

# Lecture 6: Value Function Approximation

David Silver

# Outline

\* 5장까지는 table lookup >fft.

## 1 Introduction

## 2 Incremental Methods

## 3 Batch Methods

# Outline

1 Introduction

2 Incremental Methods

3 Batch Methods

# Large-Scale Reinforcement Learning

기전까지는 정해진规模의 **Table**을 만들어놓고 사용했다.

현실의 문제로 오게되면 **State 수가 굉장히 많아진다.** → Table을 만들 수 없게 된다.

이를 대처할 방법이 **Function Approximation**

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

- Backgammon:  $10^{20}$  states
- Computer Go:  $10^{170}$  states
- Helicopter: continuous state space → State  $\infty$

# 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: continuous state space

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

# Value Function Approximation

FAQ에서 자주 묻는 것은 Parameter  $W$ 를 update하는 것이다.

- So far we have represented value function by a *lookup table*
  - Every state  $s$  has an entry  $V(s) \rightarrow S$  만큼 Table
  - Or every state-action pair  $s, a$  has an entry  $Q(s, a) \rightarrow S \times a$  만큼 Table
- 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*

인별 state에 대해서도  
올바른 value가  
나오게 해준다.

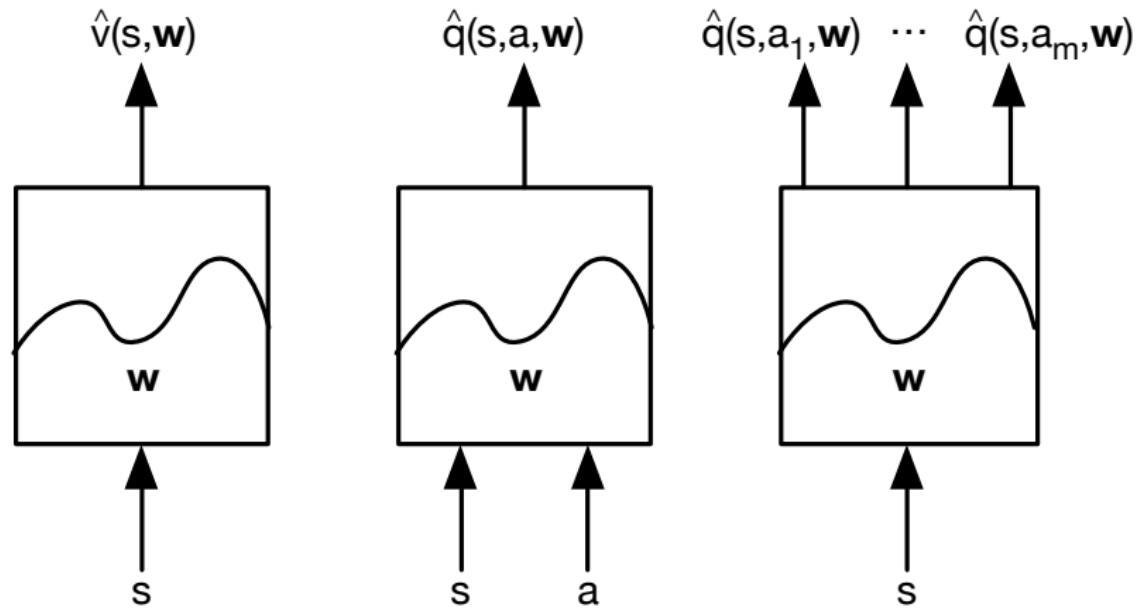
Approximator [  $\hat{v}(s, \underline{\mathbf{w}}) \approx v_\pi(s)$  ] 설계 function이면,  
 or  $\hat{q}(s, a, \underline{\mathbf{w}}) \approx q_\pi(s, a)$  Parameter

- Generalise from seen states to unseen states

- Update parameter  $w$  using MC or TD learning

$W$ 를 업데이트해준다.

# Types of Value Function Approximation



# Which Function Approximator?

There are many function approximators, e.g.

- Linear combinations of features
- Neural network → 비선형
- Decision tree → 의사결정트리
- Nearest neighbour
- Fourier / wavelet bases
- ...

# Which Function Approximator?

↳ 미분가능해야这样才能 update 할 수 있다.

We consider differentiable function approximators, e.g.

- Linear combinations of features      ] 미분가능.
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Furthermore, we require a training method that is suitable for non-stationary, non-iid data

↓  
모든퍼셉트론연합.

↳ Independent 하지 않음. (non-independent and identically distributed)

# Outline

1 Introduction

2 Incremental Methods

3 Batch Methods

## Gradient Descent



- Let  $J(\mathbf{w})$  be a differentiable function of parameter vector  $\mathbf{w}$

- Define the *gradient* of  $J(\mathbf{w})$  to be  
 $J(\cdot)$ 을 최소화하는 Input  $\mathbf{w}$ 를 찾고자 함.

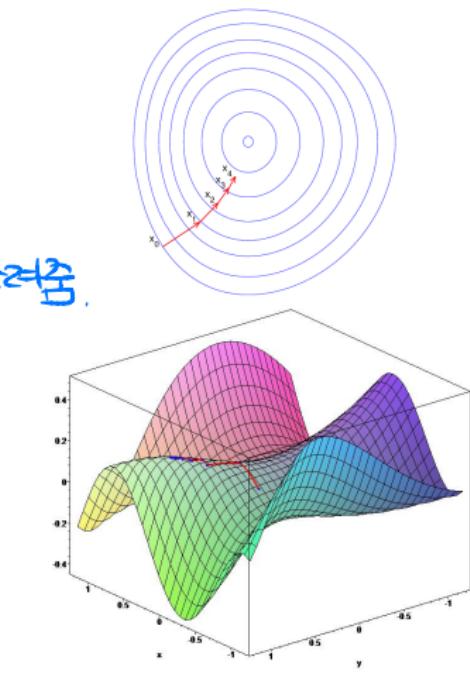
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{pmatrix} \rightarrow \text{경사 경로.}$$

- To find a local minimum of  $J(\mathbf{w})$
- Adjust  $\mathbf{w}$  in direction of -ve gradient

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

수학적 편의를 위한.  
Learning rate  
(학습률이 적어도)

where  $\alpha$  is a step-size parameter



## SGD

## Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector  $\mathbf{w}$  minimising mean-squared error between approximate value fn  $\hat{v}(s, \mathbf{w})$  and true value fn  $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[ (v_\pi(S) - \hat{v}(S, \mathbf{w}))^2 \right]$$

음, 양수차이 같게 하기 위한

Loss      true value      approximator

- Gradient descent finds a local minimum

$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})] \end{aligned}$$

sampling으로 대체해 Stochastic gradient descent를 적용한다

- Stochastic gradient descent *samples* the gradient

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

- Expected update is equal to full gradient update

# Feature Vectors

- Represent state by a *feature vector*

state의 feature vector

정의하기 나름이다.

$$\mathbf{x}(S) = \begin{pmatrix} \mathbf{x}_1(S) \\ \vdots \\ \mathbf{x}_n(S) \end{pmatrix}$$

- For example:

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

]*feature* 일수 있다.

# Linear Value Function Approximation

$$\vec{c} + \vec{d} \cdot \vec{s}$$

- Represent value function by a linear combination of features

$$\mathbf{x}(S) = \begin{bmatrix} x_1(S) \\ \vdots \\ x_n(S) \end{bmatrix}$$

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^T \mathbf{w} = \sum_{j=1}^n x_j(S) w_j \quad \text{inner product}$$

- Objective function is quadratic in parameters  $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[ (\hat{v}(S, \mathbf{w}) - v_\pi(S))^2 \right]$$

Approximator

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

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \mathbf{x}(S)$$

$$-\hat{v}(S, \mathbf{w}) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \Rightarrow \Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \mathbf{x}(S)$$

linear  
value  
function  
approx.

Update = step-size × prediction error × feature value

# Table Lookup Features

- Table lookup is a special case of linear value function approximation
- Using *table lookup features*

$$\mathbf{x}^{table}(S) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix}$$

→ 가능한 모든 \$S\$를 \$I\$라고  
나타진 것이다.

- Parameter vector  $\mathbf{w}$  gives value of each individual state

$$\hat{v}(S, \mathbf{w}) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix}$$

# Incremental Prediction Algorithms

true value function 자리에  
MC, TD 방법 적용.

- 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  $G_t$

$\nabla_w \hat{v}(S_t, w)$  하면  
얼만큼 움직여야하는지 나옴.

$$\Delta w = \alpha(G_t - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$$

- For TD(0), the target is the TD target  $R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$

$$\Delta w = \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$$

- For TD( $\lambda$ ), the target is the  $\lambda$ -return  $G_t^\lambda$  → 가중평균 사용(기하평균)

$$\Delta w = \alpha(G_t^\lambda - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$$

# Monte-Carlo with Value Function Approximation

Reward는 discount된 종합.

- Return  $G_t$  is an unbiased, noisy sample of true value  $v_\pi(S_t)$
- Can therefore apply supervised learning to “training data”:

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

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

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t - \hat{v}(S_t, \mathbf{w})) \mathbf{x}(S_t)\end{aligned}$$

- 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  $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$  is a *biased sample* of true value  $v_\pi(S_t)$
- Can still apply supervised learning to “training data”:

$\uparrow$  training data  
지금까지 훈련된  
자료

$$\langle S_1, \underline{R_2 + \gamma \hat{v}(S_2, \mathbf{w})} \rangle, \langle S_2, R_3 + \gamma \hat{v}(S_3, \mathbf{w}) \rangle, \dots, \langle S_{T-1}, R_T \rangle$$

*TD target*

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

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(\cancel{R} + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \\ &= \alpha \delta \mathbf{x}(S)\end{aligned}$$

"Time reverse"  $\cancel{R}$ 는 디버깅을 위해.  
"Time reverse"  $\cancel{R}$ 은 학습하는 데에 안 좋다.

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

learning rate · TD error · feature vector

# TD( $\lambda$ ) with Value Function Approximation

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

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

- Forward view linear TD( $\lambda$ )

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \mathbf{x}(S_t)\end{aligned}$$

- Backward view linear TD( $\lambda$ )

*eligibility trace*

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$$

$$E_t = \gamma \lambda \underbrace{E_{t-1}}_{\text{eligibility trace}} + \mathbf{x}(S_t)$$

$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

# TD( $\lambda$ ) with Value Function Approximation

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

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

- Forward view linear TD( $\lambda$ )

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \mathbf{x}(S_t)\end{aligned}$$

- Backward view linear TD( $\lambda$ )

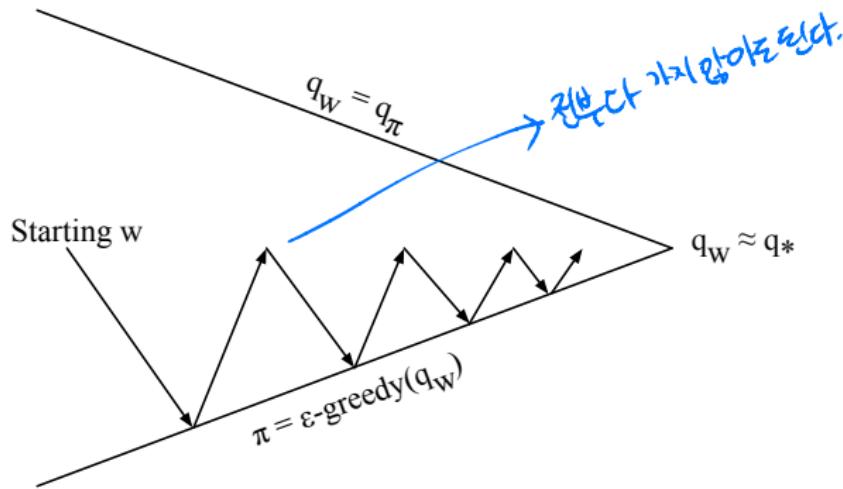
$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$$

$$E_t = \gamma \lambda E_{t-1} + \mathbf{x}(S_t)$$

$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

Forward view and backward view linear TD( $\lambda$ ) are equivalent

## Control with Value Function Approximation



Policy evaluation **Approximate** policy evaluation,  $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement  $\epsilon$ -greedy policy improvement

# Action-Value Function Approximation

자료별 Q 학습방법

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

- Minimise mean-squared error between approximate action-value fn  $\hat{q}(S, A, \mathbf{w})$  and true action-value fn  $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi [(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2]$$

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

$$\Delta \mathbf{w} = \alpha (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

# Linear Action-Value Function Approximation

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

$$\mathbf{x}(S, A) = \begin{pmatrix} \mathbf{x}_1(S, A) \\ \vdots \\ \mathbf{x}_n(S, A) \end{pmatrix}$$

- Represent action-value fn by linear combination of features

$$\hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)^\top \mathbf{w} = \sum_{j=1}^n \mathbf{x}_j(S, A) w_j$$

- Stochastic gradient descent update

$$\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)$$

$$\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

$$\Delta \mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \mathbf{x}(S, A)$$

# Incremental Control Algorithms

value function  $\hat{q}$ 함수  
 $V(S) \rightarrow Q(S, A)$

- Like prediction, we must substitute a *target* for  $q_{\pi}(S, A)$ 
  - For MC, the target is the return  $G_t$

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

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

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For forward-view TD( $\lambda$ ), target is the action-value  $\lambda$ -return

$$\Delta \mathbf{w} = \alpha(q_t^{\lambda} - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For backward-view TD( $\lambda$ ), equivalent update is

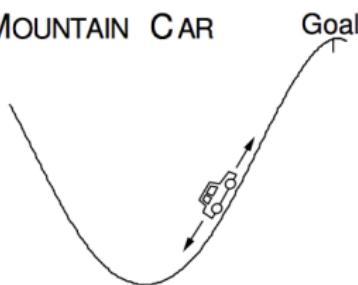
$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$

$$E_t = \gamma \lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

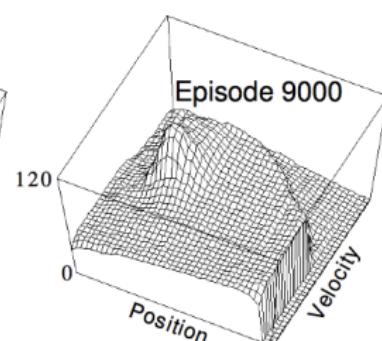
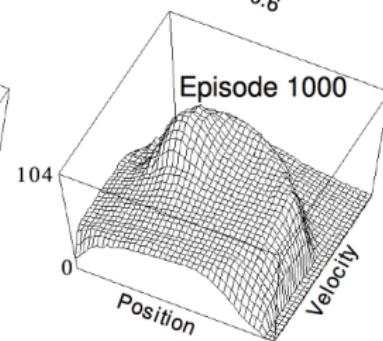
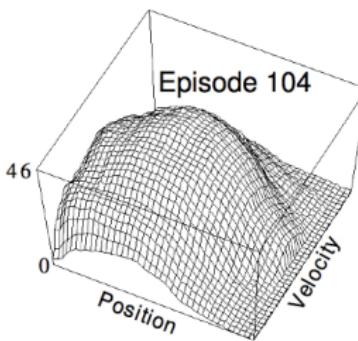
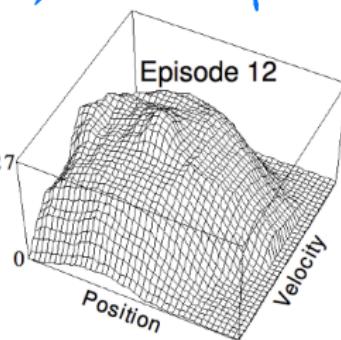
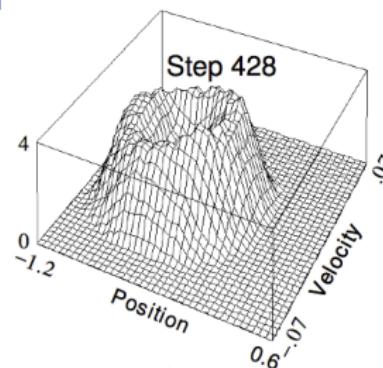
$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

# Linear Sarsa with Coarse Coding in Mountain Car

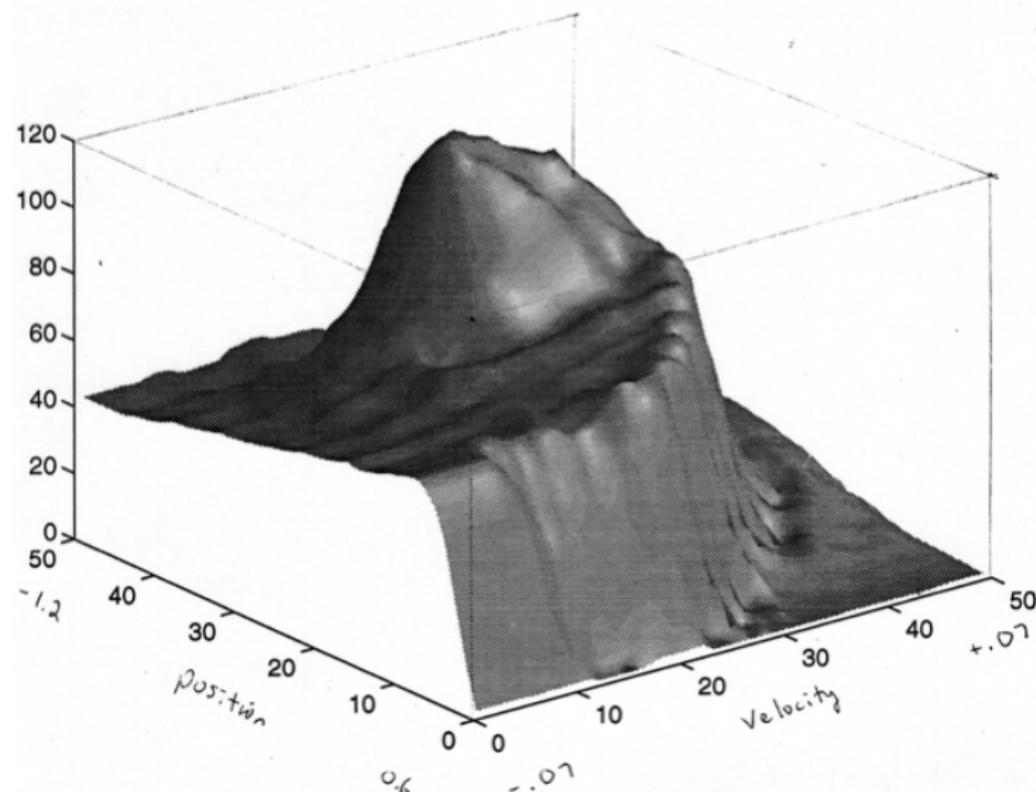
MOUNTAIN CAR



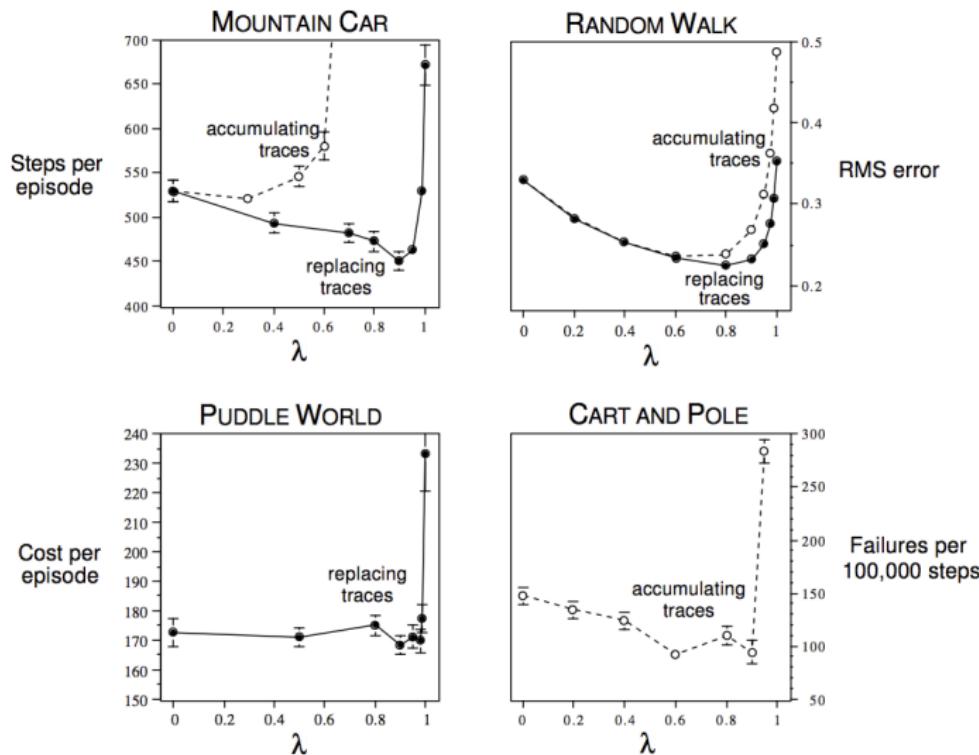
every single step update



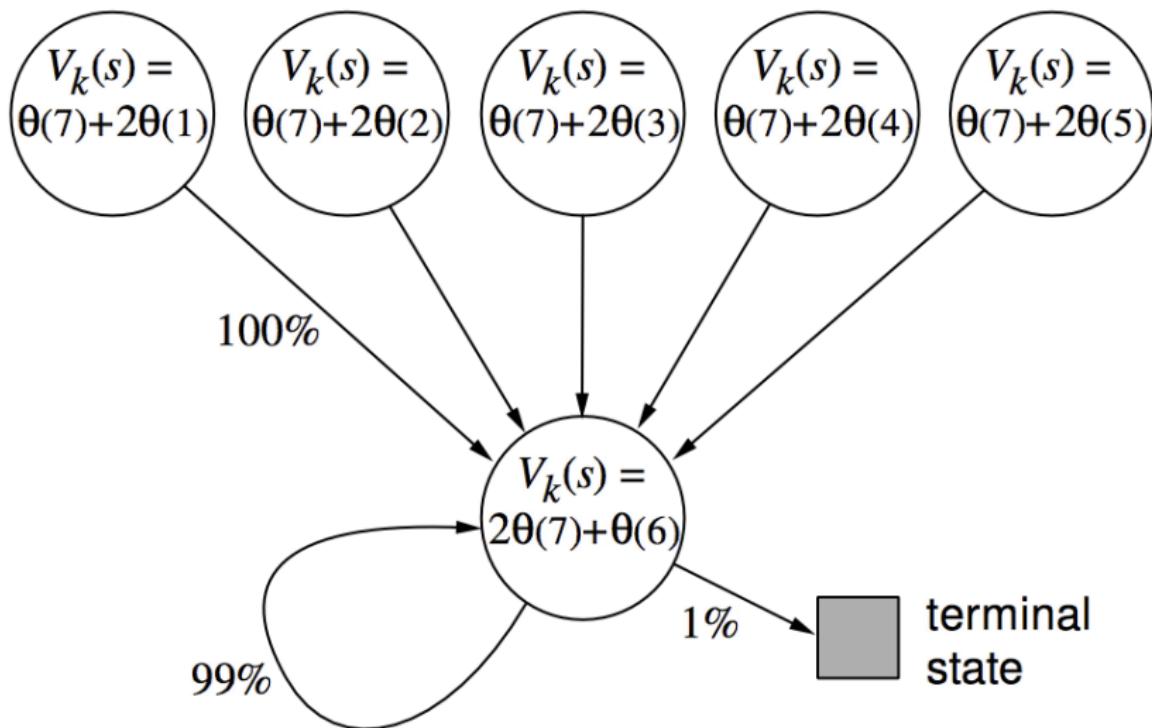
# Linear Sarsa with Radial Basis Functions in Mountain Car



# Study of $\lambda$ : Should We Bootstrap?

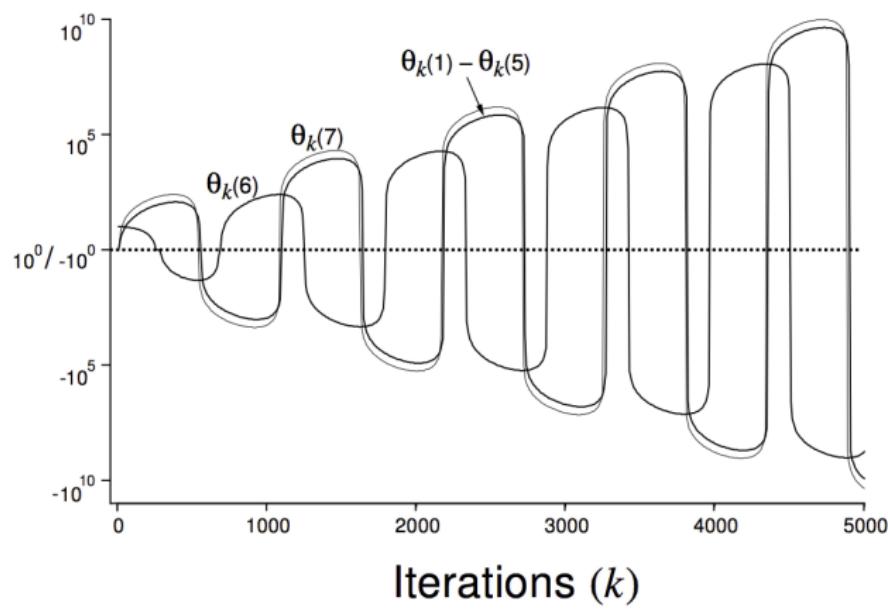


## Baird's Counterexample



# Parameter Divergence in Baird's Counterexample

Parameter values,  $\theta_k(i)$   
(log scale,  
broken at  $\pm 1$ )



# Convergence of Prediction Algorithms

*linear Approximation*

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

# 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

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
on policy Sarsa	✓	(✓)	✗
off policy Q-learning	✓	✗	✗
Gradient Q-learning	✓	✓	✗

(✓) = chatters around near-optimal value function

# Outline

1 Introduction

2 Incremental Methods

3 Batch Methods

# Batch Reinforcement Learning

- Gradient descent is simple and appealing
- But it is *not* sample efficient → sampleol 한 번 스트리밍해도 같은 결과를 얻을 수 있다.
- Batch methods seek to find the best fitting value function
- Given the agent's experience ("training data")

# Least Squares Prediction

Approximate true

- Given value function approximation  $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$
- And *experience*  $\mathcal{D}$  consisting of  $\langle \text{state}, \text{value} \rangle$  pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \dots, \langle s_T, v_T^\pi \rangle\}$$

- Which parameters  $\mathbf{w}$  give the *best fitting* value fn  $\hat{v}(s, \mathbf{w})$ ?
- Least squares** algorithms find parameter vector  $\mathbf{w}$  minimising sum-squared error between  $\hat{v}(s_t, \mathbf{w})$  and target values  $v_t^\pi$ ,

$$\begin{aligned} LS(\mathbf{w}) &= \sum_{t=1}^T (v_t^\pi - \hat{v}(s_t, \mathbf{w}))^2 && \text{sum square} \\ &= \mathbb{E}_{\mathcal{D}} [(v^\pi - \hat{v}(s, \mathbf{w}))^2] && \text{mean square} \end{aligned}$$

# Stochastic Gradient Descent with Experience Replay

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

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \dots, \langle s_T, v_T^\pi \rangle\}$$

→ **Don't Sampling**  
 ↓  
**SGD 적용**

Repeat: **every step,**

- 1 Sample state, value from experience

$$\underbrace{\langle s, v^\pi \rangle}_{\text{target}} \sim \mathcal{D}$$

- 2 Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

→ 같은 pair가 생기니까 재사용성 ↑

이거로 Experience Replay

# Stochastic Gradient Descent with Experience Replay

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

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \dots, \langle s_T, v_T^\pi \rangle\}$$

Repeat:

- 1 Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

- 2 Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

Converges to least squares solution

$$\mathbf{w}^\pi = \operatorname{argmin}_{\mathbf{w}} LS(\mathbf{w})$$

# Experience Replay in Deep Q-Networks (DQN)

DQN uses **experience replay** and **fixed Q-targets**

- Take action  $a_t$  according to  $\epsilon$ -greedy policy
- Store transition  $(s_t, a_t, r_{t+1}, s_{t+1})$  in replay memory  $\mathcal{D}$  → 64� batch
- Sample random mini-batch of transitions  $(s, a, r, s')$  from  $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters  $w^-$
- Optimise MSE between Q-network and Q-learning targets

SGD 사용

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[ \left( r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

여전버전  
파라미터

최신  
파라미터

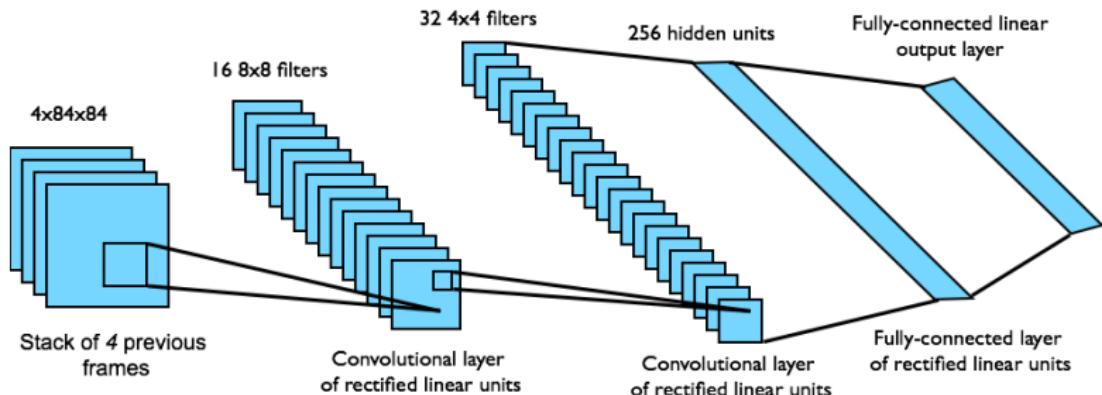
↳ fixed Q-target

- Using variant of stochastic gradient descent

# DQN in Atari

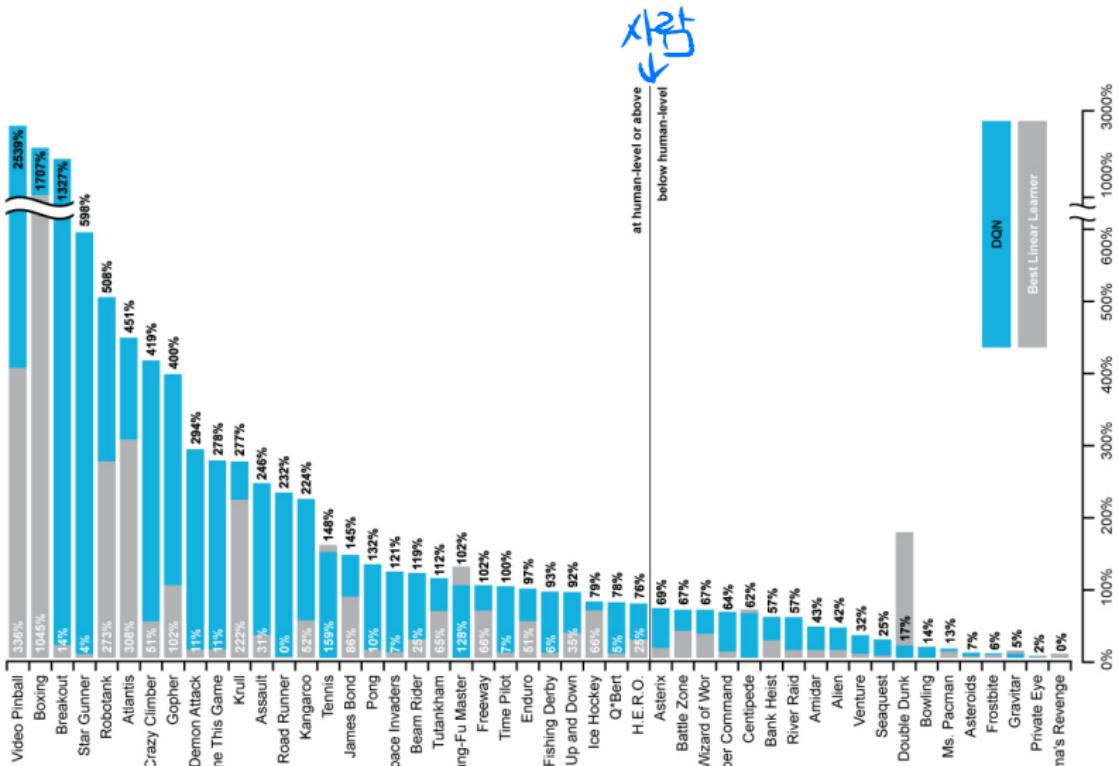
- End-to-end learning of values  $Q(s, a)$  from pixels  $s$
- Input state  $s$  is stack of raw pixels from last 4 frames
- Output is  $Q(s, a)$  for 18 joystick/button positions
- Reward is change in score for that step

↗ CNN



Network architecture and hyperparameters fixed across all games

# DQN Results in Atari



# How much does DQN help?

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

# Linear Least Squares Prediction

- Experience replay finds least squares solution
- But it may take many iterations
- Using *linear* value function approximation  $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w}$
- We can solve the least squares solution directly

## Linear Least Squares Prediction (2)

- At minimum of  $LS(\mathbf{w})$ , the expected update must be zero

$$\mathbb{E}_{\mathcal{D}} [\Delta \mathbf{w}] = 0$$

$$\cancel{\alpha} \sum_{t=1}^T \mathbf{x}(s_t)(v_t^\pi - \mathbf{x}(s_t)^\top \mathbf{w}) = 0$$

$\sum_{t=1}^T \mathbf{x}(s_t)v_t^\pi = \sum_{t=1}^T \mathbf{x}(s_t) \hat{V}$   
 w만 냉장고 옮기면

$$\left[ \sum_{t=1}^T \mathbf{x}(s_t)v_t^\pi = \left[ \sum_{t=1}^T \mathbf{x}(s_t)\mathbf{x}(s_t)^\top \right]^{-1} \mathbf{w} \right]$$

$$\mathbf{w} = \left( \sum_{t=1}^T \mathbf{x}(s_t)\mathbf{x}(s_t)^\top \right)^{-1} \sum_{t=1}^T \mathbf{x}(s_t)v_t^\pi$$

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

# Linear Least Squares Prediction Algorithms

- We do not know true values  $v_t^\pi$
- In practice, our “training data” must use noisy or biased samples of  $v_t^\pi$

LSMC Least Squares Monte-Carlo uses return

$$v_t^\pi \approx \underbrace{G_t}_\text{red}$$

LSTD Least Squares Temporal-Difference uses TD target

$$v_t^\pi \approx \underbrace{R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})}_\text{red}$$

LSTD( $\lambda$ ) Least Squares TD( $\lambda$ ) uses  $\lambda$ -return

$$v_t^\pi \approx \underbrace{G_t^\lambda}_\text{red}$$

- In each case solve directly for fixed point of MC / TD / TD( $\lambda$ )

## Linear Least Squares Prediction Algorithms (2)

각적용식 **LSMC**

$$0 = \sum_{t=1}^T \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left( \sum_{t=1}^T \mathbf{x}(S_t) \mathbf{x}(S_t)^\top \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t) G_t$$

**LSTD**

$$0 = \sum_{t=1}^T \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

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

**LSTD( $\lambda$ )**

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

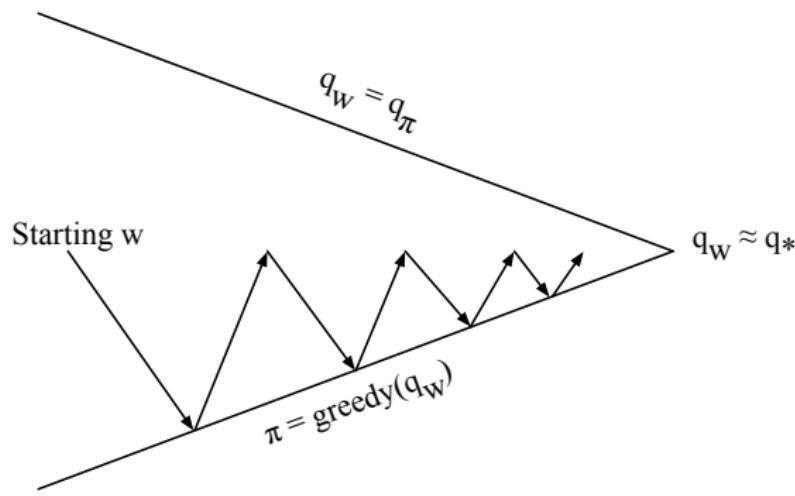
$$\mathbf{w} = \left( \sum_{t=1}^T E_t (\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^\top \right)^{-1} \sum_{t=1}^T E_t R_{t+1}$$

# 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  $\mathbf{x}(s, a)$

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)^\top \mathbf{w} \approx q_\pi(s, a)$$

- Minimise least squares error between  $\hat{q}(s, a, \mathbf{w})$  and  $q_\pi(s, a)$
- from experience generated using policy  $\pi$
- consisting of  $\langle (state, action), value \rangle$  pairs

$$\mathcal{D} = \{ \langle (s_1, a_1), v_1^\pi \rangle, \langle (s_2, a_2), v_2^\pi \rangle, \dots, \langle (s_T, a_T), v_T^\pi \rangle \}$$

# 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  $\hat{q}(S_t, A_t, \mathbf{w})$  towards value of alternative action  
 $R_{t+1} + \gamma \hat{q}(S_{t+1}, A', \mathbf{w})$

# Least Squares Q-Learning

- Consider the following linear Q-learning update

$$\delta = R_{t+1} + \gamma \hat{q}(S_{t+1}, \pi(S_{t+1}), \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$

$$\Delta \mathbf{w} = \alpha \delta \mathbf{x}(S_t, A_t)$$

- LSTDQ algorithm: solve for total update = zero

$$0 = \sum_{t=1}^T \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, \pi(S_{t+1}), \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})) \mathbf{x}(S_t, A_t)$$

$$\mathbf{w} = \left( \sum_{t=1}^T \mathbf{x}(S_t, A_t) (\mathbf{x}(S_t, A_t) - \gamma \mathbf{x}(S_{t+1}, \pi(S_{t+1})))^\top \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t, A_t) R_{t+1}$$

# Least Squares Policy Iteration Algorithm

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

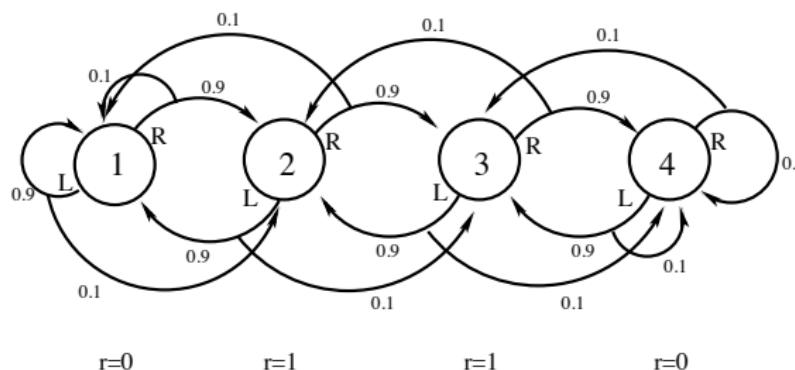
```
function LSPI-TD( $\mathcal{D}, \pi_0$ )
     $\pi' \leftarrow \pi_0$ 
    repeat
         $\pi \leftarrow \pi'$ 
         $Q \leftarrow \text{LSTDQ}(\pi, \mathcal{D})$ 
        for all  $s \in \mathcal{S}$  do
             $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q(s,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

# Convergence of Control Algorithms

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
LSPI	✓	(✓)	-

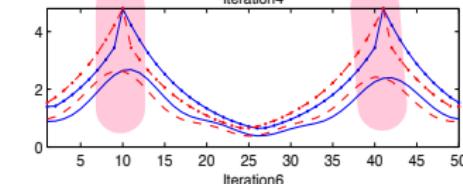
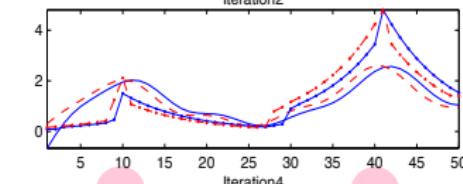
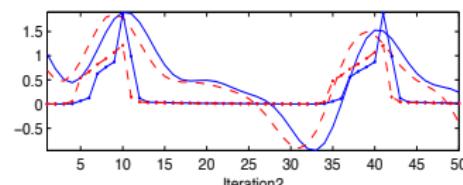
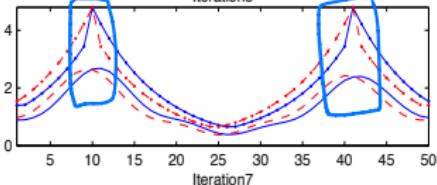
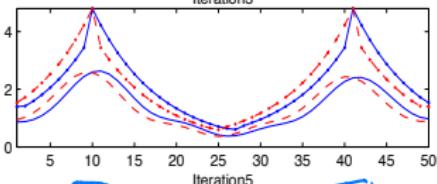
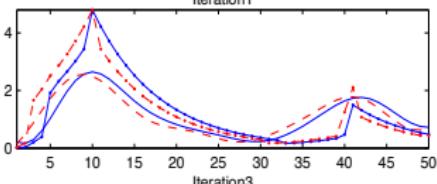
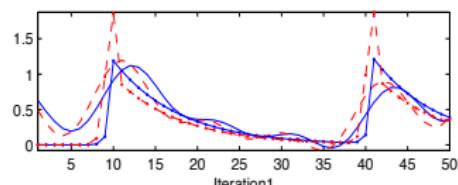
(✓) = chatters around near-optimal value function

# Chain Walk Example

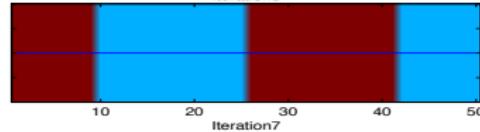
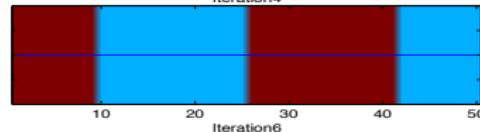
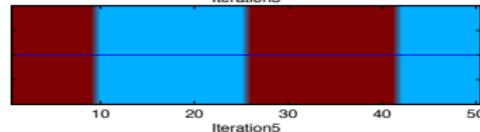
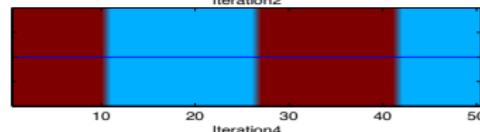
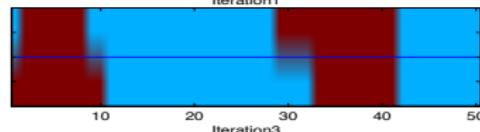
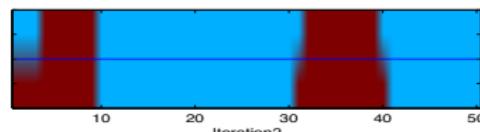
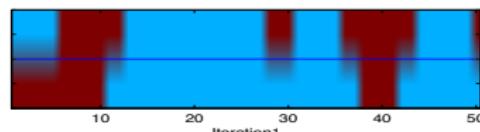


- Consider the 50 state version of this problem
- Reward +1 in states 10 and 41, 0 elsewhere *27th to 41st state*.
- Optimal policy: R (1-9), L (10-25), R (26-41), L (42, 50)
- Features: 10 evenly spaced Gaussians ( $\sigma = 4$ ) for each action
- Experience: 10,000 steps from random walk policy

# LSPI in Chain Walk: Action-Value Function



# LSPI in Chain Walk: Policy



# Questions?