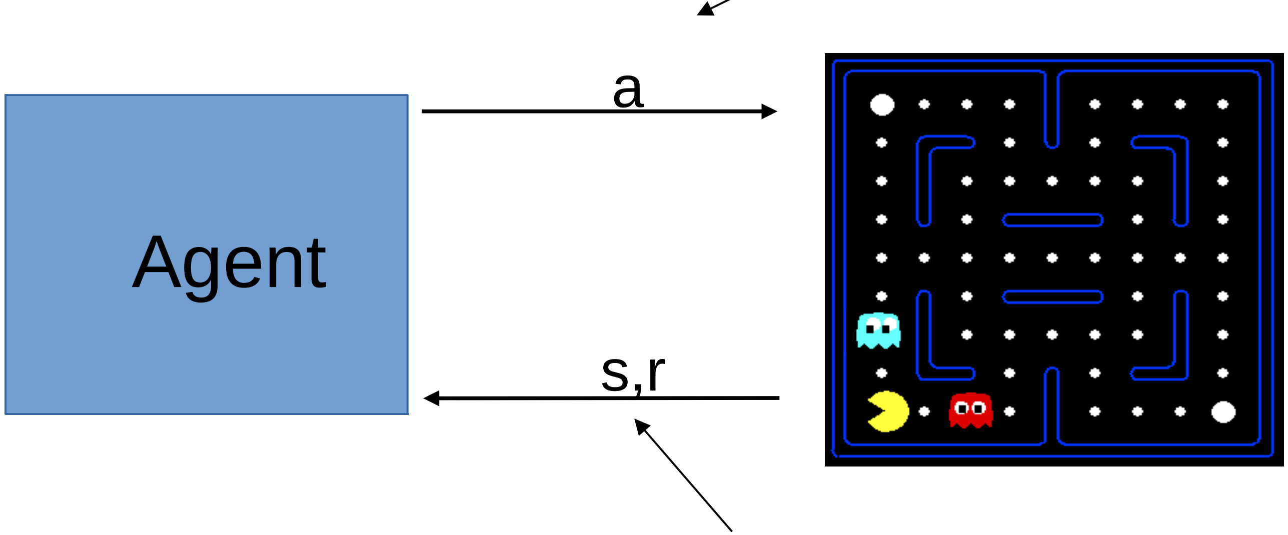
A Closer Look at Function Approximation

Robert Platt
Northeastern University

The problem of large and continuous state spaces

Example of a large state space: Atari Learning Environment
- state: video game screen
- actions: joystick actions
- reward: game score

Agent takes actions



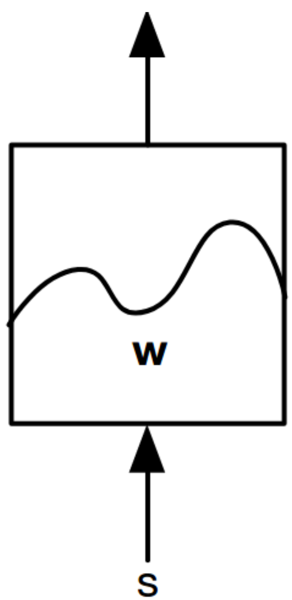
Agent perceives states and rewards

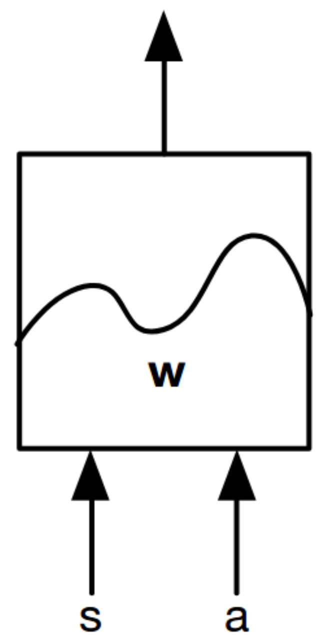
Why are large state spaces a problem for tabular methods?
1. many states may never be visited
2. there is no notion that the agent should behave similarly in “similar” states.

Function approximation

Approximating the Value function using function approximator:

$$\hat{V}(s, w) = V(s) \qquad \hat{Q}(s, a, w) \approx Q(s, a)$$





Some kind of function approximator parameterized by *w*

Which Function Approximator?

There are many function approximators, e.g.

- Linear combinations of features
- Neural networks
- Decision tree
- Nearest Neighbour
- Fourier / wavelet bases

We will require the function approximator to be differentiable

Need to be able to handle non-stationary, non-iid data

Approximating value function using SGD

For starters, let's focus on policy evaluation, i.e. estimating $V^\pi(s)$

Goal: find parameter vector *w* minimizing mean-squared error between approximate value fn, $\hat{V}(s,w)$, and the true value function, $V^\pi(s)$

Approach: do gradient descent on <u>this</u> cost function

$$J(w) = \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - \hat{V}(s,w))^2]$$

Approximating value function using SGD

For starters, let's focus on policy evaluation, i.e. estimating $V^\pi(s)$

Goal: find parameter vector *w* minimizing mean-squared error between approximate value fn, $\hat{V}(s,w)$, and the true value function, $V^\pi(s)$

Approach: do gradient descent on this cost function

$$J(w) = \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - \hat{V}(s,w))^2]$$

Here's the gradient:
$$\begin{aligned} \Delta w &= -\alpha \nabla_w J(w) \\ &= \alpha \mathbb{E}_\pi[(V^\pi(s) - \hat{V}(s,w))\nabla_w \hat{V}(s,w)] \end{aligned}$$

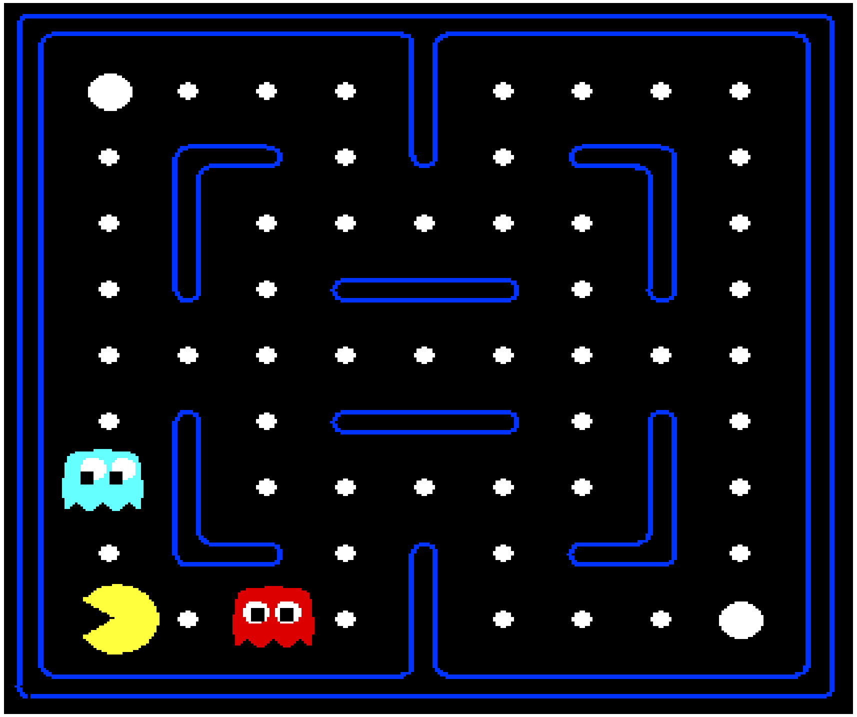
Linear value function approximation

Let's approximate $V^\pi(s)$ as a linear function of features:

$$\hat{V}(s,w) = x(s)^T w = \sum_{j=1}^{n} x_j(s) w_j$$

where x(s) is the feature vector: $x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$

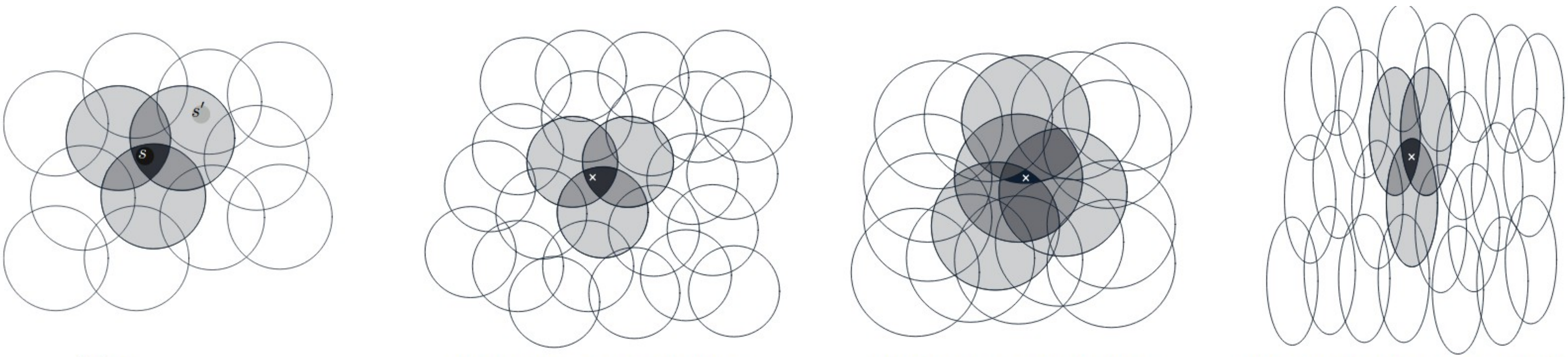
Think-pair-share



Can you think of some good features for pacman?

Linear value function approx: coarse coding

For example, the elts in x(s) could correspond to regions of state space:



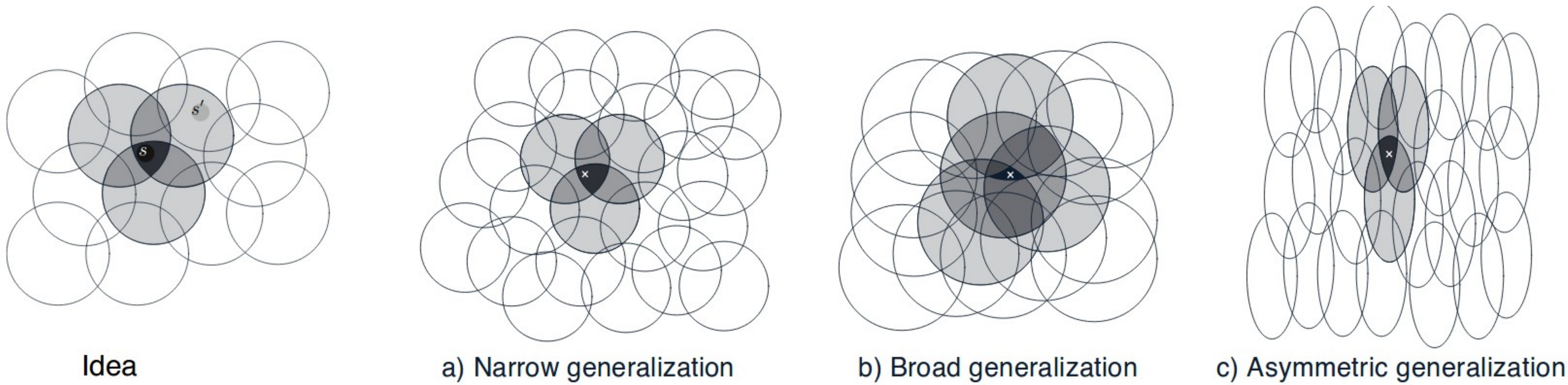
Idea　　a) Narrow generalization　　b) Broad generalization　　c) Asymmetric generalization

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

Binary features
– one feature for each circle (above)

Linear value function approx: coarse coding

For example, the elts in x(s) could correspond to regions of state space:



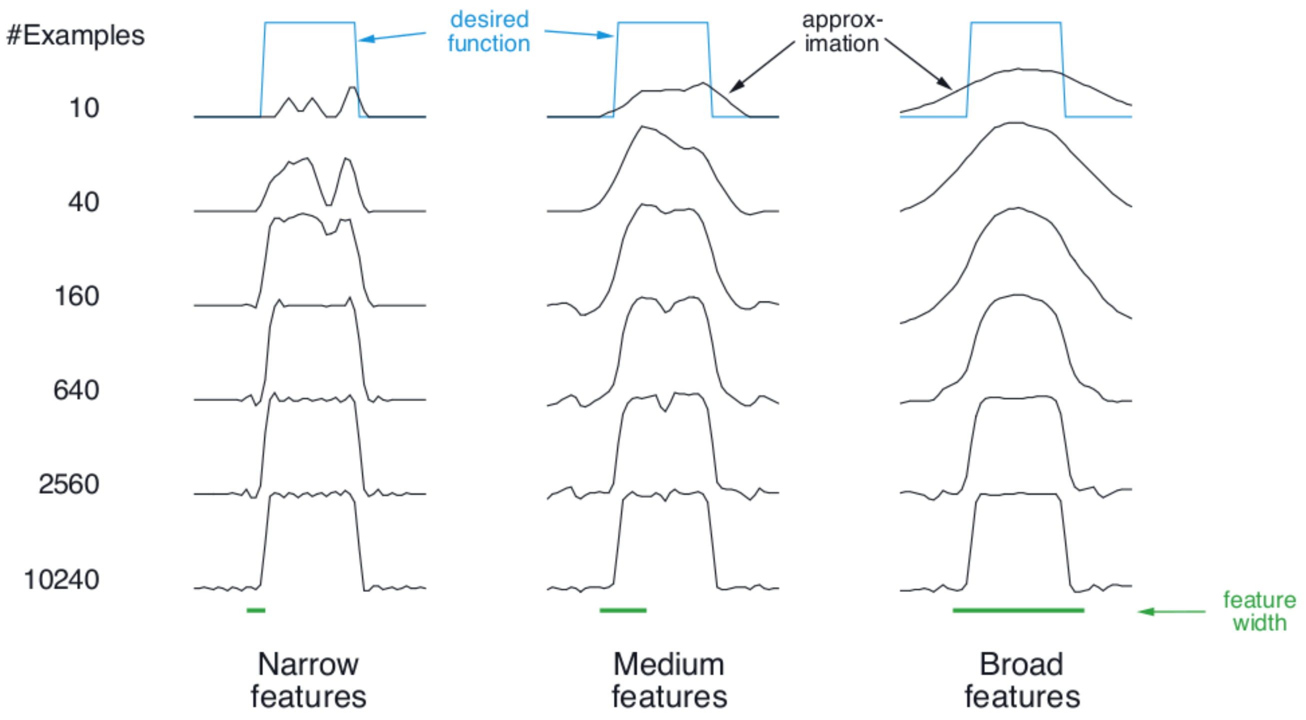
$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

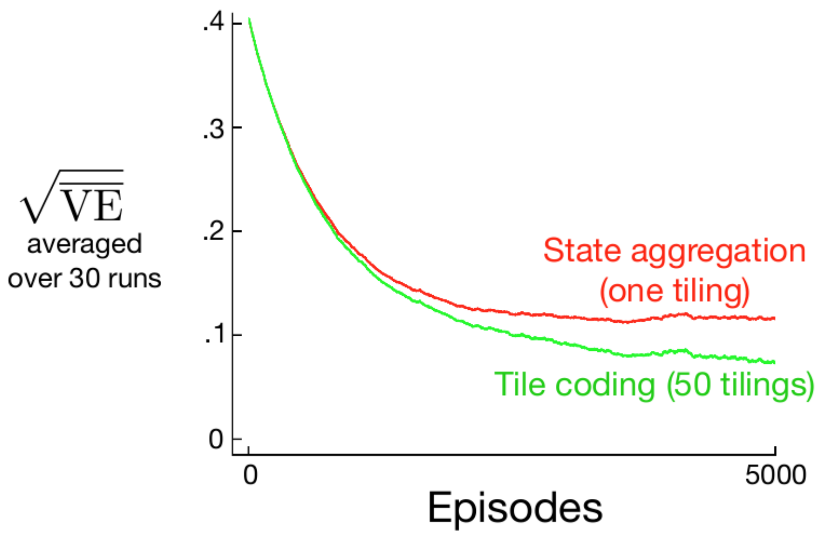
Binary features
– one feature for each circle (above)

The value function is encoded by the combination of all tiles that a state intersects

$$\hat{V}(s, w) = x(s)^T w = \sum_{j=1}^{n} x_j(s) w_j$$

The effect of overlapping feature regions

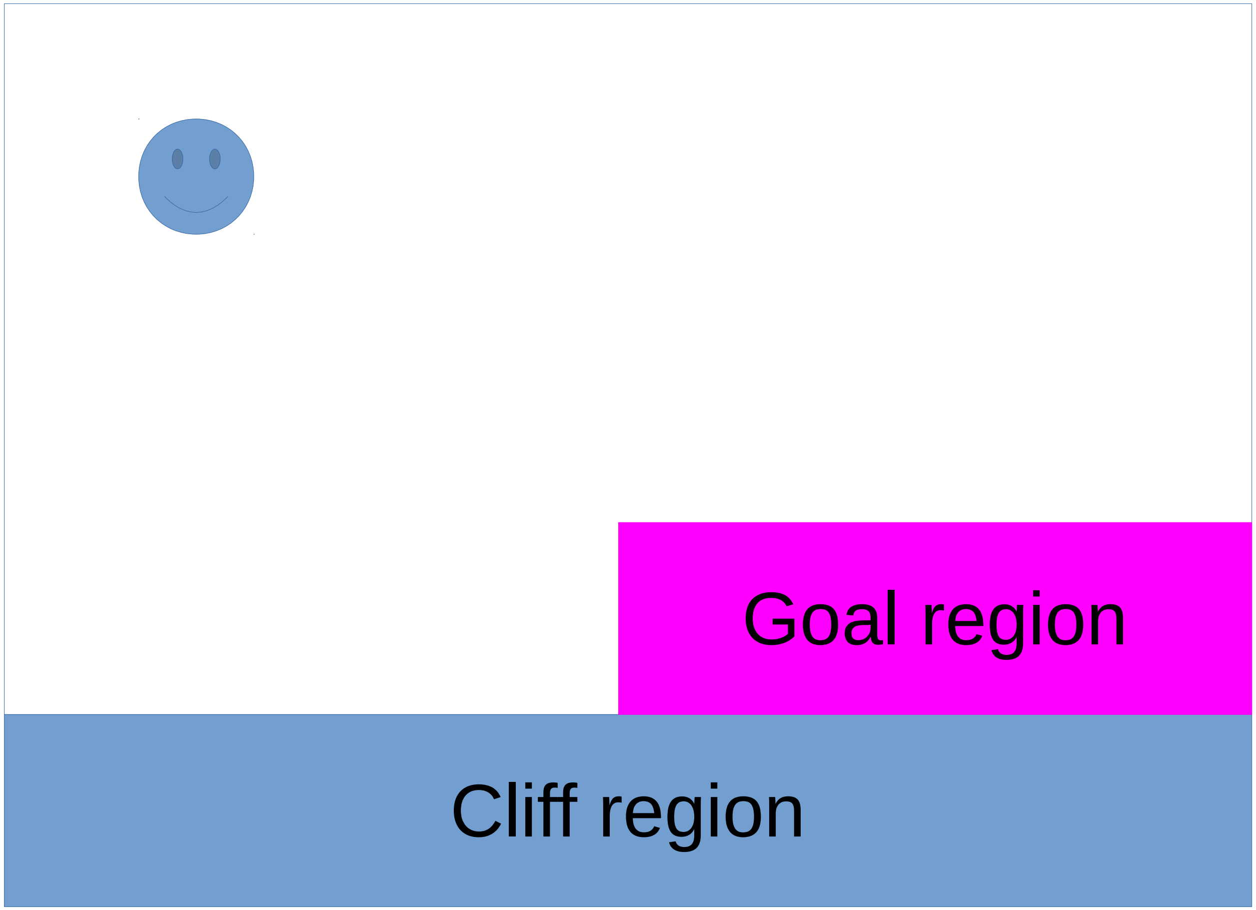




Think-pair-share

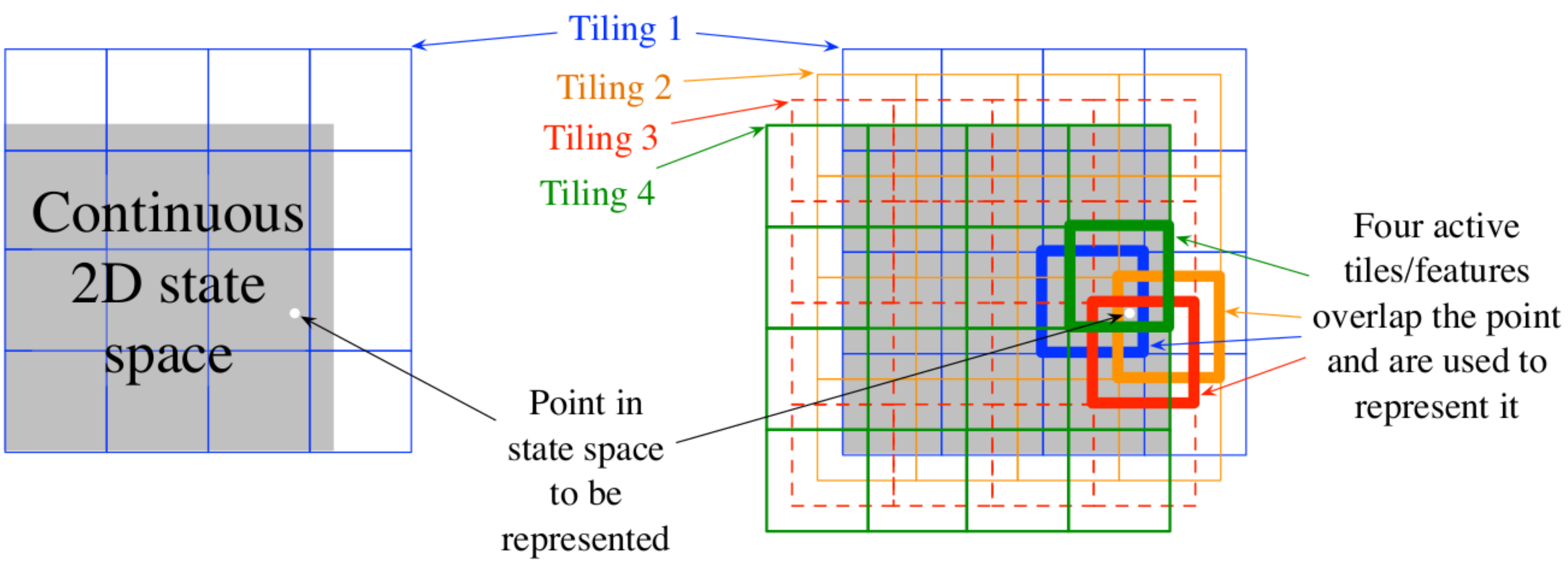
What type of linear features might be appropriate for this problem?

What is the relationship between feature shape and generalization?



Linear value function approx: tile coding

For example, x(s) could be constructed using *tile coding*:



– Each *tiling* is a partition of the state space.
– Assigns each state to a unique *tile*.

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

Binary features
n = num tiles x num tilings
In this example: n = 16 x 4

Think-pair-share

Binary features
n = num tiles x num tilings
In this example: n = 16 x 4

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

The value function is encoded by the combination of all tiles that a state intersects

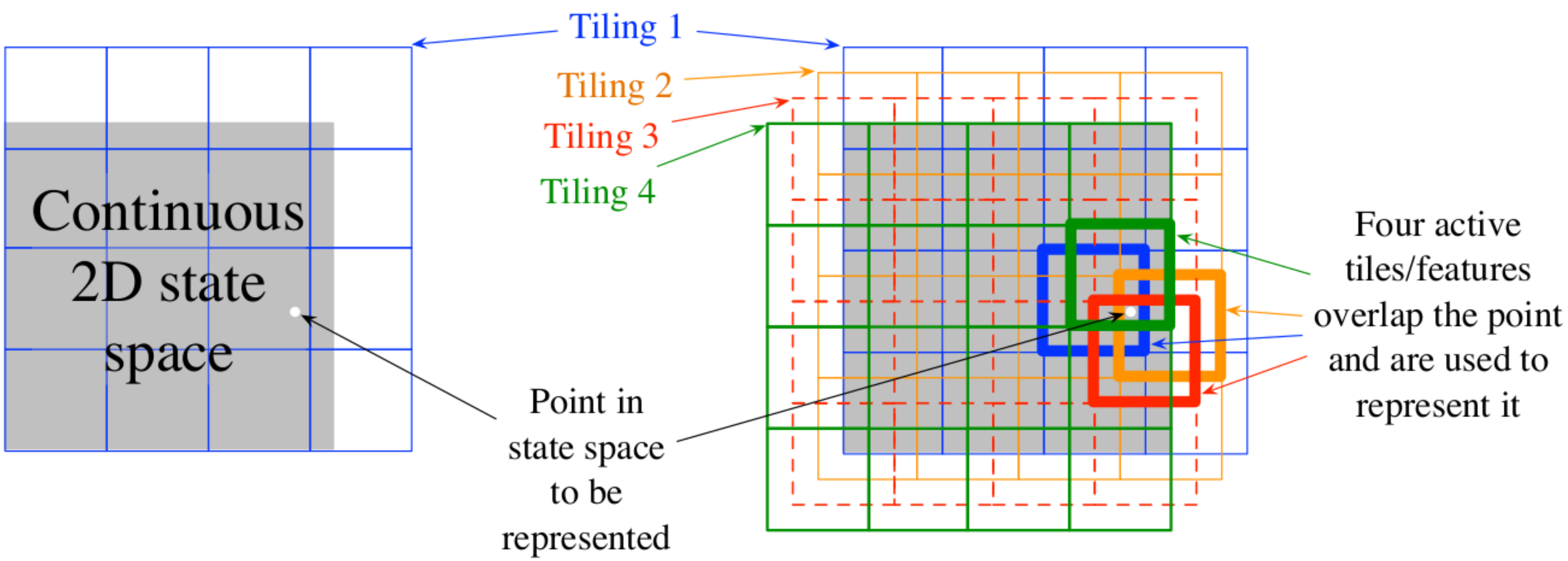
$$\hat{V}(s, w) = x(s)^T w = \sum_{j=1}^{n} x_j(s) w_j$$

State aggregation is a special case of tile coding.

How many tilings in this case?

What do the weights correspond to in this case?

Think-pair-share



$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

Binary features
n = num tiles x num tilings
In this example: n = 16 x 4

– what are the pros/cons of rectangular tiles like this?
– what are the pros/cons to evenly spacing the tilings vs placing them at uneven offsets?

Recall monte carlo policy evaluation algorithm

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

Input: a policy π to be evaluated

Initialize:

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

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

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

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

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

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Let’s think about how to do the same thing using function approximation...

Gradient monte carlo policy evaluation

<u>Goal</u>: calculate $\Delta w = \alpha \mathbb{E}_\pi [(V^\pi(s) - \hat{V}(s,w)) \nabla_w \hat{V}(s,w)]$

Notice that in MC, the return G_t is an unbiased, noisy sample of the true value, $V^\pi(s_t)$

Can therefore apply supervised learning to "training data":

$$\langle s_1, G_1 \rangle, \langle s_2, G_2 \rangle, \ldots, \langle s_T, G_T \rangle$$

The weight update "sampled" from the training data is:

$$\Delta w = \alpha (G_t - \hat{V}(s,w)) \nabla_w \hat{V}(s,w)$$

Gradient monte carlo policy evaluation

Goal: calculate $\Delta w = \alpha \mathbb{E}_\pi [(V^\pi(s) - \hat{V}(s,w))\nabla_w \hat{V}(s,w)]$

Notice that in MC, the return G_t is an unbiased, noisy sample of the true value, $V^\pi(s_t)$

Can therefore apply supervised learning to "training data":

$$\langle s_1, G_1 \rangle, \langle s_2, G_2 \rangle, \ldots, \langle s_T, G_T \rangle$$

The weight update "sampled" from the training data is:

$$\Delta w = \alpha(G_t - \hat{V}(s,w))\nabla_w \hat{V}(s,w)$$

For a linear function approximator, this is:

$$\Delta w = \alpha(G_t - \hat{V}(s,w))x(s)$$

Gradient monte carlo policy evaluation

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

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

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

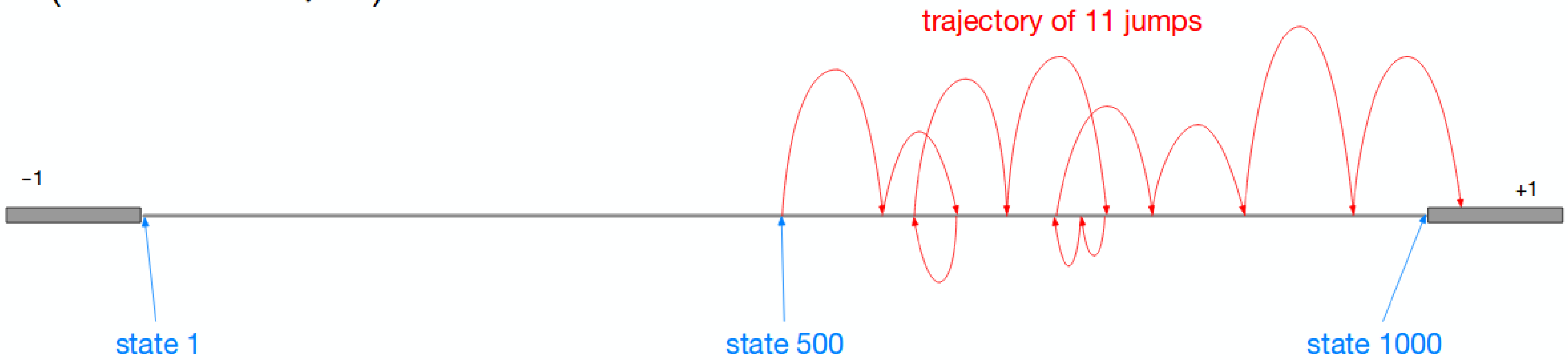
For linear function approximation, gradient MC converges to the weights that minimize MSE wrt the true value function.

Even for non-linear function approximation, gradient MC converges to a local optimum.

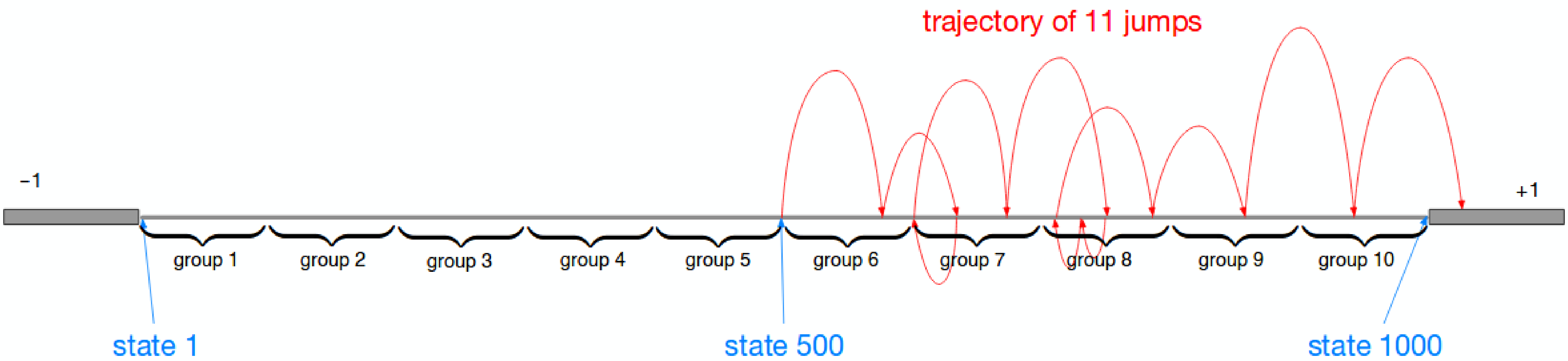
However, since this is MC, the estimates are high-variance.

Gradient MC example: 1000-state random walk

- States are numbered 1 to 1000
- Walks start in the near middle, at state 500 $\quad S_0 = 500$
- At each step, *jump* to one of the 100 states to the right, or to one of the 100 states to the left $\quad S_1 \in \{400..499\} \cup \{501..600\}$
- If the jump goes beyond 1 or 1000, terminates with a reward of −1 or +1 (otherwise R_t=0)

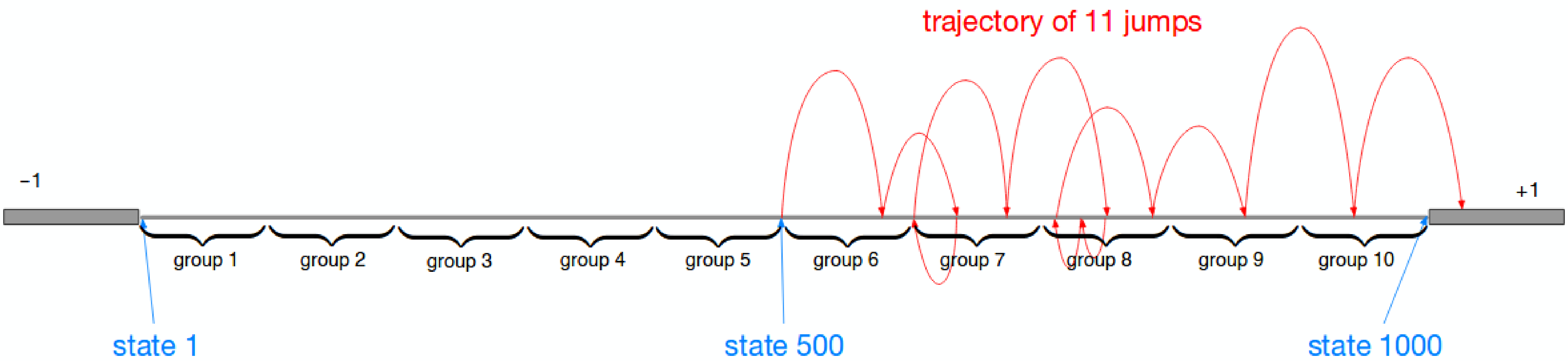


Gradient MC example: 1000-state random walk



The whole value function over 1000 states
will be approximated with 10 numbers!

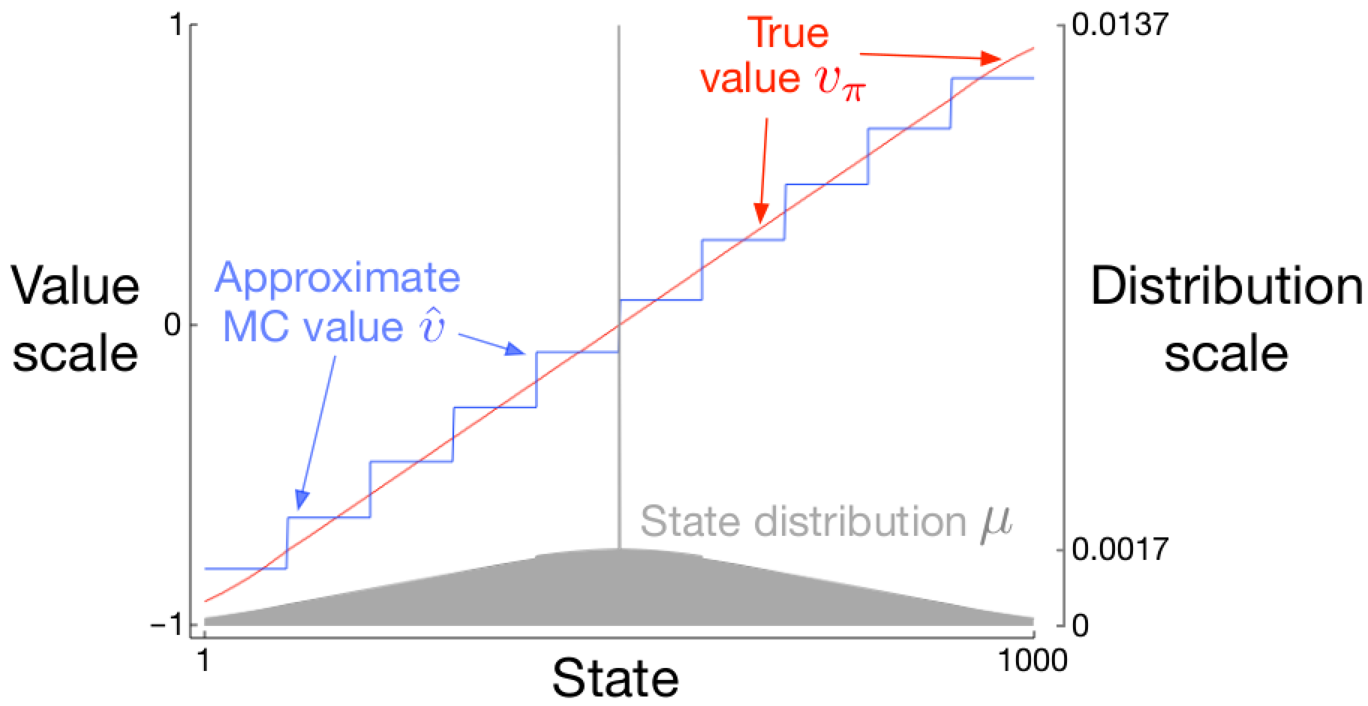
Question



The whole value function over 1000 states will be approximated with 10 numbers!

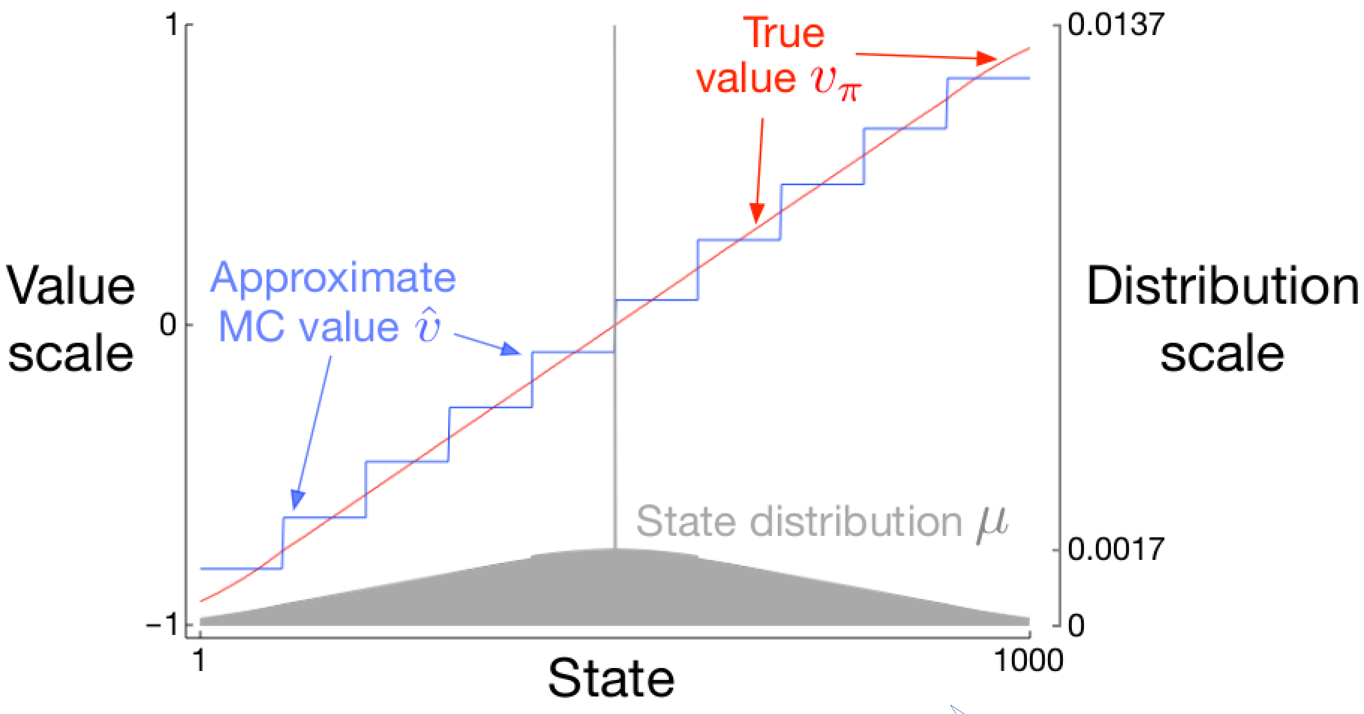
How many tilings are here?

Gradient MC example: 1000-state random walk



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

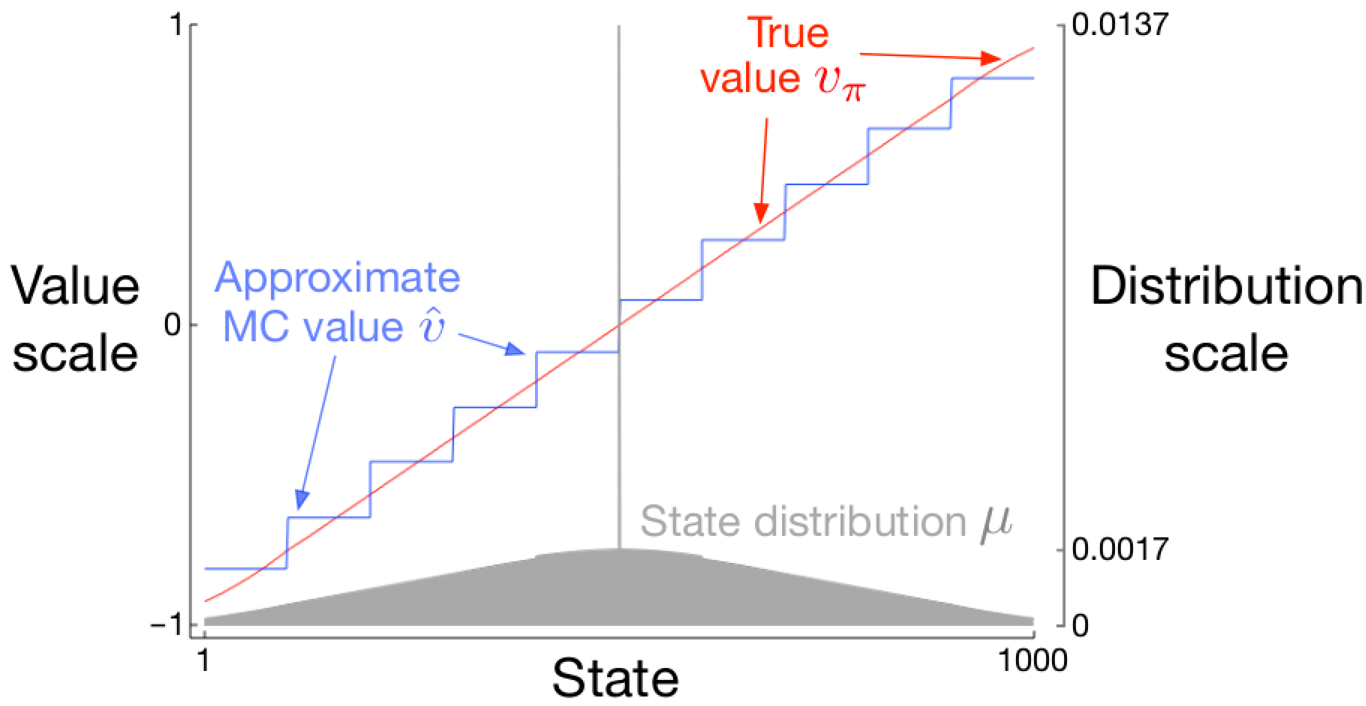
Gradient MC example: 1000-state random walk



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

Converges to unbiased value estimate

Question



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

What is the relationship between the state distribution (mu) and the policy?

How do you correct for following a policy that visits states differently?

TD Learning with value function approximation

The TD target, $R_{t+1} + \gamma \hat{V}(s_{t+1}, w)$ is an estimate of the true value, $V^\pi(s_t)$

But, let's ignore that and use the TD target anyway…

Training data:

$$\langle s_1, R_2 + \gamma \hat{V}(s_2, w) \rangle, \langle s_2, R_3 + \gamma \hat{V}(s_3, w) \rangle, \ldots, \langle s_{T-1}, R_T \rangle$$

TD Learning with value function approximation

The TD target, $R_{t+1} + \gamma \hat{V}(s_{t+1}, w)$ is an estimate of the true value, $V^{\pi}(s_t)$

But, let's ignore that and use the TD target anyway…

Training data:

$$\langle s_1, R_2 + \gamma \hat{V}(s_2, w) \rangle, \langle s_2, R_3 + \gamma \hat{V}(s_3, w) \rangle, \ldots, \langle s_{T-1}, R_T \rangle$$

This gives us TD(0) policy evaluation with:

$$\Delta w = \alpha(R + \gamma \hat{V}(s', w) - \hat{V}(s, w)) \nabla_w \hat{V}(s, w)$$

TD Learning with value function approximation

The TD target, $R_{t+1} + \gamma \hat{V}(s_{t+1}, w)$ is an estimate of the true value, $V^{\pi}(s_t)$

But, let's ignore that and use the TD target anyway…

Training data:

$$\langle s_1, R_2 + \gamma \hat{V}(s_2, w)\rangle, \langle s_2, R_3 + \gamma \hat{V}(s_3, w)\rangle, \ldots, \langle s_{T-1}, R_T\rangle$$

This gives us TD(0) policy evaluation with:

$$\Delta w = \alpha(R + \gamma \hat{V}(s', w) - \hat{V}(s, w))\nabla_w \hat{V}(s, w)$$

Next state

TD Learning with value function approximation

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

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

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

Think-pair-share

Why is this called “semi-gradient”?

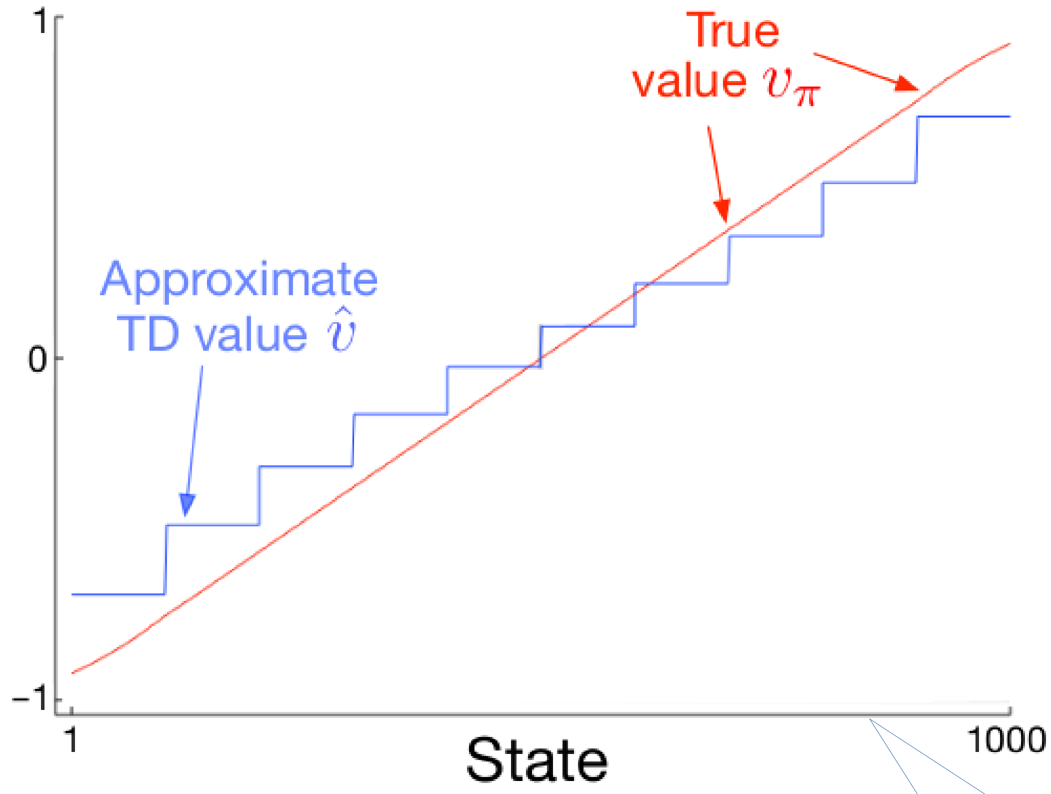
Here’s the update rule we’re using: $$\Delta w = \alpha(R + \gamma \hat{V}(s', w) - \hat{V}(s, w))\nabla_w \hat{V}(s, w)$$

Is this really the gradient?

What is the gradient actually?

Loss function: $$\begin{aligned} J(w) &= \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - \hat{V}(s, w))^2] \\ &= \frac{1}{2}(R + \gamma \hat{V}(s', w) - \hat{V}(s, w))^2 \end{aligned}$$

Semi-gradient TD(0) ex: 1000-state random walk



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

- state distribution affects accuracy

Converges to **biased** value estimate

Convergence results summary

1. Gradient-MC converges for both linear and non-linear fn approx
2. Gradient-MC converges to optimal value estimates
   – converges to values that min MSE

3. Semi-gradient-TD(0) converges for linear fn approx
4. Semi-gradient-TD(0) converges to a biased estimate
   – converges to a point, w_{TD}, that does does not minimize MSE
   – but we have:

$$J(w_{TD}) \leq \frac{1}{1-\gamma} \underbrace{\min_w J(w)}$$

Fixed point for semi-gradient TD

Point that min MSE

TD Learning with value function approximation

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

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

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

For linear function approximation, gradient TD(0) converges to biased estimate of weights such that:

$$J(w_{TD}) \leq \frac{1}{1-\gamma} \underbrace{\min_w J(w)}$$

Fixed point for semi-gradient TD

Point that min MSE

Think-pair-share

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

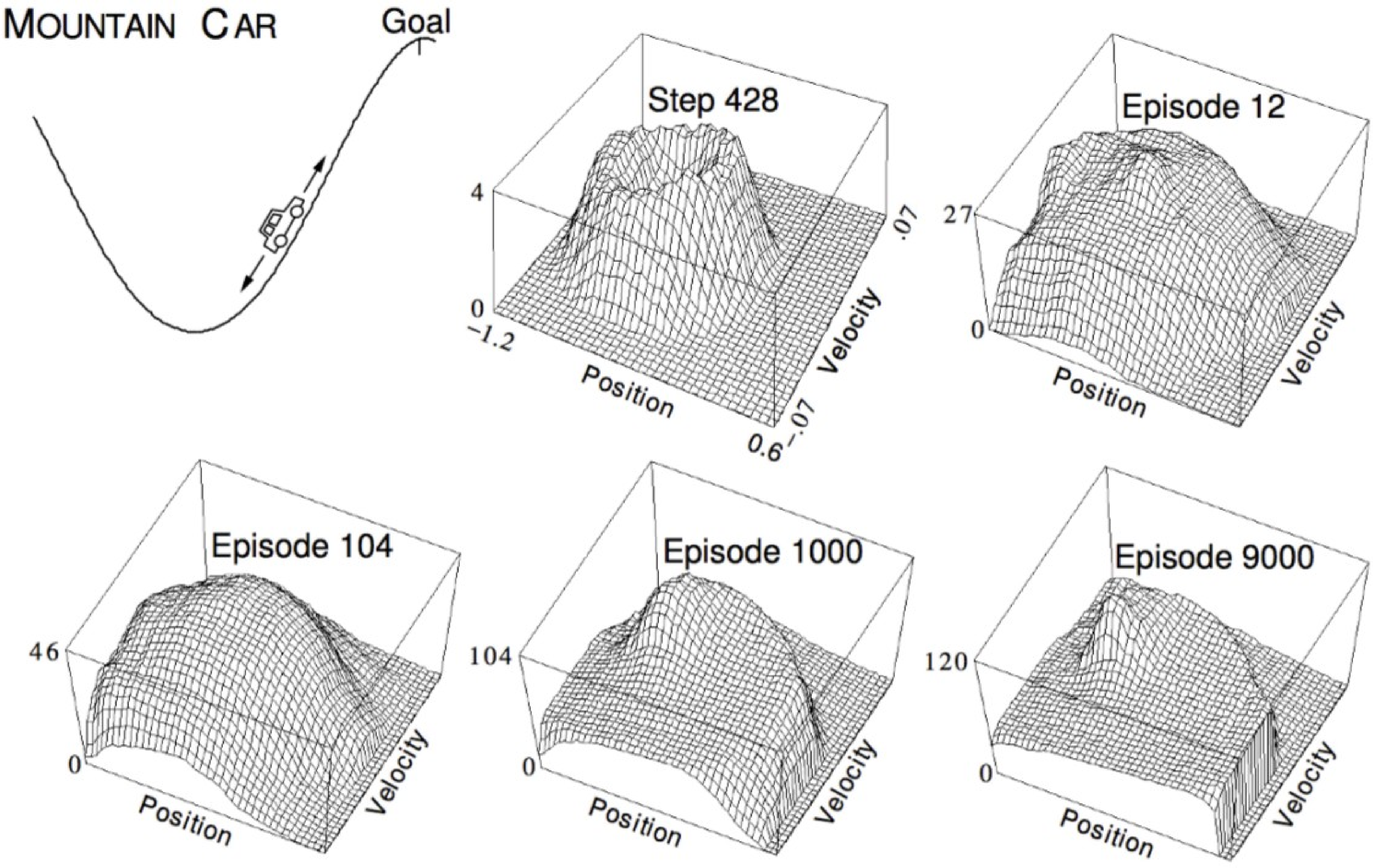
Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g.,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

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

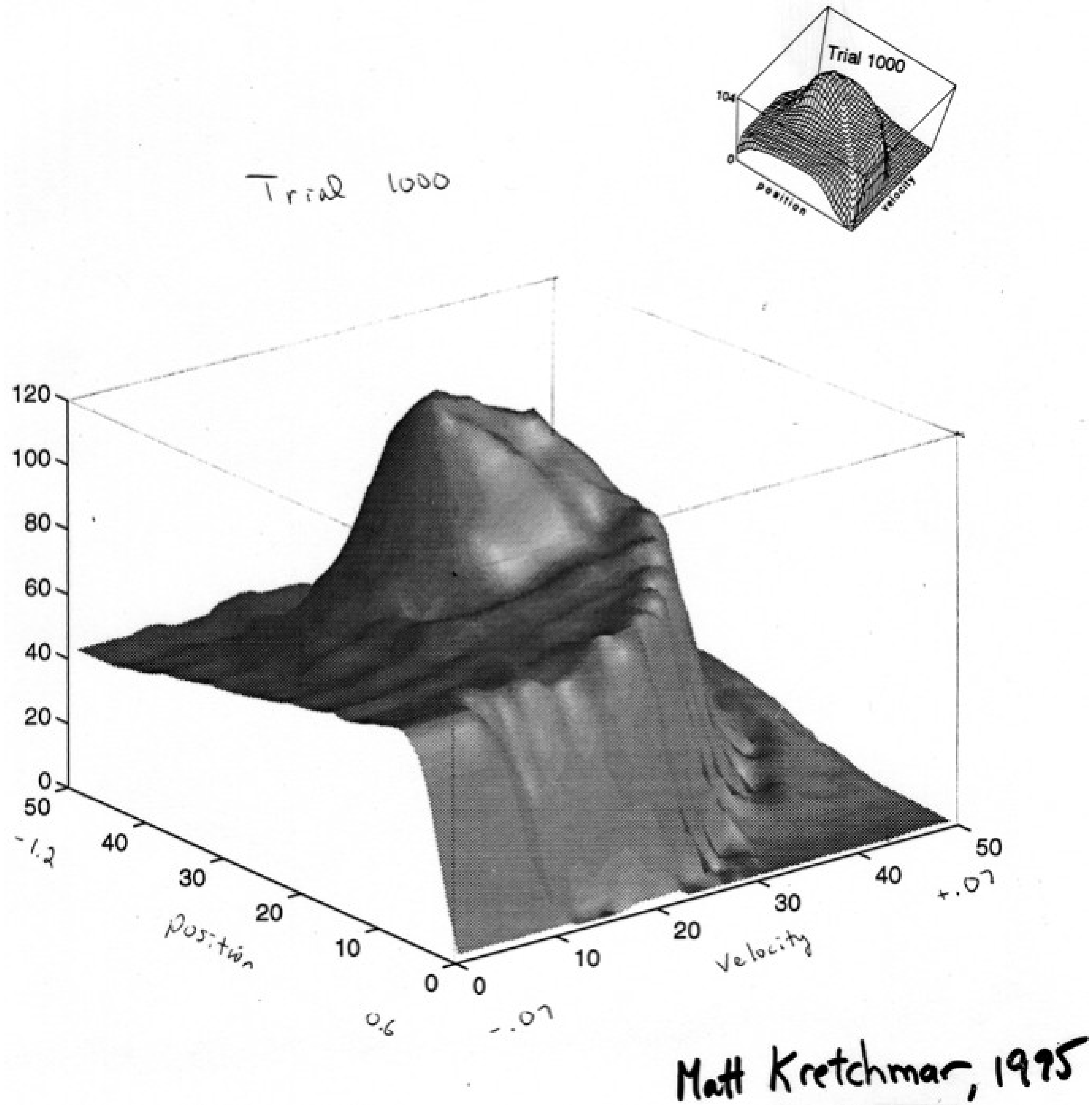
Write the semi-gradient weight update equation for the special case of linear function approximation.

How would you update this algorithm for q-learning?

Linear Sarsa with Coarse Coding in Mountain Car



Linear Sarsa with Coarse Coding in Mountain Car



Least Squares Policy Iteration (LSPI)

Recall that for linear function approximation, J(w) is quadratic in the weights:

$$
\begin{aligned}
J(w) &= \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - \hat{V}(s,w))^2] \\
&= \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - x(s)^T w)^2]
\end{aligned}
$$

We can solve for *w* that min *J(w)* directly.

First, let's think about this in the context of batch policy evaluation.

Policy evaluation

Given:

– a dataset $\mathcal{D} = \{(s_1, G_1), \dots, (s_n, G_n)\}$ generated using policy π

Find *w* that min:
$$\begin{aligned} J(w) &= \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - x(s)^T w)^2] \\ &\approx \frac{1}{2|\mathcal{D}|}\sum_{(s,G)\in\mathcal{D}}[(G - x(s)^T w)^2] \end{aligned}$$

Question

Given:

– a dataset $\mathcal{D} = \{(s_1, G_1), \dots, (s_n, G_n)\}$ generated using policy π

Find *w* that min:
$$
\begin{aligned}
J(w) &= \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - x(s)^T w)^2] \\
&\approx \frac{1}{2|\mathcal{D}|}\sum_{(s,G)\in\mathcal{D}}[(G - x(s)^T w)^2]
\end{aligned}
$$

HOW?

Think-pair-share

Given: a dataset $\mathcal{D} = \{(a_1, b_1), \ldots, (a_n, b_n)\}$

Find *w* that min: $J(w) = \frac{1}{2} \sum_{(a,b) \in \mathcal{D}} (a - bw)^2$

where *a, b, w* are scalars.

What if *b* is a vector?

Policy evaluation

Given:

– a dataset $\mathcal{D} = \{(s_1, G_1), \ldots, (s_n, G_n)\}$ generated using policy π

Find *w* that min:

$$\begin{aligned} J(w) &= \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - x(s)^T w)^2] \\ &\approx \frac{1}{2|\mathcal{D}|} \sum_{(s,G)\in\mathcal{D}} [(G - x(s)^T w)^2] \end{aligned}$$

1. Set derivative to zero:

$$\nabla_w J(w) = -\frac{1}{|\mathcal{D}|} \sum_{(s,G)\in\mathcal{D}} x(s)[G - x(s)^T w] = 0$$

Policy evaluation

Given:

– a dataset $\mathcal{D} = \{(s_1, G_1), \dots, (s_n, G_n)\}$ generated using policy π

Find *w* that min: $$J(w) = \frac{1}{2}\mathbb{E}_\pi[(V^\pi(s) - x(s)^T w)^2] \approx \frac{1}{2|\mathcal{D}|}\sum_{(s,G)\in\mathcal{D}} [(G - x(s)^T w)^2]$$

1. Set derivative to zero:

$$\nabla_w J(w) = -\frac{1}{|\mathcal{D}|}\sum_{(s,G)\in\mathcal{D}} x(s)[G - x(s)^T w] = 0$$

2. Solve for *w*:

$$w = \left(\sum_{(s,G)\in\mathcal{D}} x(s)x(s)^T\right)^{-1} \sum_{(s,G)\in\mathcal{D}} x(s)G$$

LSMC policy evaluation

1. collect a bunch of experience $\mathcal{D} = \{(s_1, G_1), \ldots, (s_n, G_n)\}$ under policy π

2. calculate weights using:

$$w = \left(\sum_{(s,G)\in\mathcal{D}} x(s)x(s)^T \right)^{-1} \sum_{(s,G)\in\mathcal{D}} x(s)G$$

LSMC policy evaluation

1. collect a bunch of experience $\mathcal{D} = \{(s_1, G_1), \ldots, (s_n, G_n)\}$ under policy π

2. calculate weights using:

$$w = \underbrace{\left(\sum_{(s,G)\in\mathcal{D}} x(s)x(s)^T \right)}^{-1} \sum_{(s,G)\in\mathcal{D}} x(s)G$$

How to we ensure this matrix is well conditioned?

Question

1. collect a bunch of experience $\mathcal{D} = \{(s_1, G_1), \ldots, (s_n, G_n)\}$ under policy π

2. calculate weights using:

$$w = \left(\sum_{(s,G)\in\mathcal{D}} x(s)x(s)^T + \epsilon I \right)^{-1} \sum_{(s,G)\in\mathcal{D}} x(s)G$$

What effect does this term have?

What cost function is being minimized now?

LSMC policy iteration

1. Take an action according current policy, π_w

2. Add experience to buffer: $\mathcal{D} = \{(s_1, G_1), \ldots, (s_n, G_n)\}$

3. Calculate new LS weights using:

$$w = \left(\sum_{(s,G) \in \mathcal{D}} x(s)x(s)^T + \epsilon I \right)^{-1} \sum_{(s,G) \in \mathcal{D}} x(s)G$$

4. Goto step 1

Is there a TD version of this?

1. Take an action according current policy, π_w

2. Add experience to buffer: $\mathcal{D} = \{(s_1, G_1), \ldots, (s_n, G_n)\}$

3. Calculate new LS weights using:

$$w = \left(\sum_{(s,G)\in\mathcal{D}} x(s)x(s)^T + \epsilon I \right)^{-1} \sum_{(s,G)\in\mathcal{D}} x(s)G$$

4. Goto step 1

MC target

LSTD policy evaluation

In TD learning, the target is: $G = r + \gamma x(s')^T w$

Substituting into the gradient of *J(w)*:

$$-\frac{1}{|\mathcal{D}|} \sum_{(s,G)\in\mathcal{D}} x(s)[r + \gamma x(s')^T w - x(s)^T w] = 0$$

Solving for *w*:

$$w = \left(\sum_{(s,G)\in\mathcal{D}} x(s)(x(s)^T - \gamma x(s')^T) \right)^{-1} \sum_{(s,G)\in\mathcal{D}} x(s) r$$

LSTD policy evaluation

In TD learning, the target is: $G = r + \gamma x(s')^T w$

Substituting into the gradient of *J(w)*:

$$-\frac{1}{|\mathcal{D}|} \sum_{(s,G)\in\mathcal{D}} x(s)[r + \gamma x(s')^T w - x(s)^T w] = 0$$

Solving for *w* (and add regularization term):

$$w = \left(\sum_{(s,G)\in\mathcal{D}} x(s)(x(s)^T - \gamma x(s')^T) + \epsilon I \right)^{-1} \sum_{(s,G)\in\mathcal{D}} x(s) r$$

LSTD policy evaluation

In TD learning, the target is: $G = r + \gamma x(s')^T w$

Substituting into the gradient of *J(w)*:

$$-\frac{1}{|\mathcal{D}|} \sum_{(s,G)\in\mathcal{D}} x(s)[r + \gamma x(s')^T w - x(s)^T w] = 0$$

Solving for *w* (and add regularization term):

$$w = \left(\sum_{(s,G)\in\mathcal{D}} x(s)(x(s)^T - \gamma x(s')^T) + \epsilon I \right)^{-1} \sum_{(s,G)\in\mathcal{D}} x(s) r$$

Notice this is slightly different from what was used for LSMC

LSTD policy evaluation

1. collect a bunch of experience $\mathcal{D} = \{(s_1, s'_1, r_1), \ldots, (s_n, s'_n, r_n)\}$ under policy π

2. calculate weights using:

$$w = \left(\sum_{(s,G) \in \mathcal{D}} x(s)(x(s)^T - \gamma x(s')^T) + \epsilon I \right)^{-1} \sum_{(s,G) \in \mathcal{D}} x(s) r$$

LSTDQ

Approximate Q function as: $\hat{Q}(s, a, w) = x(s, a)^T w$

Now, the update is:

$$w = \left(\sum_{(s,G) \in \mathcal{D}} x(s, a)(x(s, a)^T - \gamma x(s', \pi(s'))^T) + \epsilon I \right)^{-1} \sum_{(s,G) \in \mathcal{D}} x(s, a) r$$

LSPI-TD

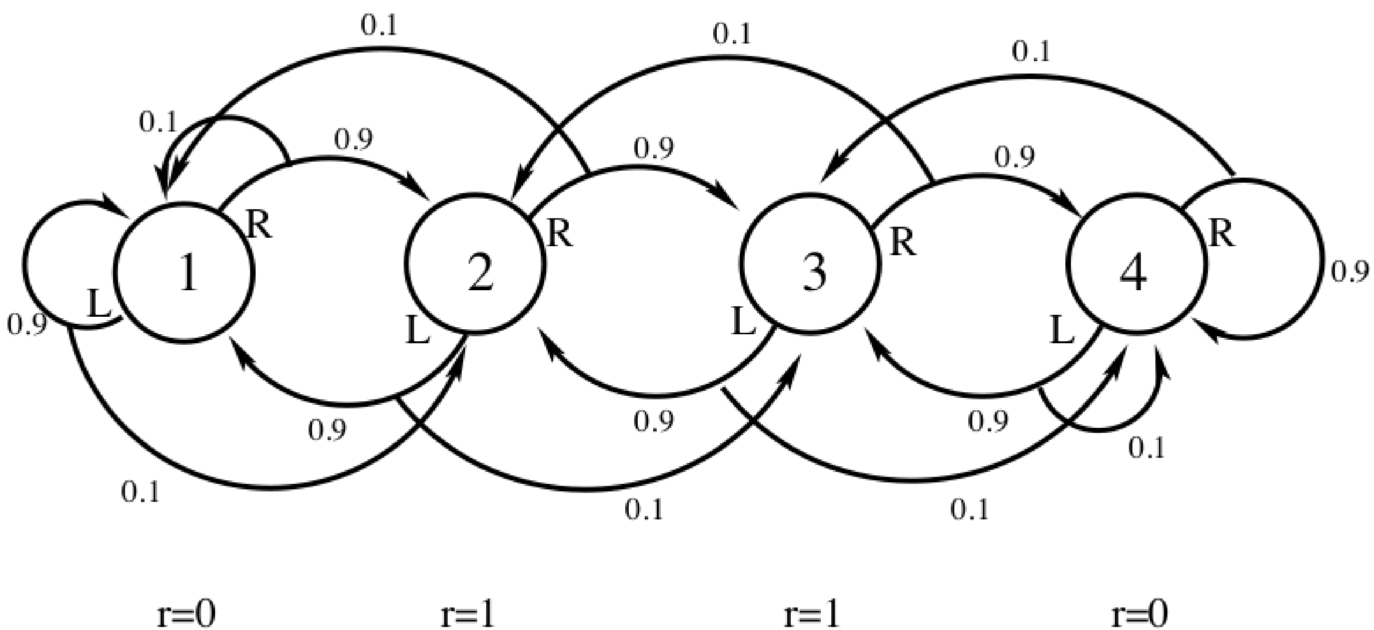
function **LSPI-TD**$(\mathcal{D}, \pi_0)$
  $\pi' \leftarrow \pi_0$
  repeat
    $\pi \leftarrow \pi'$
    $Q \leftarrow$ **LSTDQ**$(\pi, \mathcal{D})$
    for all $s \in \mathcal{S}$ **do**
      $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$
    end for
  until $(\pi \approx \pi')$
  return π
end function

$$w = \left(\sum_{(s,G) \in \mathcal{D}} x(s,a)(x(s,a)^T - \gamma x(s', \pi(s'))^T) + \epsilon I \right)^{-1} \sum_{(s,G) \in \mathcal{D}} x(s,a) r$$

Policy improvement

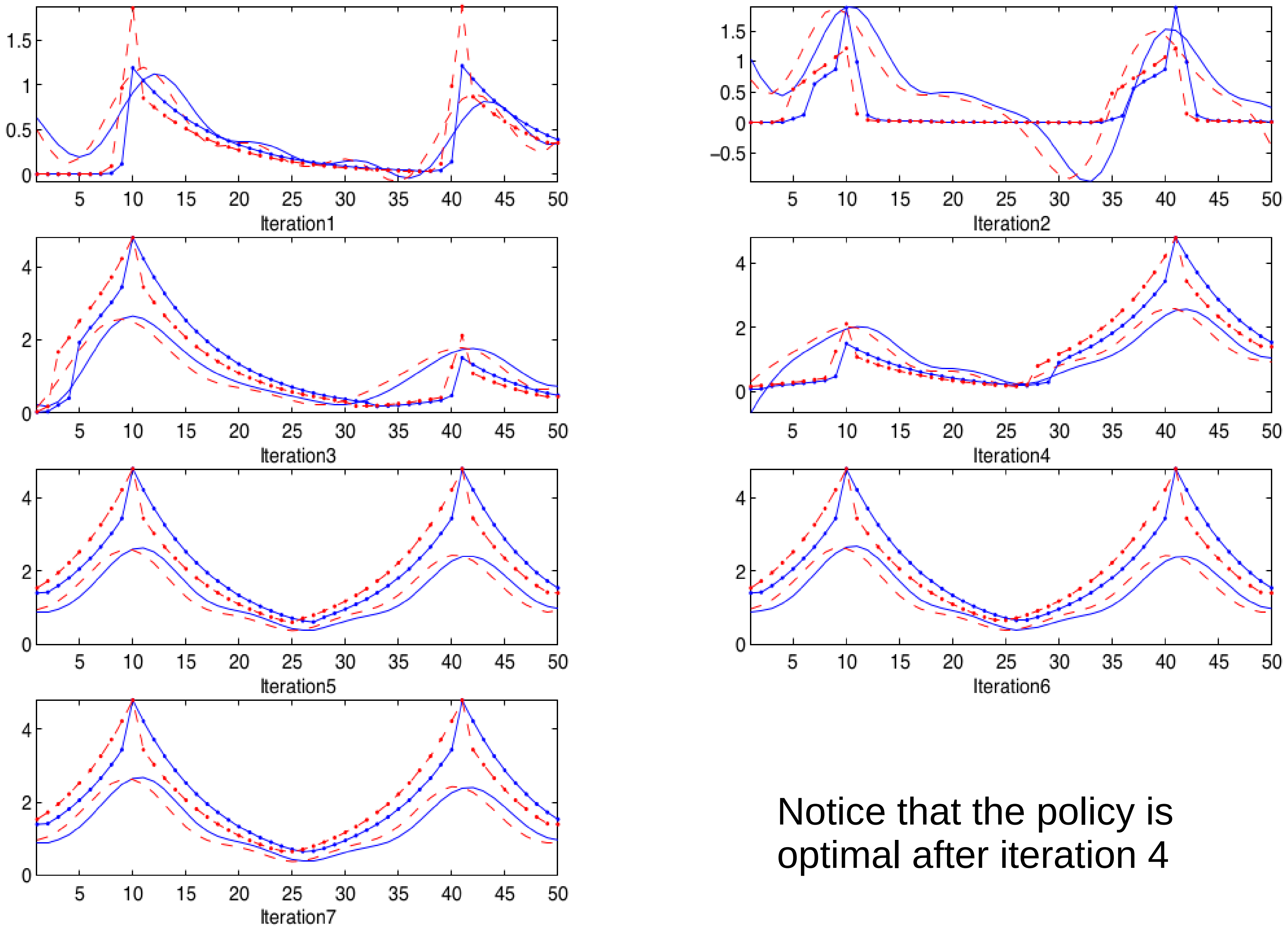
Guaranteed to converge to near-optimal (linear fn approx)

Chain Walk Example



- Consider the 50 state version of this problem
- Reward +1 in states 10 and 41, 0 elsewhere
- Optimal policy: R (1-9), L (10-25), R (26-41), L (42, 50)
- Features: 10 evenly spaced Gaussians ($\sigma = 4$) for each action
- Experience: 10,000 steps from random walk policy

LSPI in Chain Walk: Action-Value Function



Notice that the policy is optimal after iteration 4