Lecture 7: Value Function Approximation

Joseph Modayil

Outline

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve *large* problems, e.g.

- Backgammon: 10^{20} states
- Computer Go: 10^{170} states
- Helicopter/Mountain Car: continuous state space
- Robots: informal state space (physical universe)

How can we scale up the model-free methods for *prediction* and *control* from the last two lectures?

Value Function Approximation

- So far we have represented value function by a *lookup table*
  - Every state s has an entry $V(s)$
  - Or every state-action pair s, a has an entry $Q(s, a)$
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with *function approximation*

$$V_\theta(s) \approx v^\pi(s)$$
$$\text{or } Q_\theta(s, a) \approx q^\pi(s, a)$$

  - *Generalise* from seen states to unseen states
  - *Update* parameter θ using MC or TD learning

Which Function Approximator?

There are many function approximators, e.g.

- Artificial neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- Coarse coding

In principle, *any* function approximator can be used. However, the choice may be affected by some properties of RL:

- Experience is not i.i.d. - successive time-steps are correlated
- During control, value function $v^{\pi}(s)$ is *non-stationary*
- Agent's actions affect the subsequent data it receives
- Feedback is delayed, not instantaneous

Classes of Function Approximation

- Tabular (No FA): a table with an entry for each MDP state
- State Aggregation: Partition environment states
- Linear function approximation: fixed features (or fixed kernel)
- Differentiable (nonlinear) function approximation: neural nets

So what should you choose? Depends on your goals.

- Top: good theory but weak performance
- ⋮
- Bottom: excellent performance but weak theory
- Linear function approximation is a useful middle ground
- Neural nets now commonly give the highest performance

Gradient Descent

- Let $J(\theta)$ be a differentiable function of parameter vector θ
- Define the *gradient* of $J(\theta)$ to be

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

- To find a local minimum of $J(\theta)$
- Adjust the parameter θ in the direction of -ve gradient

$$\Delta\theta = -\frac{1}{2}\alpha\nabla_\theta J(\theta)$$

where α is a step-size parameter

Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector θ minimising mean-squared error between approximate value fn $V_\theta(s)$ and true value fn $v^\pi(s)$

$$J(\theta) = \mathbb{E}_\pi \left[(v^\pi(S) - V_\theta(S))^2 \right]$$

  Note: The notation $\mathbb{E}_\pi[]$ means that the random variable S is drawn from a distribution induced by π. $\mathbb{E}_\pi[f(S)] = \sum_s f(s) d_\pi(s)$

- Gradient descent finds a local minimum

$$\Delta\theta = -\frac{1}{2}\alpha \nabla_\theta J(\theta) = \alpha \mathbb{E}_\pi \left[(v^\pi(S) - V_\theta(S)) \nabla_\theta V_\theta(S) \right]$$

- Stochastic gradient descent *samples* the gradient

$$\Delta\theta = \alpha (v^\pi(s) - V_\theta(s)) \nabla_\theta V_\theta(s)$$

- Expected update is equal to full gradient update

Feature Vectors

- Represent state by a *feature vector*

$$\phi(s) = \begin{pmatrix} \phi_1(s) \\ \vdots \\ \phi_n(s) \end{pmatrix}$$

- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess

Linear Value Function Approximation

- Approximate value function by a linear combination of features

$$V_\theta(s) = \phi(s)^\top \theta = \sum_{j=1}^{n} \phi_j(s)\theta_j$$

- Objective function is quadratic in parameters θ

$$J(\theta) = \mathbb{E}_\pi \left[(v^\pi(S) - \phi(S)^\top \theta)^2 \right]$$

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

$$\nabla_\theta V_\theta(s) = \phi(s)$$
$$\Delta\theta = \alpha(v^\pi(s) - V_\theta(s))\phi(s)$$

Update $=$ *step-size* $\times$ *prediction error* $\times$ *feature vector*

Table Lookup Features

- Table lookup can be implemented as a special case of linear value function approximation
- Let the n states be given by $\mathcal{S} = \{s^{(1)}, \ldots, s^{(n)}\}$.
- Using *table lookup features*

$$\phi^{table}(s) = \begin{pmatrix} \mathbf{1}(s = s^{(1)}) \\ \vdots \\ \mathbf{1}(s = s^{(n)}) \end{pmatrix}$$

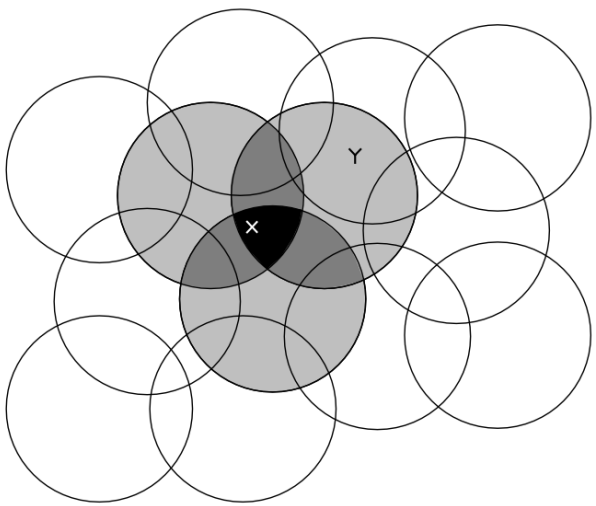
- Parameter vector θ gives value of each individual state

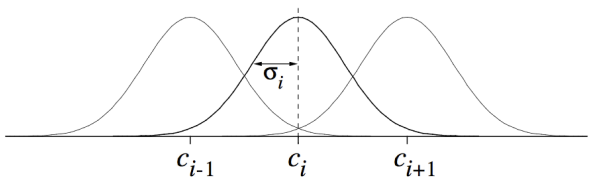
$$V(s) = \begin{pmatrix} \mathbf{1}(s = s^{(1)}) \\ \vdots \\ \mathbf{1}(s = s^{(n)}) \end{pmatrix} \cdot \begin{pmatrix} \theta_1 \\ \vdots \\ \theta_n \end{pmatrix}$$

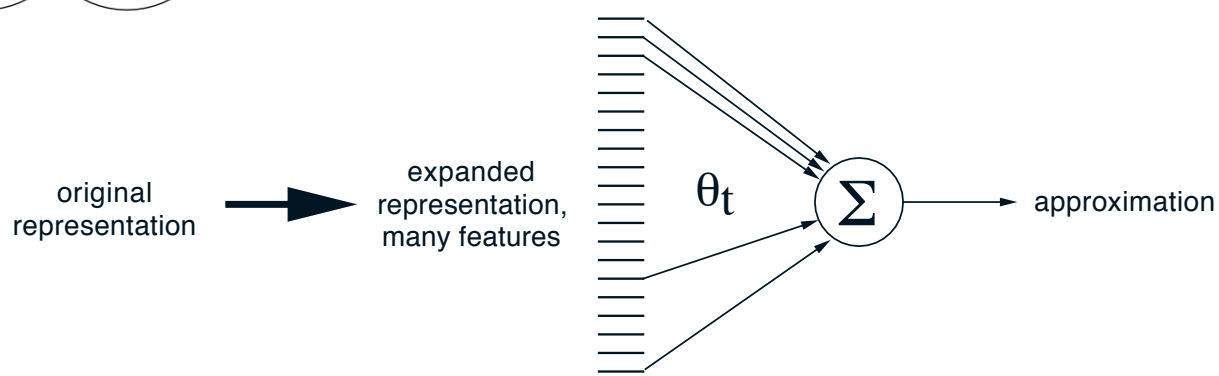
Coarse Coding

Example of linear value function approximation:

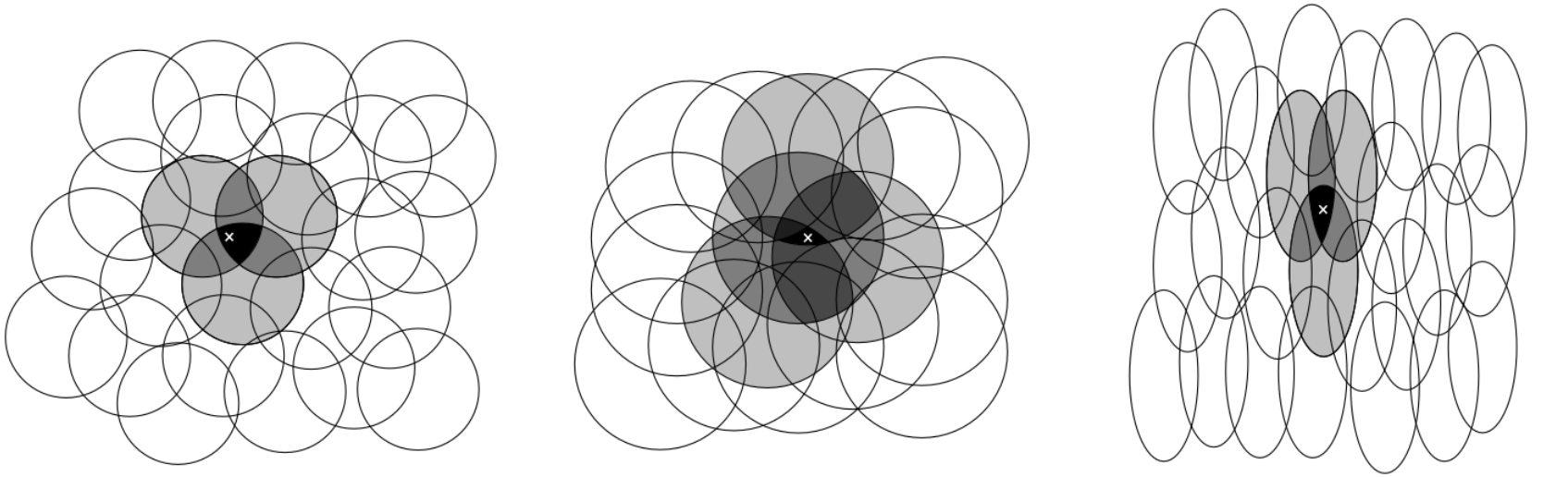
- *Coarse coding* provides large feature vector $\phi(s)$
- Parameter vector θ gives a value to each feature







Generalization in Coarse Coding

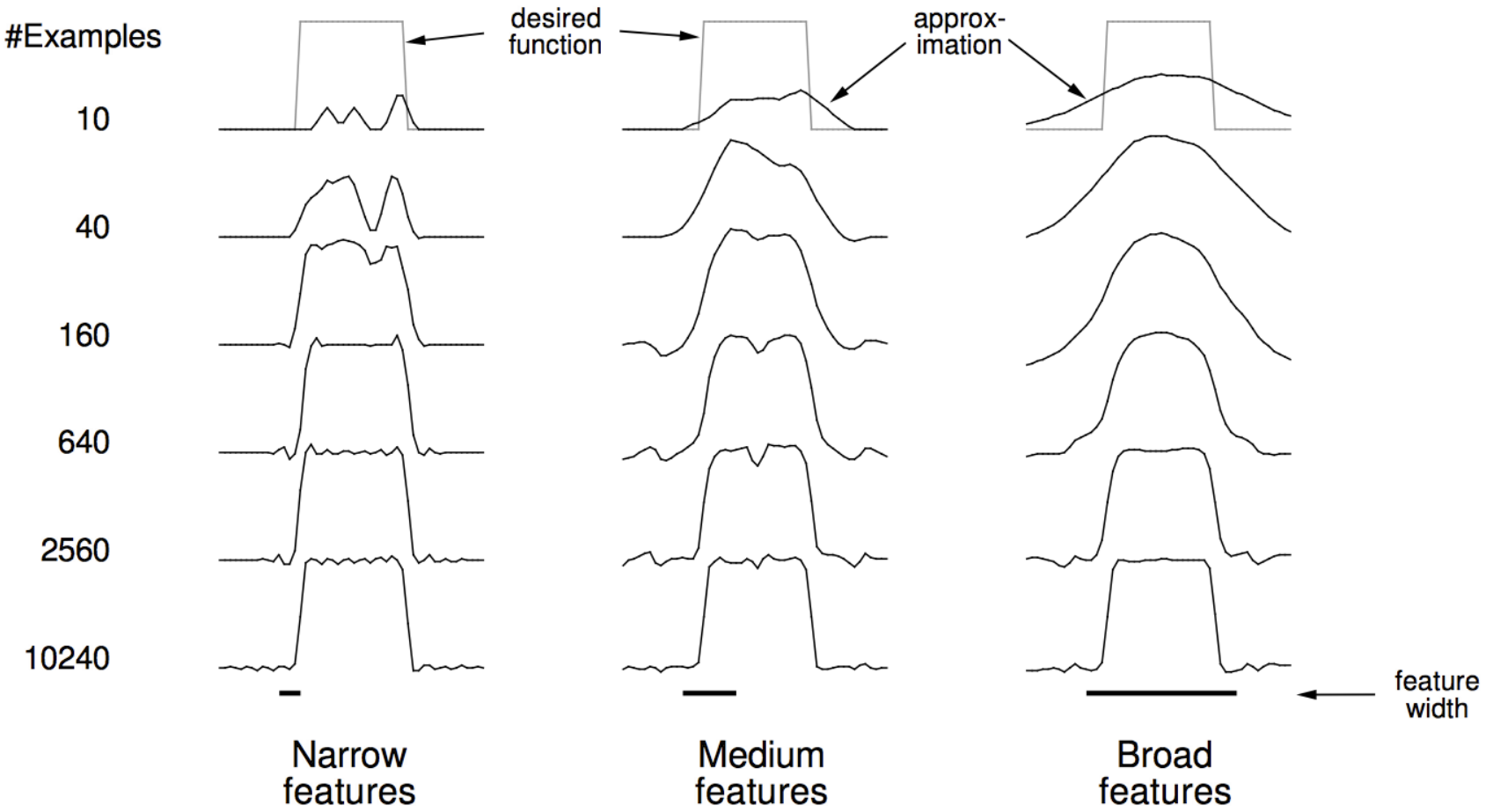


a) Narrow generalization

b) Broad generalization

c) Asymmetric generalization

Stochastic Gradient Descent with Coarse Coding



Incremental Prediction Algorithms

- Have assumed true value function $v^\pi(s)$ given by supervisor
- But in RL there is no supervisor, only rewards
- In practice, we substitute a *target* for $v^\pi(s)$
  - For MC, the target is the return G_t

$$\Delta\theta = \alpha(G_t - V_\theta(s))\nabla_\theta V_\theta(s)$$

  - For TD(0), the target is the TD target $r + \gamma V_\theta(s')$

$$\Delta\theta = \alpha(r + \gamma V_\theta(s') - V_\theta(s))\nabla_\theta V_\theta(s)$$

  - For TD(λ), the target is the λ-return G_t^λ

$$\Delta\theta = \alpha(G_t^\lambda - V_\theta(s))\nabla_\theta V_\theta(s)$$

Monte-Carlo with Value Function Approximation

- The return G_t is an unbiased, noisy sample of true value $v^\pi(s)$
- Can therefore apply supervised learning to "training data":

$$\langle S_1, G_1 \rangle, \langle S_2, G_2 \rangle, ..., \langle S_T, G_T \rangle$$

- For example, using *linear Monte-Carlo policy evaluation*

$$\begin{aligned} \Delta\theta &= \alpha(G_t - V_\theta(s))\nabla_\theta V_\theta(s) \\ &= \alpha(G_t - V_\theta(s))\phi(s) \end{aligned}$$

- Monte-Carlo evaluation converges to a local optimum
- Even when using non-linear value function approximation

TD Learning with Value Function Approximation

- The TD-target $R_{t+1} + \gamma V_\theta(S_{t+1})$ is a *biased* sample of true value $v^\pi(S_t)$
- Can still apply supervised learning to "training data":

$$\langle S_1, R_2 + \gamma V_\theta(S_2) \rangle, \langle S_2, R_3 + \gamma V_\theta(S_3) \rangle, ..., \langle S_{T-1}, R_T \rangle$$

- For example, using *linear TD(0)*

$$\begin{aligned} \Delta\theta &= \alpha(r + \gamma V_\theta(s') - V_\theta(s))\nabla_\theta V_\theta(s) \\ &= \alpha\delta\phi(s) \end{aligned}$$

- Linear TD(0) converges (close) to global optimum

TD(λ) with Value Function Approximation

- The λ-return G_t^λ is also a biased sample of true value $v^\pi(s)$
- Can again apply supervised learning to "training data":

$$\left\langle S_1, G_1^\lambda \right\rangle, \left\langle S_2, G_2^\lambda \right\rangle, ..., \left\langle S_{T-1}, G_{T-1}^\lambda \right\rangle$$

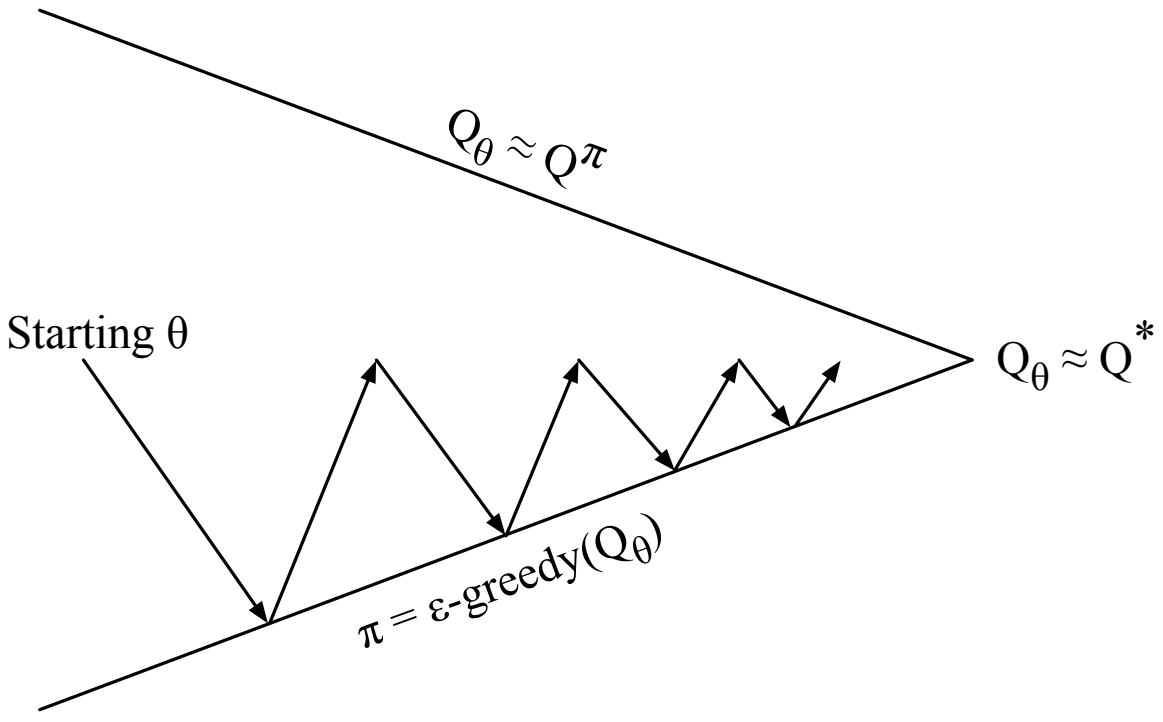
- Forward view linear TD(λ)

$$\begin{aligned}\Delta\theta &= \alpha(G_t^\lambda - V_\theta(S_t))\nabla_\theta V_\theta(S_t) \\ &= \alpha(G_t^\lambda - V_\theta(S_t))\phi(S_t)\end{aligned}$$

- Backward view linear TD(λ)

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma V_\theta(S_{t+1}) - V_\theta(S_t) \\ e_t &= \gamma\lambda e_{t-1} + \phi(S_t) \\ \Delta\theta &= \alpha\delta_t e_t\end{aligned}$$

Control with Value Function Approximation



Policy evaluation Approximate policy evaluation, $Q_\theta \approx q^\pi$

Policy improvement ϵ-greedy policy improvement

Action-Value Function Approximation

- Approximate the action-value function

$$Q_\theta(s, a) \approx q^\pi(s, a)$$

- Minimise mean-squared error between approximate action-value fn $Q_\theta(s, a)$ and true action-value fn $q^\pi(s, a)$

$$J(\theta) = \mathbb{E}_\pi \left[(q^\pi(S, A) - Q_\theta(S, A))^2 \right]$$

Here, $\mathbb{E}_\pi$ [] means both S and A are drawn from a distribution induced by π.

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2}\nabla_\theta J(\theta) = (q^\pi(s, a) - Q_\theta(s, a))\nabla_\theta Q_\theta(s, a)$$

$$\Delta\theta = \alpha(q^\pi(s, a) - Q_\theta(s, a))\nabla_\theta Q_\theta(s, a)$$

Linear Action-Value Function Approximation

- Represent state *and* action by a *feature vector*

$$\phi(s,a) = \begin{pmatrix} \phi_1(s,a) \\ \vdots \\ \phi_n(s,a) \end{pmatrix}$$

- Represent action-value fn by linear combination of features

$$Q_\theta(s,a) = \phi(s,a)^\top \theta = \sum_{j=1}^{n} \phi_j(s,a)\theta_j$$

- Stochastic gradient descent update

$$\nabla_\theta Q_\theta(s,a) = \phi(s,a)$$
$$\Delta\theta = \alpha(q^\pi(s,a) - Q_\theta(s,a))\phi(s)$$

Incremental Linear Control Algorithms

- Like prediction, we must substitute a *target* for $q^\pi(s, a)$
  - For MC, the target is the return G_t

$$\Delta\theta = \alpha(G_t - Q_\theta(S_t, a_t))\phi(S_t, A_t)$$

  - For SARSA(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

$$\Delta\theta = \alpha(R_{t+1} + \gamma Q_\theta(S_{t+1}, A_{t+1}) - Q_\theta(S_t, A_t))\phi(S_t, A_t)$$

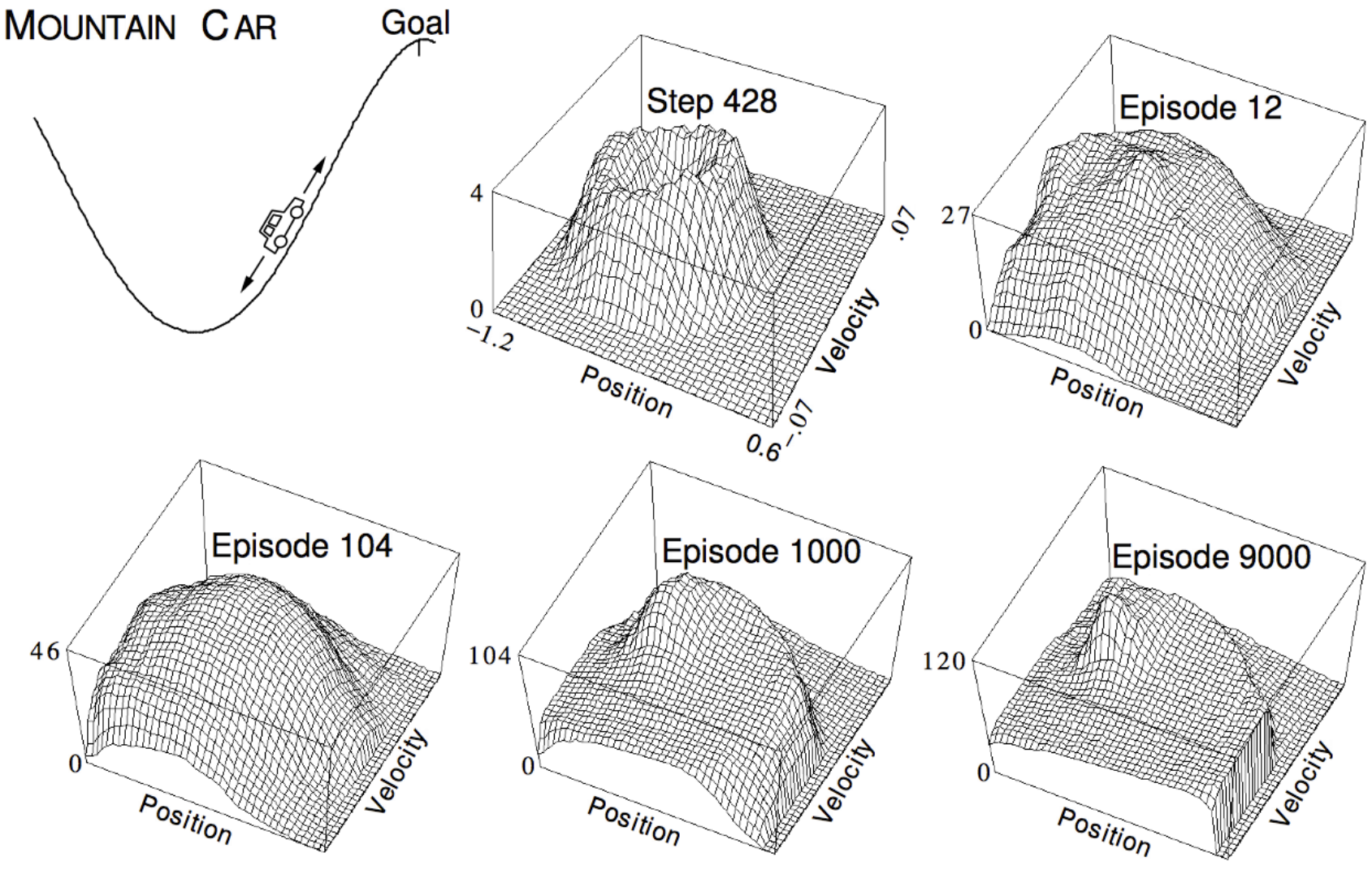
  - For forward-view Sarsa(λ), target is the λ-return with action-values

$$\Delta\theta = \alpha(G_t^\lambda - Q_\theta(S_t, A_t))\phi(S_t, A_t)$$

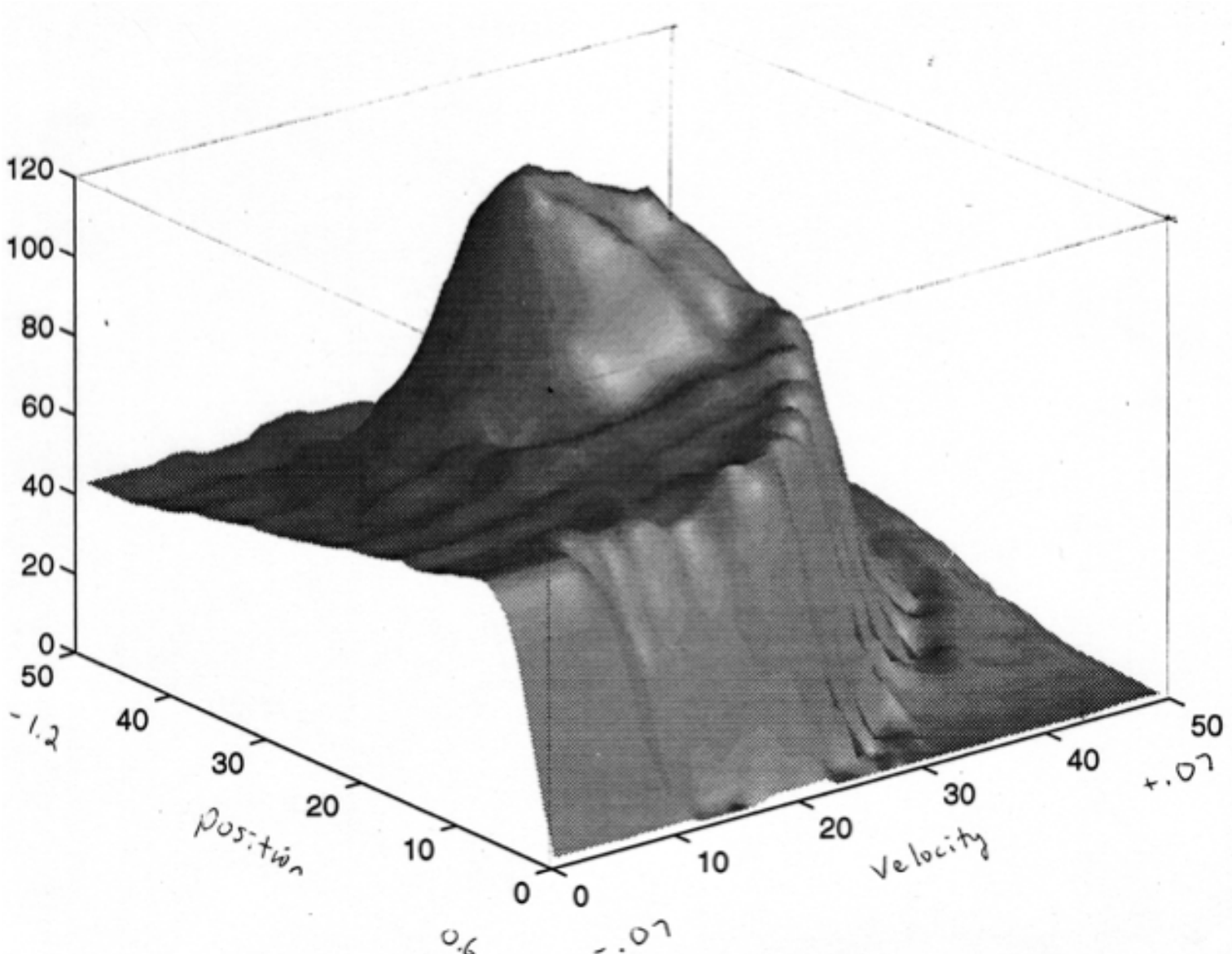
  - For backward-view Sarsa(λ), equivalent update is

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma Q_\theta(S_{t+1}, A_{t+1}) - Q_\theta(S_t, A_t) \\ e_t &= \gamma\lambda e_{t-1} + \phi(S_t, A_t) \\ \Delta\theta &= \alpha\delta_t e_t\end{aligned}$$

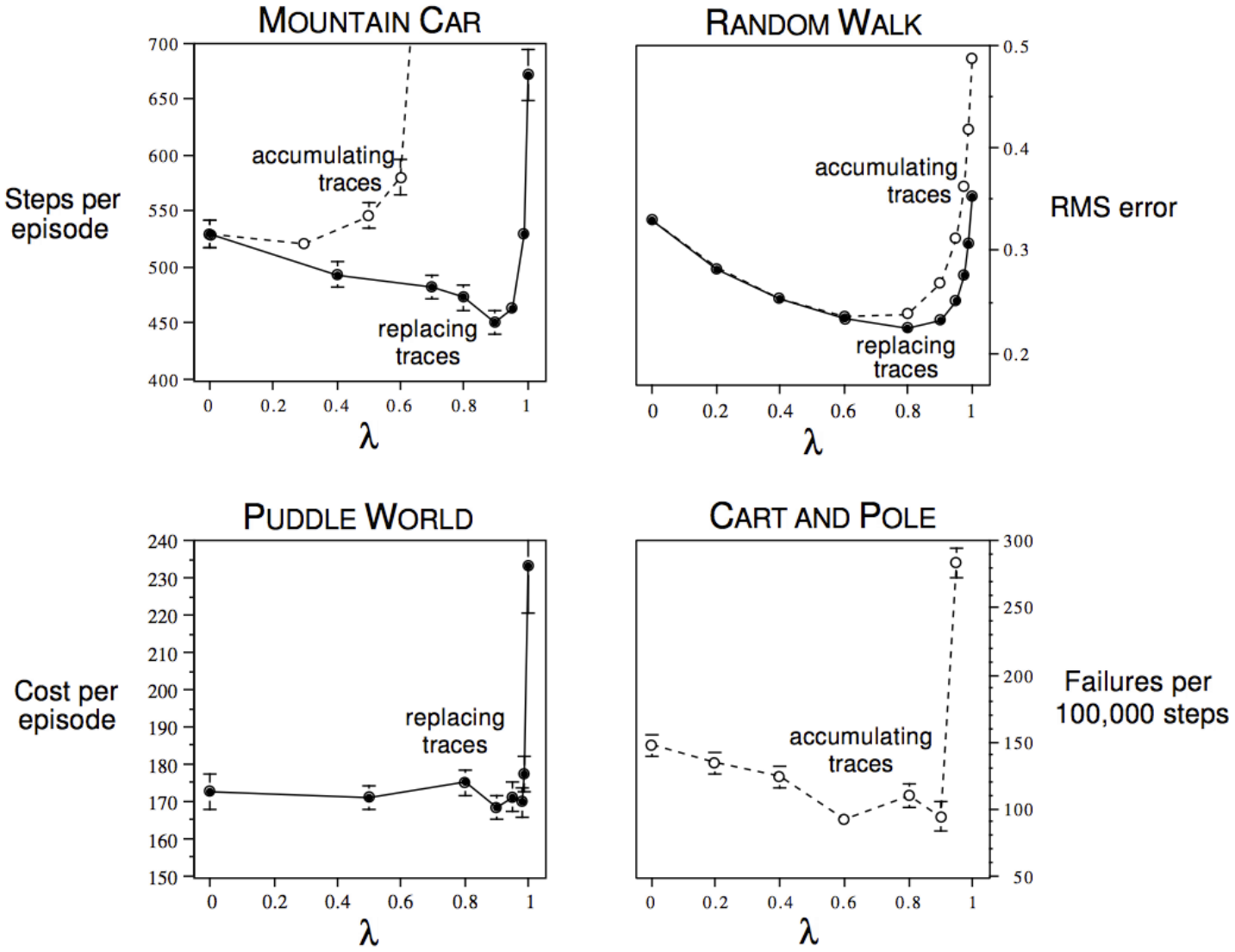
Linear Sarsa with Coarse Coding in Mountain Car



Linear Sarsa with Radial Basis Functions in Mountain Car



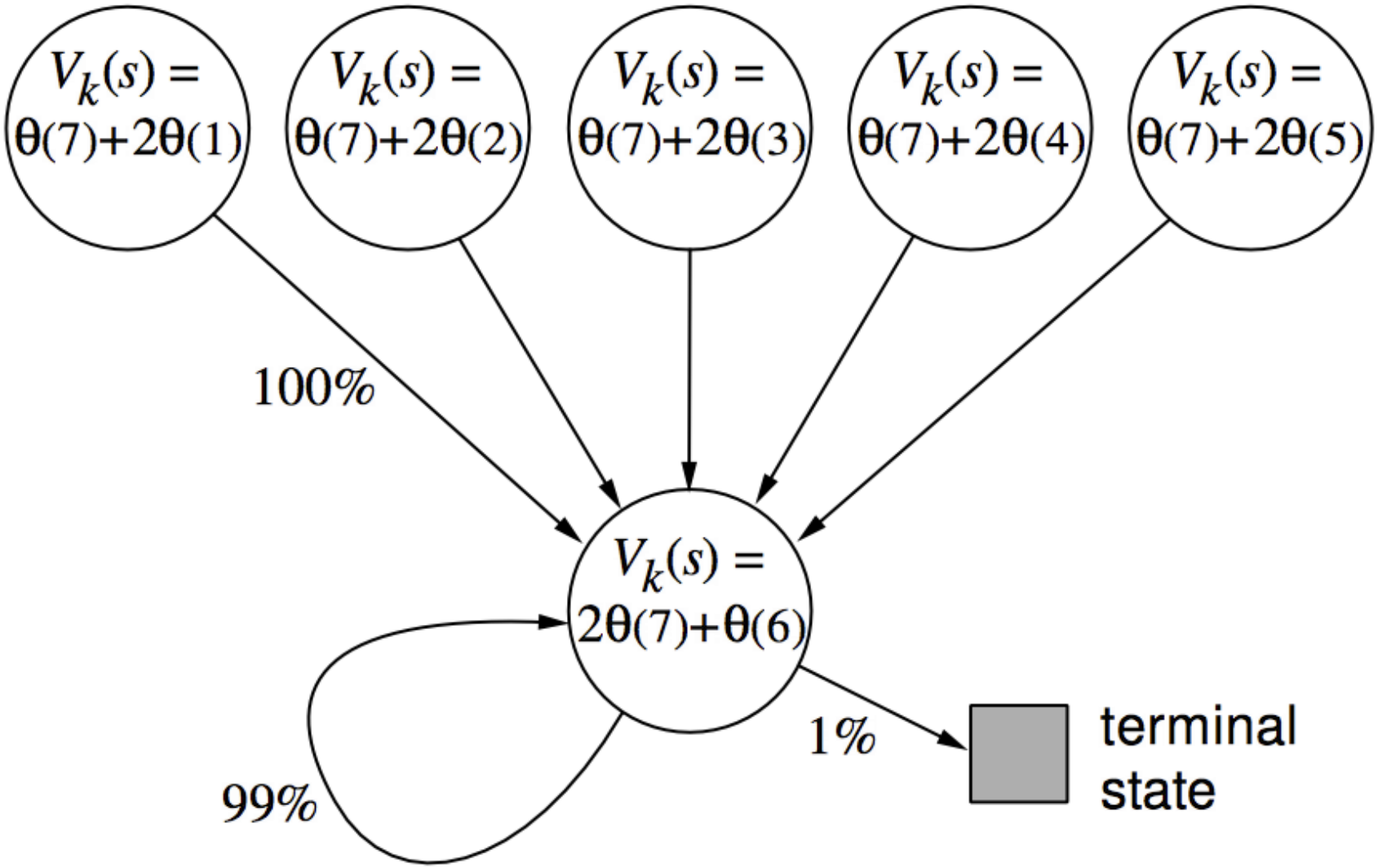
Study of λ: Should We Bootstrap?



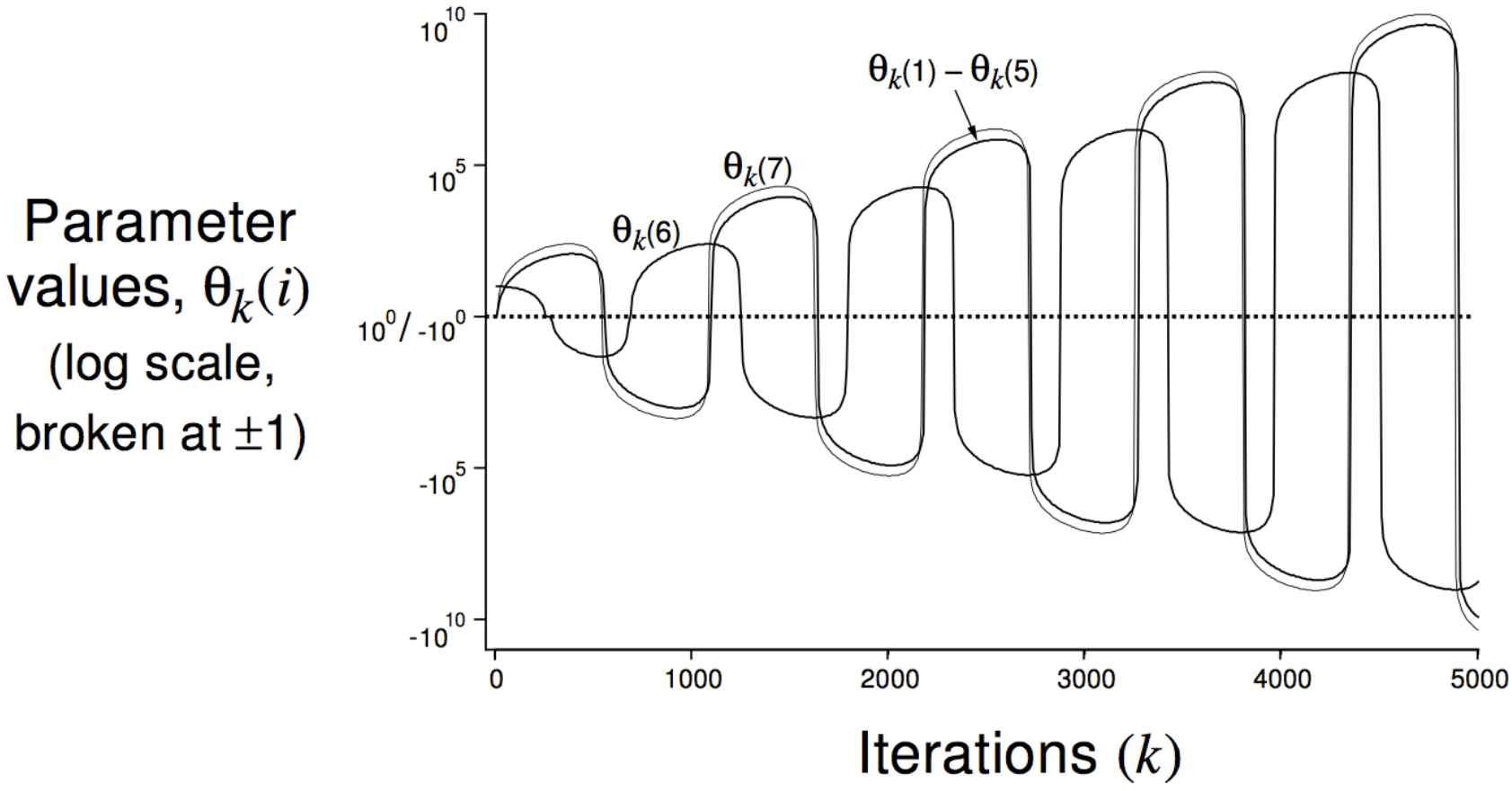
Convergence Questions

- The previous results show it is desirable to bootstrap
- But now we consider convergence issues
- When do incremental prediction algorithms converge?
  - When using bootstrapping (i.e. TD with $\lambda < 1$)?
  - When using linear value function approximation?
  - When using off-policy learning?
- Ideally, we would like algorithms that converge in all cases

Baird's Counterexample



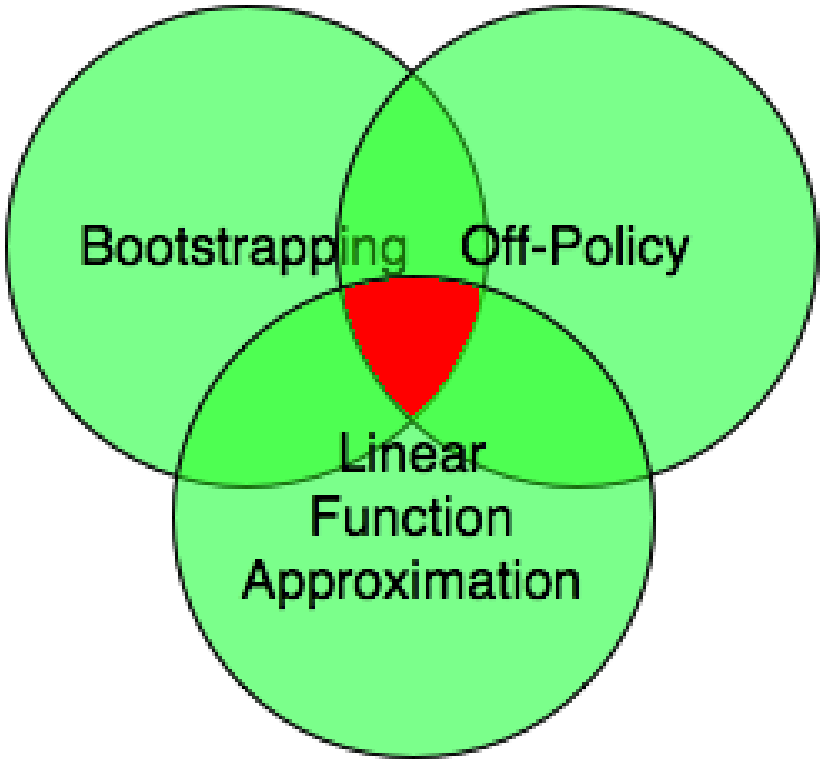
Parameter Divergence in Baird's Counterexample



Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Gruesome Threesome



We have not *quite* achieved our ideal goal for prediction algorithms.

Gradient Temporal-Difference Learning

- TD does not follow the gradient of *any* objective function
- This is why TD can diverge when off-policy or using non-linear function approximation
- Gradient TD follows true gradient of projected Bellman error

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓

Convergence of Control Algorithms

- In practice, the tabular control learning algorithms are extended to find a control policy (with linear FA or with neural nets).
- In theory, many aspects of control are not as simple to specify under function approximation.
- e.g. The starting state distribution is required before specifying an optimal policy, unlike in the tabular setting. The optimal policy can differ starting from state $s^{(1)}$ and from state $s^{(2)}$, but the state aggregation may not be able to distinguish between them.
- Such situations commonly arise in large environments (e.g. robotics), and *tracking* is often preferred to convergence. (continually adapting the policy instead of converging to a fixed policy).

Batch Reinforcement Learning

- Gradient descent is simple and appealing
- But it is *not* sample efficient
- Batch methods seek to find the best fitting value function for a given a set of past experience ("training data")

Least Squares Prediction

- Given value function approximation $V_\theta(s) \approx v^\pi(s)$
- And *experience* $\mathcal{D}$ consisting of $\langle$*state*, *estimated value*$\rangle$ pairs

$$\mathcal{D} = \{\left\langle S_1, \hat{V}_1^\pi \right\rangle, \left\langle S_2, \hat{V}_2^\pi \right\rangle, ..., \left\langle S_T, \hat{V}_T^\pi \right\rangle\}$$

- Which parameters θ give the *best fitting* value fn $V_\theta(s)$?
- Least squares algorithms find parameter vector θ minimising sum-squared error between $V_\theta(S_t)$ and target values $\hat{V}_t^\pi$,

$$\begin{aligned} LS(\theta) &= \sum_{t=1}^{T} (\hat{V}_t^\pi - V_\theta(S_t))^2 \\ &= \mathbb{E}_{\mathcal{D}} \left[(\hat{V}^\pi - V_\theta(s))^2 \right] \end{aligned}$$

Stochastic Gradient Descent with Experience Replay

Given experience consisting of $\langle state, value \rangle$ pairs

$$\mathcal{D} = \{\left\langle S_1, \hat{V}_1^\pi \right\rangle, \left\langle S_2, \hat{V}_2^\pi \right\rangle, ..., \left\langle S_T, \hat{V}_T^\pi \right\rangle\}$$

Repeat:

1. Sample state, value from experience

$$\left\langle s, \hat{V}^\pi \right\rangle \sim \mathcal{D}$$

2. Apply stochastic gradient descent update

$$\Delta\theta = \alpha(\hat{V}^\pi - V_\theta(s))\nabla_\theta V_\theta(s)$$

Converges to least squares solution

$$\theta^\pi = \underset{\theta}{\text{argmin}}\ LS(\theta)$$

Linear Least Squares Prediction

- Experience replay finds least squares solution
- But it may take many iterations
- Using *linear* value function approximation $V_\theta(s) = \phi(s)^\top \theta$
- We can solve the least squares solution directly

Linear Least Squares Prediction (2)

- At minimum of $LS(\theta)$, the expected update must be zero

$$\mathbb{E}_{\mathcal{D}}\left[\Delta\theta\right] = 0$$

$$\alpha \sum_{t=1}^{T} \phi(S_t)(\hat{V}_t^{\pi} - \phi(S_t)^{\top}\theta) = 0$$

$$\sum_{t=1}^{T} \phi(S_t)\hat{V}_t^{\pi} = \sum_{t=1}^{T} \phi(S_t)\phi(S_t)^{\top}\theta$$

$$\theta = \left(\sum_{t=1}^{T} \phi(S_t)\phi(S_t)^{\top}\right)^{-1} \sum_{t=1}^{T} \phi(S_t)\hat{V}_t^{\pi}$$

- For N features, direct solution time is $O(N^3)$
- Incremental solution time is $O(N^2)$ using Shermann-Morrison

Linear Least Squares Prediction Algorithms

- We do not know true values v_t^π (have estimates $\hat{V}_t^\pi$)
- In practice, our "training data" must use noisy or biased samples of v_t^π

LSMC	Least Squares Monte-Carlo uses return $v_t^\pi \approx G_t$
LSTD	Least Squares Temporal-Difference uses TD target $v_t^\pi \approx R_{t+1} + \gamma V_\theta(S_{t+1})$
LSTD(λ)	Least Squares TD(λ) uses λ-return $v_t^\pi \approx V_t^\lambda$

- In each case solve directly for fixed point of MC / TD / TD(λ)

Linear Least Squares Prediction Algorithms (2)

LSMC

$$0 = \sum_{t=1}^{T} \alpha(G_t - V_\theta(S_t))\phi(S_t)$$

$$\theta = \left(\sum_{t=1}^{T} \phi(S_t)\phi(S_t)^\top\right)^{-1} \sum_{t=1}^{T} \phi(S_t)G_t$$

LSTD

$$0 = \sum_{t=1}^{T} \alpha(R_{t+1} + \gamma V_\theta(S_{t+1}) - V_\theta(S_t))\phi(S_t)$$

$$\theta = \left(\sum_{t=1}^{T} \phi(S_t)(\phi(S_t) - \gamma\phi(S_{t+1}))^\top\right)^{-1} \sum_{t=1}^{T} \phi(S_t)R_{t+1}$$

LSTD(λ)

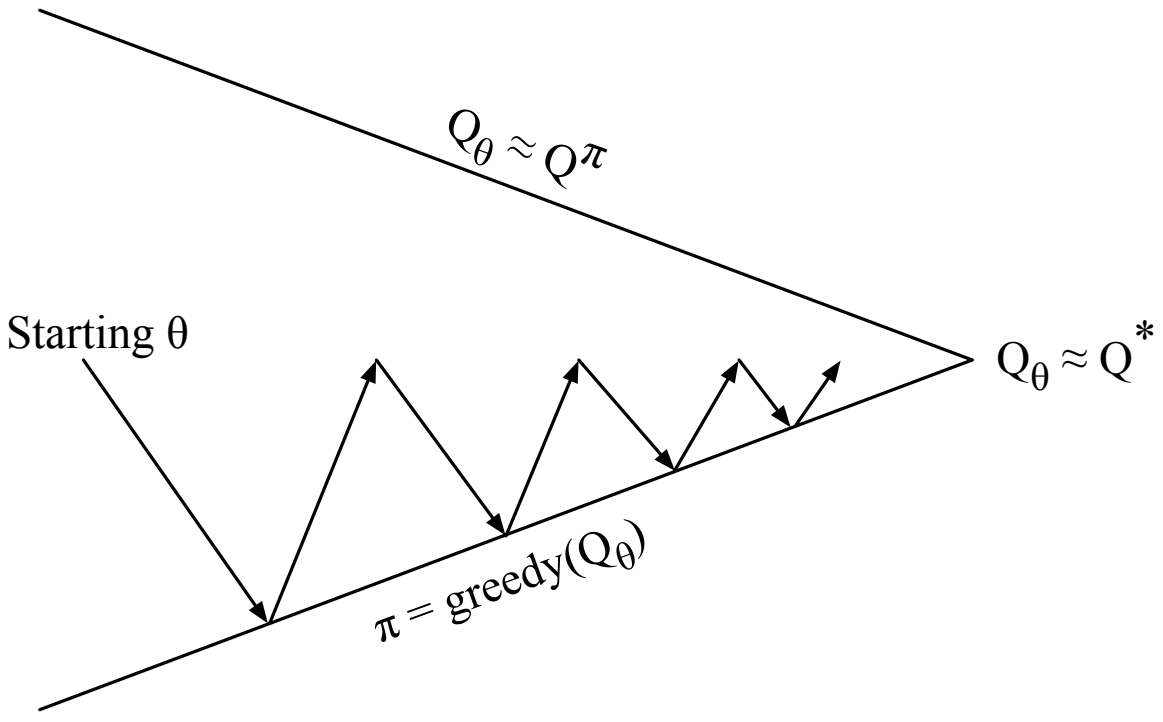
$$0 = \sum_{t=1}^{T} \alpha\delta_t e_t$$

$$\theta = \left(\sum_{t=1}^{T} e_t(\phi(S_t) - \gamma\phi(S_{t+1}))^\top\right)^{-1} \sum_{t=1}^{T} e_t R_{t+1}$$

Convergence of Linear Least Squares Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✗	✗
	LSTD	✓	✓	-

Least Squares Policy Iteration



Policy evaluation Policy evaluation by least squares Q-learning

Policy improvement Greedy policy improvement

Least Squares Action-Value Function Approximation

- Approximate action-value function $q^\pi(s, a)$
- using linear combination of features $\phi(s, a)$

$$Q_\theta(s, a) = \phi(s, a)^\top \theta \approx q^\pi(s, a)$$

- Minimise least squares error between $Q_\theta(s, a)$ and $q^\pi(s, a)$
- from experience generated using policy π
- consisting of $\langle (state, action), value \rangle$ pairs

$$\mathcal{D} = \{\left\langle (S_1, A_1), \hat{V}_1^\pi \right\rangle, \left\langle (S_2, A_2), \hat{V}_2^\pi \right\rangle, ..., \left\langle (S_T, A_T), \hat{V}_T^\pi \right\rangle\}$$

Least Squares Control

- For policy evaluation, we want to efficiently use all experience
- For control, we also want to improve the policy
- This experience is generated from many policies
- So to evaluate $q^\pi(s, a)$ we must learn off-policy
- We use the same idea as Q-learning:
  - Use experience generated by old policy
    $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$
  - Consider alternative successor action $a' = \pi_{new}(S_{t+1})$
  - Update $Q_\theta(S_t, A_t)$ towards value of alternative action
    $R_{t+1} + \gamma Q_\theta(S_{t+1}, a')$

Least Squares Q-Learning

- Consider the following linear Q-learning update

$$\delta = R_{t+1} + \gamma Q_\theta(S_{t+1}, \pi(S_{t+1})) - Q_\theta(S_t, A_t)$$
$$\Delta\theta = \alpha\delta\phi(S_t, A_t)$$

- LSTDQ algorithm: solve for total update = zero

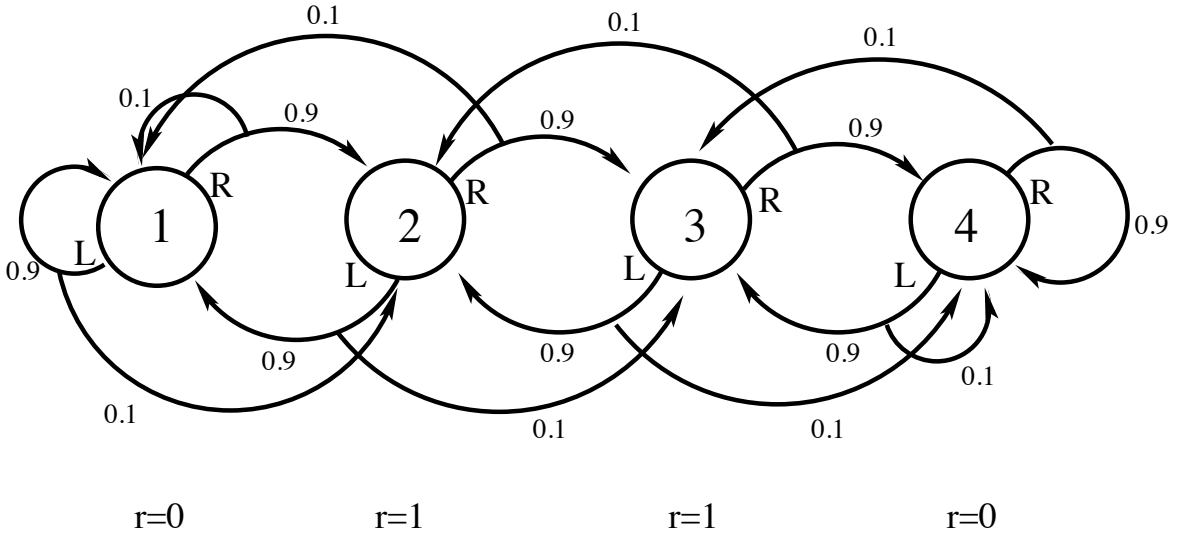
$$0 = \sum_{t=1}^{T} \alpha(R_{t+1} + \gamma Q_\theta(S_{t+1}, \pi(S_{t+1})) - Q_\theta(S_t, A_t))\phi(S_t, A_t)$$
$$\theta = \left(\sum_{t=1}^{T} \phi(S_t, A_t)(\phi(S_t, A_t) - \gamma\phi(S_{t+1}, \pi(S_{t+1})))^\top\right)^{-1} \sum_{t=1}^{T} \phi(S_t, A_t)R_{t+1}$$

Least Squares Policy Iteration Algorithm

- The following pseudocode uses LSTDQ for policy evaluation
- It repeatedly re-evaluates experience $\mathcal{D}$ with different policies

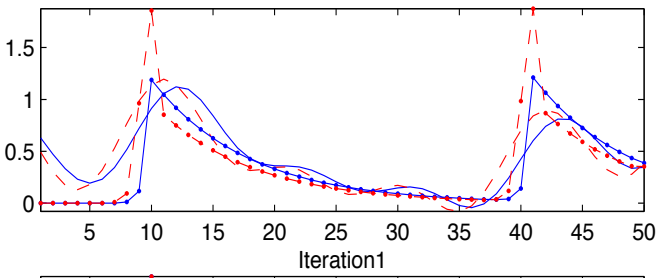
function LSPI-TD$(\mathcal{D}, \pi_0)$
  $\pi' \leftarrow \pi_0$
  repeat
    $\pi \leftarrow \pi'$
    $Q \leftarrow$ **LSTDQ**$(\pi, \mathcal{D})$
    for all $s \in \mathcal{S}$ **do**
      $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$
    end for
  until $(\pi \approx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

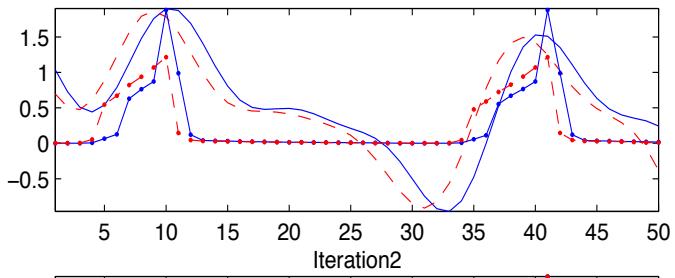
Chain Walk Example

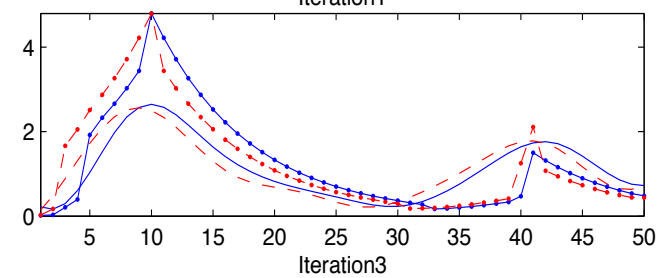


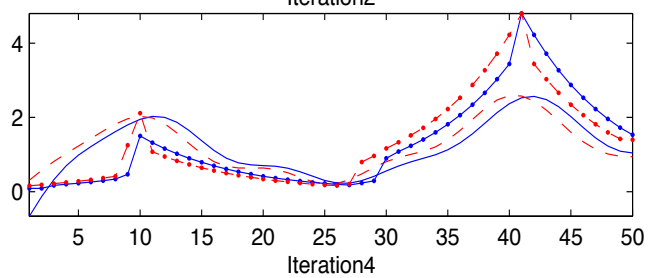
- Consider the 50 state version of this problem
- Reward $+1$ in states 10 and 41, 0 elsewhere
- Optimal policy: R (1-9), L (10-25), R (26-41), L (42, 50)
- Features: 10 evenly spaced Gaussians ($\sigma = 4$) for each action
- Experience: 10,000 steps from random walk policy

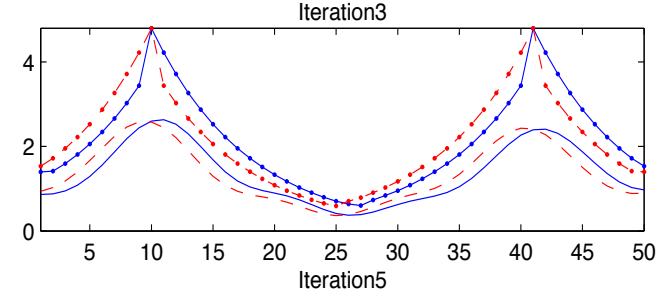
LSPI in Chain Walk: Action-Value Function

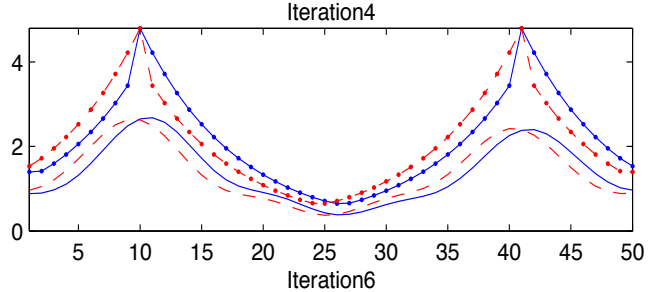


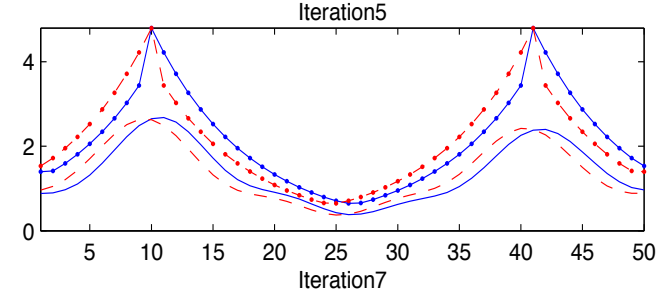












Exact (solid blue)
Function Approx. (red dashes)

Questions?