its diagonal entries are positive and greater than the sum of the absolute values of the corresponding off-diagonal entries.*

For our key matrix, $\mathbf{D}(\mathbf{I}-\alpha\mathbf{A})$, the diagonal entries are positive and the off-diagonal entries are negative. The row sums are positive since $\mathbf{P}$ is a stochastic matrix and $\gamma < 1$. It remains to show that the column sums are nonnegative.

Note that the row vector (column sums) of $\mathbf{M}$ can be written as $\mathbf{1}^T\mathbf{M}$. Since $\mu$ is stationary, for $|\mathcal{S}|$-vector $\mu$, $\mu = \mathbf{P}\mu$. Then

$$\begin{aligned}\mathbf{1}^T\mathbf{D}(\mathbf{I}-\gamma\mathbf{P}) &= \mathbf{u}^T(\mathbf{I}-\gamma\mathbf{P}) \\ &= \mu^T - \gamma\mu^T\mathbf{P} \\ &= \mu^T - \gamma\mu^T \\ &= (1-\gamma)\mu^T\end{aligned}$$

all components of which are positive. Thus, the key matrix and its A matrix are positive definite, and on-policy TD(0) is stable.■

### Remarks on TD fixed point

At the TD fixed point, it has also been proven (in the continuing case) that the $\overline{\mathbf{VE}}$ is within a bounded expansion of the lowest possible error:

$$\overline{\text{VE}}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma}\min_{\mathbf{w}}\overline{\text{VE}}(\mathbf{w}) \tag{9.14}$$

That is, the asymptotic error of the TD method is no more than $\frac{1}{\gamma}$ times the smallest possible error, that attained in the limit by the Monte Carlo method.

Because $\gamma$ is often near 1, the factor can be quite large, so there is substantial potential loss in asymptotic performance with the TD method.

## 6 Feature Basis for Linear Methods

### Polynomial Basis

Suppose a reinforcement learning problem has states with two numerical dimensions, $s = (s_1, s_2) \in \mathbb{R}^2$. If we simply choose $\mathbf{x}(s) = (s_1, s_2)^T$ as its feature vector, then we neglect the **interaction** of $s_1$ and $s_2$.

We can choose for example, $\mathbf{x}(s) = (1, s_1, s_2, s_1s_2)^T$ as the feature vector of $s$. We can also choose $\mathbf{x}(s) = (1, s_1, s_2, s_1s_2, s_1^2, s_2^2, s_1^2s_2, s_1s_2^2, s_1^2s_2^2)$. Such feature vectors enable approximations as arbitrary quadratic functions of the state numbers—even though the approximation is still linear in the weights that have to be learned.

*Remark.* Each component has $n+1$ choices and we have $k$ components, so there are $(n+1)^k$ possible features.

### 6.1 Fourier Basis

Suppose each state $s$ corresponds to a vector of $k$ numbers, $s = (s_1, s_2, ..., s_k)^T$ with each $s_i \in [0, 1]$. The $i$th feature in the order-$n$ Fourier cosine basis can

Suppose each state $s$ corresponds to $k$ numbers, $s_1$, $s_2$, ..., $s_k$, with each $s_i \in \mathbb{R}$. For this $k$-dimensional state space, each order-$n$ polynomial-basis feature $x_i$ can be written as

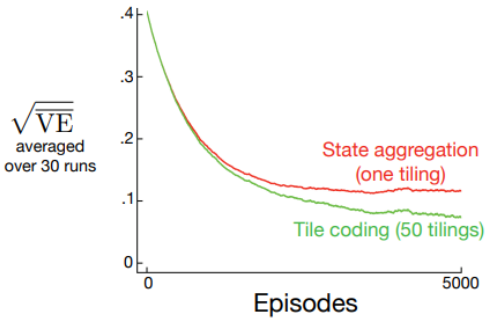
$$x_i(s) = \Pi_{j=1}^{k} s_j^{c_{i,j}}, \tag{9.17}$$

where each $c_{i,j}$ is an integer in the set $\{0, 1, \ldots, n\}$ for an integer $n \geq 0$. These features make up the order-$n$ polynomial basis for dimension $k$, which contains $(n+1)^k$ different features.

then be written

$$x_i(s) = \cos(\pi \mathbf{s}^T \mathbf{c}^i) \tag{9.8}$$

for $\mathbf{c}^i = (c_1^i, c_2^i, ..., c_k^i)$, $c_j^i \in \{0, 1, ..., n\}$ for $j = 1, ..., k$ and $i = 1, ..., (n+1)^k$. Each $x_i(s)$ thus has $(n+1)^k$ possible outcome. The number of features in the order-$n$ Fourier basis grows exponentially with the dimension of the state space. Conclusion: in general, we do not recommend using polynomials for




online learning.

# 7 Coarse Coding and Tile Coding

## Coarse Coding

The features correspond to the aspects of the state space, so how we define the features does matter.

Imagine that each circle is a feature. Each point is a state.

- present (in the circle): 1
- absent: (out of the circle) 0

This is also called the **binary feature**. However, this is not flexible since we ignore the **degree** and the **interactions** of features. We can define the features in another way.

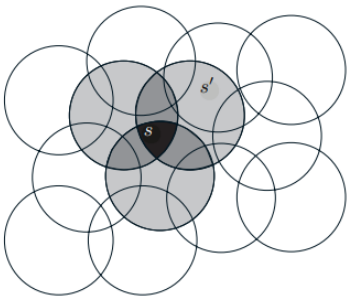



**Figure 9.6** Coarse coding

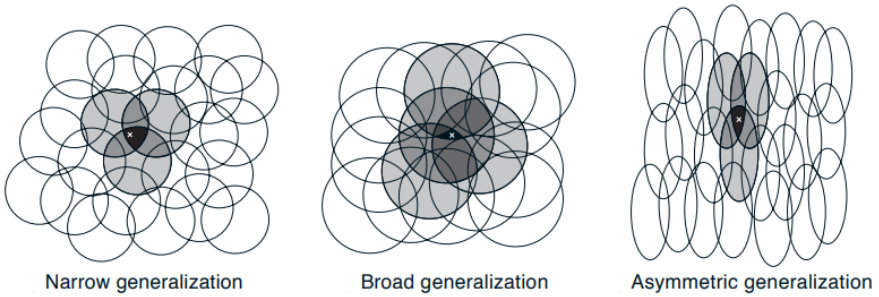



**Figure 9.7** generalization of coarse coding

Let each circle be a single weight (a component of $\mathbf{w}$) that is affected by learning. If we train at one state, then the weights of all circles intersecting that state will be affected. Thus, the approximate value function will be affected at all states within the union of the circles.

**Example 9.3: Coarseness of Coarse Coding** This example illustrates the effect on learning of the size of the receptive fields in coarse coding. Linear function approximation based on coarse coding and (9.7) was used to learn a one-dimensional square-wave function.

- target: $U_t$
- intervals: narrow, medium, broad
- the same density of features
- $\alpha = \frac{0.2}{n}$

Result:

- broad features: the generalization tended to be broad
- narrow features: only the close neighbors of each trained point were changed

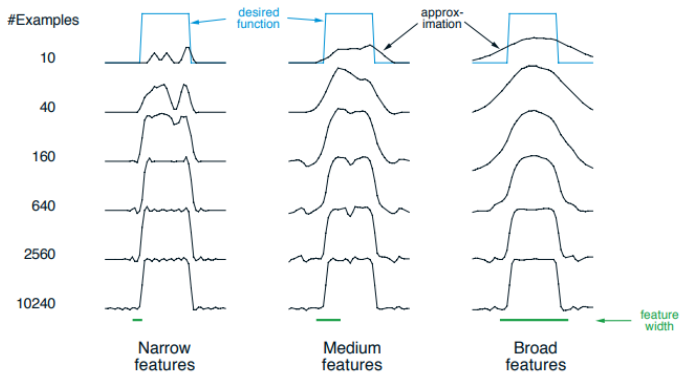



**Figure 9.8**

- the final function learned was affected only slightly by the width of the features

Conclusion: receptive field shape tends to have a strong effect on generalization but little effect on asymptotic solution quality.

## Tile Coding

Tile coding is a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient. **It may be the most practical feature representation for modern sequential digital computers.**

### Partition and offset

The receptive fields of the features are grouped into **partitions** of the state space. Each such partition is called a **tiling**, and each element of the partition is called a **tile**. The tiles or receptive field here are squares rather than the circles.

- **offset → overlap**
- step size → $\alpha = 1/n$, $n$ is the number of tilings
- For each state vector $\mathbf{x}(s)$, $\mathbf{x}(s)$ has $4 \times 4 \times 4$ components in this example, and each component has either 0 or 1.

Figure 9.10 shows the advantage of tile coding (also coarse coding). The task is 1000-state random walk.

- red: single tiling with 5 tiles 200 states wide, step size $\alpha = 0.0001$
- green: 50 tilings with 5 tiles 4 states wide, step size $\alpha = 0.0001/50$

Result: use coarse coding

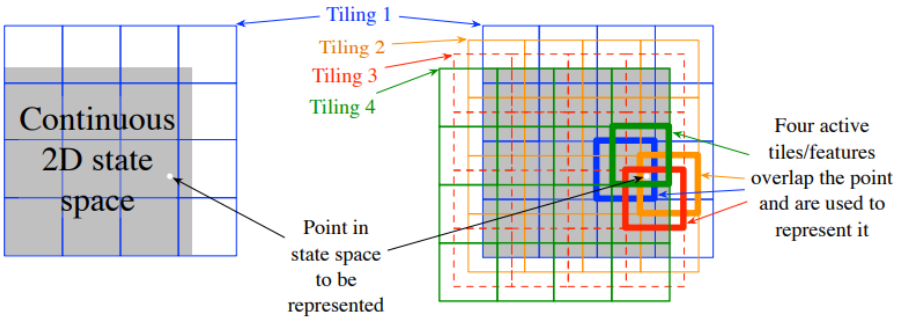



**Figure 9.8** Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

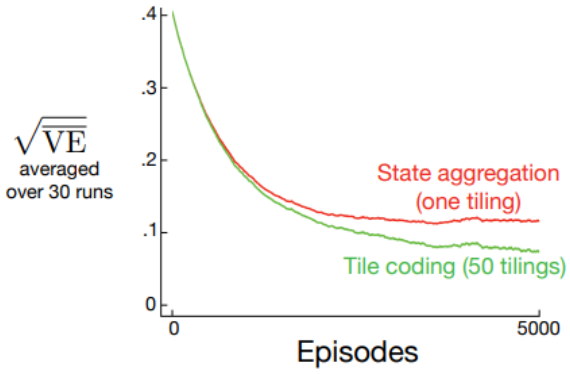



**Figure 9.10**

### Asymmetrical Offsets

Instead of offsetting tilings uniformly, we can offset with different weights. If $w$ denotes the tile width and $n$ the number of tilings, then $\frac{w}{n}$ is a fundamental unit.

Result: **asymmetrical offsets are preferred in tile coding**.

- small black plus → strength of generalization
- uniformly offset → ngs are uniformly offset (above), then there are diagonal artifacts and substantial variations
- asymmetrically offset → more spherical and homogeneous

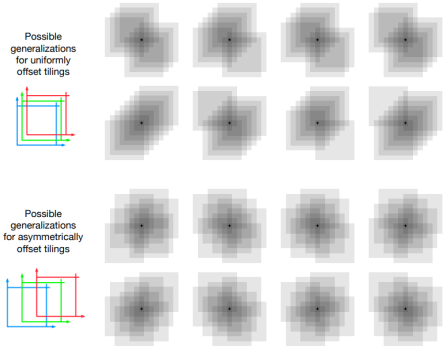



**Figure 9.11**

### Radial Basis Functions

continuous-valued features

- $s \in [0, 1]$
- $c_i$: center, $\sigma_i$: width

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

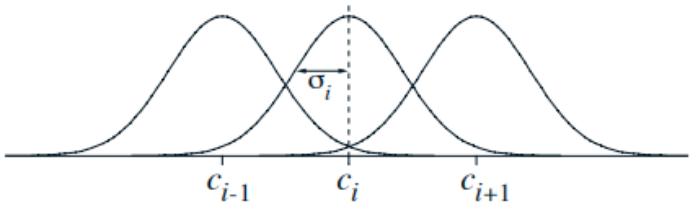



**Figure 9.13:** One-dimensional radial basis functions.

The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. Although this is appealing, in most cases it has no practical significance.

## 8 Selecting Step-Size Parameters Manually

- stochastic approximation conditions (2.7) are sufficient to guarantee convergence

- $\alpha_t = 1/t$ is not appropriate for TD methods, for nonstationary problems (why?)

So how to select step size manually? We could first think how many experiences do we have.

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

If the step size is 1, then the sample error after one target is eliminated. We usually want to learn slower than this.

- In general, $\alpha = \tau$, then we need about $\tau$ experiences.

Suppose we wanted to learn in about experiences with substantially the same feature vector.

$$\alpha \doteq \left(\tau \mathbb{E}[\mathbf{x}^T \mathbf{x}]\right)^{-1} \tag{9.19}$$

# 9 Least-Squares TD

Original update:

$$\mathbf{w_{t+1}} = \mathbf{w}_t + \alpha(R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T \mathbf{w}_t)$$

with time complexity $O(d^3)$.

*Remark.* The time complexity of matrix transposition is $O(d^2)$.

Recall the TD fixed point:

$$\mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b}$$

with

$$\mathbf{A} \doteq \mathbf{E}[\mathbf{x_t}(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T] \text{ and } \mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]$$

**Motivation:** Compute the **estimates** of $\mathbf{A}$ and $\mathbf{b}$, and then directly compute the TD fixed point.

$$\hat{\mathbf{A}}_t \doteq \sum_{k=0}^{t-1} \mathbf{x}_k(\mathbf{x}_k - \gamma \mathbf{x}_{k+1})^T + \epsilon \mathbf{I} \text{ and } \hat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{k+1}\mathbf{x}_k \tag{9.20}$$

where $\mathbf{I}$ is the identity matrix and $\varepsilon \mathbf{I}$ ensures that $\hat{A}_t$ is always invertible (why?). The estimate for TD fixed point is

$$\mathbf{w}_t \doteq \hat{\mathbf{A}}_t^{-1}\hat{\mathbf{b}}_t \tag{9.21}$$

This algorithm is the most data efficient form of linear TD(0), but it is also more expensive computationally.

**Disadvantages**

- semi-gradient TD(0) → memory complexity/time complexity: $O(d)$, $d$=number of components
- LSTD → time complexity: $O(d^2)$ (matrix update), memory complexity: $O(d^2)$
- inverse matrix:

$$\hat{\mathbf{A}}_t^{-1} = \left(\hat{\mathbf{A}}_{t-1} + \mathbf{x}_{t-1}\left(\mathbf{x}_{t-1} - \gamma\mathbf{x}_t\right)^T\right)^{-1} \qquad \text{(from 9.20)}$$

$$= \hat{\mathbf{A}}_{t-1}^{-1} - \frac{\hat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_{t-1}(\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^T\hat{\mathbf{A}}_{t-1}^{-1}}{1 + (\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^T\hat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_{t-1}} \qquad (9.22)$$

  for $t > 0, \hat{\mathbf{A}_0} \doteq \varepsilon\mathbf{I}$. This is also called **Sherman-Morrison formula** with complexity $O(d^2)$.
- $\varepsilon \to$ small: the sequence of inverses can vary wildly
- $\varepsilon \to$ large: learning is slow
- no step-size parameter → problematic if the target policy changes e.g. GPI.

**LSTD for estimating $\hat{v} = \mathbf{w}^\top\mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)**

Input: feature representation $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal) = \mathbf{0}$
Algorithm parameter: small $\varepsilon > 0$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — A $d \times d$ matrix
$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$ — A $d$-dimensional vector
Loop for each episode:
  Initialize $S$; $\mathbf{x} \leftarrow \mathbf{x}(S)$
  Loop for each step of episode:
    Choose and take action $A \sim \pi(\cdot|S)$, observe $R, S'$; $\mathbf{x}' \leftarrow \mathbf{x}(S')$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top(\mathbf{x} - \gamma\mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - (\widehat{\mathbf{A}^{-1}}\mathbf{x})\mathbf{v}^\top/(1 + \mathbf{v}^\top\mathbf{x})$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\mathbf{w} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until $S'$ is terminal

# 10 Memory-based Function Approximation

- save training examples in memory
- **nonparametric** approximation: more training examples, more accurate approximations

### Examples

- **nearest neighbor method**: returns the example whose state is closest to the query state
- **weighted average method**: retrieve a set of nearest neighbor examples and return a weighted average of their target values (decreases with the distance from the query state)
- **Locally weighted regression**: fits a surface to the values of a set of nearest states by means of a parametric approximation method

### Local Approximation

- Memory-based local methods can focus on local neighborhoods of states (or state–action pairs) visited in real or simulated trajectories.
- We don't need global approximation because many areas of the state space will never (or almost never) be reached.

## 11 Kernel-based Function Approximation

- kernel (function) $k : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ is the weight, influence of $s$ to $s'$

### Kernel Regression

- **memory-based method** that computes a kernel weighted average of the targets of **all** examples stored in memory, assigning the result to the query state
- $\mathcal{D}$: set of stored example, $g(s')$ target

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s') g(s') \tag{9.23}$$

- We can choose the kernel to be

$$k(s, s') = \mathbf{x}(s)^T \mathbf{x}(s') \tag{9.24}$$

### Kernel Trick

For many sets of feature vectors, (9.24) has a compact functional form that can be evaluated without any computation taking place in the d-dimensional feature space.