Table of contents

10 Value Function Approximation (Deep RL)

Value Function Approximation

Large-Scale Reinforcement Learning

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

- Backgammon: 10^{20} states
- Computer Go: 10^{170} states
- The number of distinct 1-minute, 60 fps, 4K, 24-bit videos: $\approx 10^{10,216,000,000,000}$
- Humanoid Robot: continuous state space, *uncountable* number of states

How can we scale up the model-free methods for *prediction* and *control* using generalisation?

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 $Q(s, a)$,
  sufficient to do *control*, however . . .

Problem with large MDPs

- Too many states and/or actions to store in memory
- Too slow to learn the value of each state individually

Solution for large MDPs

Estimate value function with function approximation

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\text{or} \quad \hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

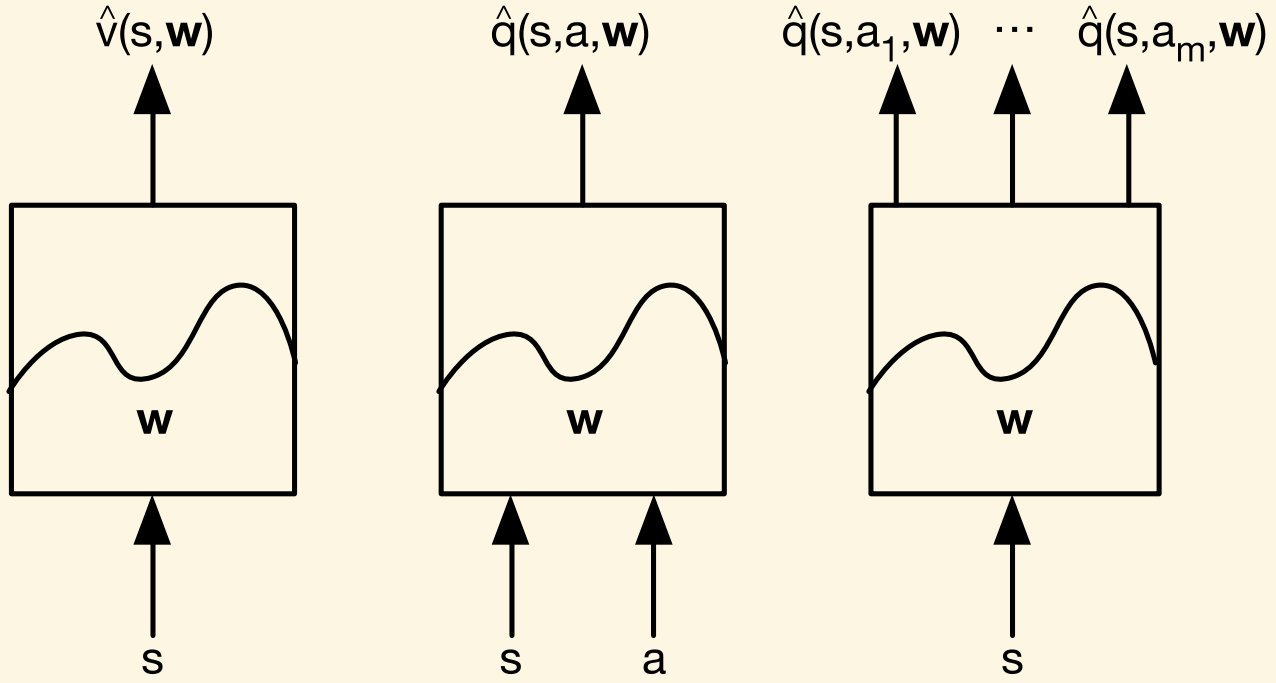
Generalises from seen states to unseen states

- Parametric function approximation to the *true* value function, $v_\pi(s)$ or action-value function $q_\pi(s, a)$

Update parameter $\mathbf{w}$ (a vector) using *MC* or *TD* learning

- Less memory required, based only on number of weights in vector $\mathbf{w}$, instead of number of states s (s *becomes implicit*)

Types of Value Function Approximation



Left: value function approximation tells us how much reward we will get at some input state. **Middle**: action-in value function approximation. **Right**: action-out value function approximation (over *all* actions, e.g. Atari)

Which Function Approximator?

- Linear combinations of features
- Neural networks
- Decision trees
- Nearest neighbours
- Fourier / wavelet bases
- . . .

Which Function Approximator?

We consider differentiable function approximators, e.g.

- Linear combinations of features
- Neural networks (sub-symbolic e.g. Convolution NNs (CNNs), Recurrent NNs (RNNS), transformers etc.)
- Decision trees (symbolic e.g. if-then-else rules, not differentiable)
- Nearest neighbours (e.g. clustering in ML, not differentiable)
- Fourier / wavelet bases (e.g. transform coding in video compression not differentiable)
- . . .

Furthermore, we require a *training method* that is suitable for

- non-stationary
- non-iid (independent and identically distributed) data

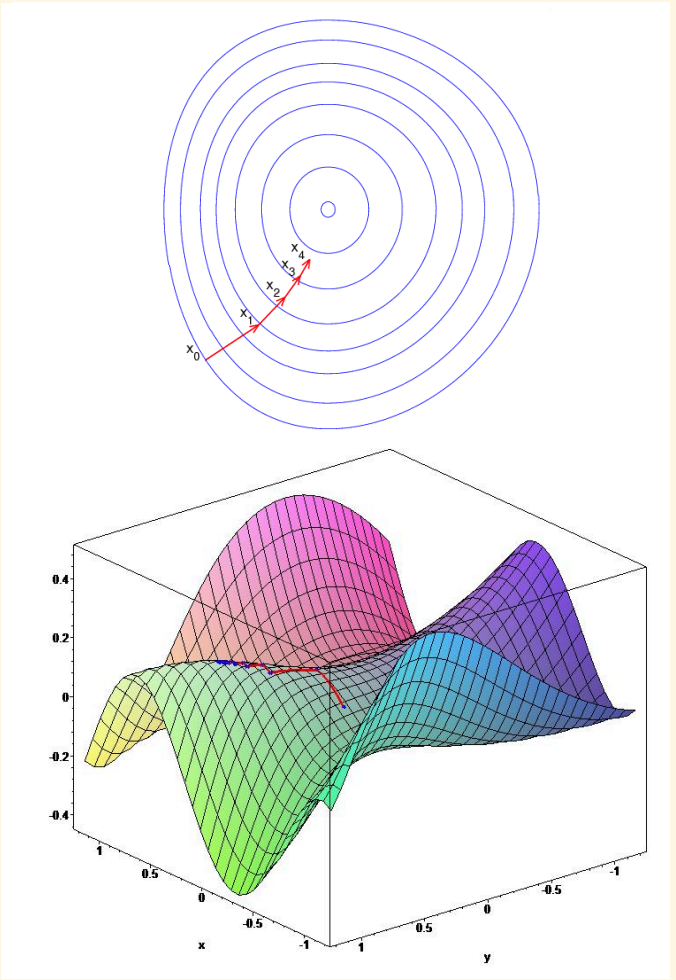
Incremental Methods - Theory

Gradient Descent

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

Define the gradient of $J(\mathbf{w})$ to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}n} \end{pmatrix}$$



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

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

i.e. move *downhill* (∇ is differential operator for vectors and α is a step-size parameter)

Value Function Approx. By Stochastic Gradient Descent

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

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

Gradient descent finds a local minimum

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

Stochastic gradient descent *samples* the gradient for states

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

Expected update equals the full gradient update.

However, this is essentially *supervised* learning, as we require an oracle

Feature Vectors

Represent *state* by a feature vector

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

For Example:

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

Linear Value Function Approximation

Represent value function by a linear combination (weighted sum) of features:

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

Objective function is quadratic in $\mathbf{w}$:

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

This means it is *convex* which means it is easy to optimise–we can be confident we won't get stuck in local minima

Stochastic gradient descent converges to the *global* optimum.

Update rule is simple

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

- Update = *step-size* × *prediction error* × *feature value*.
- $\cdots$ with respect to a supervisor or an oracle

Table Lookup Features

Table lookup is a *special case* of linear value function approximation

Using *table lookup features*

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

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 Methods - Algorithms

Incremental Prediction Algorithms

So far assumed true value function $v_\pi(s)$ is given by a supervisor

But in RL there is no supervisor, only rewards

In practice, substitute a target for $v_\pi(s)$, instead of an oracle, directly from rewards we get from experience

- *Monte Carlo*: target = return G_t

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

- *TD(0)*: target = $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$

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

- *TD(λ)*: target = λ-return G_t^λ

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

Monte-Carlo with Value Function Approximation

Return G_t is an unbiased, noisy sample of true value $v_\pi(S_t)$

Can therefore apply supervised learning to "training data" incrementally, derived from Monte Carlo episodes (roll outs) performed at each state

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

For example:, *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 for *non-linear* function approximation

TD Learning with Value Function Approximation

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":

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

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

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

- Use *TD error* $R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$ to update function approximator at each step
- Linear $TD(0)$ converges *close* to the 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 supervised learning to "training data":

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

Forward view linear *TD*(λ):

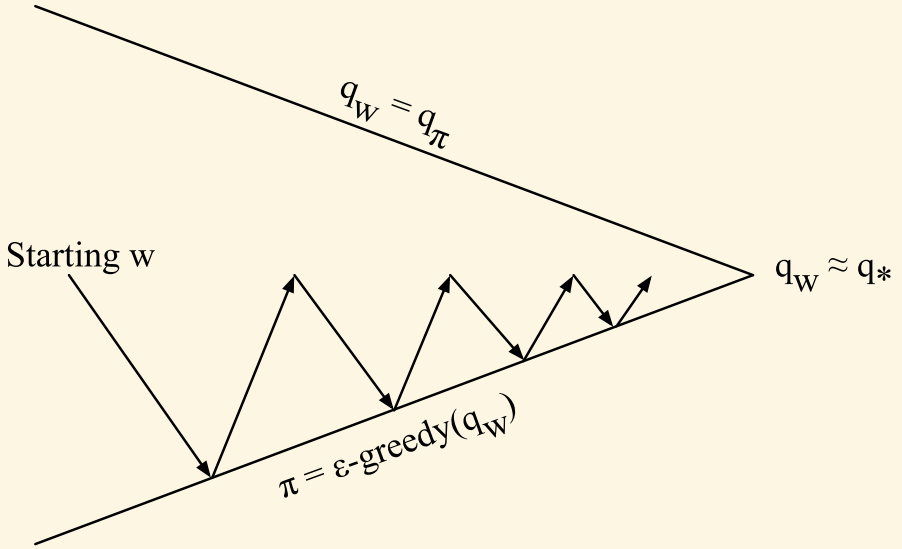
$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, w))x(S_t)\end{aligned}$$

Backward view linear *TD*(λ) using eligibility traces over *features* $\mathbf{x}$:

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \\ E_t &= \gamma \lambda E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t\end{aligned}$$

Forward and backward views are equivalent.

Control with Value Function Approximation



Policy evaluation approximate policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$
Policy improvement ϵ-greedy policy improvement

- i.e. uses value function approximator to get approximate q values
- Can update update value function approximator (e.g. neural network) at each step

Action-Value Function Approximation

Approximate the action-value function:

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

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

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

Stochastic gradient descent:

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

Linear Action-Value Function Approximation

Represent state and action by feature vector:

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

Linear action-value function:

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

Stochastic gradient descent update:

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

Incremental Control Algorithms

Like prediction, we must substitute a *target* for $q_\pi(S, A)$

For *Monte Carlo*, 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(λ)*, target is the action-value λ-return

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

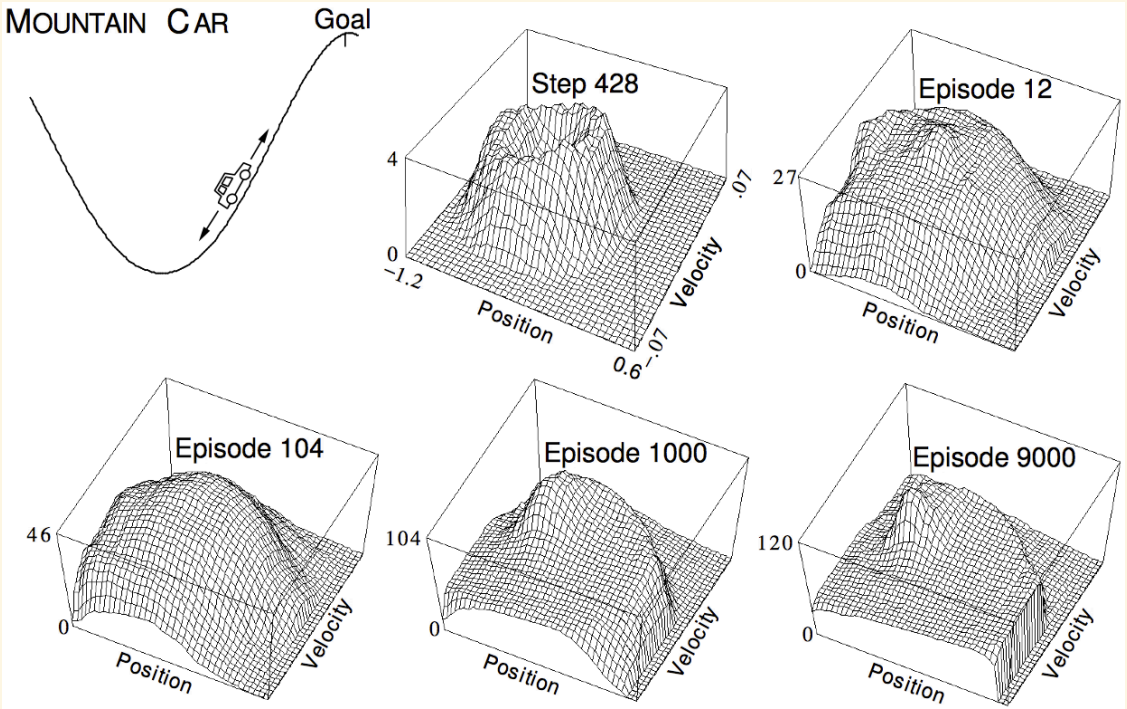
for *Backward-view TD(λ)*, the equivalent update is

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \\ E_t &= \gamma\lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w}) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t\end{aligned}$$

The reason we are using q instead of v is so we can do *control*

Examples (Incremental)

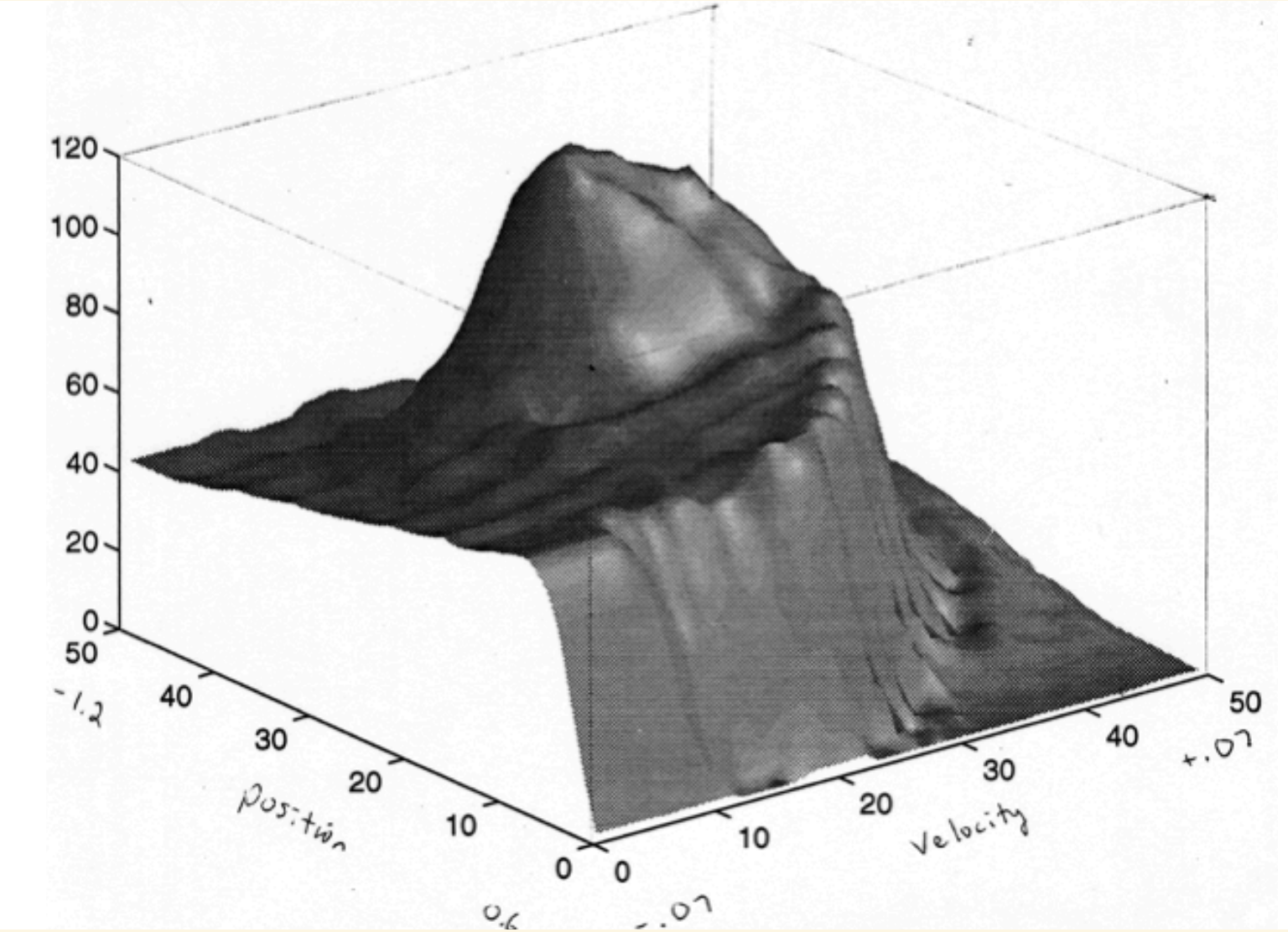
Linear Sarsa with Coarse Coding in Mountain Car



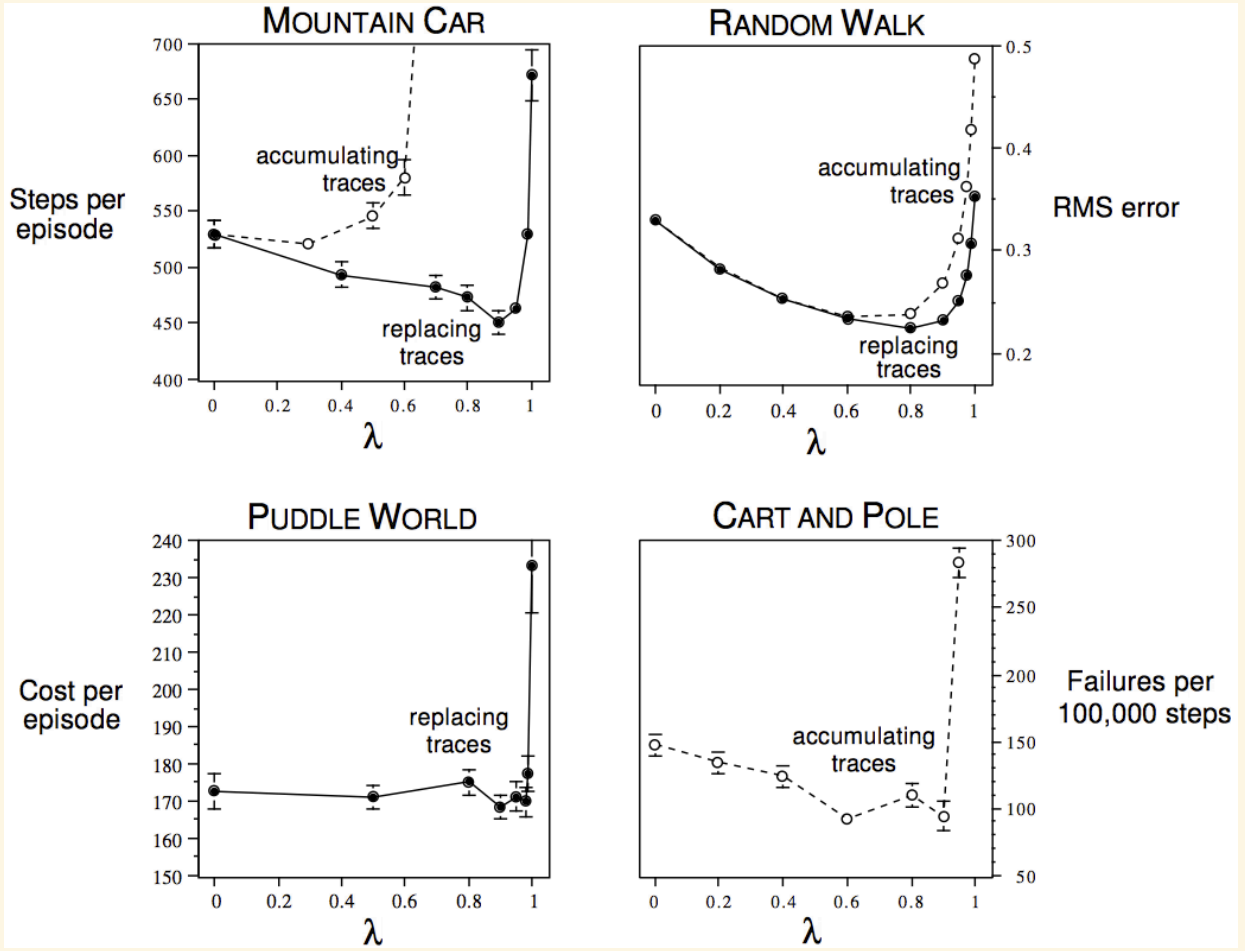
Car isn't powerful enough to drive up the mountain—it needs to drive forwards, roll backwards, drive forwards, etc. until it reaches goal

- Features in vector $\mathbf{x}$ are position and velocity
- Updates q every step using Sarsa returns

Linear Sarsa with Radial Basis Functions in Mountain Car

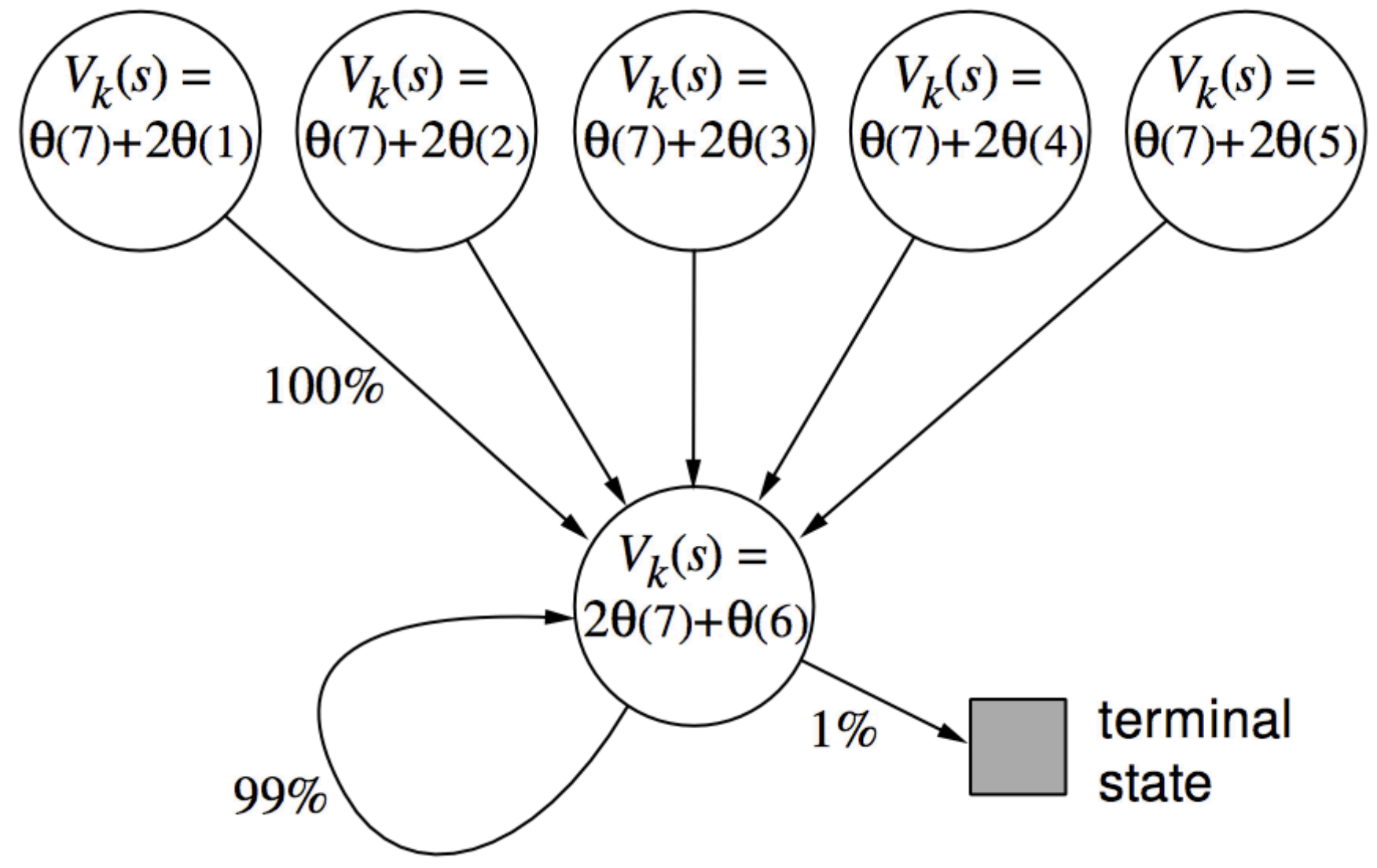


Study of λ: Should We Bootstrap?

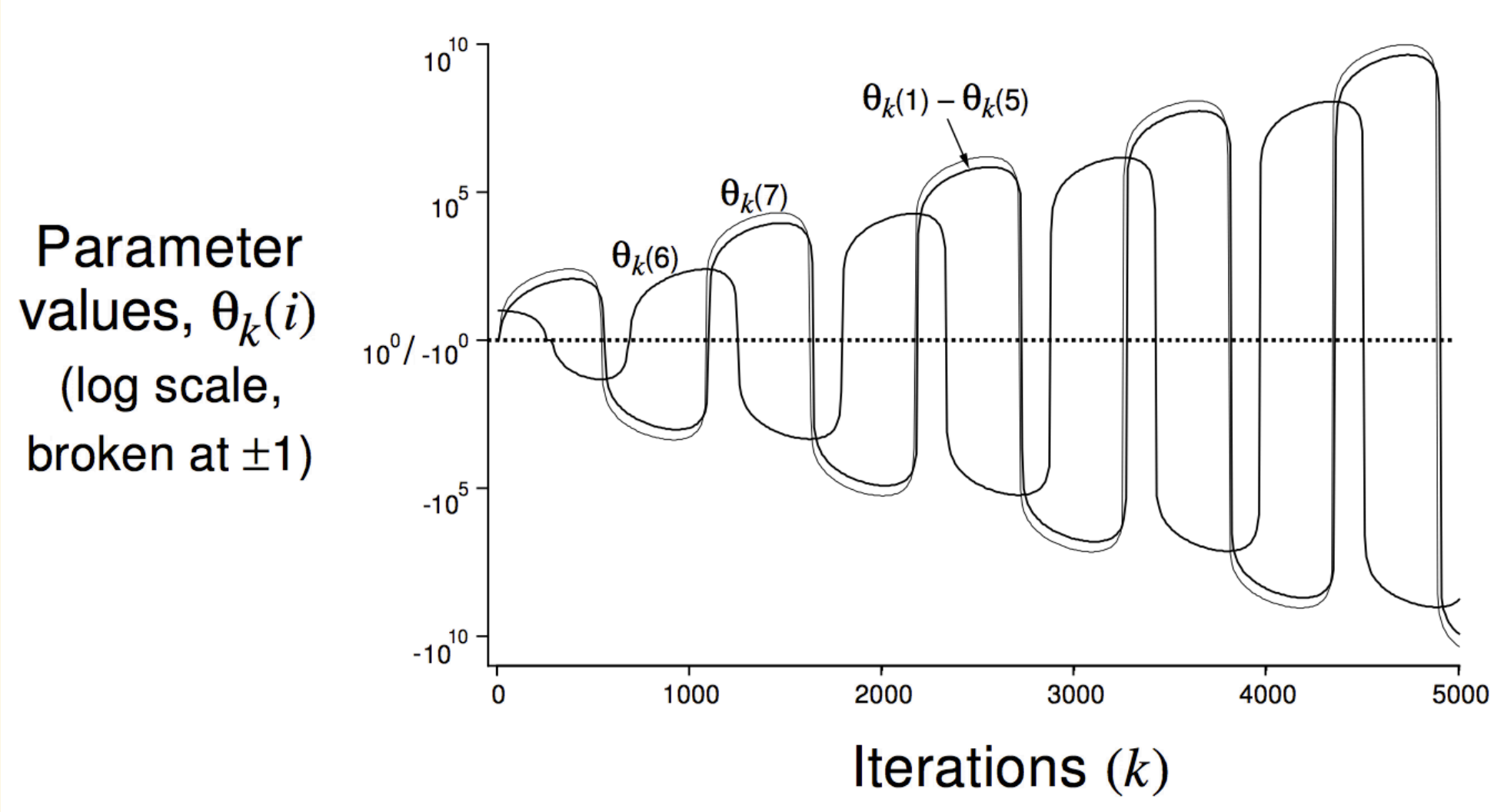


We should always bootstrap (using eligibility traces) and there is a sweet spot for λ (ignore distinction between accumulating versus replacing traces)

Baird's Counterexample–TD does not always converge



Parameter Divergence in Baird's Counterexample



Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	$TD(0)$	✓	✓	✗
	$TD(\lambda)$	✓	✓	✗
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 with non-linear function approximation.
- Gradient TD follows true gradient of projected Bellman error using a correction term

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

A principled successor of the Gradient TD technique is *Emphatic TD*

- it introduces *emphasis* weights that stabilise updates and reduce bias

Convergence of Control Algorithms

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-Learning	✓	✗	✗
Gradient Q-Learning	✓	✓	✗

(✓) = chatters (oscillates) around near-optimal value function.

Batch Methods

Batch Reinforcement Learning

Gradient descent is simple and appealing.

- But it is *not* sample efficient, as we throw away data after we have used it

Batch methods seek to find the best fitting value function

- Given the agent's experience ("training data") so far

Least Squares Prediction

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

$$D = \{\langle s1, v_1^\pi \rangle, \ldots, \langle sT, v_T^\pi \rangle\}$$

Which parameters $\mathbf{w}$ give the best fitting value function $\hat{v}(s, \mathbf{w})$ for the entire data set?

Least squares algorithms find parameter vector $\mathbf{w}$ by minimising sum-squared error between $\hat{v}(st, \mathbf{w})$ and target values v_t^π

$$LS(\mathbf{w}) = \sum_{t=1}^{T} (v_t^\pi - \hat{v}(st, \mathbf{w}))^2$$

$$= \mathbb{E}_D[(v^\pi - \hat{v}(s, \mathbf{w}))^2]$$

Stochastic Gradient Descent with Experience Replay

Given experience of $\langle state, value \rangle$ pairs, that we *store*

$$D = \{\langle s1, v_1^\pi \rangle, \ldots, \langle sT, v_T^\pi \rangle\}$$

Stochastic Gradient Descent with Experience Replay

Repeat:
1. Sample states, values randomly from stored *experience*

$$\langle s, v^\pi \rangle \sim D$$

2. Apply one *stochastic gradient descent* update toward the target to samples states

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

Converges to least squares solution: $\mathbf{w}^\pi = \arg\min_{\mathbf{w}} LS(\mathbf{w})$

Experience Replay in Deep Q-Networks (DQN)

DQN uses experience replay and fixed Q-targets (off-policy)

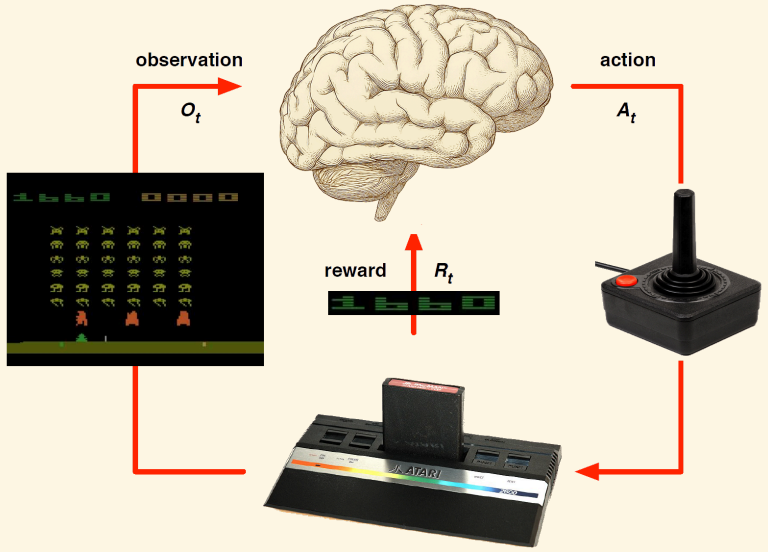
- Take action a_t according to ε-greedy policy
- Store transitions $(st, at, rt+1, st+1)$ in replay memory D
- Sample mini-batch of transitions (s, a, r, s') from D (at random from different episodes)
- Compute Q-learning targets w.r.t. fixed parameters w^-
- Optimise MSE (loss L_i) between TD-target and Q-network estimate:

$$L_i(\mathbf{w}_i) = \underbrace{\mathbb{E}_{(s,a,r,s')\sim\mathcal{D}_i}}_{\text{sampled from } D_i} \Big[\underbrace{\big(r + \gamma \max_{a'} Q(s', a'; \mathbf{w}_i^-)\big)}_{\text{TD target}} - \underbrace{Q(s, a; \mathbf{w}_i)}_{\text{Q estimate}} \Big)^2 \Big]$$

- Using variant of stochastic gradient descent ($\mathbb{E}$ is expectation or mean)

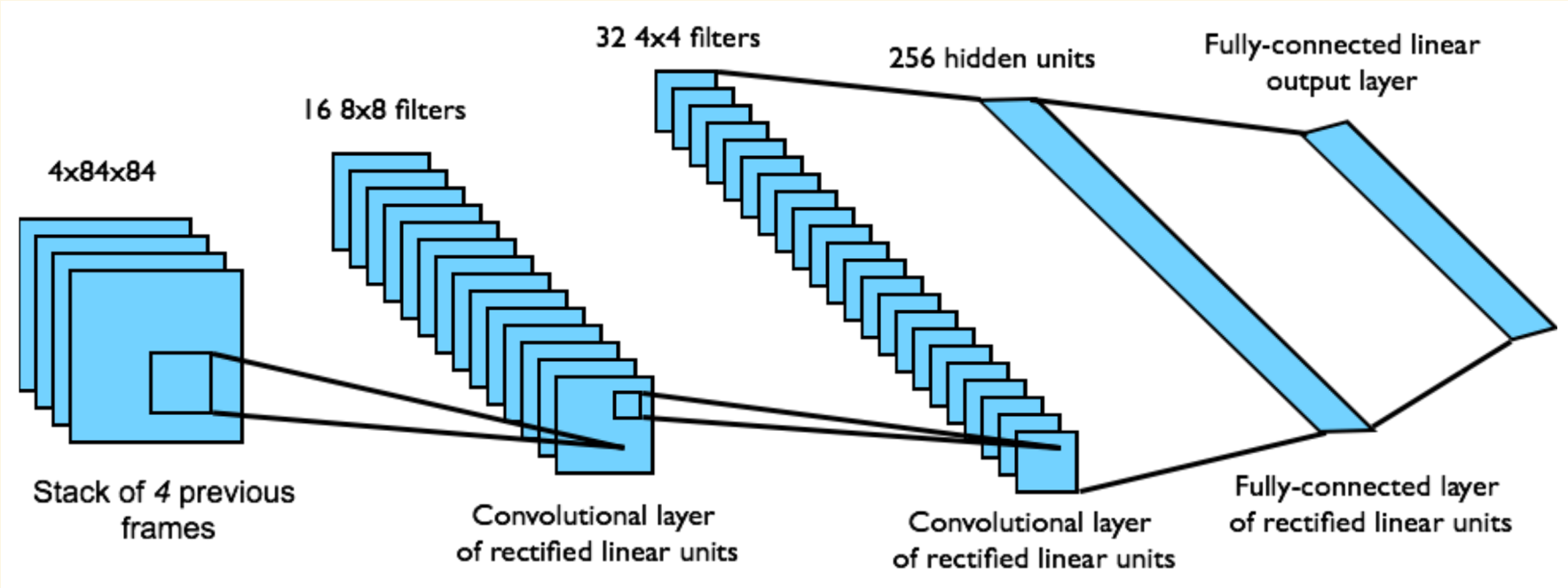
- Experience replay *decorrelates* trajectories
- Fixed Q-targets: Keep *two* different networks
  - use “frozen” target for target updates
  - switch the two networks around after a fixed number of steps (e.g. 1000 steps)
  - this helps to stabilise convergence to non-linearity

DQN in Atari (model-free learning from images)



End-to-end learning of values $Q(s, a)$ from pixels s.

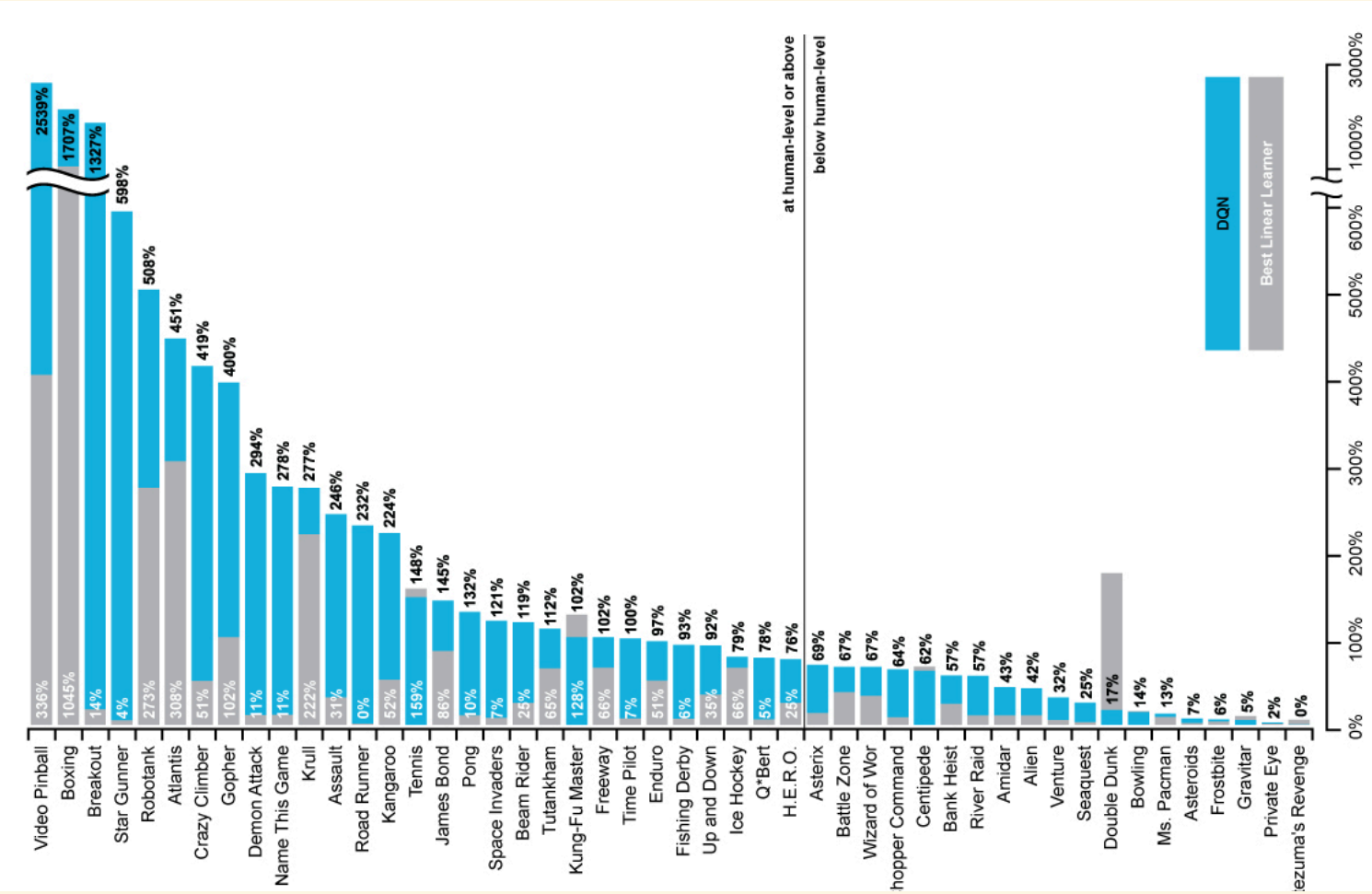
- Input: stack of raw pixels (last 4 frames)
- Output: $Q(s, a)$ for 18 joystick/button positions
- Reward: change in score for that step



Network architecture + hyper-parameters fixed across all games

- We can *backpropagate* loss through parameters by considering a single recursive non-linear function of all layers, i.e. $f(\mathbf{x}, \mathbf{w}) = fL(fL{-}1(\ldots f1(\mathbf{x}, \mathbf{w}1)\ldots \mathbf{w}L{-}1), \mathbf{w}L)$
- We can also do this for *tensor networks*, including attention-based transformers, which use matrices instead of vectors.

DQN Results in Atari Games - Above human level (left) & below human level (right)



- For details see paper Volodymyr Mnih et al., Human-level control through deep reinforcement learning by deep reinforcement learning, *Nature*, pp, 529–539, 2015

How much does DQN help?

Game	Replay + Fixed-Q	Replay + Q-learning	No replay + Fixed-Q	No replay + Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.30	831.25	141.89	29.10
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.40	822.55	1003.00	275.81
Space Invaders	1088.94	826.33	373.22	301.99

Linear Least Squares Methods

Linear Least Squares Prediction

Experience replay finds the least squares solution

- But may take many iterations.

Using the special case of *linear value function approximation*

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

We can solve the least squares solution directly using a closed form

Linear Least Squares Prediction (2)

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

$$\mathbb{E}_{\mathcal{D}}\left[\Delta\mathbf{w}\right] = 0 \quad \text{(want update zero across data set)}$$

$$\alpha \sum_{t=1}^{T} \mathbf{x}(st)\left(v_t^{\pi} - \mathbf{x}(st)^{\top}\mathbf{w}\right) = 0 \quad \text{(unwrapping expected updates)}$$

$$\sum_{t=1}^{T} \mathbf{x}(st) v_t^{\pi} = \sum_{t=1}^{T} \mathbf{x}(st)\mathbf{x}(st)^{\top}\mathbf{w}$$

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

Compute time: $O(N^3)$ for N features (direct),
or **incremental** $O(N^2)$ using Sherman–Morrison.

- Sometimes better or cheaper than experience replay, e.g. if have only a few features for the *linear* cases.

Linear Least Squares Prediction Algorithms

We do not know true values v_t^π

- 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 $v_t^\pi \approx R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$
LSTD(λ)	Least Squares TD(λ) uses λ-return $v_t^\pi \approx G_t^\lambda$

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

Linear Least Squares Prediction Algorithms (2)

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

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

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

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

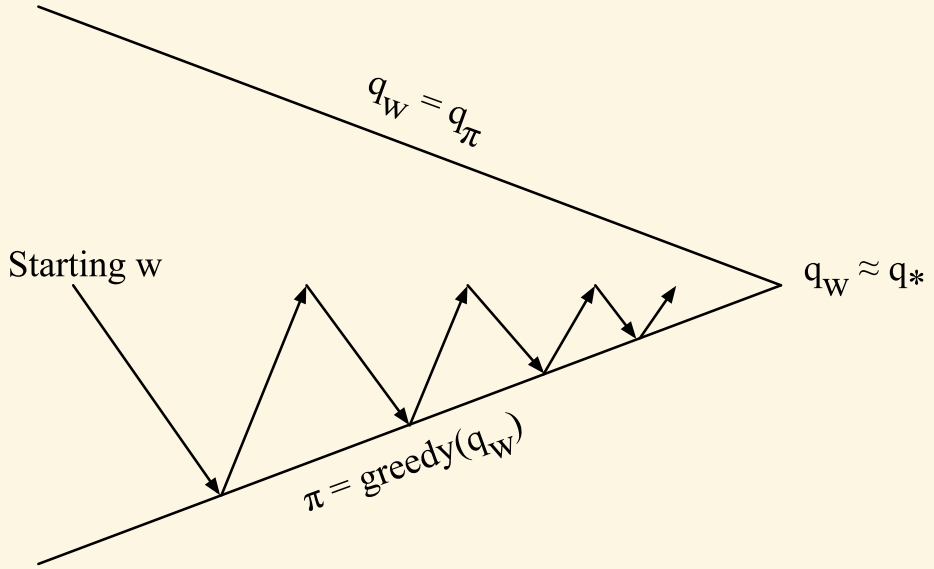
LSTD(λ)
$$0 = \sum_{t=1}^{T} \alpha\, \delta_t\, E_t,$$

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

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 π
- consisting of $\langle (state, action), value \rangle$ pairs

$$\mathcal{D} = \left\{ \langle (s1, a1), v_1^\pi \rangle, \langle (s2, a2), v_2^\pi \rangle, \ldots, \langle (sT, aT), v_T^\pi \rangle \right\}$$

Least Squares Control

For *policy evaluation*, we want to efficiently use *all experience*

For *control*, we also want to improve the policy

The 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:

- Experience from old policy: $(S_t, A_t, R_{t+1}, S_{t+1}) \sim \pi_{\text{old}}$
- Consider alternative successor action $A' = \pi_{\text{new}}(S_{t+1})$
- Update $\hat{q}(S_t, A_t, \mathbf{w})$ toward

$$R_{t+1} + \gamma\, \hat{q}(S_{t+1}, A', \mathbf{w})$$

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

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

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

Least Squares Policy Iteration Algorithm

The following pseudo-code uses LSTDQ for policy evaluation

It repeatedly re-evaluates experience $\mathcal{D}$ with different policies

Least Squares Policy Iteration Algorithm (LSTFQ variant)

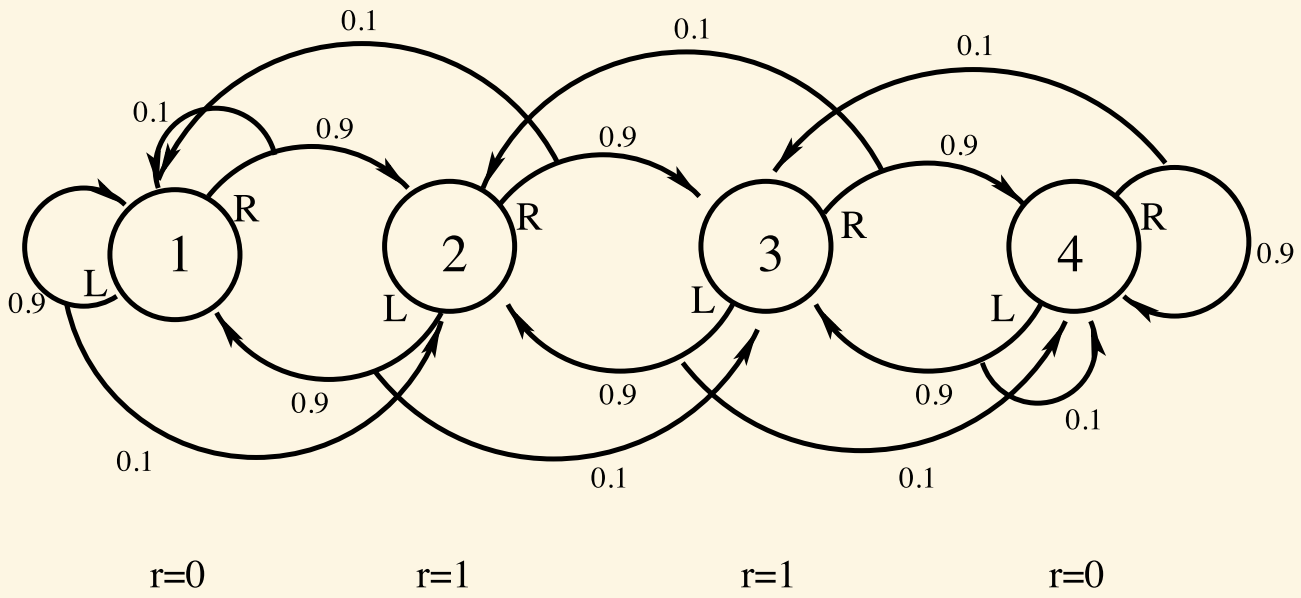
$$
\begin{array}{l}
\text{function LSPI-TD}(\mathcal{D}, \pi_0) \\
\quad \pi' \leftarrow \pi_0 \\
\quad \text{repeat} \\
\quad\quad \pi \leftarrow \pi' \\
\quad\quad Q \leftarrow \text{LSTDQ}(\pi, \mathcal{D}) \\
\quad\quad \text{for all } s \in S \text{ do} \\
\quad\quad\quad \pi'(s) \leftarrow \arg\max_{a \in A} Q(s, a) \\
\quad\quad \text{end for} \\
\quad \text{until } (\pi \approx \pi') \\
\quad \text{return } \pi \\
\text{end function}
\end{array}
$$

Convergence of Control Algorithms

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

(✓) = chatters around near-optimal value function.

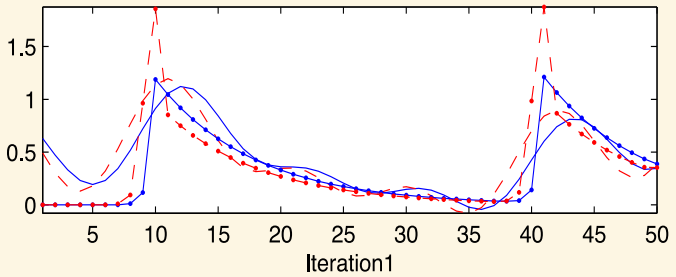
Chain Walk Example (More complicated random walk)

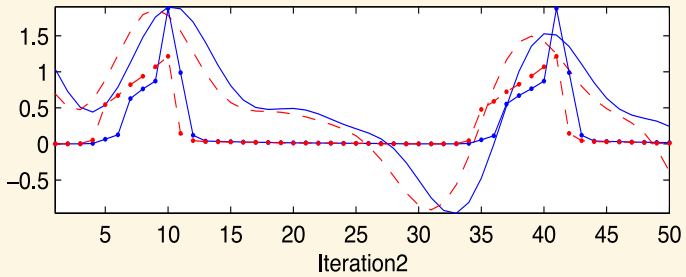


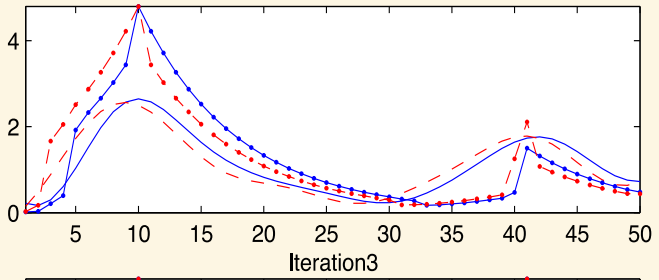
Consider the 50 state version of this problem (bigger replica of this diagram)

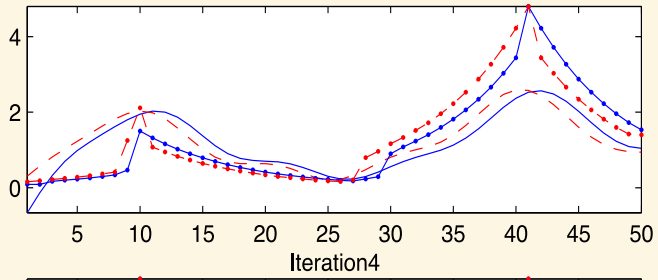
- Reward $+1$ in states 10 and 41, 0 elsewhere
- Optimal policy: R $(1-9)$, L $(10-25)$, R $(26-41)$, L $(42, 50)$
- Features: 10 evenly spaced Gaussians ($\sigma = 4$) for each action
- Experience: $10,000$ steps from random walk policy

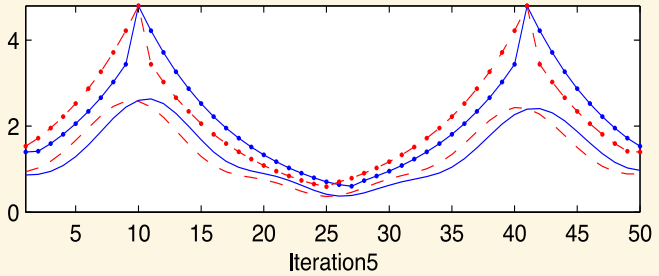
LSPI in Chain Walk: Action-Value Function

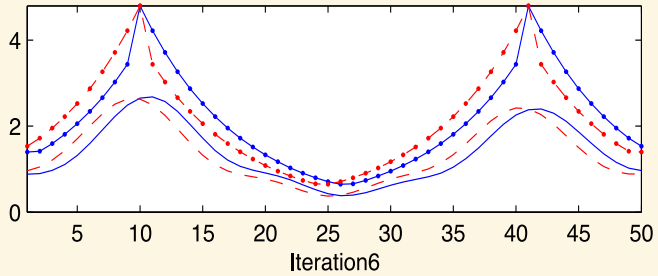


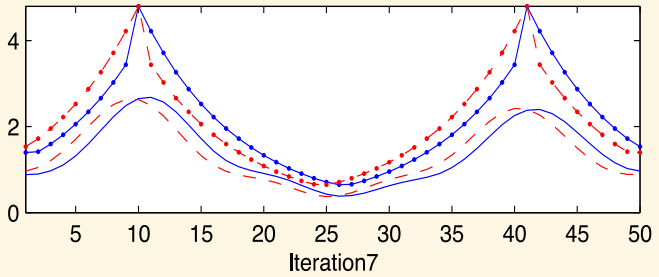








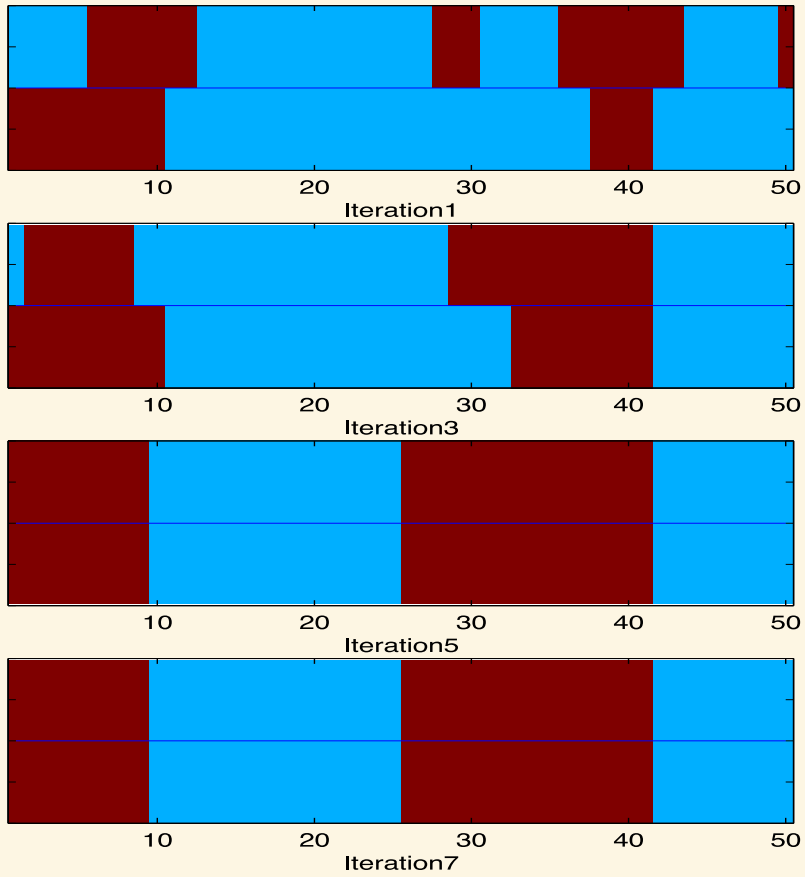


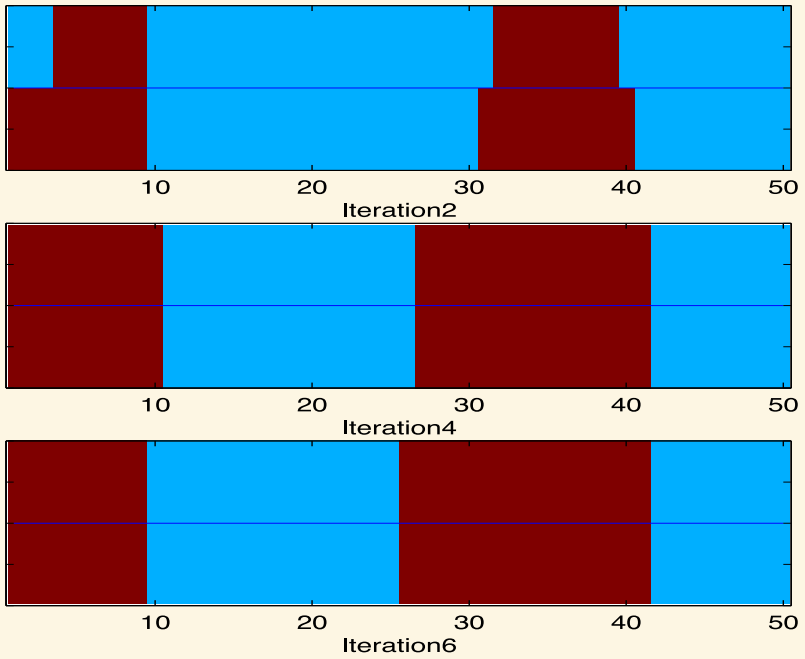


Plots show LSPI iterations on a 50-state chain with a radial basis function approximator.

- The colours represent two different actions of going left (blue) or right (red)
- The plots show the true value function (dashed) or approximate value function
- You can see it converges to the optimal policy after only 7 iterations.

LSPI in Chain Walk: Policy





Plots of policy improvement over LSPI iterations (using same 50-state chain example).

- Shows convergence of LSPI policy to optimal within a few iterations.