



# Reinforcement Learning

## On-policy Prediction with Approximation

**U Kang**  
**Seoul National University**



# In This Lecture

- On-policy value function approximation
- Gradient based methods
- Linear methods
- Nonlinear methods
- Features for methods



# Overview

- Function approximation in RL for estimating the state-value function from on-policy data
- The approximate value function is represented not as a table but as a parameterized functional form with weight vector  $w \in R^d$
- $\hat{v}(s, w) \approx v_\pi(s)$  denotes the approximate value of state  $s$  given weight vector  $w$
- E.g.,  $\hat{v}$  might be a linear function in features of the state, with  $w$  the vector of feature weights
- E.g.,  $\hat{v}$  might be the function computed by a multi-layer artificial neural network, with  $w$  the vector of connection weights in all the layers; by adjusting the weights, any of a wide range of different functions can be implemented by the network
- E.g.,  $\hat{v}$  might be the function computed by a decision tree, where  $w$  is all the numbers defining the split points and leaf values of the tree



# Overview

- Typically, the number of weights (the dimensionality of  $w$ ) is much less than the number of states ( $d \ll |S|$ ), and changing one weight changes the estimated value of many states
- Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states
- Such generalization makes the learning potentially more powerful but also potentially more difficult to manage and understand



# Outline

- ➔  **Value-function Approximation**
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion



# Value-function Approximation

- All of the prediction methods up to this point have been described as updates to an estimated value function that shifts its value at particular states toward a “backed-up value,” or update target, for that state
- Let us refer to an individual update by the notation  $s \rightarrow u$ , where  $s$  is the state updated and  $u$  is the update target that  $s$ 's estimated value is shifted toward
  - MC:  $S_t \rightarrow G_t$
  - TD(0):  $S_t \rightarrow R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t)$
  - n-step TD:  $S_t \rightarrow G_{t:t+n}$
  - DP:  $s \rightarrow E_{\pi}[R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) | S_t = s]$
  - Note that DP updates all states, while other methods update the state encountered in actual experience for each update



# Value-function Approximation

- Each update is interpreted as specifying an example of the desired input–output behavior of the value function
- In a sense, the update  $s \rightarrow u$  means that the estimated value for state  $s$  should be more like the update target  $u$
- Tabular method: the table entry for  $s$ 's estimated value is shifted a fraction of the way toward  $u$ , and the estimated values of all other states were left unchanged
- Function approximation: arbitrarily complex and sophisticated functions are used to implement the update, and updating at  $s$  generalizes so that the estimated values of many other states are changed as well
- Function approximation methods receive examples (training data) of the desired input–output behavior of the function they are trying to approximate



# Value-function Approximation

- Viewing each update as a conventional training example in this way enables us to use any of a wide range of existing function approximation methods for value prediction
- 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 RL
- The most sophisticated artificial neural network and statistical methods all assume a static training set over which multiple passes are made



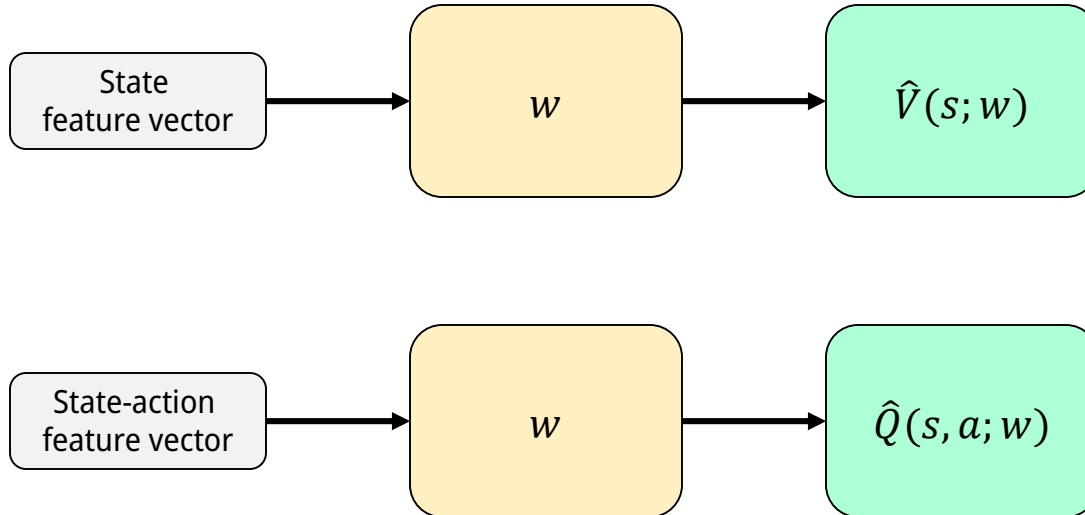


# Value-function Approximation

- In RL, however, it is important that learning be able to occur online, while the agent interacts with its environment or with a model of its environment
- To do this requires methods that are able to learn efficiently from incrementally acquired data
- In addition, RL generally requires function approximation methods able to handle nonstationary target functions (target functions that change over time)
- E.g., in control methods based on GPI (generalized policy iteration) we often seek to learn  $q_\pi$  while  $\pi$  changes
- Even if the policy remains the same, the target values of training examples are nonstationary if they are generated by bootstrapping methods (DP and TD learning)
- Methods that cannot easily handle such nonstationarity are less suitable for RL




# Value-function Approximation





# Outline

- Value-function Approximation
-   **The Prediction Objective ( $\overline{VE}$ )**
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion



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

- Up to now we have not specified an explicit objective for prediction
- In the tabular case a continuous measure of prediction quality was not necessary because the learned value function could come to equal the true value function exactly; moreover, the learned values at each state were decoupled—an update at one state affected no other
- With genuine approximation, an update at one state affects many others, and it is not possible to get the values of all states exactly correct
- By assumption we have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate
- We are obligated then to say which states we care most about
- We must specify a state distribution  $\mu(s) \geq 0, \sum_s \mu(s) = 1$ , representing how much we care about the error in each state  $s$



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

- The error in a state  $s$ : the square of the difference between the approximate value  $\hat{v}(s, w)$  and the true value  $v_\pi(s)$
- Weighting this over the state space by  $\mu$ , we obtain a natural objective function  $\overline{VE}$ , the Mean Squared Value Error

$$\overline{VE}(w) = \sum_{s \in \mathcal{S}} \mu(s) [v_\pi(s) - \hat{v}(s, w)]^2$$

- The square root of this measure, the root  $VE$ , gives a rough measure of how much the approximate values differ from the true values
- 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
- In continuing tasks, the on-policy distribution is the stationary distribution under  $\pi$



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

- The on-policy distribution in an episodic task
  - Depends on how the initial states of episodes are chosen
  - Let  $h(s)$  denote the probability that an episode begins in each state  $s$ , and let  $\eta(s)$  denote the number of time steps spent, on average, in state  $s$  in a single episode
  - Time is spent in a state  $s$  if episodes start in  $s$ , or if transitions are made into  $s$  from a preceding state  $\bar{s}$  in which time is spent

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

- This system of equations can be solved for the expected number of visits  $\eta(s)$ . The on-policy distribution is then the fraction of time spent in each state normalized to sum to one

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \text{for all } s \in \mathcal{S}$$

- If there is discounting ( $\gamma < 1$ ), we include a factor of  $\gamma$  at



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

- It is not completely clear that the VE is the right performance objective for RL
- Remember that our ultimate purpose—the reason we are learning a value function—is to find a better policy
- The best value function for this purpose is not necessarily the best for minimizing  $\overline{VE}$
- Nevertheless, it is not yet clear what a more useful alternative goal for value prediction might be




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

- An ideal goal in terms of  $\overline{VE}$  would be to find a *global* optimum, a weight vector  $w^*$  for which  $\overline{VE}(w^*) \leq \overline{VE}(w)$  for all possible  $w$
- Reaching this goal is sometimes possible for simple function approximators (FAs) such as linear ones, but is rarely possible for complex FAs such as artificial neural networks and decision trees
- Instead, complex FAs may seek to converge to a *local* optimum, a weight vector  $w^*$  which  $\overline{VE}(w^*) \leq \overline{VE}(w)$  for all  $w$  in some neighborhood of  $w^*$
- Although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear FAs, and often it is enough





# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
-   **Stochastic-gradient and Semi-gradient Methods**
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion



# Stochastic-gradient and Semi-gradient Methods

- We discuss learning methods for function approximation (FA) in value prediction based on stochastic gradient descent (SGD)
- SGD methods are among the most widely used for all FA methods and are particularly well suited to online RL



# Stochastic-gradient and Semi-gradient Methods

- In SGD, the weight vector  $\mathbf{w} = (w_1, w_2, \dots, w_d)^T$  is a column vector with a fixed number of real valued components, and the approximate value function  $\hat{v}(s, \mathbf{w})$  is a differentiable function of  $\mathbf{w}$  for all  $s$
- $\mathbf{w}_t$  means the weight vector updated at each step  $t=0, 1, 2, \dots$
- Assume that, on each step, we observe a new example  $S_t \rightarrow v_\pi(S_t)$  consisting of a state  $S_t$  and its true value under the policy
- 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 FA has limited resources and thus limited resolution
- There is generally no  $\mathbf{w}$  that gets all the states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in examples



# Stochastic-gradient and Semi-gradient Methods

- We assume that states appear in examples with the same distribution,  $\mu$ , over which we are trying to minimize the  $\overline{VE}$
- A good strategy is to minimize error on the observed examples
- SGD methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2} \alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w}_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t) \end{aligned}$$

- where  $\alpha > 0$  is a step size
- $\nabla f(\mathbf{w})$  means the gradient of  $f$  wrt  $\mathbf{w}$

$$\nabla f(\mathbf{w}) = \left( \frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \dots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^T$$



# Aside: Gradient

- Functions with multiple inputs:  $f : R^n \rightarrow R$
- Partial derivative  $\frac{\partial}{\partial x_i} f(\mathbf{x})$  measures how  $f$  changes as only the variable  $x_i$  increases at point  $\mathbf{x}$ .
- Gradient  $\nabla_{\mathbf{x}} f(\mathbf{x})$  of  $f$  is the vector containing all the partial derivatives
- Critical points: every element of the gradient is equal to 0



# Aside: Gradient

- $f(\mathbf{x}) = \mathbf{c}^T \mathbf{x}$ 
  - $\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{c}$
  
- $f(\mathbf{x}) = \mathbf{x}^T \mathbf{x}$ 
  - $\nabla_{\mathbf{x}} f(\mathbf{x}) = 2\mathbf{x}$
  
- $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$ , for symmetric  $\mathbf{A}$ 
  - $\nabla_{\mathbf{x}} f(\mathbf{x}) = 2\mathbf{A} \mathbf{x}$



# Aside: Chain Rule of Calculus

- Let  $x$  be a real number, and  $f$  and  $g$  be functions from  $\mathbb{R}$  to  $\mathbb{R}$ . Suppose  $y = g(x)$ , and  $z = f(g(x)) = f(y)$ . Then the chain rule states that 
$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$
- Suppose  $x \in \mathbb{R}^m$ ,  $y \in \mathbb{R}^n$ ,  $g: \mathbb{R}^m \rightarrow \mathbb{R}^n$ ,  $f: \mathbb{R}^n \rightarrow \mathbb{R}$ . If  $\mathbf{y} = g(\mathbf{x})$  and  $z = f(\mathbf{y})$ , then 
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$
- In vector notation:  $\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$ , where  $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$  is the  $n \times m$  Jacobian matrix of  $g$ 
  - E.g., suppose  $z = (y - c)^2$ , and  $y = g(\mathbf{x})$ . Then  $\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z = 2(y - c) \nabla_{\mathbf{x}} g(\mathbf{x})$



# Stochastic-gradient and Semi-gradient Methods

- We turn now to the case in which the target output  $U_t \in R$  of the  $t$  th training example,  $S_t \rightarrow U_t$  is not the true value  $v_\pi(S_t)$ , but some, possibly random, approximation to it
- E.g.,  $U_t$  might be a noise-corrupted version of  $v_\pi(S_t)$ , or it might be one of the bootstrapping targets
- In these cases we cannot perform the exact update because  $v_\pi(S_t)$  is unknown, but we can approximate it by substituting  $U_t$  for  $v_\pi(S_t)$
- This leads to the general SGD method for state-value prediction:
$$w_{t+1} = w_t + \alpha[U_t - \hat{v}(S_t, w_t)]\nabla\hat{v}(S_t, w_t)$$
- If  $U_t$  is an unbiased estimate, that is, if  $E[U_t|S_t = s] = v_\pi(S_t)$ , for each  $t$ , then  $w_t$  is guaranteed to converge to a local optimum under the usual stochastic approximation conditions for decreasing  $\alpha$





# Stochastic-gradient and Semi-gradient Methods

- For example, suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy  $\pi$
- Because the true value of a state is the expected value of the return following it, the MC target  $U_t = G_t$  is by definition an unbiased estimate of  $v_\pi(S_t)$
- With this choice, the general SGD method converges to a locally optimal approximation to  $v_\pi(S_t)$
- Thus, the gradient-descent version of MC state-value prediction is guaranteed to find a locally optimal solution



# Stochastic-gradient and Semi-gradient Methods

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

Input: the policy  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$

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  $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$  using  $\pi$

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

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$



# Stochastic-gradient and Semi-gradient Methods

- One does not obtain the same guarantees if a bootstrapping estimate of  $v_\pi(S_t)$  is used as the target  $U_t$
- Bootstrapping targets such as n-step returns  $G_{t:t+n}$  or the DP target  $\sum_{a,s',r} \pi(a|S_t)p(s',r|S_t,a)[r + \gamma\hat{v}(s',w_t)]$  all depend on the current value of the weight vector  $w_t$ , which implies that they will be biased and that they will not produce a true gradient-descent method

$$\begin{aligned}w_{t+1} &= w_t - \frac{1}{2} \alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, w_t)]^2 \\ &= w_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t)\end{aligned}$$

- Note that the key gradient step relies on the target being independent of  $w_t$ . This step would not be valid if a bootstrapping estimate were used in place of  $v_\pi(S_t)$



# Stochastic-gradient and Semi-gradient Methods

$$\begin{aligned}w_{t+1} &= w_t - \frac{1}{2} \alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, w_t)]^2 \\ &= w_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t)\end{aligned}$$

- Bootstrapping methods are *not* in fact instances of true gradient descent
- They take into account the effect of changing the weight vector  $w_t$  on the estimate, but ignore its effect on the target
- They include only a part of the gradient and, accordingly, we call them semi-gradient methods



# Stochastic-gradient and 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
- Advantages of semi-gradient methods
  - They typically enable significantly faster learning (e.g., TD)
  - 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
  - E.g., semi-gradient TD(0) uses  $U_t = R_{t+1} + \hat{v}(S_{t+1}, w)$



# Stochastic-gradient and Semi-gradient Methods

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

Input: the policy  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  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  $R, S'$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$

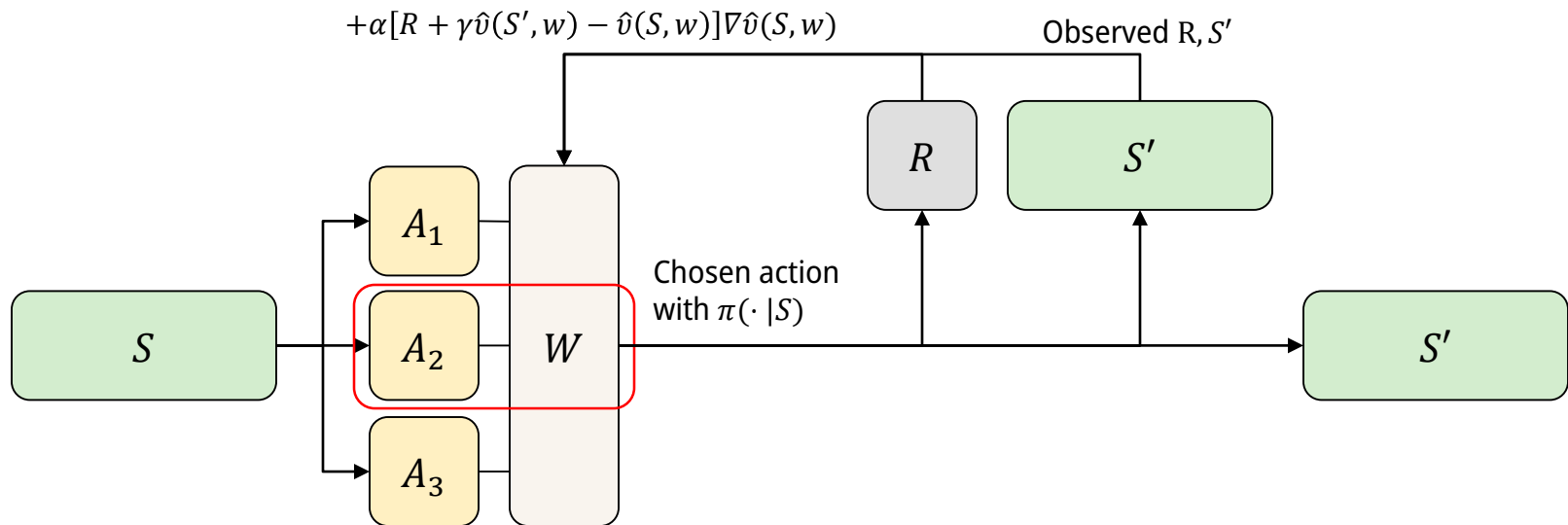
$S \leftarrow S'$

  until  $S$  is terminal

Sutton and Barto,  
Reinforcement  
Learning, 2018



# Stochastic-gradient and Semi-gradient Methods





# Stochastic-gradient and Semi-gradient Methods

- 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  $w$ ) for each group
- The value of a state is estimated as its group's component, and when the state is updated, that component alone is updated
- State aggregation is a special case of SGD where the gradient,  $\nabla \hat{v}(S_{t+1}, w_t)$  is 1 for  $S_t$ 's group's component and 0 for the other components

State	Value
$s_1$	$w_1$
$s_2$	$w_1$
$s_3$	$w_1$
$s_4$	$w_2$
$s_5$	$w_2$
$s_6$	$w_2$

$$\left. \begin{array}{l} s_1 \\ s_2 \\ s_3 \end{array} \right\} x(s) = [1 \ 0]^T$$
$$\left. \begin{array}{l} s_4 \\ s_5 \\ s_6 \end{array} \right\} x(s) = [0 \ 1]^T$$

$$\hat{v}(s, w) = w^T x(s)$$



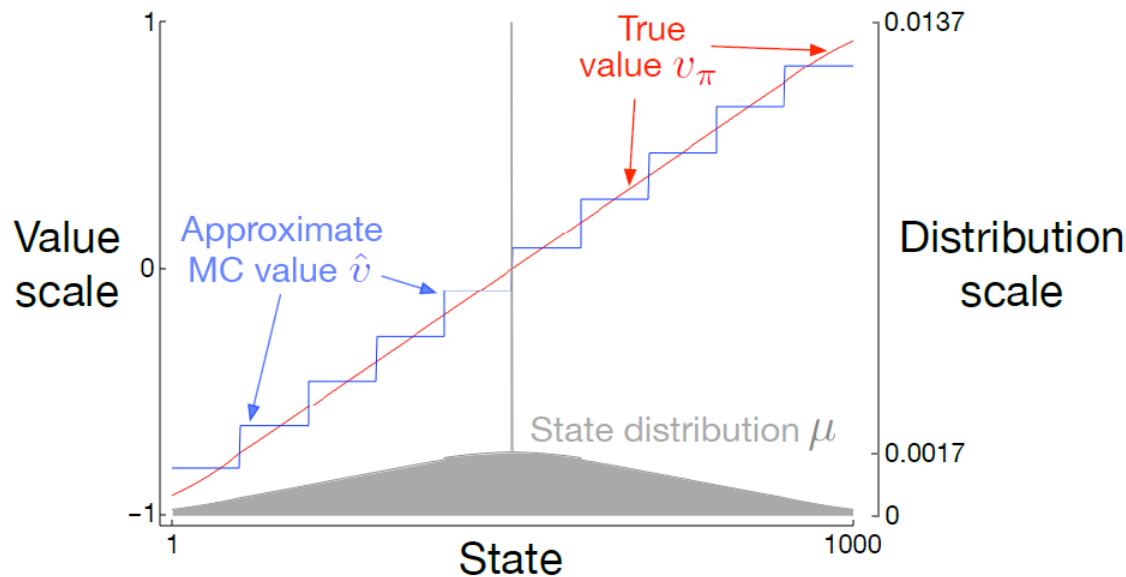


# Example: State Aggregation

- Consider a 1000-state version of the random walk task
- 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
- 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)
- 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



# Example: State Aggregation

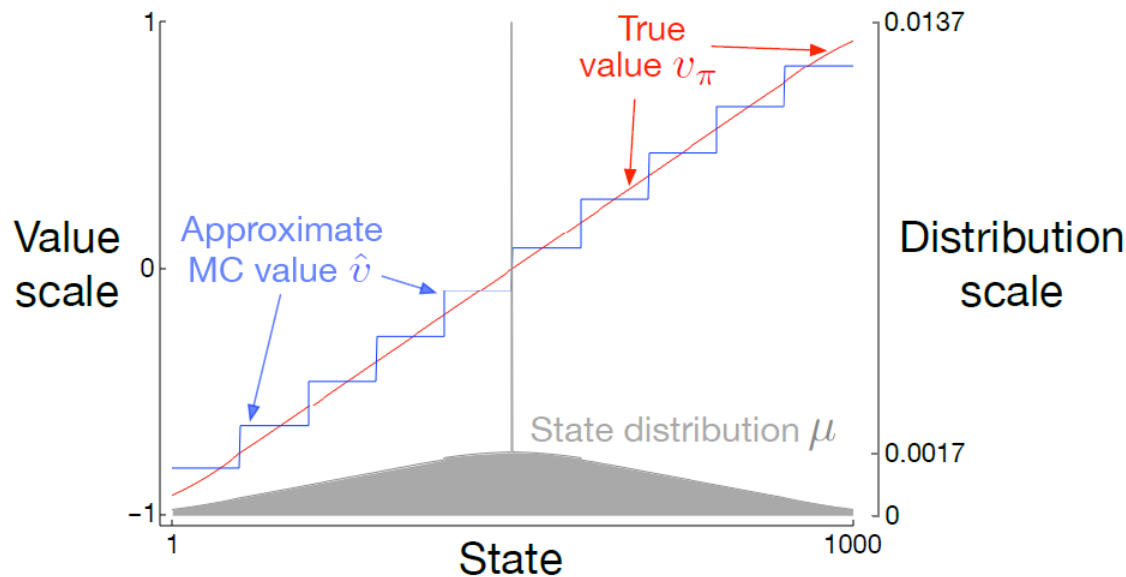


Sutton and Barto,  
Reinforcement  
Learning, 2018

- The true value function  $v_\pi$  is nearly a straight line
- The final approximate value function is shown as well; it is learned by the gradient MC algorithm with state aggregation after 100,000 episodes with a step size of  $\alpha = 2 * 10^{-5}$
- 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)



# Example: State Aggregation



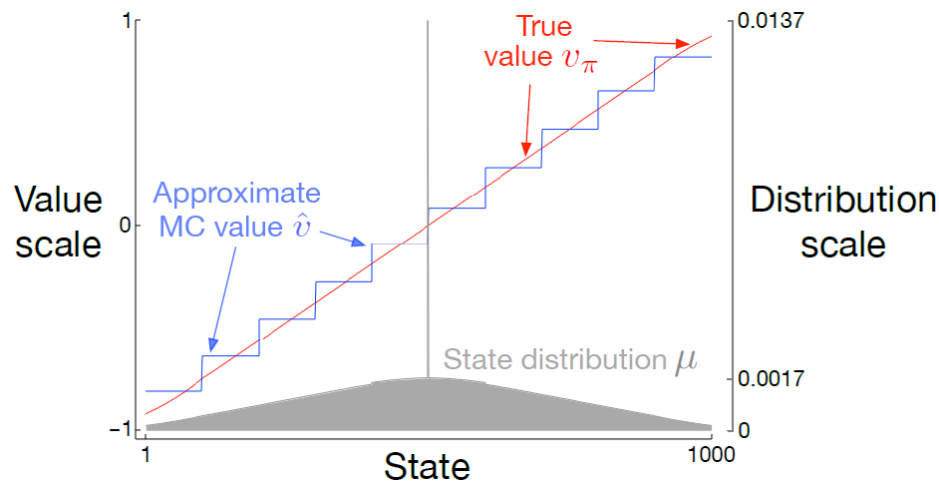
Sutton and Barto,  
Reinforcement  
Learning, 2018

- The staircase effect is typical of state aggregation; within each group, the approximate value is constant, and it changes abruptly from one group to the next
- These approximate values are close to the global minimum of the  $\overline{VE}$




# Example: State Aggregation

- State 500, in the center, is the first state of every episode, but is rarely visited again. On average, about 1.37% of the time steps are spent in the start state
- The most visible effect of the distribution is on the leftmost groups, whose values are clearly shifted higher than the unweighted average of the true values of states within the group, and on the rightmost groups, whose values are clearly shifted lower
- This is due to the states in these areas having the greatest asymmetry in their weightings by  $\mu$ ; e.g., in the leftmost group, state 100 is weighted more than 3 times than state 1. Thus the estimate for the group is biased toward the true value of state 100, which is higher than the true value of state 1





# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
-   **Linear Methods**
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion



# Linear Methods

- One of the most important special cases of function approximation is that in which the approximate function,  $\hat{v}(\cdot, w)$ , is a linear function of the weight vector,  $w$
- Corresponding to every state  $s$ , there is a real-valued vector  $x(s) = (x_1(s), x_2(s), \dots, x_d(s))^T$  with the same number of components as  $w$
- Linear methods approximate state-value function by the inner product between  $w$  and  $x(s)$ :

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

- In this case the approximate value function is said to be linear in the weights, or simply linear
- The vector  $x(s)$  is called a feature vector representing state  $s$ . Each component  $x_i(s)$  of  $x(s)$  is the value of a function  $x_i: S \rightarrow R$



# Linear Methods

- It is natural to use SGD updates with linear function approximation.
- The gradient of the approximate value function with respect to  $w$  in this case is:  $\nabla \hat{v}(s, w) = x(s)$
- In the linear case the general SGD update reduces to a simple form
$$w_{t+1} = w_t + \alpha [U_t - \hat{v}(S_t, w_t)] x(S_t)$$
- Linear SGD is one of the most favorable for mathematical analysis due to its simplicity
- In the linear case there is only one optimum, and thus any method that is guaranteed to converge to or near a local optimum is automatically guaranteed to converge to or near the global optimum
- E.g., the gradient MC converges to the global optimum of the  $\overline{VE}$  under linear function approximation if  $\alpha$  is reduced over time according to the usual conditions



# Linear Methods

- The semi-gradient TD(0) algorithm also converges under linear FA
- The weight vector converged to is not the global optimum, but rather a point near the local optimum
- The update of the weight vector at each time  $t$  is

$$\begin{aligned}w_{t+1} &= w_t + \alpha(R_{t+1} + \gamma w_t^T x_{t+1} - w_t^T x_t)x_t \\ &= w_t + \alpha(R_{t+1}x_t - x_t(x_t - \gamma x_{t+1})^T w_t)\end{aligned}$$

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

$$\mathbb{E}[w_{t+1}|w_t] = w_t + \alpha(b - Aw_t)$$

- where

$$b = \mathbb{E}[R_{t+1}x_t] \in \mathbb{R}^d \text{ and } A = \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T] \in \mathbb{R}^d \times \mathbb{R}^d$$





# Linear Methods

- It is clear that, if the system converges, it must converge to the weight vector  $w_{TD}$  at which

$$\begin{aligned}b - Aw_{TD} &= 0 \\ \rightarrow b &= Aw_{TD} \\ \rightarrow w_{TD} &= A^{-1}b\end{aligned}$$

- This quantity is called the TD fixed point. Linear semi-gradient TD(0) converges to this point



# Linear Methods

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

$$\overline{VE}(w_{TD}) \leq \frac{1}{1-\gamma} \min_w \overline{VE}(w)$$

- That is, the asymptotic error of the TD method is no more than  $\frac{1}{1-\gamma}$  times the smallest possible error, that attained in the limit by the MC method
- Because  $\gamma$  is often near one, this expansion factor can be quite large, so there is substantial potential loss in asymptotic performance with the TD method
- On the other hand, recall that the TD methods are often of vastly reduced variance compared to MC methods, and thus faster
- Which method is better depends on the nature of the approximation and problem, and on how long learning continues



# Linear Methods

- A similar bound applies to other on-policy bootstrapping methods
- E.g., linear semi-gradient DP

$$w_{t+1} = w_t + \alpha[U_t - \hat{v}(S_t, w_t)]\nabla\hat{v}(S_t, w_t)$$

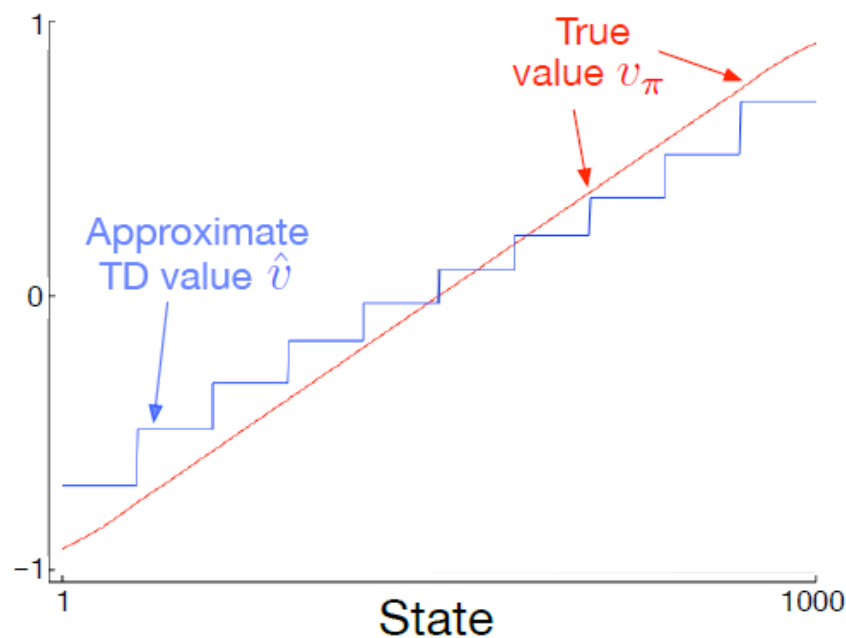
$$U_t = \sum_a \pi(a|S_t)p(s', r|S_t, a)[r + \gamma\hat{v}(s', w_t)]$$

- with updates according to the on-policy distribution will also converge to the TD fixed point
- One-step semi-gradient action-value methods, such as semi-gradient Sarsa(0) converge to an analogous fixed point and an analogous bound
- For episodic tasks, there is a slightly different but related bound
- Critical to these convergence results is that states are updated according to the on-policy distribution. For other update distributions, bootstrapping methods using function approximation may actually diverge to infinity



# Example: Bootstrapping on 1000-state Random Walk

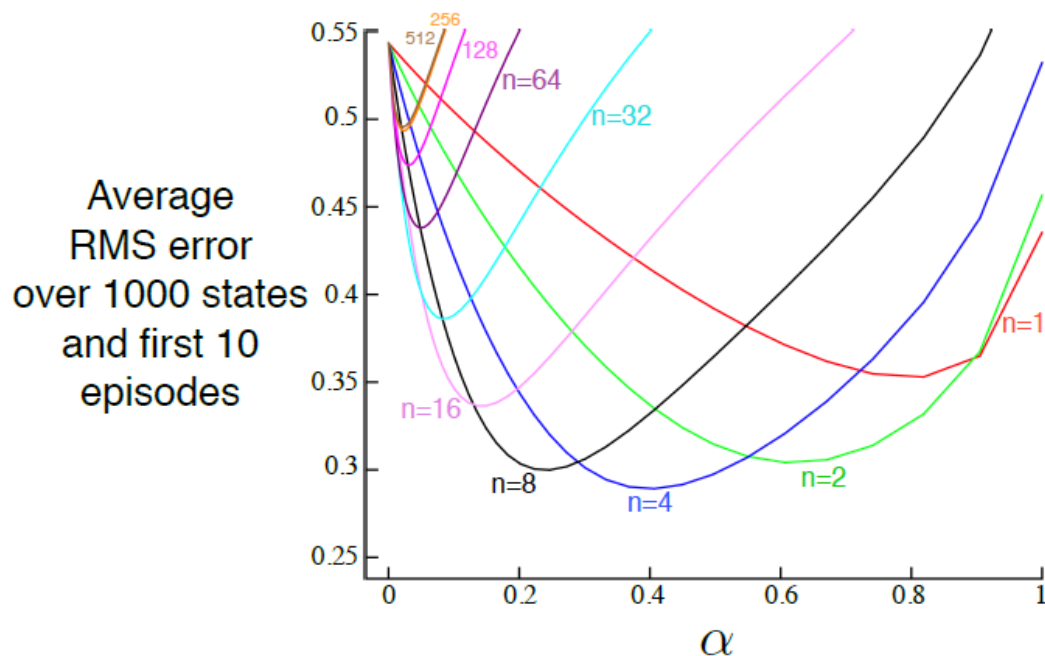
- State aggregation is a special case of linear function approximation
- The figure shows the final value function learned by the semi-gradient TD(0) algorithm using the state aggregation, in 1000-state random walk example
- Note that the near-asymptotic TD approximation is indeed farther from the true values than the Monte Carlo approximation





# Example: Bootstrapping on 1000-state Random Walk

- Nevertheless, TD methods retain large potential advantages in learning rate, and generalize MC methods
- The figure shows results with an n-step semi-gradient TD method using state aggregation on the 1000-state random walk that are strikingly similar to those we obtained earlier with tabular methods and the 19-state random walk





# Linear Methods

- The semi-gradient n-step TD algorithm is the natural extension of the tabular n-step TD algorithm to semi-gradient FA

- Key equation:

$$w_{t+n} = w_{t+n-1} + \alpha [G_{t:t+n} - \hat{v}(S_t, w_{t+n-1})] \nabla \hat{v}(S_t, w_{t+n-1}), 0 \leq t < T$$

- where the n-step return is generalized to

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, w_{t+n-1}), 0 \leq t \leq T - n$$



# Linear Methods

## $n$ -step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  such that  $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameters: step size  $\alpha > 0$ , a positive integer  $n$

Initialize value-function weights  $\mathbf{w}$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

All store and access operations ( $S_t$  and  $R_t$ ) can take their index mod  $n + 1$

Loop for each episode:

    Initialize and store  $S_0 \neq \text{terminal}$

$T \leftarrow \infty$

    Loop for  $t = 0, 1, 2, \dots$  :

        If  $t < T$ , then:

            Take an action according to  $\pi(\cdot | S_t)$

            Observe and store the next reward as  $R_{t+1}$  and the next state as  $S_{t+1}$

            If  $S_{t+1}$  is terminal, then  $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$  ( $\tau$  is the time whose state's estimate is being updated)

        If  $\tau \geq 0$ :

$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$


            If  $\tau + n < T$ , then:  $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$  ( $G_{\tau:\tau+n}$ )

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_\tau, \mathbf{w})] \nabla \hat{v}(S_\tau, \mathbf{w})$

    Until  $\tau = T - 1$



# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
-   **Feature Construction for Linear Methods**
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion





# Feature Construction for Linear Methods

- Linear methods are interesting because of their convergence guarantees, but also because in practice they can be very efficient in terms of both data and computation
- Choosing features appropriate to the task is an important way of adding prior domain knowledge to RL systems
- Intuitively, the features should correspond to the aspects of the state space along which generalization may be appropriate
- If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function
- If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, and so on



# Feature Construction for Linear Methods

- A limitation of the linear form is that it cannot take into account any interactions between features, such as the presence of feature  $i$  being good only in the absence of feature  $j$
- E.g., in the pole-balancing task, high angular velocity can be either good or bad depending on the angle. If the angle is high, then high angular velocity means an imminent danger of falling—a bad state—whereas if the angle is low, then high angular velocity means the pole is righting itself—a good state
- A linear value function could not represent this if its features are coded separately for the angle and the angular velocity
- It needs instead, or in addition, features for combinations of these two underlying state dimensions



# Polynomials

- Polynomials make up one of the simplest families of features used for interpolation and regression
- Suppose a RL problem has states  $\in R^2$ ; i.e., a state  $s$  is represented by two scalars  $s_1$  and  $s_2$
- If we set  $x(s) = (s_1, s_2)^T$ , we cannot take into account any interactions between these dimensions; also, if both  $s_1$  and  $s_2$  are 0, then the value function would be 0
- Setting  $x(s) = (1, s_1, s_2, s_1s_2)^T$  addresses the problem
- We can even set  $x(s) = (1, s_1, s_2, s_1s_2, s_1^2, s_2^2, s_1s_2^2, s_1^2s_2, s_1^2s_2^2)^T$  to take more complex interaction into account



# Polynomials

- In general, let each state  $s$  corresponds to  $k$  numbers  $s_1, s_2, \dots, s_k$
- Each order- $n$  polynomial-basis feature  $x_i$  is given by  $x_i = \prod_{j=1}^k s_j^{c_j}$  where each  $c_j$  is an integer in the set  $\{0, 1, \dots, n\}$  for an integer  $n \geq 0$
- These features make up the order- $n$  polynomial basis for dimension  $k$ , which contains  $(n + 1)^k$  different features
  
- Higher-order polynomial bases allow for more accurate approximations of more complicated functions
- But because the number of features in an order- $n$  polynomial basis grows exponentially with the dimension  $k$  of the natural state space, it is generally necessary to select a subset of them
- This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods developed for polynomial regression can be adapted to RL



# Fourier Basis

- Fourier series expresses periodic functions as weighted sums of sine and cosine basis functions (features) of different frequencies
- A function  $f$  is periodic if  $f(x) = f(x + \tau)$  for all  $x$  and some period  $\tau$
- Fourier series and the more general Fourier transform are widely used in applied sciences in part because if a function to be approximated is known, then the basis function weights are given by simple formulae and, further, with enough basis functions essentially any function can be approximated as accurately as desired
- In RL, where the functions to be approximated are unknown, Fourier basis functions are of interest because they are easy to use and can perform well in a range of RL problems

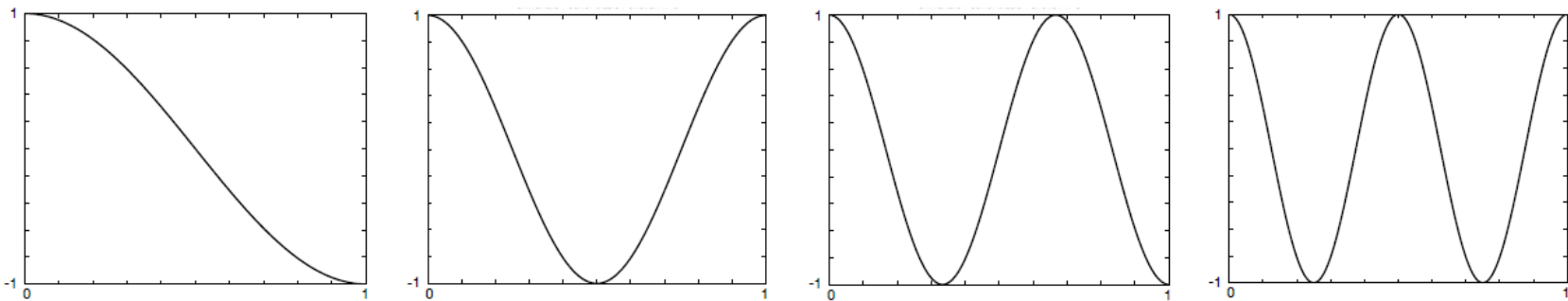


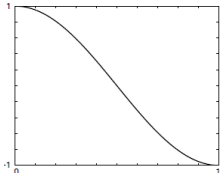
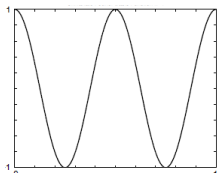
# Fourier Basis

- Assume the features are defined over the interval  $[0, 1]$ . The one-dimensional order- $n$  Fourier cosine basis consists of the  $n + 1$  features

$$x_i(s) = \cos(i\pi s), \quad s \in [0, 1]$$

- for  $i=0, \dots, n$
- 1-D Fourier cosine features  $x_i, i=1, 2, 3, 4$ , for approximating functions over the interval  $[0, 1]$



- Intuition:  $f(s) = w_1$    $+ w_2$    $\dots$

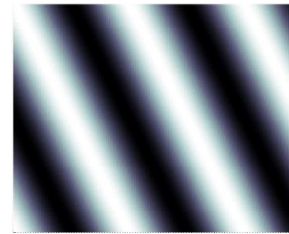


# Fourier Basis

- Fourier cosine series approximation in k-D case
- Suppose each state  $s$  corresponds to a vector of  $k$  numbers,  $s = (s_1, s_2, \dots, s_k)^T$ , with each  $s_i \in [0, 1]$
- The  $i$ th feature in the order- $n$  Fourier cosine basis can be written  $x_i(s) = \cos(\pi s^T c^i)$  where  $c^i = (c_1^i, \dots, c_k^i)^T$ , with  $c_j^i \in \{0, \dots, n\}$  for  $j = 1, \dots, k$  and  $i = 0, \dots, (n + 1)^k$
- This defines a feature for each of the  $(n + 1)^k$  possible integer vectors  $c^i$
- The inner product  $s^T c^i$  has the effect of assigning an integer in  $\{0, \dots, n\}$  to each dimension of  $s$
- Intuition:  $f(s) = w_1$



+  $w_2$

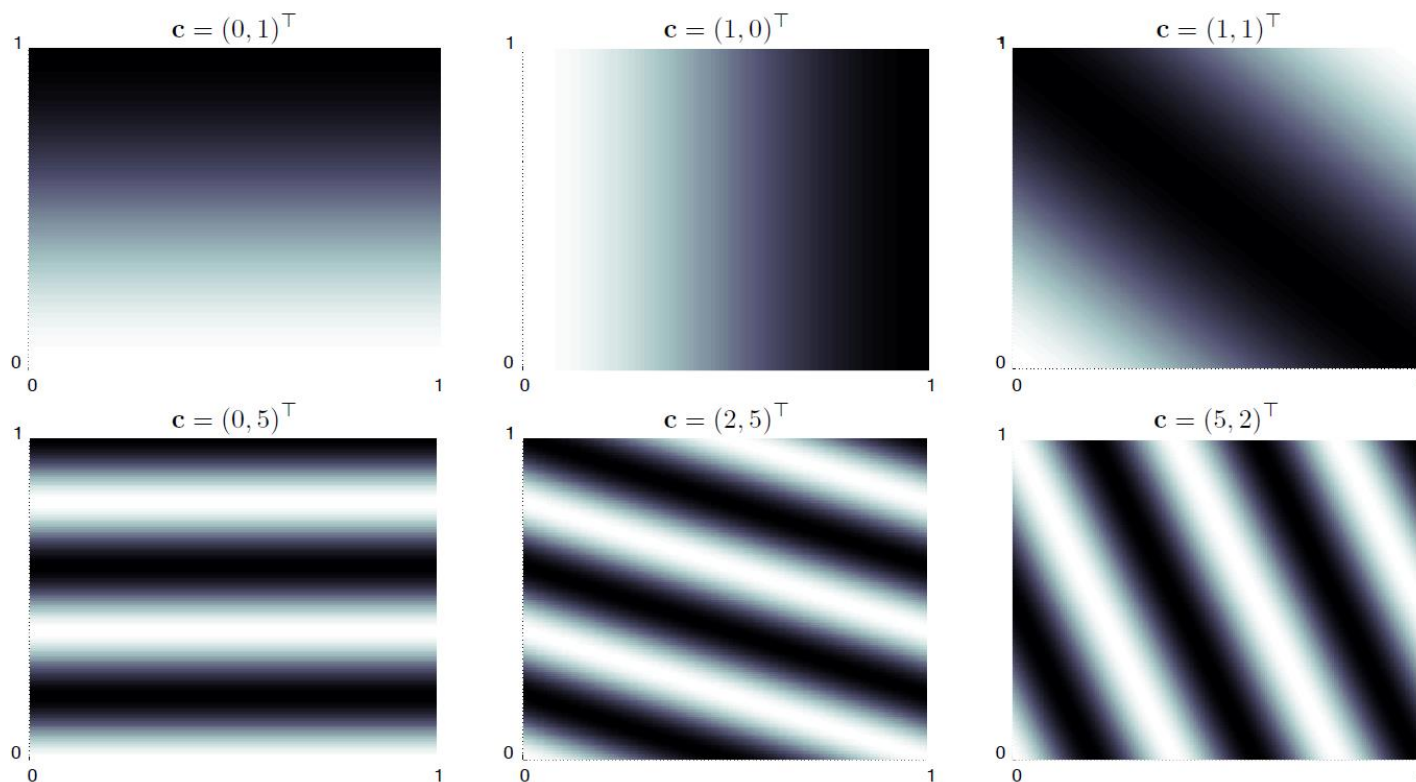


...



# Fourier Basis

- The  $i$ th feature in the order- $n$  Fourier cosine basis can be written  $x_i(s) = \cos(\pi s^T c^i)$  where  $c^i = (c_1^i, \dots, c_k^i)^T$ , with  $c_j^i \in \{0, \dots, n\}$  for  $j = 1, \dots, k$  and  $i = 0, \dots, (n + 1)^k$
- E.g., 2-D Fourier cosine features







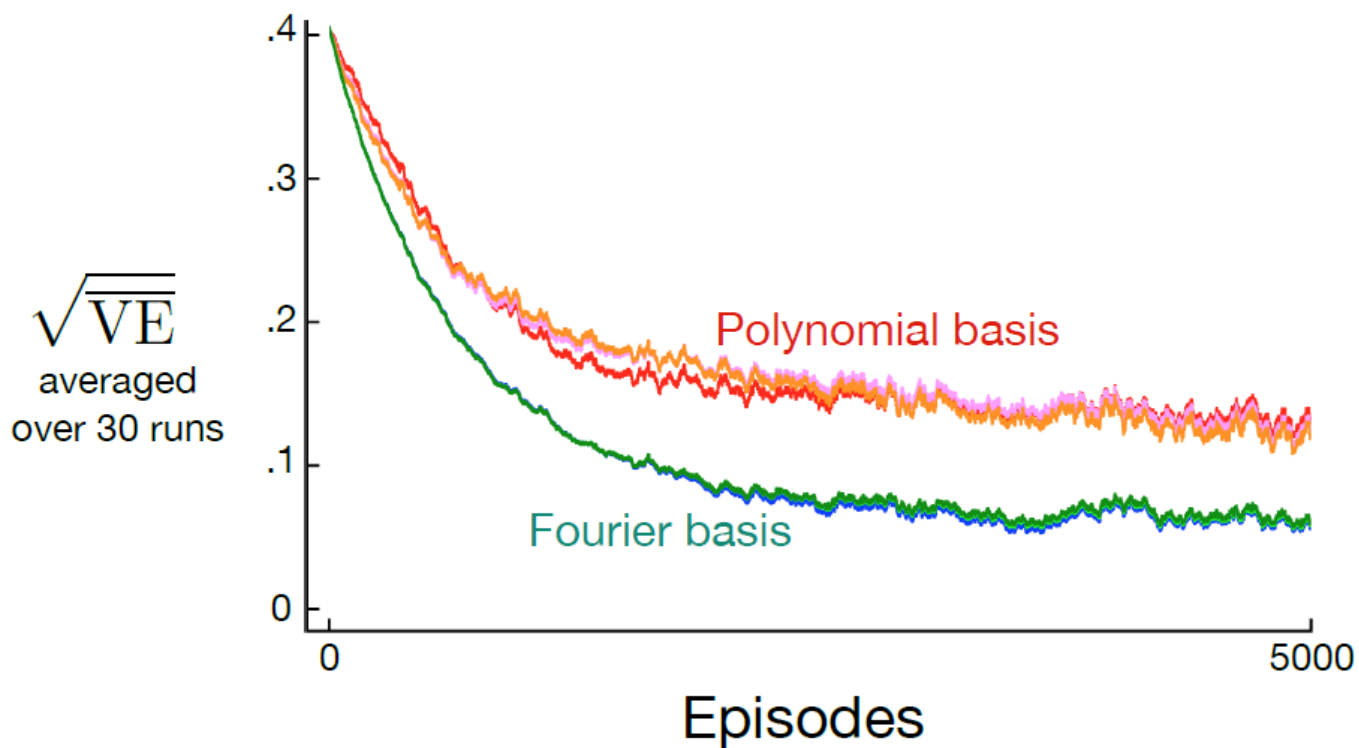
# Fourier Basis

- Fourier cosine features with Sarsa can produce good performance compared to several other collections of basis functions, including polynomial and radial basis functions
- The number of features in the order- $n$  Fourier basis grows exponentially with the dimension of the state space, but if that dimension is small enough (e.g.,  $k \leq 5$ ), then one can select  $n$  so that all of the order- $n$  Fourier features can be used
- This makes the selection of features more-or-less automatic
- For high dimension state spaces, however, it is necessary to select a subset of these features. This can be done using prior beliefs, and some automated selection methods



# Fourier Basis

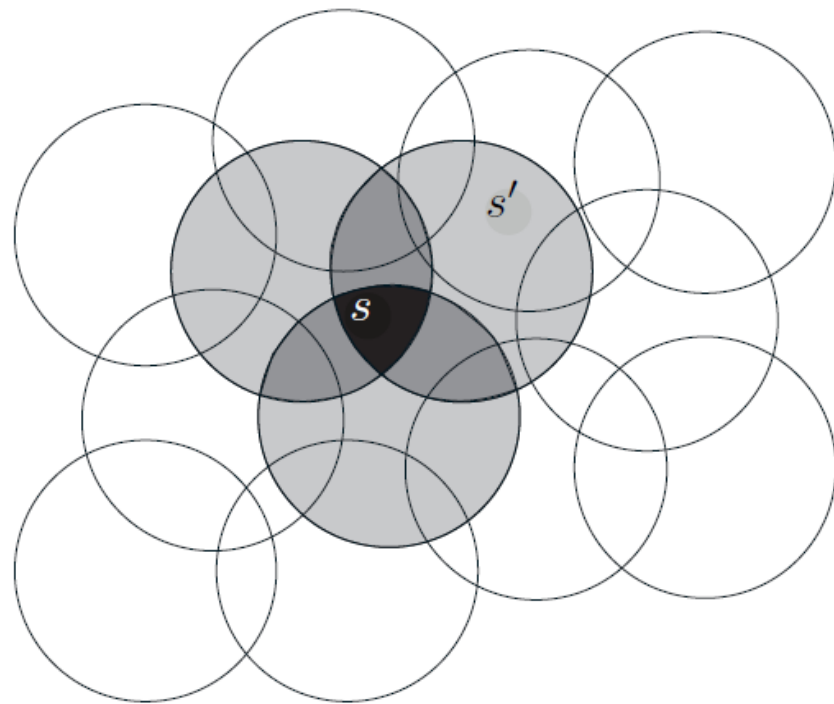
- Fourier basis vs polynomials on the 1000-state random walk





# Coarse Coding

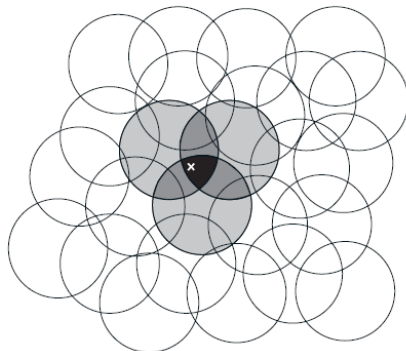
- Coarse coding
  - Consider a task in which the natural representation of the state set is a continuous 2-D space
  - One kind of representation for this case is made up of features corresponding to circles in state space
  - If the state is inside a circle, then the corresponding feature has the value 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



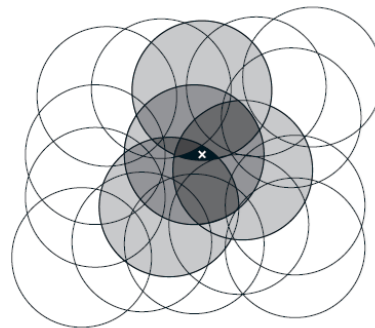


# Coarse Coding

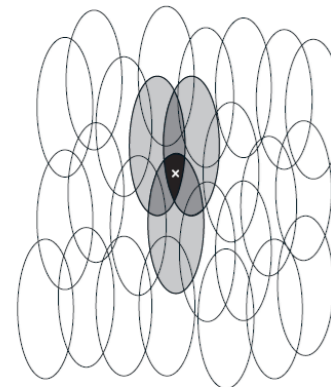
- Assuming linear gradient-descent FA, consider the effect of the size and density of the circles
- Corresponding to each circle is a single weight (a component of  $w$ ) that is affected by learning
- If we train at one state, a point in the space, then the weights of all circles intersecting that state will be affected
- If the circles are small, then the generalization will be over a short distance, whereas if they are large, it will be over a large distance



Narrow generalization



Broad generalization

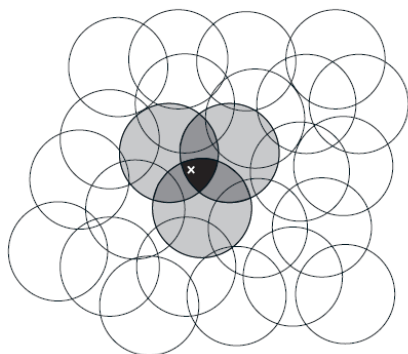


Asymmetric generalization

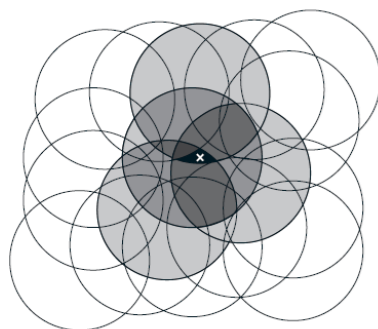


# Coarse Coding

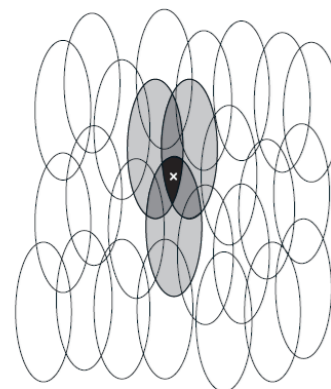
- The shape of the features will determine the nature of the generalization. E.g., if they are not strictly circular, but are elongated in one direction, then generalization will be similarly affected
- Features with large receptive fields give broad generalization, but might also seem to limit the learned 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



Narrow generalization



Broad generalization

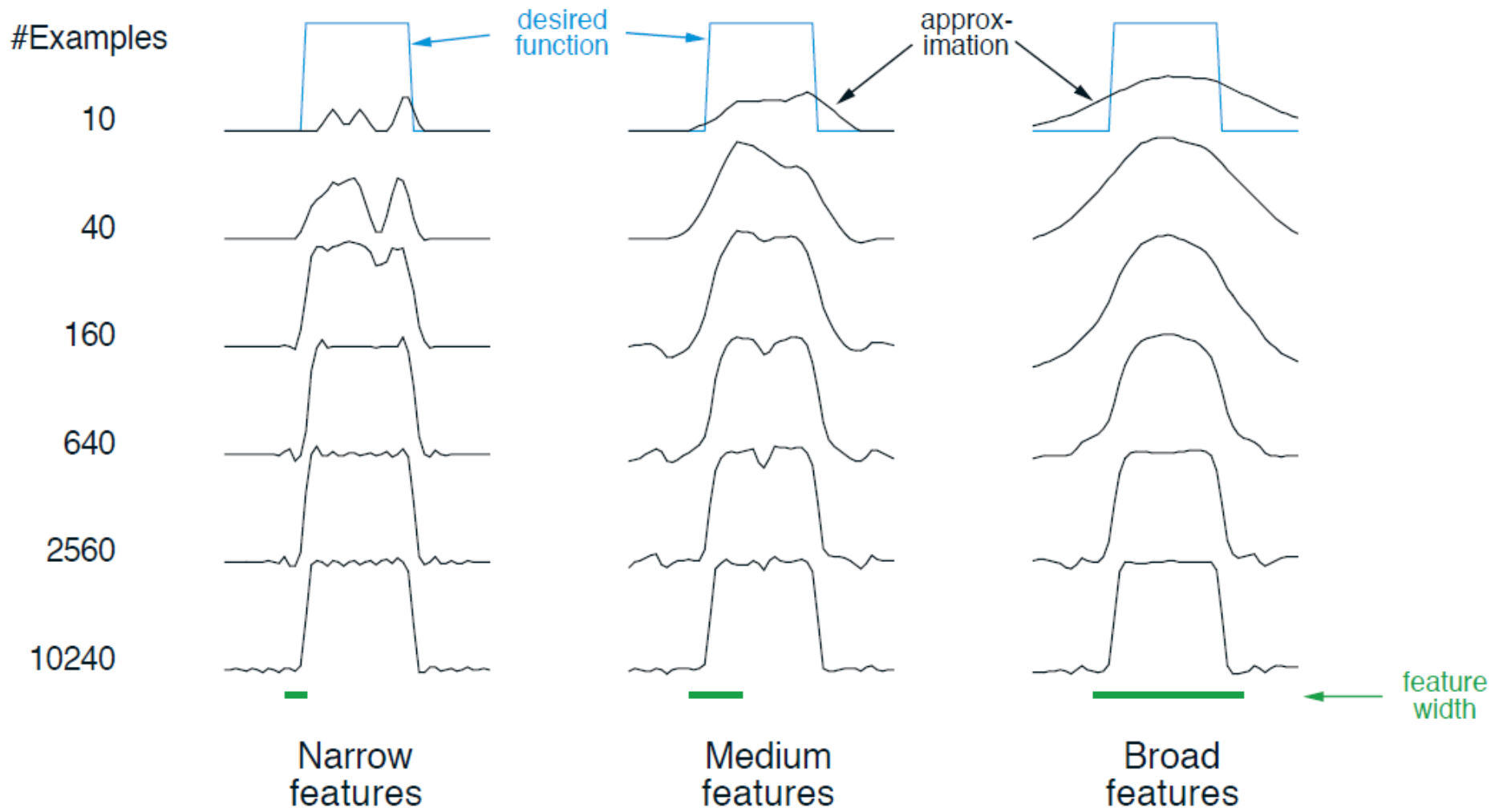


Asymmetric generalization



# Example: Coarseness of Coarse Coding

Sutton and Barto, Reinforcement Learning, 2018





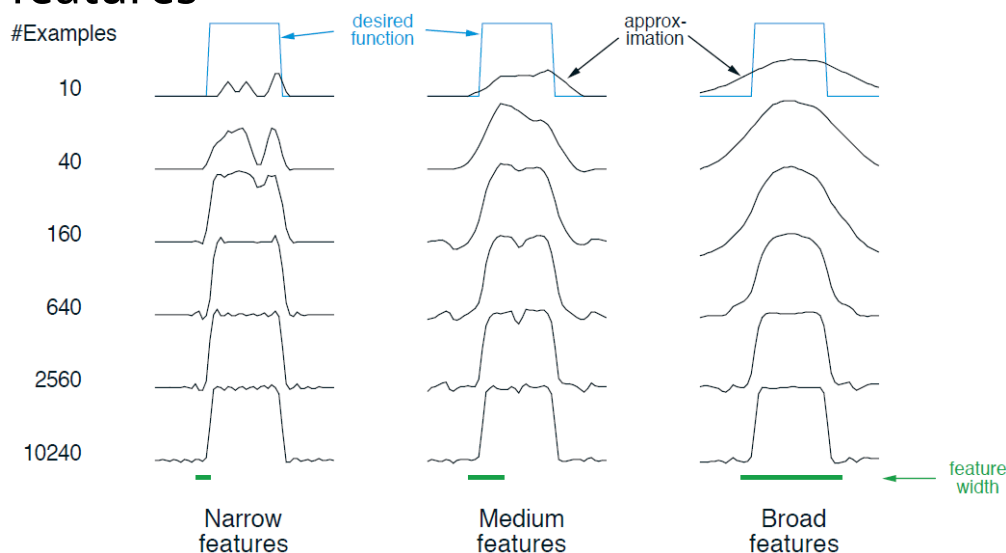
# Example: Coarseness of Coarse Coding

- This example illustrates the effect on learning of the size of the receptive fields in coarse coding
- Linear FA based on coarse coding was used to learn a one-dimensional square-wave function
- The values of this function were used as the targets,  $U_t$
- With just one dimension, the receptive fields were intervals rather than circles
- Learning was repeated with three different sizes of the intervals: narrow, medium, and broad
- 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



# Example: Coarseness of Coarse Coding

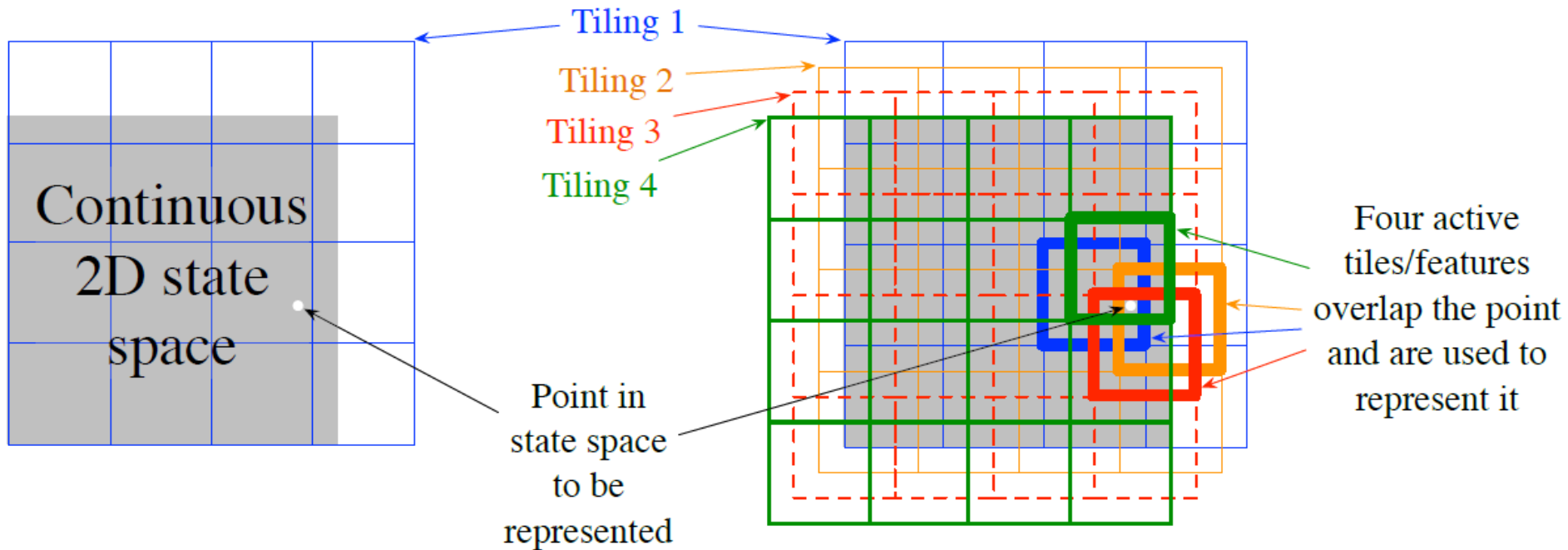
- The figure shows the functions learned in all three cases over the course of learning
- Note that the width of the features had a strong effect early in learning
- With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of each trained point were changed, causing the function learned to be more bumpy
- However, the final function learned was affected only slightly by the width of the features







# Tile Coding





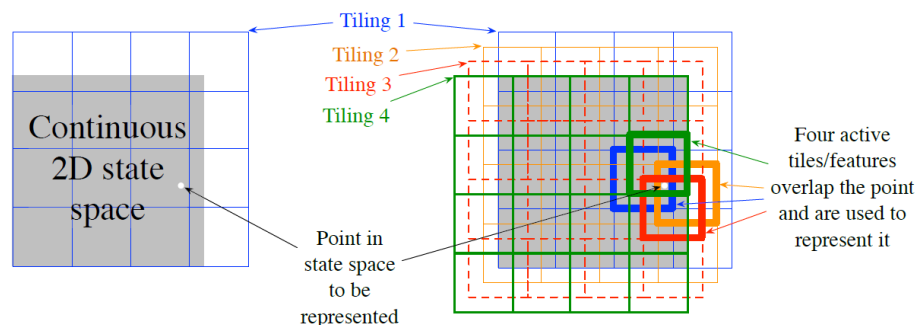
# Tile Coding

- Tile coding is a form of coarse coding for multi-dimensional continuous spaces that is computationally efficient
- In tile coding the receptive fields of the features are grouped into partitions of the state space
- Each such partition is called a tiling, and each element of the partition is called a tile
- E.g., the simplest tiling of a two-dimensional state space is a uniform grid; the tiles or receptive field here are squares rather than the circles
- If just this single tiling were used, then the state indicated by a point would be represented by the single feature whose tile it falls within; generalization would be complete to all states within the same tile and nonexistent to states outside it
- With just one tiling, we would not have coarse coding but just a case of state aggregation



# Tile Coding

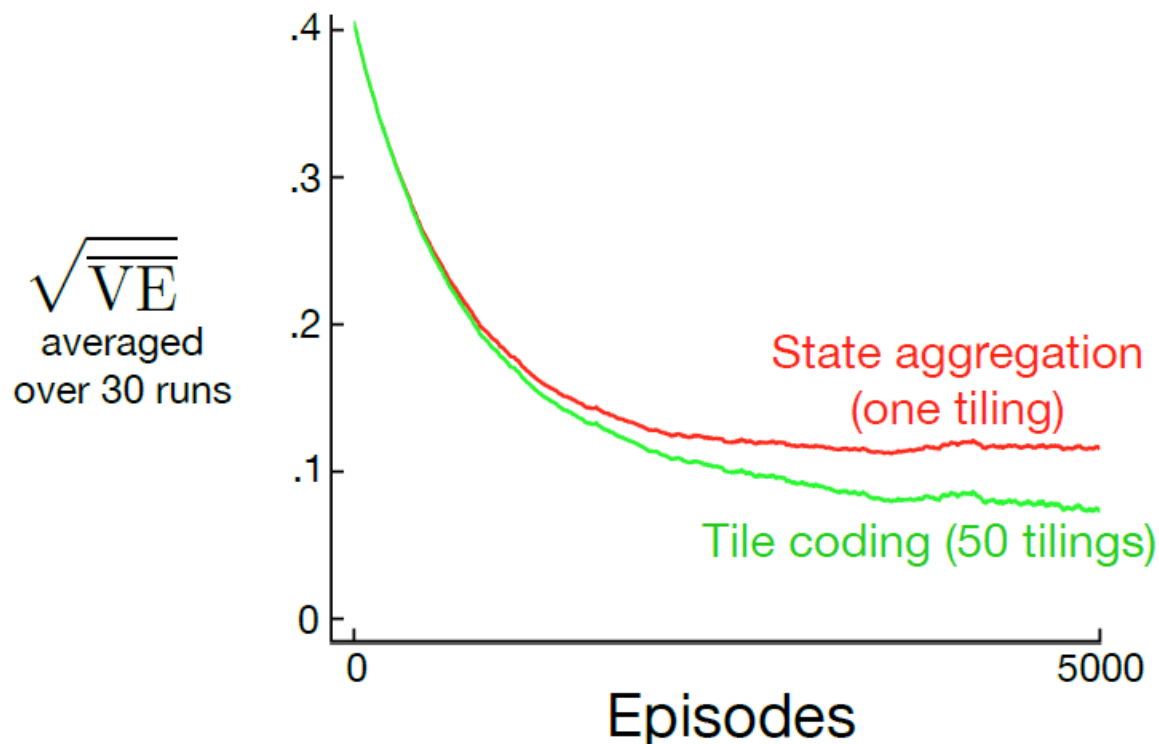
- To get the strengths of coarse coding requires overlapping receptive fields, and by definition the tiles of a partition do not overlap
- To get true coarse coding with tile coding, multiple tilings are used, each offset by a fraction of a tile width
- Every state, such as that indicated by the white spot, falls in exactly one tile in each of the four tilings
- These four tiles correspond to four features that become active when the state occurs; specifically, the feature vector  $x(s)$  has one component for each tile in each tiling
- In this example there are  $4 \times 4 \times 4 = 64$  components, all of which will be 0 except for the four corresponding to the tiles that  $s$  falls within





# Tile Coding

- Advantage of multiple offset tilings (coarse coding) over a single tiling on the 1000-state random walk example





# Tile Coding

- An immediate practical advantage of tile coding is that, because it works with partitions, the overall number of features that are active at one time is the same for any state; exactly one feature is present in each tiling, so the total number of features present is always the same as the number of tilings
- This allows the step-size parameter  $\alpha$  to be set in an easy, intuitive way; e.g., choosing  $\alpha = \frac{1}{n}$ , where  $n$  is the number of tilings, results in exact one-trial learning

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

- If the example  $s \rightarrow v$  is trained on, then whatever the prior estimate,  $\hat{v}(s, \mathbf{w}_t)$ , the new estimate will be  $\hat{v}(s, \mathbf{w}_{t+1}) = v$
- Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. For example, one might choose  $\alpha = \frac{1}{10n}$ , in which case the estimate for the trained state would move one-tenth of the way to the target in one update, and neighboring states will be moved less, proportional to the number of tiles they have in common



# Tile Coding

- 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 is almost trivial to compute

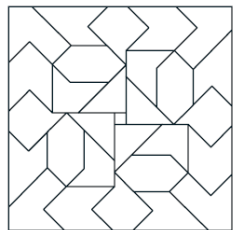
$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^d w_i x_i(s)$$

- Rather than performing  $d$  multiplications and additions, one simply computes the indices of the  $n \ll d$  active features and then adds up the  $n$  corresponding components of the weight vector

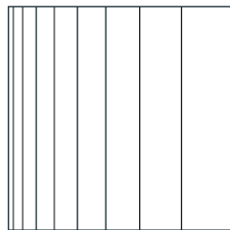


# Tile Coding

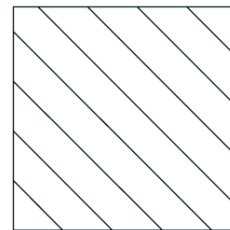
- In choosing a tiling strategy, one has to pick the number of the tilings and the shape of the tiles which determine the resolution and the generalization, respectively
- Square tiles will generalize roughly equally in each dimension
- Tiles that are elongated along one dimension, such as the stripe tilings, will promote generalization along that dimension
- The tilings in the middle figure are also denser and thinner on the left, promoting discrimination along the horizontal dimension at lower values along that dimension
- The diagonal stripe tiling will promote generalization along one diagonal. In higher dimensions, axis-aligned stripes correspond to ignoring some of the dimensions in some of the tilings, that is, to hyperplanar slices
- Irregular tilings are also possible, though rare in practice and beyond the standard software



Irregular



Log stripes



Diagonal stripes

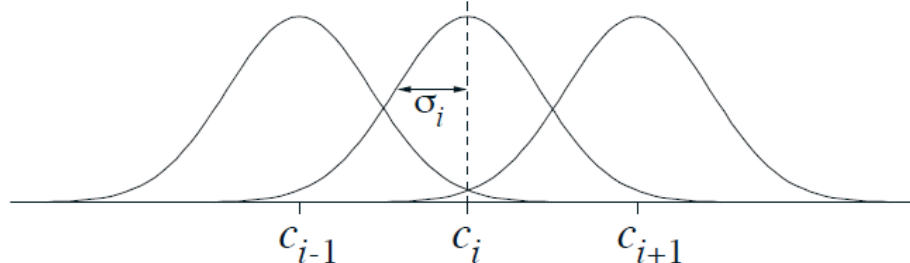


# Radial Basis Functions

- Radial basis functions (RBFs) are the natural generalization of coarse coding to continuous valued features
- Rather than each feature being either 0 or 1, it can be anything in the interval  $[0, 1]$ , reflecting various degrees to which the feature is present
- A typical RBF feature,  $x_i$ , has a Gaussian (bell-shaped) response  $x_i(s)$  dependent only on the distance between the state,  $s$ , and the feature's prototypical or center state  $c_i$ , and relative to the feature's width  $\sigma_i$

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

- E.g., 1-D RBF with a Euclidean distance metric (L2 norm)








# 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 (substantial additional computation without performance gain)
- RBF network is a linear FA using RBFs for its features
- In addition, some learning methods for RBF networks change the centers and widths of the features as well, bringing them into the realm of nonlinear FA
- Nonlinear methods may fit target functions much more precisely
- The downside to RBF networks, and to nonlinear RBF networks especially, is greater computational complexity and, often, more manual tuning (of centers and widths) before learning is robust and efficient



# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
-   **Selecting Step-Size Parameters Manually**
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion



# Selecting Step-Size Parameters Manually

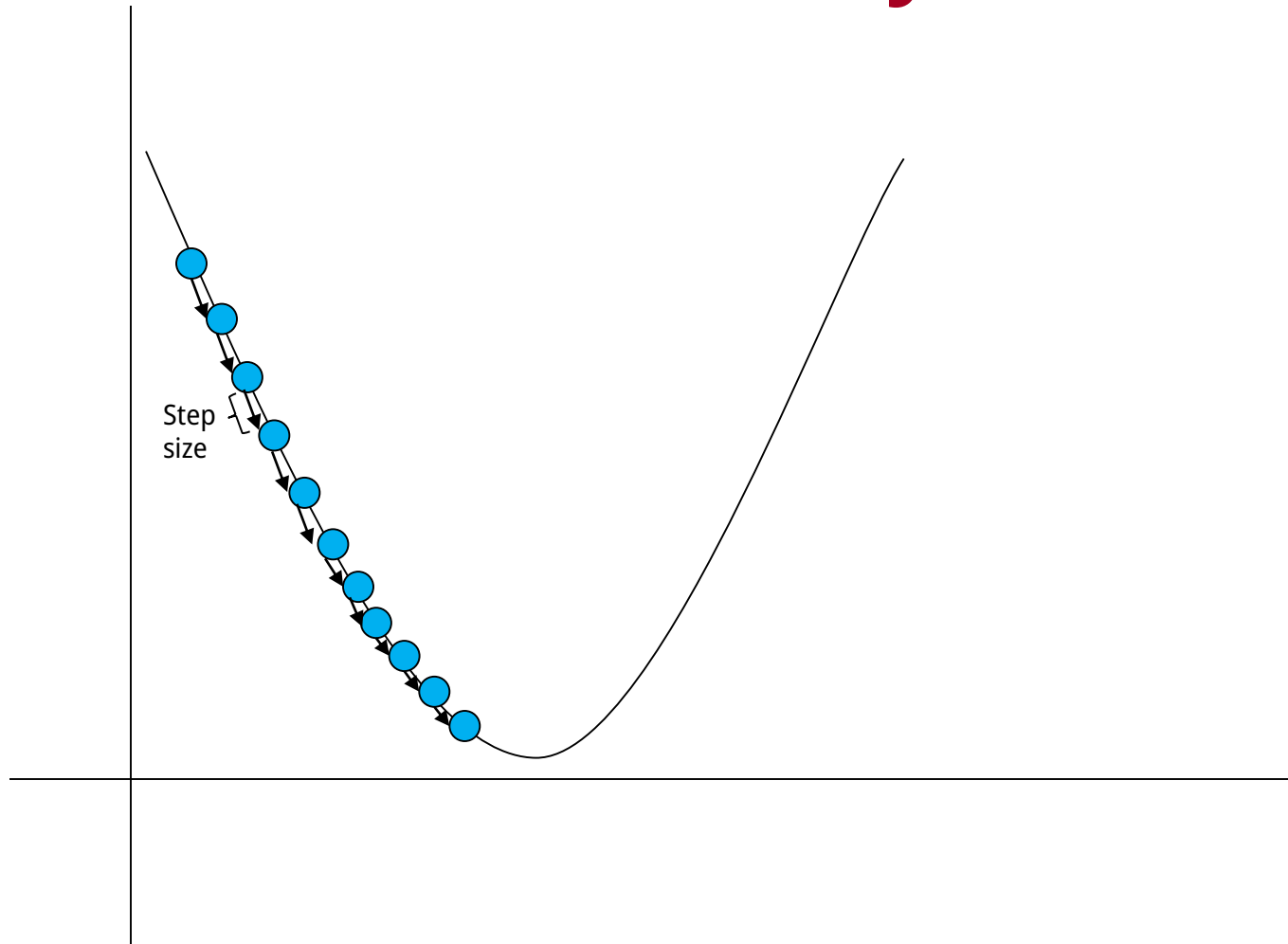
- Most SGD methods require the designer to select an appropriate step-size parameter  $\alpha$

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

- Ideally this selection would be automated, and in some cases it has been, but for most cases it is still common practice to set it manually
- To do this, and to better understand the algorithms, it is useful to develop some intuitive sense of the role of the step-size parameter
- Can we say in general how it should be set?



# Selecting Step-Size Parameters Manually





# Selecting Step-Size Parameters Manually

- Theoretical considerations are unfortunately of little help
- The theory of stochastic approximation gives us conditions on a slowly decreasing step-size sequence that are sufficient to guarantee convergence, but these tend to result in learning that is too slow.
- The classical choice  $\alpha_t = 1/t$ , which produces sample averages in tabular MC methods, is not appropriate for TD methods, for nonstationary problems, or for any method using FA
- For linear methods, there are recursive least-squares methods, and these methods can be extended to TD learning as in the LSTD method, but these require  $O(d^2)$  step-size parameters, or  $d$  times more parameters than we are learning
- For this reason we rule them out for use on large problems where FA is most needed



# Selecting Step-Size Parameters Manually

- To get some intuitive feel for how to set the step-size parameter manually, it is best to go back momentarily to the tabular case
- There we can understand that a step size of  $\alpha = 1$  will result in a complete elimination of the sample error after one target  
$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$
- We usually want to learn slower than this
- In the tabular case, a step size of  $\alpha = \frac{1}{10}$  would take about 10 experiences to converge approximately to their mean target, and if we wanted to learn in 100 experiences we would use  $\alpha = \frac{1}{100}$
- In general, if  $\alpha = \frac{1}{\tau}$ , then the tabular estimate for a state will approach the mean of its targets, with the most recent targets having the greatest effect, after about  $\tau$  experiences with the state



# Selecting Step-Size Parameters Manually

- With general FA there is not such a clear notion of number of experiences with a state, as each state may be similar to and dissimilar from all the others to various degrees
- However, there is a similar rule that gives similar behavior in the case of linear FA
- Suppose you wanted to learn in about  $\tau$  experiences with substantially the same feature vector
- A good rule of thumb for setting the step-size parameter of linear SGD methods is then  $\alpha = (\tau \mathbb{E}[x^T x])^{-1}$

- Note the linear SGD update rule

$$w_{t+1} = w_t + \alpha [U_t - \hat{v}(S_t, w_t)] x(S_t)$$

- where  $x$  is a random feature vector chosen from the same distribution as input vectors in the SGD. This method works best if the feature vectors do not vary greatly in length; ideally  $x^T x$  is a constant



# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks**
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion





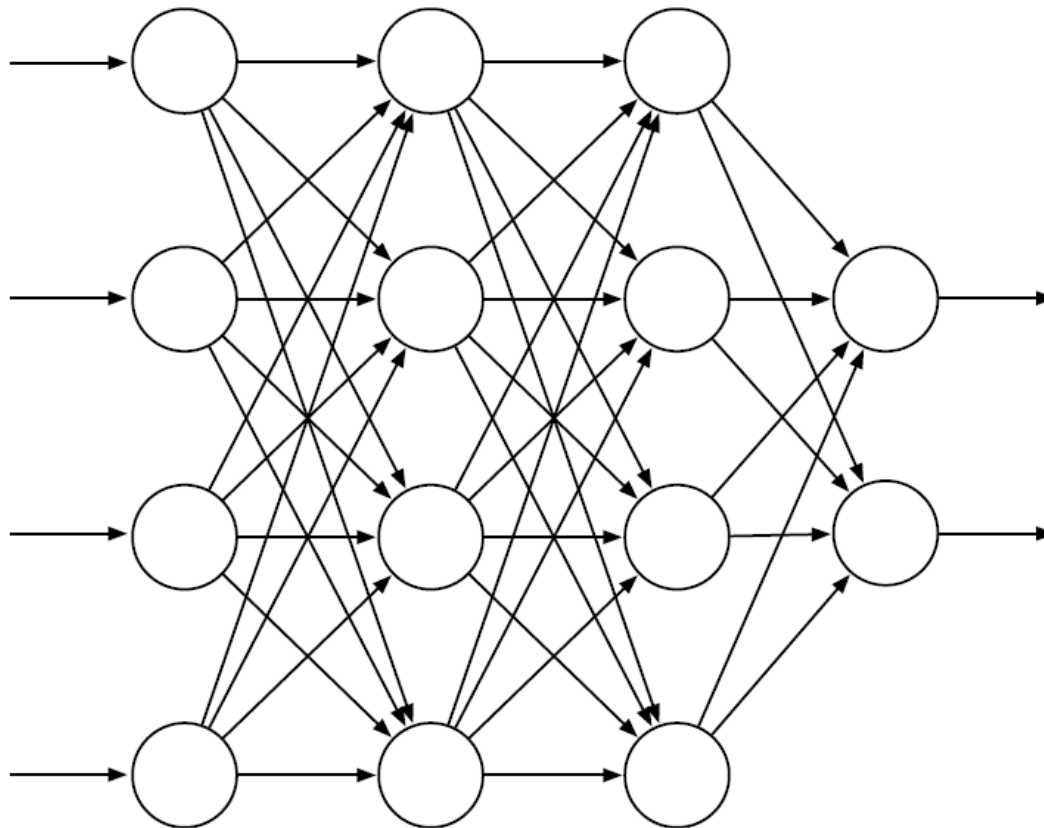
# Nonlinear FA: Artificial Neural Networks

- Artificial neural networks (ANNs) are widely used for nonlinear FA
- An ANN is a network of interconnected units that have some of the properties of neurons, the main components of nervous systems
- ANNs have a long history, with the latest advances in training deeply-layered ANNs (deep learning) being responsible for some of the most impressive abilities of machine learning systems, including RL systems



# Nonlinear FA: Artificial Neural Networks

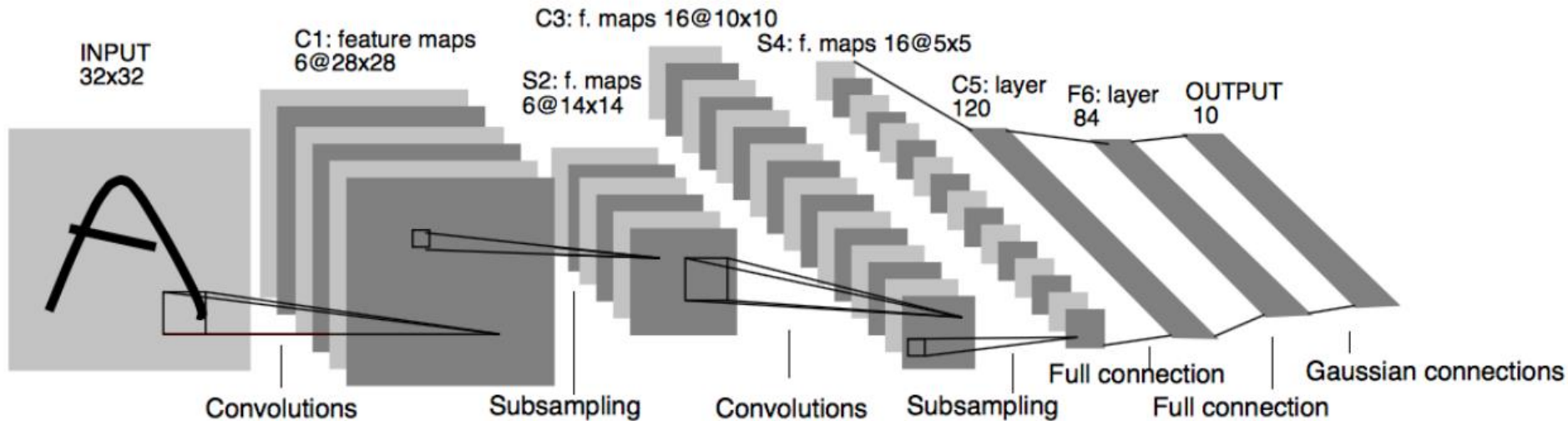
- A generic feedforward ANN with 4 input units, two output units, and two hidden layers






# Nonlinear FA: Artificial Neural Networks

- Deep convolutional network





# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
-   **Least-Squares TD**
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion



# Least-Squares TD

- All the methods we have discussed so far in this chapter have required computation per time step proportional to the number of parameters
- With more computation, however, one can do better
- We discuss a method for linear FA that is arguably the best that can be done for this case
- As discussed, TD(0) with linear FA converges asymptotically (for appropriately decreasing step sizes) to the TD fixed point

$$w_{TD} = A^{-1}b$$

- where

$$A = \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T] \text{ and } b = \mathbb{E}[R_{t+1}x_t]$$

- Can we compute A and b this directly, not iteratively, and thus get the TD fixed point?
  - Yes! The answer is Least-squares TD (LSTD)



# Least-Squares TD

- LSTD forms the natural estimates

$$\hat{A}_t = \sum_{k=0}^{t-1} x_k (x_k - \gamma x_{k+1})^T + \epsilon I \quad \text{and} \quad \hat{b}_t = \sum_{k=0}^{t-1} R_{k+1} x_k$$

- where  $I$  is the identity matrix, and  $\epsilon I$ , for some small  $\epsilon > 0$ , ensures that  $\hat{A}_t$  is always invertible
  - A strictly diagonally dominant matrix is invertible
  - [https://en.wikipedia.org/wiki/Diagonally\\_dominant\\_matrix](https://en.wikipedia.org/wiki/Diagonally_dominant_matrix)
- Note that  $1/t$  is not written since it is canceled out

$$w_t = \hat{A}_t^{-1} \hat{b}_t$$

- This algorithm is the most data efficient form of linear TD(0), but it is also more expensive computationally. Recall that semi-gradient TD(0) requires memory and per-step computation that is only  $O(d)$



# Least-Squares TD

- How complex is LSTD? As it is written above the complexity seems to increase with  $t$ , but the two approximations ( $\hat{A}_t$  and  $\hat{b}_t$ ) could be implemented incrementally so that they can be done quickly per step
- Even so, the update for  $\hat{A}_t$  would involve an outer product (a column vector times a row vector) and thus would be a matrix update; its computational complexity would be  $O(d^2)$ , and of course the memory required to hold the  $\hat{A}_t$  matrix would be  $O(d^2)$



# Least-Squares TD

- A potentially greater problem is that our final computation  $w_t = \hat{A}_t^{-1} \hat{b}_t$  uses the inverse of  $\hat{A}_t$ , and the computational complexity of a general inverse computation is  $O(d^3)$
- Fortunately, an inverse of a matrix of our special form—a sum of outer products—can also be updated incrementally with only  $O(d^2)$  computations (Sherman-Morrison formula)

$$\begin{aligned}\hat{A}_t^{-1} &= \left( \hat{A}_{t-1} + x_t(x_t - \gamma x_{t+1})^T \right)^{-1} \\ &= \hat{A}_{t-1}^{-1} - \frac{\hat{A}_{t-1}^{-1} x_t (x_t - \gamma x_{t+1})^T \hat{A}_{t-1}^{-1}}{1 + (x_t - \gamma x_{t+1})^T \hat{A}_{t-1}^{-1} x_t}\end{aligned}$$

- for  $t > 0$ , with  $\hat{A}_0 = \epsilon I$
- It involves only vector-matrix and vector-vector multiplications and thus is only  $O(d^2)$





# Least-Squares TD

Sutton and Barto,  
Reinforcement  
Learning, 2018

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

Input: feature representation  $\mathbf{x} : \mathcal{S}^+ \rightarrow \mathbb{R}^d$  such that  $\mathbf{x}(\text{terminal}) = \mathbf{0}$

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

$$\begin{aligned} \hat{A}_t^{-1} &= (\hat{A}_{t-1} + x_t(x_t - \gamma x_{t+1})^T)^{-1} \\ &= \hat{A}_{t-1}^{-1} - \frac{\hat{A}_{t-1}^{-1} x_t (x_t - \gamma x_{t+1})^T \hat{A}_{t-1}^{-1}}{1 + (x_t - \gamma x_{t+1})^T \hat{A}_{t-1}^{-1} x_t} \end{aligned}$$




# Least-Squares TD

- Of course,  $O(d^2)$  is still significantly more expensive than the  $O(d)$  of semi-gradient TD
- Whether the greater data efficiency of LSTD is worth this computational expense depends on how large  $d$  is, how important it is to learn quickly, and the expense of other parts of the system
- LSTD does not require a step size, but it does requires  $\epsilon$ ; if  $\epsilon$  is chosen too small the sequence of inverses can vary wildly, and if  $\epsilon$  is chosen too large then learning is slowed
- In addition, LSTD's lack of a step-size parameter means that it never forgets. This is sometimes desirable, but it is problematic if the target policy  $\pi$  changes as it does in RL and GPI
- In control applications, LSTD typically has to be combined with some other mechanism to induce forgetting



# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
-   **Memory-based Function Approximation**
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion



# Memory-based FA

- So far we have discussed the parametric approach to approximating value functions. In this approach, a learning algorithm adjusts the parameters of a functional form intended to approximate the value function over a problem's entire state space
- Each update,  $s \rightarrow g$ , is a training example used by the learning algorithm to change the parameters with the aim of reducing the approximation error
- After the update, the training example can be discarded (although it might be saved to be used again)
- When an approximate value of a state (which we will call the query state) is needed, the function is simply evaluated at that state using the latest parameters produced by the learning algorithm



# Memory-based FA

- Memory-based FA methods are very different. They simply save training examples in memory as they arrive (or at least save a subset of the examples) without updating any parameters
- Whenever a query state's value estimate is needed, a set of examples is retrieved from memory and used to compute a value estimate for the query state
- This approach is sometimes called lazy learning because processing training examples is postponed until the system is queried to provide an output
- Memory-based FA methods are prime examples of *nonparametric* methods. Unlike parametric methods, the approximating function's form is not limited to a fixed parameterized class of functions, such as linear functions or polynomials, but is instead determined by the training examples themselves, together with some means for combining them to output estimated values for query states
- As more training examples accumulate in memory, one expects nonparametric methods to produce increasingly accurate approximations of any target function



# Memory-based FA

- There are many different memory-based methods depending on how the stored training examples are selected and how they are used to respond to a query
- We focus on *local-learning* methods that approximate a value function only locally in the neighborhood of the current query state
- These methods retrieve a set of training examples from memory whose states are judged to be the most relevant to the query state, where relevance usually depends on the distance between states: the closer a training example's state is to the query state, the more relevant it is considered to be, where distance can be defined in many different ways
- After the query state is given a value, the local approximation is discarded



# Memory-based FA

- The simplest example of the memory-based approach is the nearest neighbor method, which simply finds the example in memory whose state is closest to the query state and returns that example's value as the approximate value of the query state
- Slightly more complicated are weighted average methods that retrieve a set of nearest neighbor examples and return a weighted average of their target values, where the weights generally decrease with increasing distance between their states and the query state



# Memory-based FA

- Locally weighted regression is similar, but it fits a surface to the values of a set of nearest states by means of a parametric approximation method that minimizes a weighted error measure, where the weights depend on distances from the query state
  - In typical linear regression, we find  $\theta$  to minimize  $\sum_{i=1}^m (y^{(i)} - \theta^T x^{(i)})^2$ ; then we output  $\theta^T x$  for a given query point  $x$
  - In locally weighted regression, we instead find  $\theta$  to minimize  $\sum_{i=1}^k w^{(i)} (\tilde{y}^{(i)} - \theta^T \tilde{x}^{(i)})^2$  where  $\tilde{x}^{(i)}$  for  $i=1\dots k$  are the  $k$  nearest neighbors of a query point  $x$ ;  $w^{(i)}$  is the weight given to the  $i$ -th nearest neighbor, e.g.,  
$$w^{(i)}(x^{(i)}) = \exp\left(-\frac{(x^{(i)}-x)^2}{2\tau^2}\right)$$
- The value returned is the evaluation of the locally-fitted surface at the query state, after which the local approximation surface is discarded





# Memory-based FA

- Being nonparametric, memory-based methods have the advantage over parametric methods of not limiting approximations to pre-specified functional forms
- This allows accuracy to improve as more data accumulates
- Memory-based local approximation methods have other properties that make them well suited for RL
- Because trajectory sampling is of such importance in RL, memory-based local methods can focus FA on local neighborhoods of states (or state–action pairs) visited in real or simulated trajectories
- There may be no need for global approximation because many areas of the state space will never (or almost never) be reached
- In addition, memory-based methods allow an agent’s experience to have a relatively immediate affect on value estimates in the neighborhood of the current state, unlike parametric methods which incrementally adjust parameters of a global approximation



# Memory-based FA

- Avoiding global approximation is also a way to address the curse of dimensionality
- For example, for a state space with  $k$  dimensions, a tabular method storing a global approximation requires memory exponential in  $k$
- On the other hand, in storing examples for a memory-based method, each example requires memory proportional to  $k$ , and the memory required to store, say,  $n$  examples is linear in  $n$
- Of course, the critical remaining issue is whether a memory-based method can answer queries quickly enough to be useful to an agent
- A related concern is how speed degrades as the size of the memory grows; finding nearest neighbors in a large database can take too long to be practical in many applications



# Memory-based FA

- Speeding up nearest neighbor search
  - Using parallel computers or special purpose hardware
  - Using special multi-dimensional data structures to store the training data; e.g., k-d tree which recursively splits a k-dimensional space into regions arranged as nodes of a binary tree
- Speeding up locally weighted regression
  - Forgetting entries in order to keep the size of the database within bounds



# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation**
- Looking Deeper at On-policy Learning: Interest and Emphasis
- Conclusion



# Kernel-based FA

- Memory-based methods such as the weighted average and locally weighted regression methods described above depend on assigning weights to examples  $s' \rightarrow g$  in the database depending on the distance between  $s'$  and a query states  $s$
- The function that assigns these weights is called a kernel (function)
- In the weighted average and locally weighted regressions methods, for example, a kernel function assigns weights to distances between states
- More generally, weights do not have to depend on distances; they can depend on some other measure of similarity between states
- In this case,  $k: S \times S \rightarrow R$ , so that  $k(s, s')$  is the weight given to data about  $s'$  in its influence on answering queries about  $s$
- Viewed slightly differently,  $k(s, s')$  is a measure of the strength of generalization from  $s'$  to  $s$
- Kernel functions numerically express how relevant knowledge about any state is to any other state



# Kernel-based FA

- Kernel regression is the memory-based method that computes a kernel weighted average of the targets of all examples stored in memory, assigning the result to the query state
- If  $D$  is the set of stored examples, and  $g(s')$  denotes the target function for state  $s'$  in a stored example, then kernel regression approximates the target function, in this case a value function depending on  $D$ , as

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

- The weighted average method is a special case in which  $k(s, s')$  is non-zero only when  $s$  and  $s'$  are close to one another so that the sum need not be computed over all of  $D$



# Kernel-based FA

- A common kernel is the Gaussian radial basis functions (RBFs) used in FA
- When used in parametric FA, RBFs are features which express the weight between a state and a center

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

- Those centers and widths are either fixed from the start, with centers presumably concentrated in areas where many examples are expected to fall, or are adjusted in some way during learning
- Barring methods that adjust centers and widths, this is a linear parametric method whose parameters are the weights of each RBF, which are typically learned by stochastic gradient, or semi-gradient, descent
- The form of the approximation is a linear combination of the pre-determined RBFs




# Kernel-based FA

- Kernel regression with a RBF kernel differs from the parametric method in two ways
  - 1) It is memory-based: the RBFs are centered on the states of the stored examples
  - 2) It is nonparametric: there are no parameters to learn





# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
-   **Looking Deeper at On-policy Learning: Interest and Emphasis**
- Conclusion



# Looking Deeper at On-policy Learning: Interest and Emphasis

- The algorithms we have considered so far have treated all the states encountered equally, as if they were all equally important
- In some cases, we are more interested in some states than others
- In discounted episodic problems, we may be more interested in accurately valuing early states in the episode than in later states where discounting may have made the rewards much less important to the value of the start state
- Or, if an action-value function is being learned, it may be less important to accurately value poor actions whose value is much less than the greedy action
- FA resources are always limited, and if they were used in a more targeted way, then performance could be improved



# Looking Deeper at On-policy Learning: Interest and Emphasis

- New concepts: interest and emphasis
- Interest
  - A non-negative scalar measure, a random variable  $I_t$ , called interest indicates the degree to which we are interested in accurately valuing the state (or state–action pair) at time  $t$
  - If we don't care at all about the state, then the interest should be 0; if we fully care, it might be 1, though it is formally allowed to take any non-negative value
  - The interest can be set in any causal way; for example, it may depend on the trajectory up to time  $t$  or the learned parameters at time  $t$ . The distribution  $\mu$  in the  $\overline{VE}$  is then defined as the distribution of states encountered while following the target policy, weighted by the interest



# Looking Deeper at On-policy Learning: Interest and Emphasis

## ■ Emphasis

- A non-negative scalar random variable  $M_t$ , called emphasis, multiplies the learning update and thus emphasizes or de-emphasizes the learning done at time  $t$

- The general  $n$ -step learning rule using emphasis is

$$w_{t+n} = w_{t+n-1} + \alpha M_t [G_{t:t+n} - \hat{v}(S_t, w_{t+n-1})] \nabla \hat{v}(S_t, w_{t+n-1}), 0 \leq t \leq T$$

- with the  $n$ -step return given and the emphasis determined recursively from the interest by

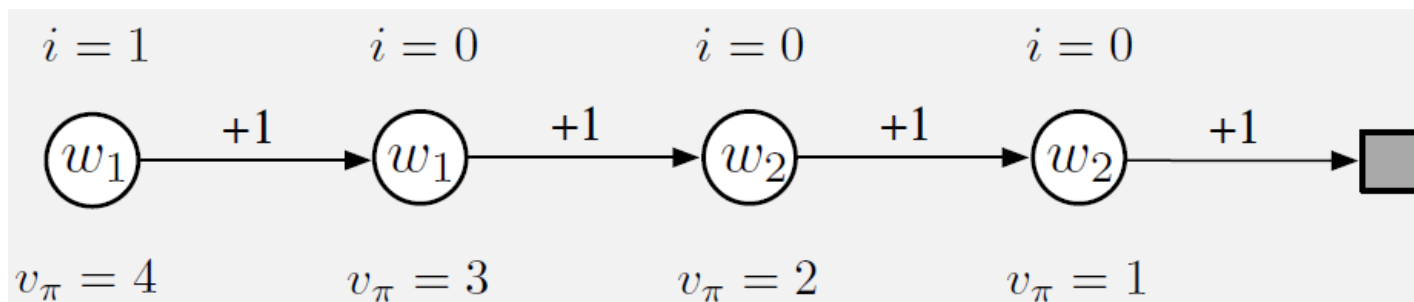
$$M_t = I_t + \gamma^n M_{t-n}, 0 \leq t \leq T$$

- with  $M_t = 0$  for all  $t < 0$
- These equations are taken to include the MC case, for which  $G_{t:t+n} = G_t$ , all the updates are made at end of the episode,  $n = T - t$ , and  $M_t = I_t$



# Example: Interest and Emphasis

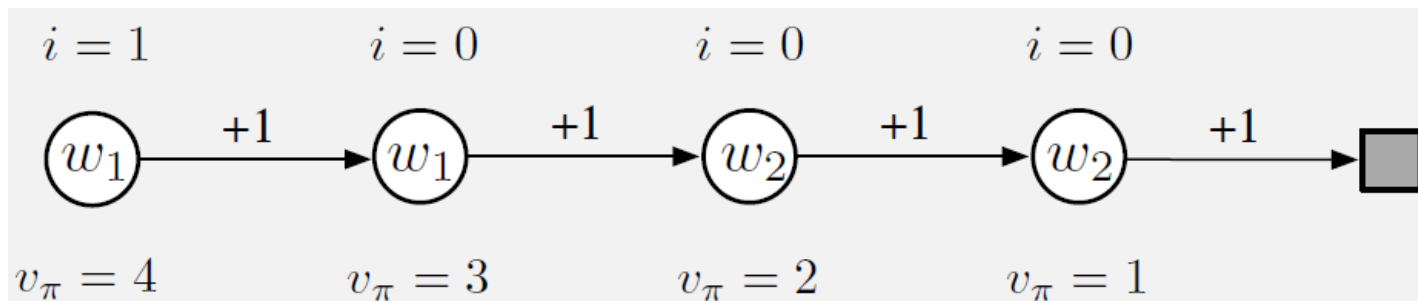
- Consider a four-state Markov reward process
- Episodes start in the leftmost state, then transition one state to the right, with a reward of +1, on each step until the terminal state is reached
- The true value of the first state is thus 4, of the second state 3, and so on as shown below each state
- These are the true values; the estimated values can only approximate these because they are constrained by the parameterization. There are two components to the parameter vector  $w = (w_1, w_2)^T$ , and the parameterization is as written inside each state





# Example: Interest and Emphasis

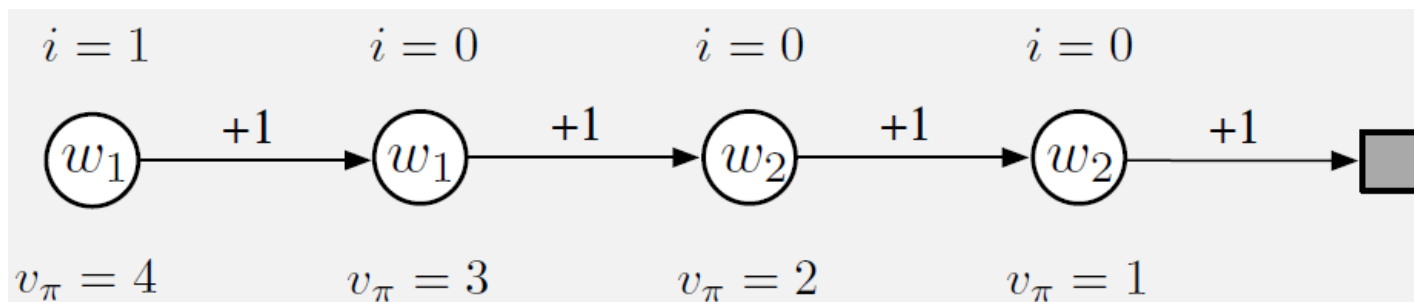
- The estimated values of the first two states are given by  $w_1$  alone and thus must be the same even though their true values are different
- Similarly, the estimated values of the third and fourth states are given by  $w_2$  alone and must be the same even though their true values are different
- Suppose that we are interested in accurately valuing only the leftmost state; we assign it an interest of 1 while all the other states are assigned an interest of 0, as indicated above the states





# Example: Interest and Emphasis

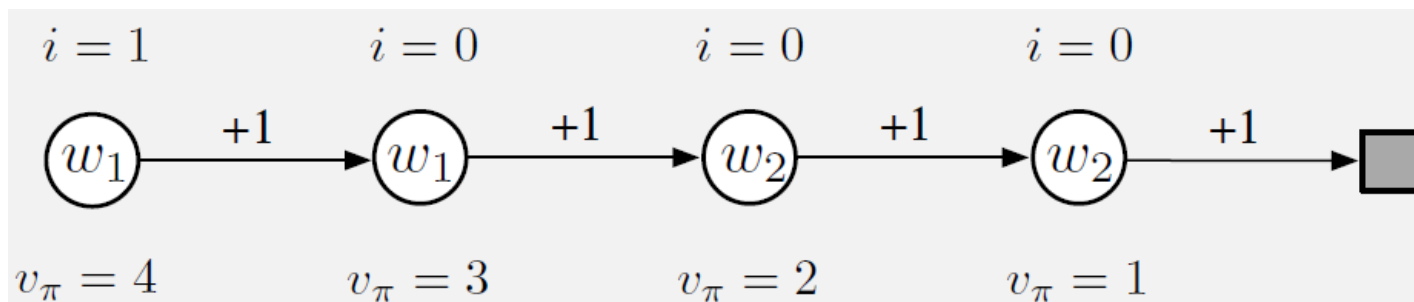
- First consider applying gradient MC algorithms to this problem
- The algorithms that do not take into account interest and emphasis will converge (for decreasing step sizes) to the parameter vector  $w_\infty = (3.5, 1.5)$ , which gives the first state—the only one we are interested in—a value of 3.5 (i.e., intermediate between the true values of the first and second states)
- The methods presented that do use interest and emphasis, on the other hand, will learn the value of the first state exactly correctly;  $w_1$  will converge to 4 while  $w_2$  will never be updated because the emphasis is zero in all states except the leftmost state





# Example: Interest and Emphasis


- Now consider applying two-step semi-gradient TD methods
- The methods without interest and emphasis will again converge to  $w_\infty = (3.5, 1.5)$ , while the methods with interest and emphasis converge to  $w_\infty = (4, 2)$
- The latter produces the exactly correct values for the first state and for the third state (which the first state bootstraps from) while never making any updates corresponding to the second or fourth states







# Outline

- Value-function Approximation
- The Prediction Objective ( $\overline{VE}$ )
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Selecting Step-Size Parameters Manually
- Nonlinear Function Approximation: Neural Networks
- Least-Squares TD
- Memory-based Function Approximation
- Kernel-based Function Approximation
- Looking Deeper at On-policy Learning: Interest and Emphasis
-   **Conclusion**



# Conclusion

- RL systems must be capable of generalization if they are to be applicable to AI or to large engineering applications
- To do this, supervised learning methods can be used simply by treating each update as a training example
- We discussed supervised learning methods using parameterized FA, where the policy is parameterized by a weight vector  $w$
- Although the weight vector has many components, the state space is much larger still, and we must settle for an approximate solution
- We defined the mean squared value error,  $\overline{VE}(w)$ , as a measure of the error in the values  $v_{\pi_w}(s)$  for a weight vector  $w$  under the on-policy distribution  $\mu$
- $\overline{VE}$  gives us a clear way to rank different value-function approximations in the on-policy case



# Conclusion

- To find a good weight vector, the most popular methods are variations of stochastic gradient descent (SGD)
- We have focused on the on-policy case with a fixed policy, also known as policy evaluation or prediction; a natural learning algorithm for this case is  $n$ -step semi-gradient TD, which includes gradient MC and semi-gradient TD(0) algorithms as the special cases when  $n=\infty$  and  $n=1$  respectively
- Semi-gradient TD methods are not true gradient methods. In such bootstrapping methods (including DP), the weight vector appears in the update target, yet this is not taken into account in computing the gradient—thus they are semi-gradient methods. As such, they cannot rely on classical SGD results



# Conclusion

- Nevertheless, good results can be obtained for semi-gradient methods in the special case of linear FA, where the value estimates are sums of features times corresponding weights
- The linear case works well with appropriate features
- Choosing the features
  - Polynomials: generalizes poorly in the online RL
  - Fourier basis and coarse coding with sparse overlapping receptive fields are widely used
  - Tile coding is a form of coarse coding that is efficient and flexible
  - Radial basis functions are useful for one- or two-dimensional tasks in which a smoothly varying response is important.
  - LSTD is the most data-efficient linear TD prediction method, but requires computation proportional to the square of the number of weights, whereas all the other methods are of complexity linear in the number of weights
  - Nonlinear methods include ANN



# Exercise

- (Question 1)
- Suppose we have 2x2 grid-world example presented below. We have four actions  $\{up, down, left, right\}$  that we can take for each state. Show gradient descent update steps of one-step *SARSA* for one episode. We use linear FA with the following features.

	$S_1$	$S_2$	$S_3$	$S_4$
<i>up</i>	{0.1, 0.0, 0.0, 0.1}	{0.1, 0.1, 0.0, 0.0}	{0.0, 0.1, 0.0, 0.1}	{0.1, 0.1, 0.0, 0.1}
<i>down</i>	{0.0, 0.2, 0.1, 0.0}	{0.0, 0.1, 0.1, 0.0}	{0.3, 0.1, 0.1, 0.0}	{0.1, 0.2, 0.1, 0.0}
<i>left</i>	{0.0, 0.1, 0.1, 0.0}	{0.0, 0.1, 0.1, 0.0}	{0.0, 0.1, 0.1, 0.0}	{0.0, 0.1, 0.1, 0.0}
<i>right</i>	{0.1, 0.0, 0.1, 0.1}	{0.1, 0.3, 0.0, 0.1}	{0.1, 0.3, 0.0, 0.1}	{0.1, 0.0, 0.2, 0.1}

- $\alpha = 0.1$
- $\gamma = 0.9$
- $W = \text{initialized as all zeros}$
- *Episode* :  $(S_1, down, S_3, right, S_4)$

$S_1$	$S_2, R = 0$
$S_3, R = 0$	$S_4, R = 1$



# Exercise

- (Hint)
- Remember the semi-gradient update equation

$$\begin{aligned}w_{t+1} &= w_t - \frac{1}{2} \alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, w_t)]^2 \\ &= w_t + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t)\end{aligned}$$



# Exercise

$$w_{t+1} = w_t + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t)$$

- (Answer)
- At  $S_1$ : select action down, next state  $S_3$  with  $R = 0$
- we know next action is right based on trajectory.
- So, with initialized  $w = \{0, 0, 0, 0\}$ ,
- $\hat{q}(S, A, w) = w^T x(S_1, \text{down}) = \{0, 0, 0, 0\}^T * \{0.0, 0.2, 0.1, 0.0\} = 0$
- $\hat{q}(S', A', w) = w^T x(S_3, \text{right}) = \{0, 0, 0, 0\}^T * \{0.1, 0.3, 0.0, 0.1\} = 0$
- $\nabla \hat{q}(S, A, w) = x(S_1, \text{down}) = \{0.0, 0.2, 0.1, 0.0\}$
- $w \leftarrow \{0, 0, 0, 0\} + 0.1 [0 + 0.9 * 0 - 0] * \{0.0, 0.2, 0.1, 0.0\} = \{0, 0, 0, 0\} + \{0.0, 0.0, 0.0, 0.0\} = \{0.0, 0.0, 0.0, 0.0\}$

	$S_1$	$S_2$	$S_3$	$S_4$
up	{0.1, 0.0, 0.0, 0.1}	{0.1, 0.1, 0.0, 0.0}	{0.0, 0.1, 0.0, 0.1}	{0.1, 0.1, 0.0, 0.1}
down	{0.0, 0.2, 0.1, 0.0}	{0.0, 0.1, 0.1, 0.0}	{0.3, 0.1, 0.1, 0.0}	{0.1, 0.2, 0.1, 0.0}
left	{0.0, 0.1, 0.1, 0.0}	{0.0, 0.1, 0.1, 0.0}	{0.0, 0.1, 0.1, 0.0}	{0.0, 0.1, 0.1, 0.0}
right	{0.1, 0.0, 0.1, 0.1}	{0.1, 0.3, 0.0, 0.1}	{0.1, 0.3, 0.0, 0.1}	{0.1, 0.0, 0.2, 0.1}

$S_1$	$S_2, R = 0$
$S_3, R = 0$	$S_4, R = 1$



# Exercise

$$w_{t+1} = w_t + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t)$$

- (Answer)
- So, with  $w$  from above,
- $\hat{q}(S, A, w) = w^T x(S_3, \text{right}) = \{0,0,0,0\}^T * \{0.1,0.3,0.0,0.1\} = 0$
- $\nabla \hat{q}(S, A, w) = x(S_3, \text{right}) = \{0.1,0.3,0.0,0.1\}$
- $w \leftarrow \{0.0, 0.0, 0.0, 0.0\} + 0.1[1 + 0.9 * 0] * \{0.1,0.3,0.0,0.1\} = \{0.01,0.03,0.0,0.01\}$

	$S_1$	$S_2$	$S_3$	$S_4$
<i>up</i>	{0.1,0.0,0.0,0.1}	{0.1,0.1,0.0,0.0}	{0.0,0.1,0.0,0.1}	{0.1,0.1,0.0,0.1}
<i>down</i>	{0.0,0.2,0.1,0.0}	{0.0,0.1,0.1,0.0}	{0.3,0.1,0.1,0.0}	{0.1,0.2,0.1,0.0}
<i>left</i>	{0.0,0.1,0.1,0.0}	{0.0,0.1,0.1,0.0}	{0.0,0.1,0.1,0.0}	{0.0,0.1,0.1,0.0}
<i>right</i>	{0.1,0.0,0.1,0.1}	{0.1,0.3,0.0,0.1}	{0.1,0.3,0.0,0.1}	{0.1,0.0,0.2,0.1}

$S_1$	$S_2, R = 0$
$S_3, R = 0$	$S_4, R = 1$





# Questions?