

Reinforcement Learning with Function Approximation: Value-based Methods

Doina Precup

McGill University and Google DeepMind Montreal

dprecup@cs.mcgill.ca

With thanks to Rich Sutton and the Reasoning & Learning Lab, McGill

Dedicated to my father, Prof. Univ. Stefan-Ioan Nitchi (1944-2014)

Overview

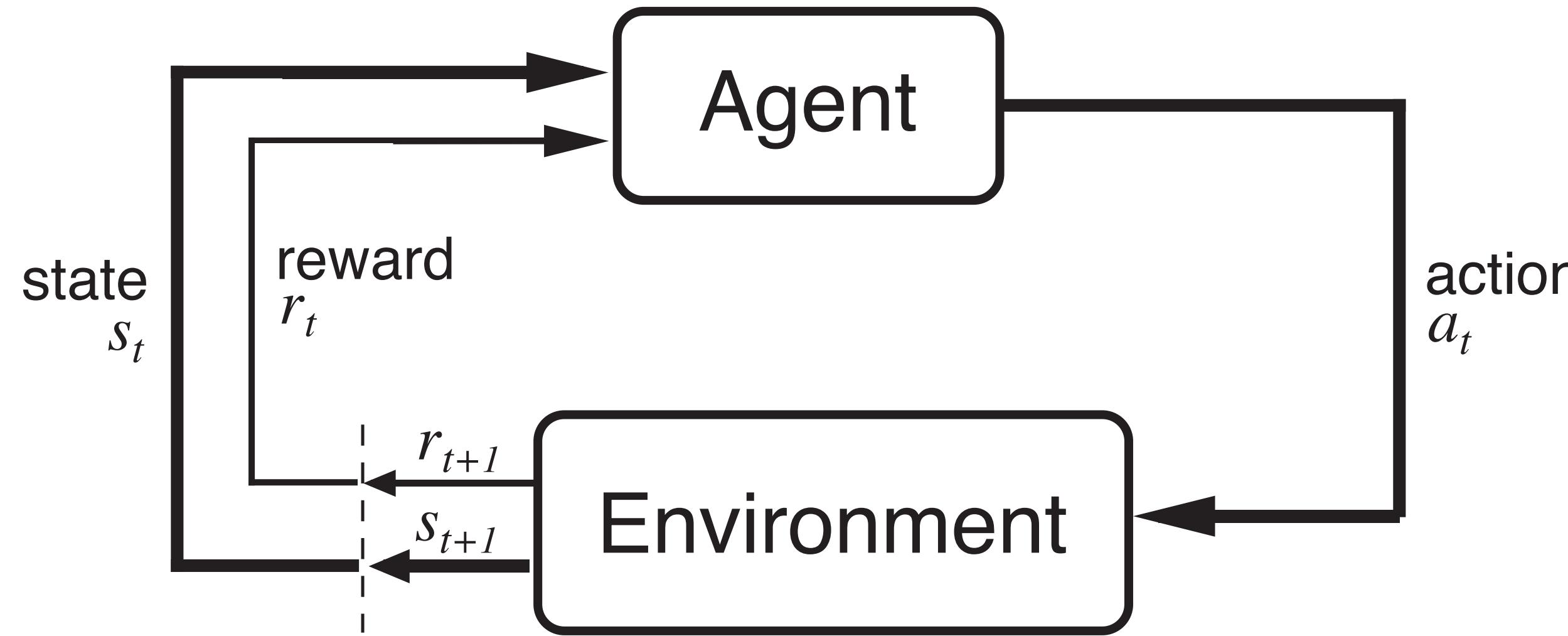
- Recap and some extra information on previously seen RL methods
- Value-based methods with function approximation for on-policy and off-policy learning
- If we have time: Temporal abstraction / hierarchical reinforcement learning

Reinforcement learning with function approximation

I. Nicholas's talk: *Policy-gradient methods*

- learn the parameters of a stochastic policy
- update by gradient ascent in performance
- includes *actor-critic methods*, which learn *both* value and policy parameters
- Today: *Value-based methods - learn the value function and use it to inform action choices*

Notation



- Notation as in Sutton & Barto: s for state, a for action
- Reward R , discounted return G
- Estimate value functions v, q
- We look at both policy evaluation as well as control

Value-based methods

- The *value of an action in a state given a policy* is the expected future reward starting from the state taking that first action, then following the policy thereafter

$$q_{\pi}(s, a) = \mathbb{E} \left[\sum_{t=1}^{\infty} \gamma^{t-1} R_t \middle| S_0 = s, A_0 = a \right]$$

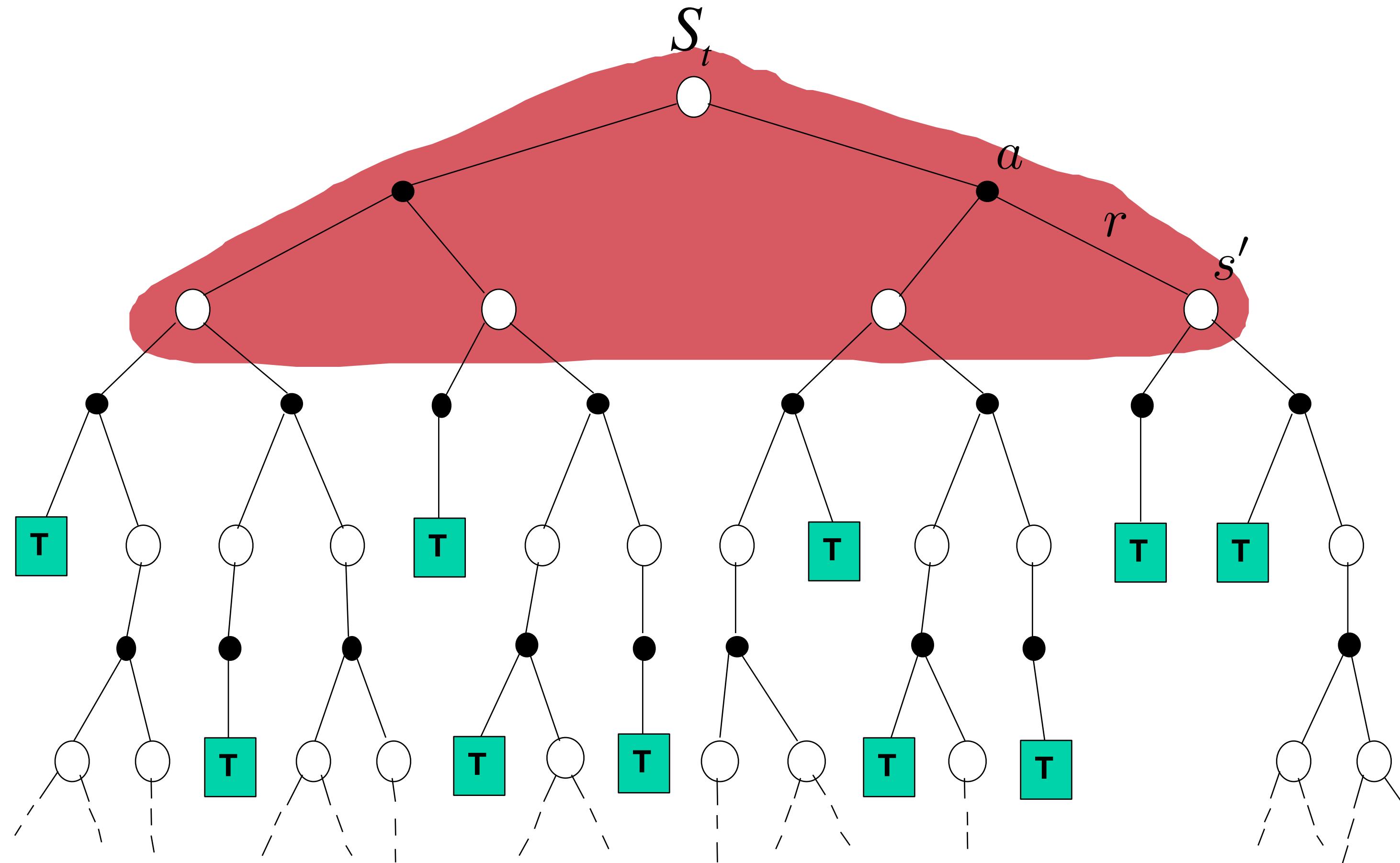
- Policy: pick the max most of the time

$$A_t = \arg \max_a \hat{Q}_t(S_t, a)$$

but sometimes pick at random (ε -greedy)

cf. Dynamic Programming

$$V(S_t) \leftarrow E_{\pi} [R_{t+1} + \gamma V(S_{t+1})] = \sum_a \pi(a|S_t) \sum_{s', r} p(s', r|S_t, a)[r + \gamma V(s')]$$



Curse of dimensionality



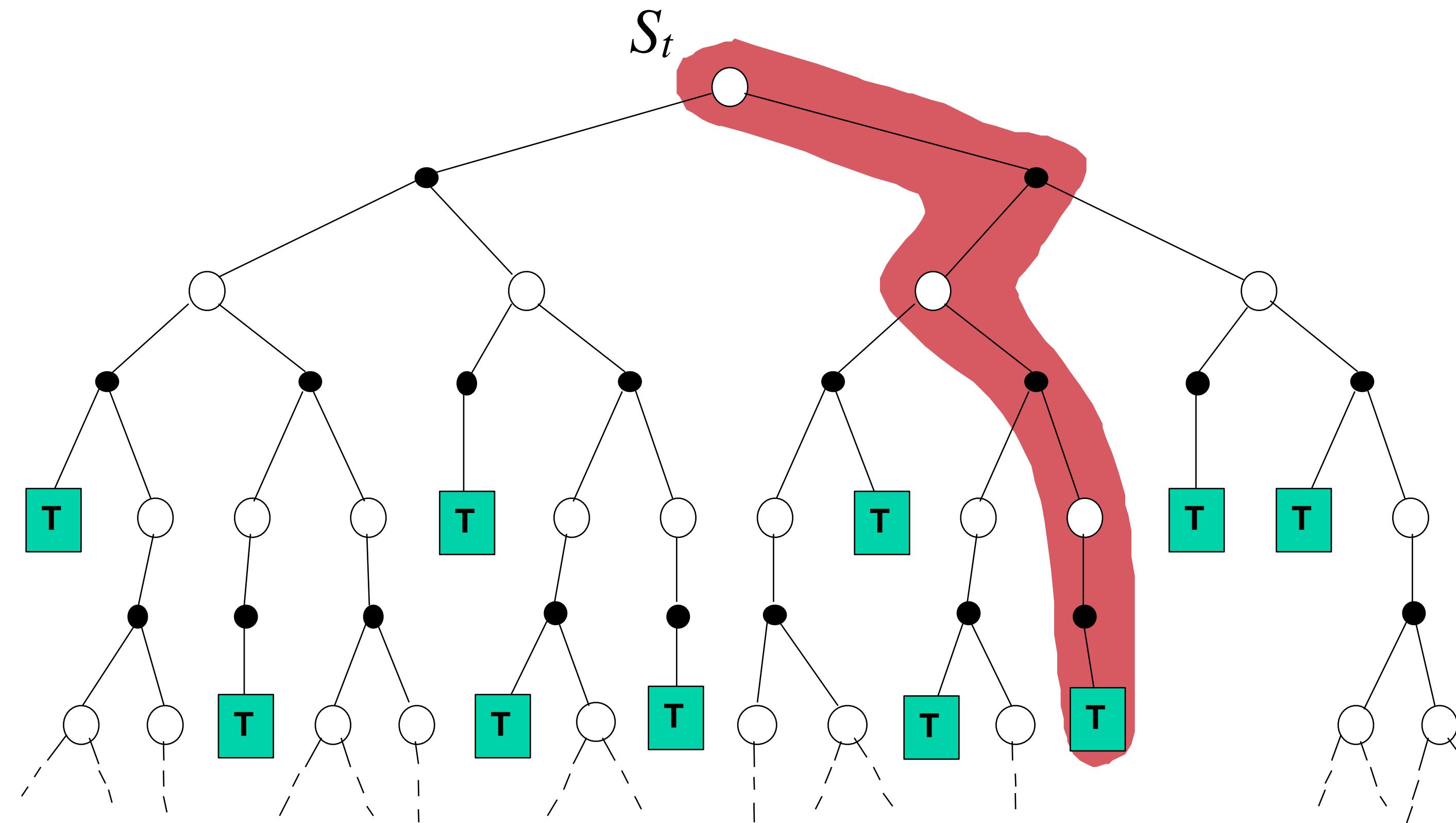
- Values are governed by nice recursive equations:

$$V_{k+1}(s) \leftarrow \max_{a \in A} \left(r_{ss'}^a + \gamma \sum_{s' \in S} p_{ss'}^a V_k(s') \right), \forall s \in S$$

- The number of states grows *exponentially* with the number of state variables (the dimensionality of the problem)
E.g. in Go, there are 10^{170} states
- The *action set* may also be very large or continuous
E.g. in Go, branching factor is ≈ 100 actions
- The solution may require *chaining many steps*
E.g. in Go games take ≈ 200 actions

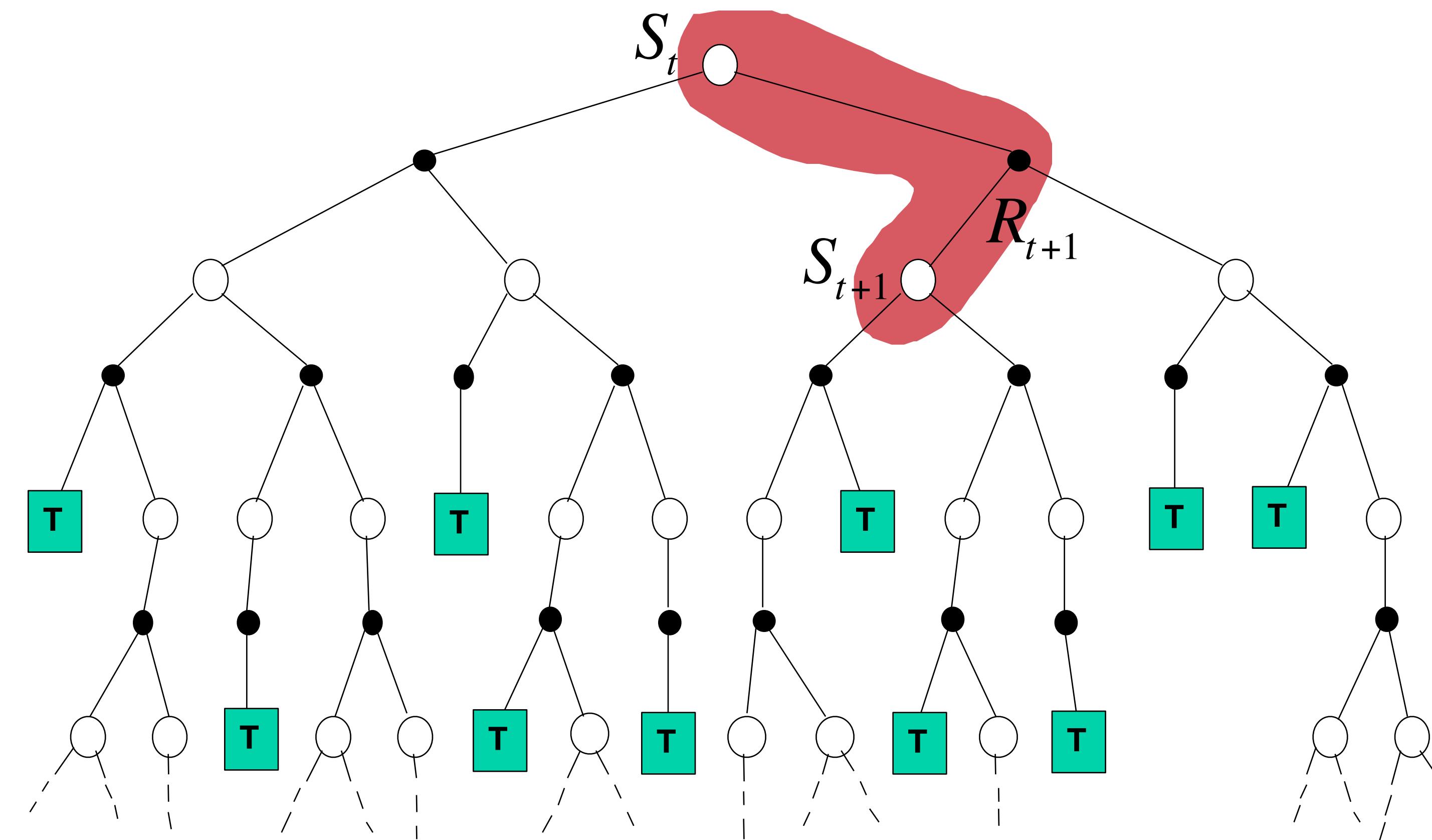
Simple Monte Carlo

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$$



Simplest TD Method

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$



TD methods bootstrap and sample

- **Bootstrapping:** update involves an *estimate*
 - MC does not bootstrap
 - DP bootstraps
 - TD bootstraps
- **Sampling:** update does not involve an *expected value*
 - MC samples
 - DP does not sample
 - TD samples

TD Prediction

Policy Evaluation (the prediction problem):

for a given policy π , compute the state-value function v_π

Recall: Simple every-visit Monte Carlo method:

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$$

target: the actual return after time t

The simplest temporal-difference method TD(0):

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

target: an estimate of the return

You are the Predictor

Suppose you observe the following 8 episodes:

A, 0, B, 0

B, 1

B, 1

$V(B) ?$

B, 1

$V(A) ?$

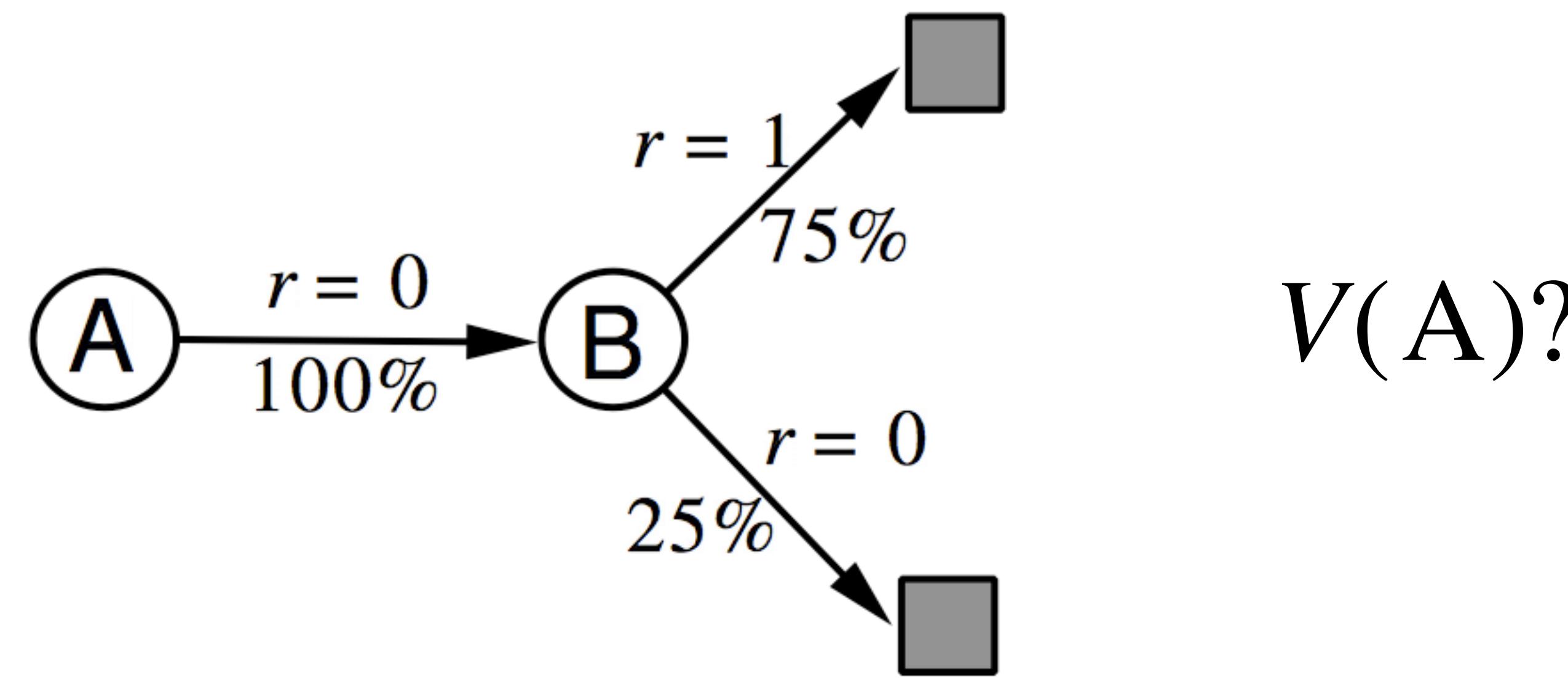
B, 1

B, 1

B, 0

Assume Markov states, no discounting ($\gamma = 1$)

You are the Predictor

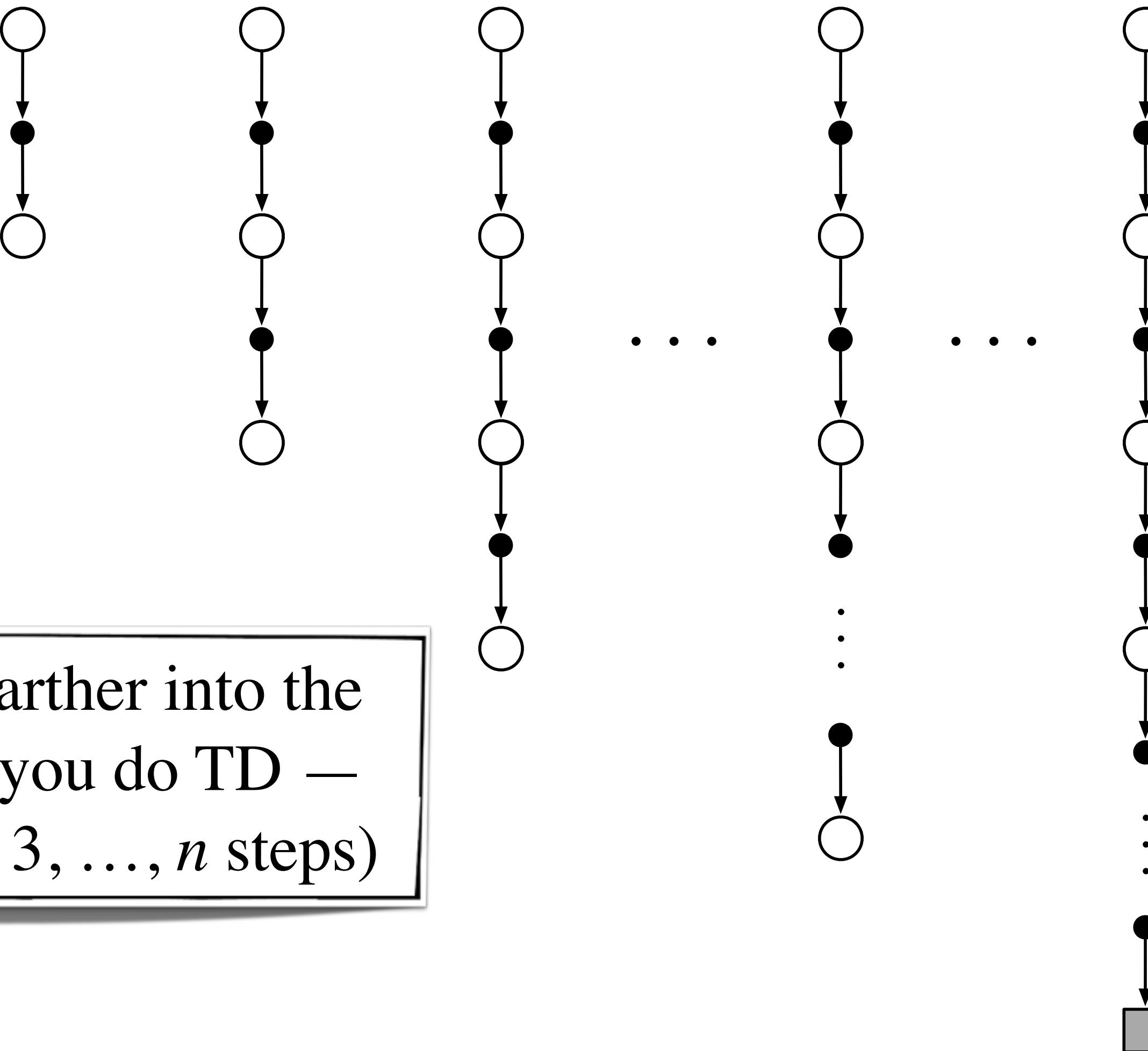


You are the Predictor

- The prediction that best matches the training data is $V(A)=0$
 - This **minimizes the mean-square-error** on the training set
 - This is what a batch Monte Carlo method gets
- If we consider the sequentiality of the problem, then we would set $V(A)=.75$
 - This is correct for the **maximum likelihood** estimate of a Markov model generating the data
 - i.e, if we do a best fit Markov model, and assume it is exactly correct, and then compute what it predicts (how?)
 - This is called the **certainty-equivalence estimate**
 - This is what TD gets

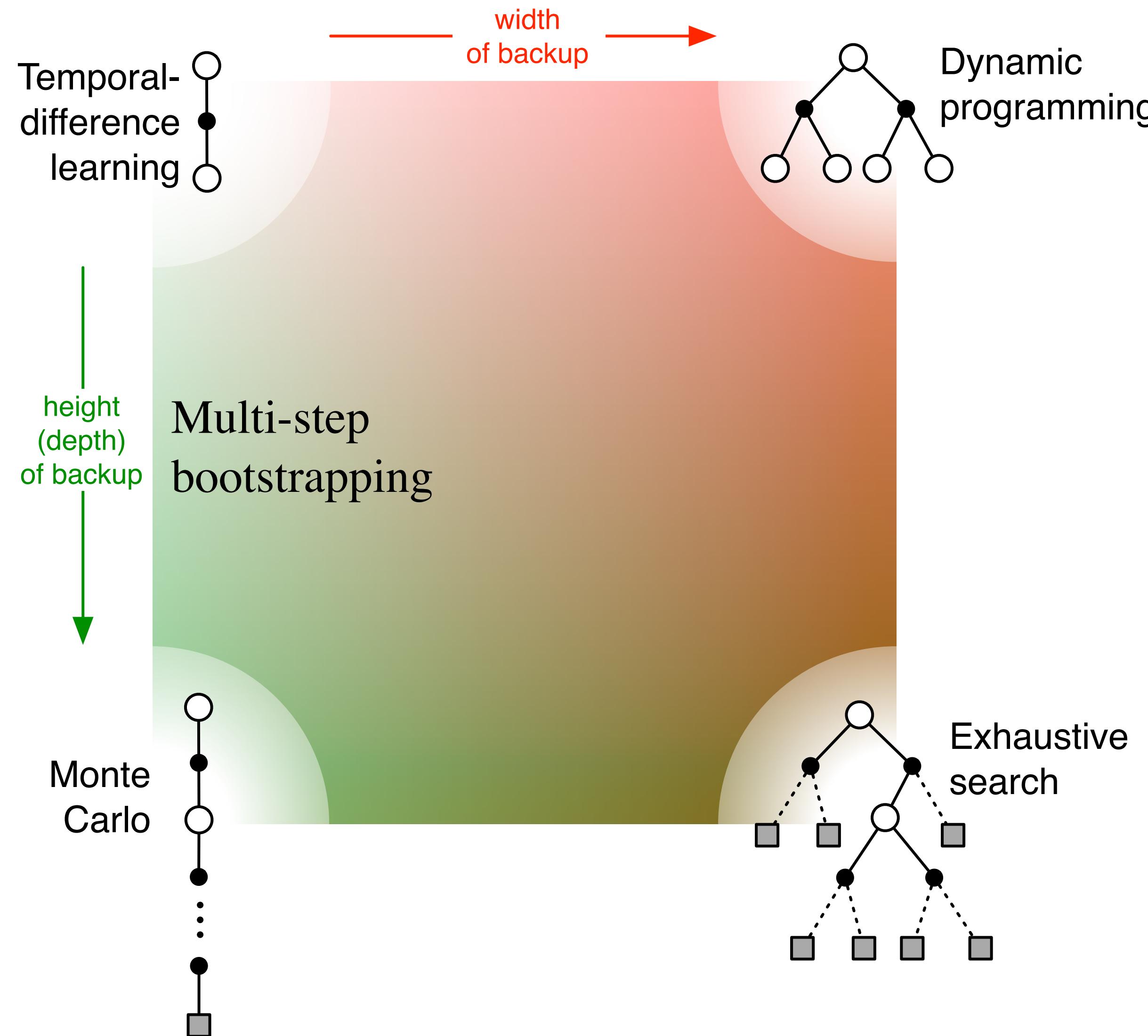
n -step TD Prediction

1-step TD
and TD(0) 2-step TD 3-step TD n -step TD ∞ -step TD
and Monte Carlo



Idea: Look farther into the future when you do TD — backup (1, 2, 3, ..., n steps)

Unified View



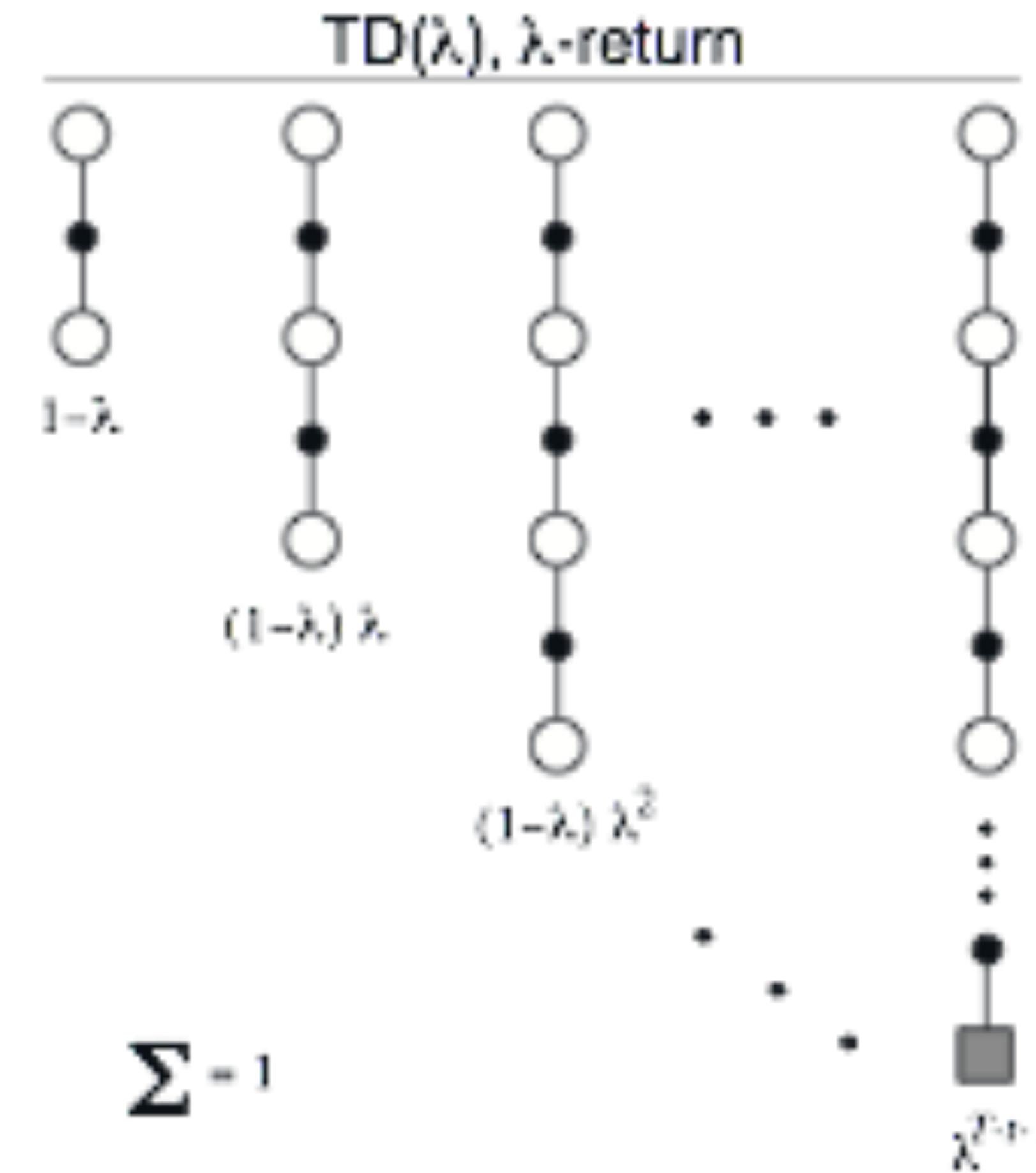
Mathematics of n -step TD Returns/Targets

- Monte Carlo: $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$
- TD: $G_t^{(1)} \doteq R_{t+1} + \gamma V_t(S_{t+1})$
 - Use V_t to estimate remaining return
- n -step TD:
 - 2 step return: $G_t^{(2)} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_t(S_{t+2})$
 - n -step return: $G_t^{(n)} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_t(S_{t+n})$
with $G_t^{(n)} \doteq G_t$ if $t + n \geq T$

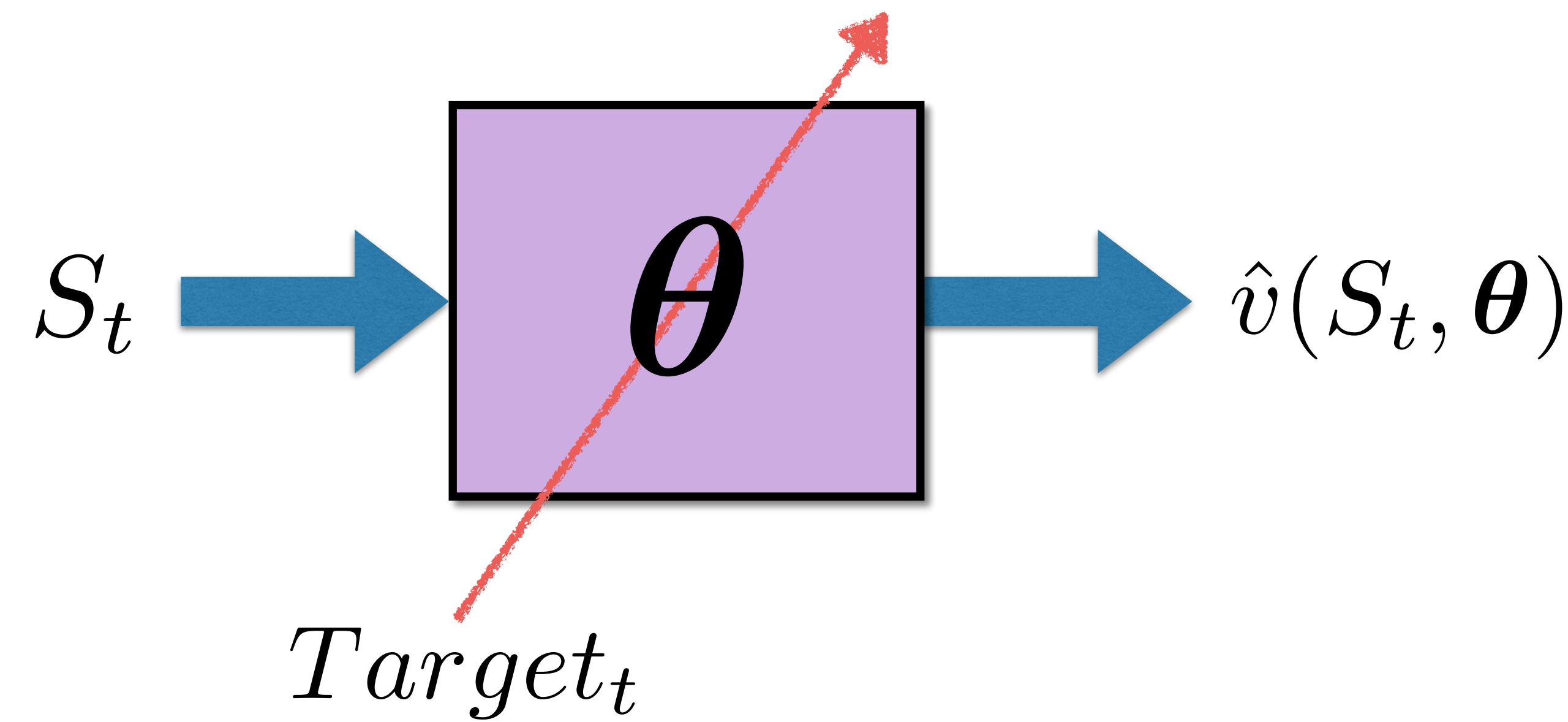
The λ -return is a compound update target

- The λ -return is a target that averages all n -step targets
 - each weighted by λ^{n-1}

$$G_t^\lambda \doteq (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$



Value function approximation (VFA) replaces the table with a general parameterized form



Target depends on the agent's behavior, and in TD, also on its current estimates!

Stochastic Gradient Descent (SGD) is the idea behind most approximate learning

General SGD:	$\theta \leftarrow \theta - \alpha \nabla_{\theta} Error_t^2$
For VFA:	$\leftarrow \theta - \alpha \nabla_{\theta} [Target_t - \hat{v}(S_t, \theta)]^2$
Chain rule:	$\leftarrow \theta - 2\alpha [Target_t - \hat{v}(S_t, \theta)] \nabla_{\theta} [Target_t - \hat{v}(S_t, \theta)]$
Semi-gradient:	$\leftarrow \theta + \alpha [Target_t - \hat{v}(S_t, \theta)] \nabla_{\theta} \hat{v}(S_t, \theta)$
Linear case:	$\leftarrow \theta + \alpha [Target_t - \hat{v}(S_t, \theta)] \phi(S_t)$
Action-value form:	$\theta \leftarrow \theta + \alpha [Target_t - \hat{q}(S_t, A_t, \theta)] \phi(S_t, A_t)$

A natural objective in VFA
is to minimize the Mean Square Value Error

$$\text{MSVE}(\boldsymbol{\theta}) \doteq \sum_{s \in \mathcal{S}} d(s) \left[v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}) \right]^2$$

where $d(s)$ is the fraction of time steps spent in state s

True SGD will converge to a local minimum of the error objective
In *linear* VFA, there is only one minimum: local=global
Monte Carlo will provide samples of the expectation

State aggregation is the simplest kind of VFA

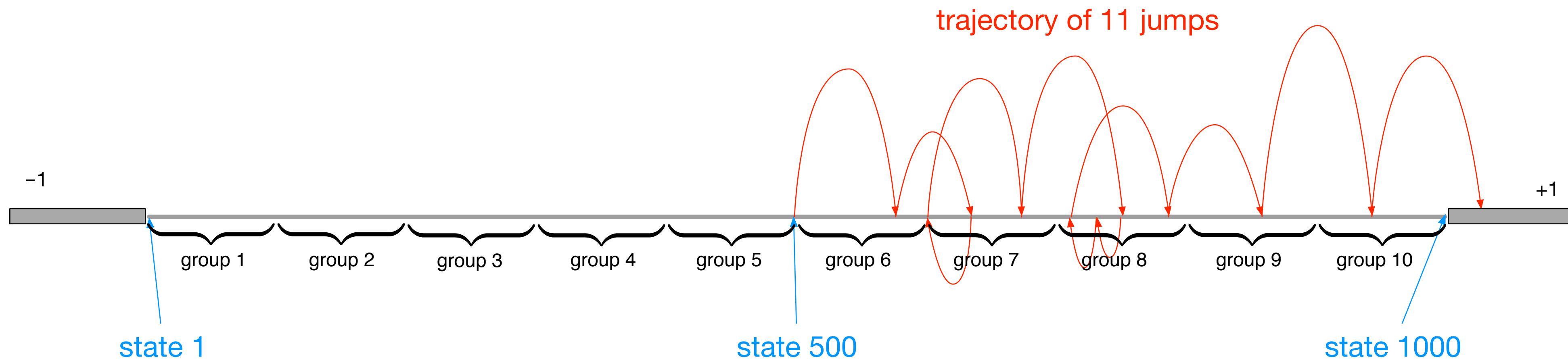
- States are partitioned into disjoint subsets (groups)
- One component of $\boldsymbol{\theta}$ is allocated to each group

$$\hat{v}(s, \boldsymbol{\theta}) \doteq \theta_{group(s)}$$

$$\nabla_{\boldsymbol{\theta}} \hat{v}(s, \boldsymbol{\theta}) \doteq [0, 0, \dots, 0, 1, 0, 0, \dots, 0]$$

Recall: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha [Target_t - \hat{v}(S_t, \boldsymbol{\theta})] \nabla_{\boldsymbol{\theta}} \hat{v}(S_t, \boldsymbol{\theta})$

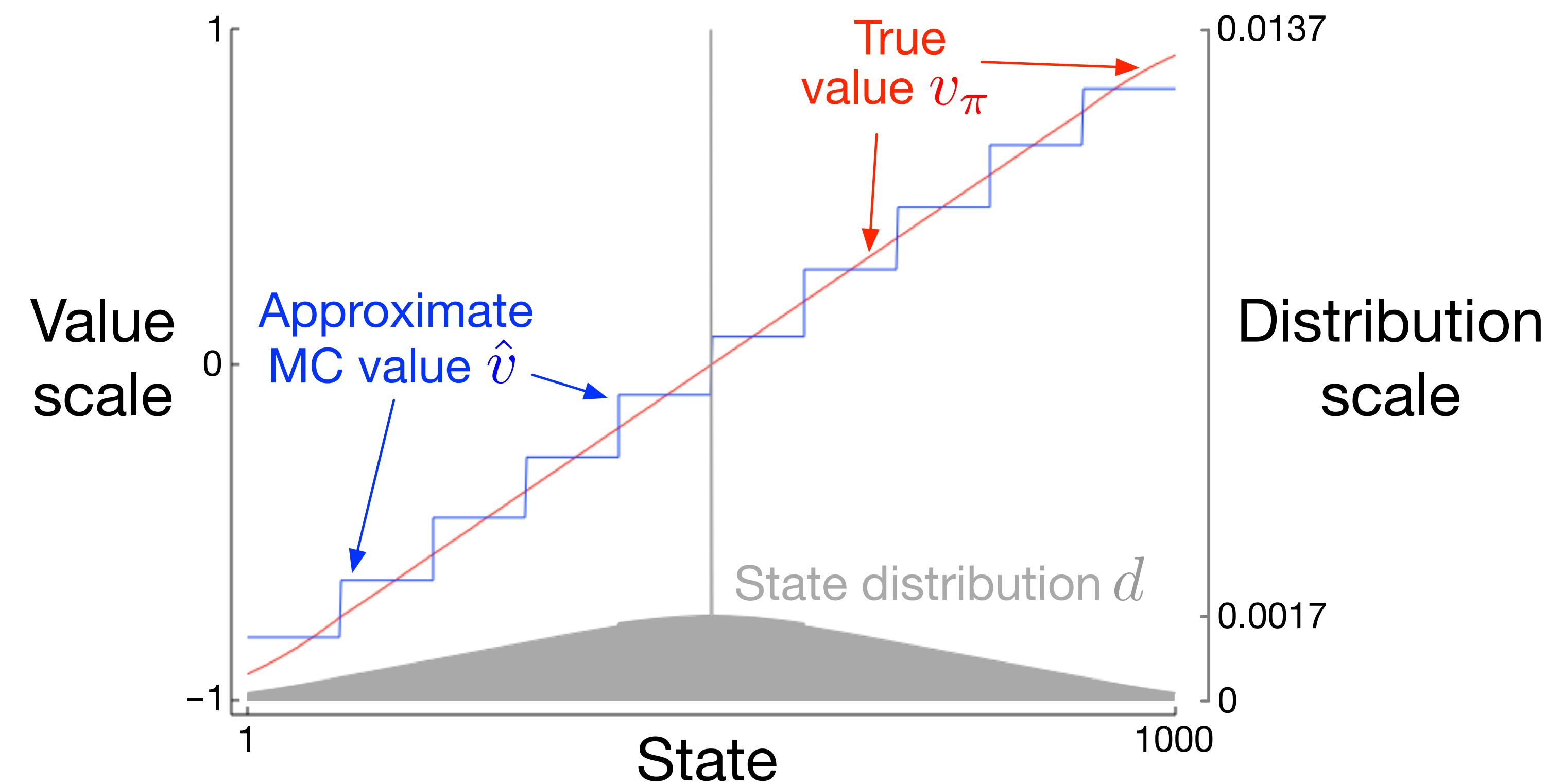
Example: Random walk with state aggregation



- States are numbered 1 to 1000 start in middle
- At each step, *jump* to one of the 100 states to the right, or to one of the 100 states to the left
- If the jump goes beyond 1 or 1000, terminates with a reward of -1 or $+1$ (otherwise $R_t=0$)
- States are aggregated into 10 bins (so only 10 values are maintained)

Gradient MC works well on the 1000-state random walk using *state aggregation*

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



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

Input: the policy π to be evaluated

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

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

Repeat (for each episode):

 Initialize S

 Repeat (for each step of episode):

 Choose $A \sim \pi(\cdot | S)$

 Take action A , observe R, S'

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

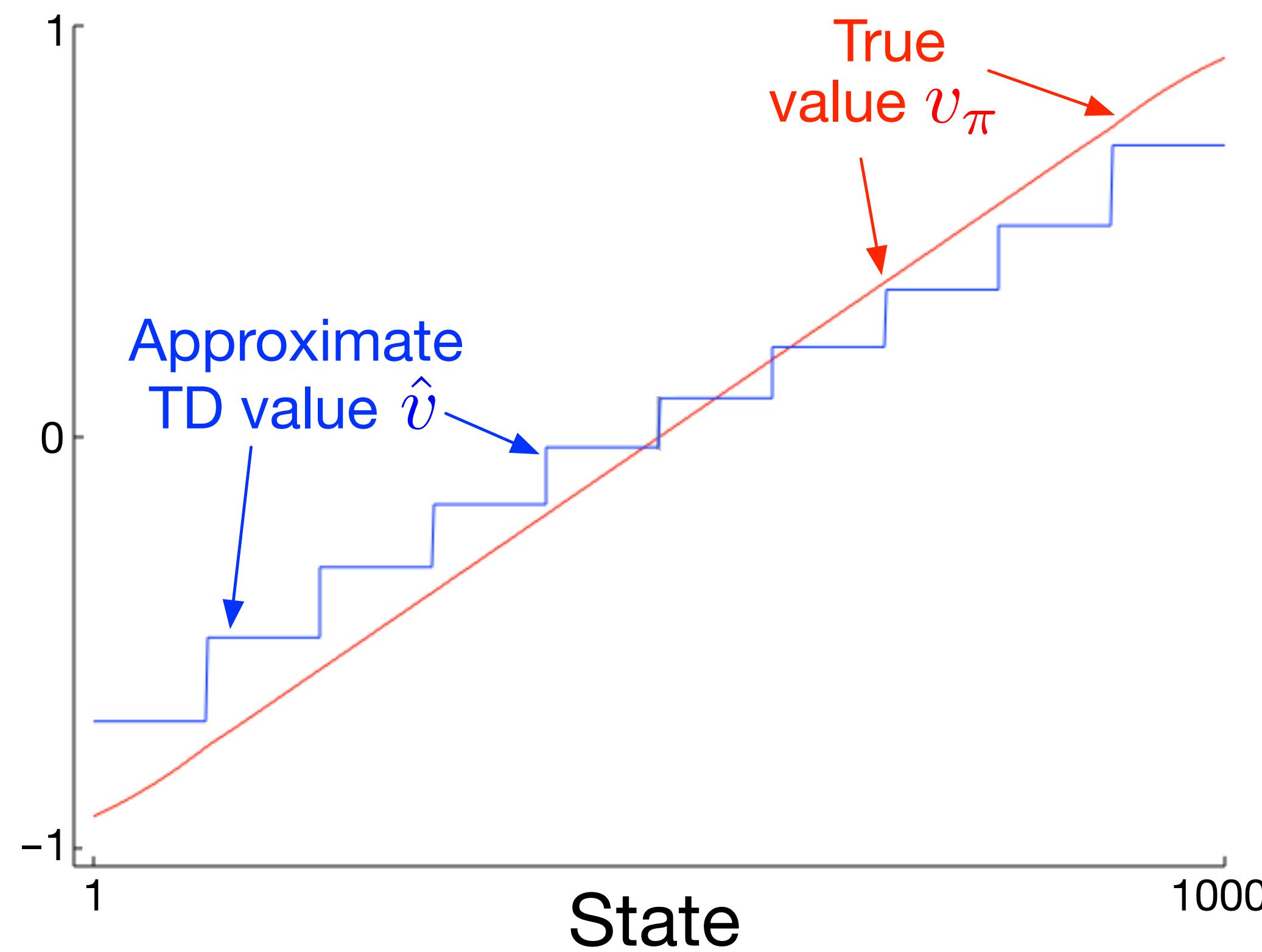
$S \leftarrow S'$

 until S' is terminal

Semi-gradient TD is less accurate than MC on the 1000-state random walk using state aggregation

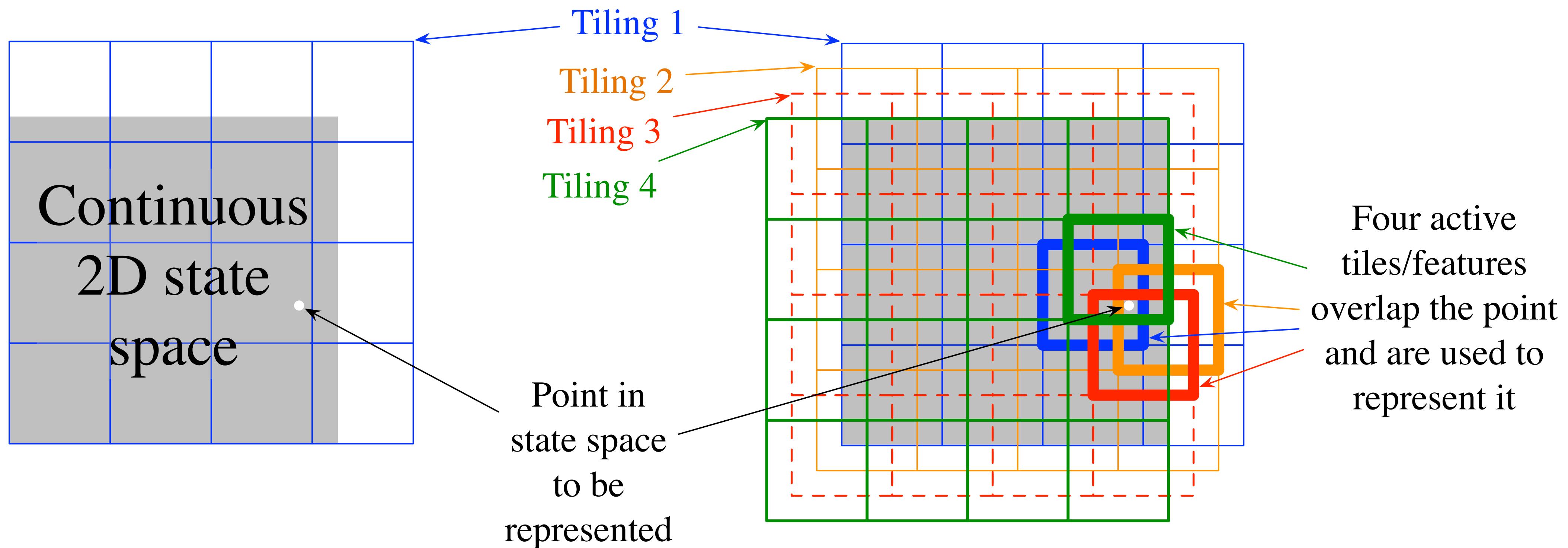
- 10 groups of 100 states
- after 100,000 episodes
- $\alpha = 2 \times 10^{-5}$

Relative values are
still pretty accurate



Tile coding is coarse coding, with rectangular receptive fields, controlled overlap

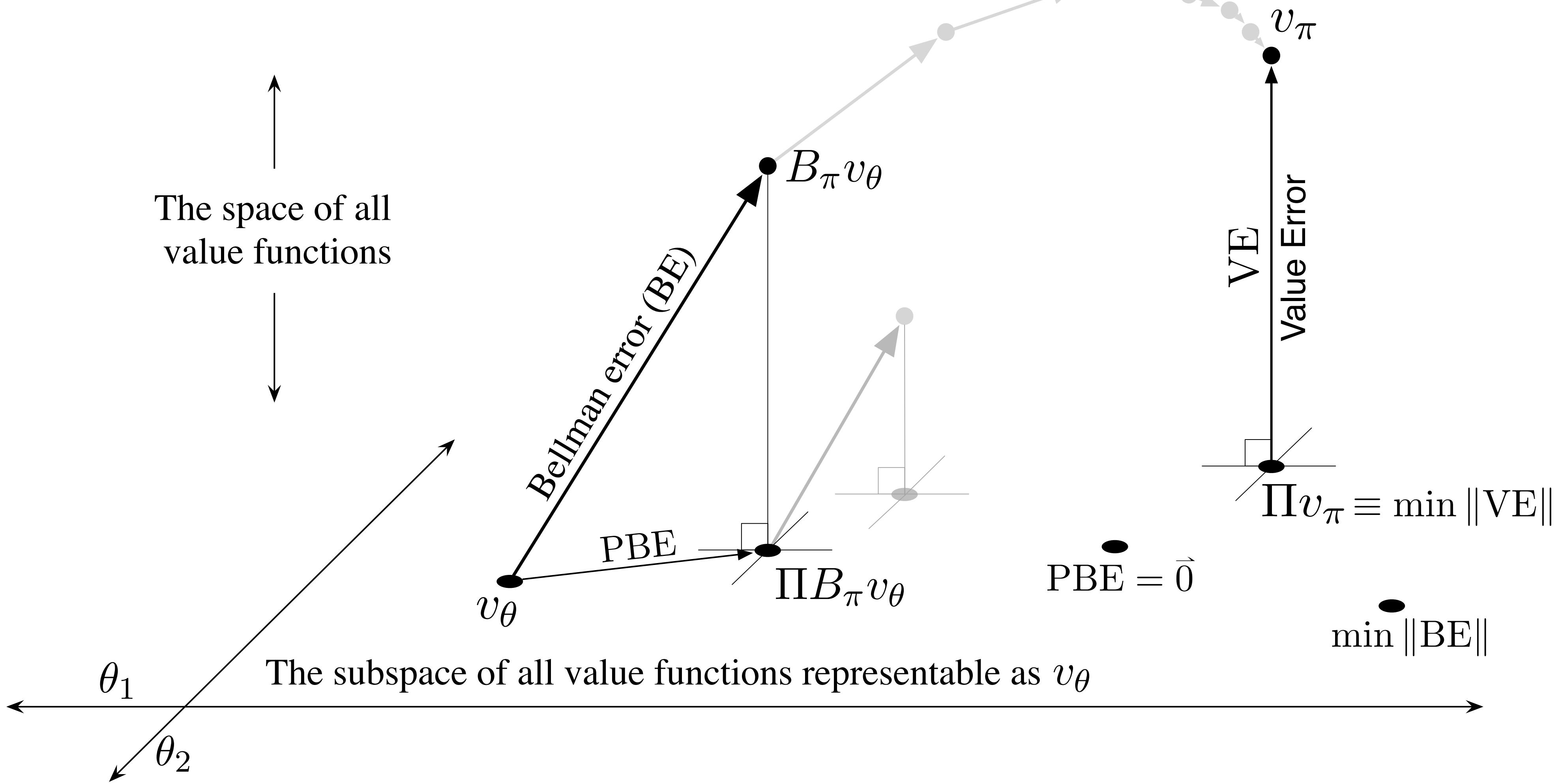
2D example



$v_\theta \doteq \hat{v}(\cdot, \theta)$ as a giant vector $\in \mathbb{R}^{|\mathcal{S}|}$

$$(B_\pi v)(s) \doteq \sum_{a \in \mathcal{A}} \pi(s, a) \left[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) v(s') \right]$$

Geometric intuition



TD converges to the TD fixedpoint, θ_{TD} ,
a biased but interesting answer

TD(0) update:

$$\begin{aligned}\theta_{t+1} &\doteq \theta_t + \alpha \left(R_{t+1} + \gamma \theta_t^\top \phi_{t+1} - \theta_t^\top \phi_t \right) \phi_t \\ &= \theta_t + \alpha \left(R_{t+1} \phi_t - \phi_t (\phi_t - \gamma \phi_{t+1})^\top \theta_t \right)\end{aligned}$$

Fixed-point analysis:

$$\begin{aligned}\mathbf{b} - \mathbf{A}\theta_{TD} &= 0 \\ \Rightarrow \quad \mathbf{b} &= \mathbf{A}\theta_{TD} \\ \Rightarrow \quad \theta_{TD} &\doteq \mathbf{A}^{-1}\mathbf{b}\end{aligned}$$

In expectation:

$$\mathbb{E}[\theta_{t+1} | \theta_t] = \theta_t + \alpha(\mathbf{b} - \mathbf{A}\theta_t),$$

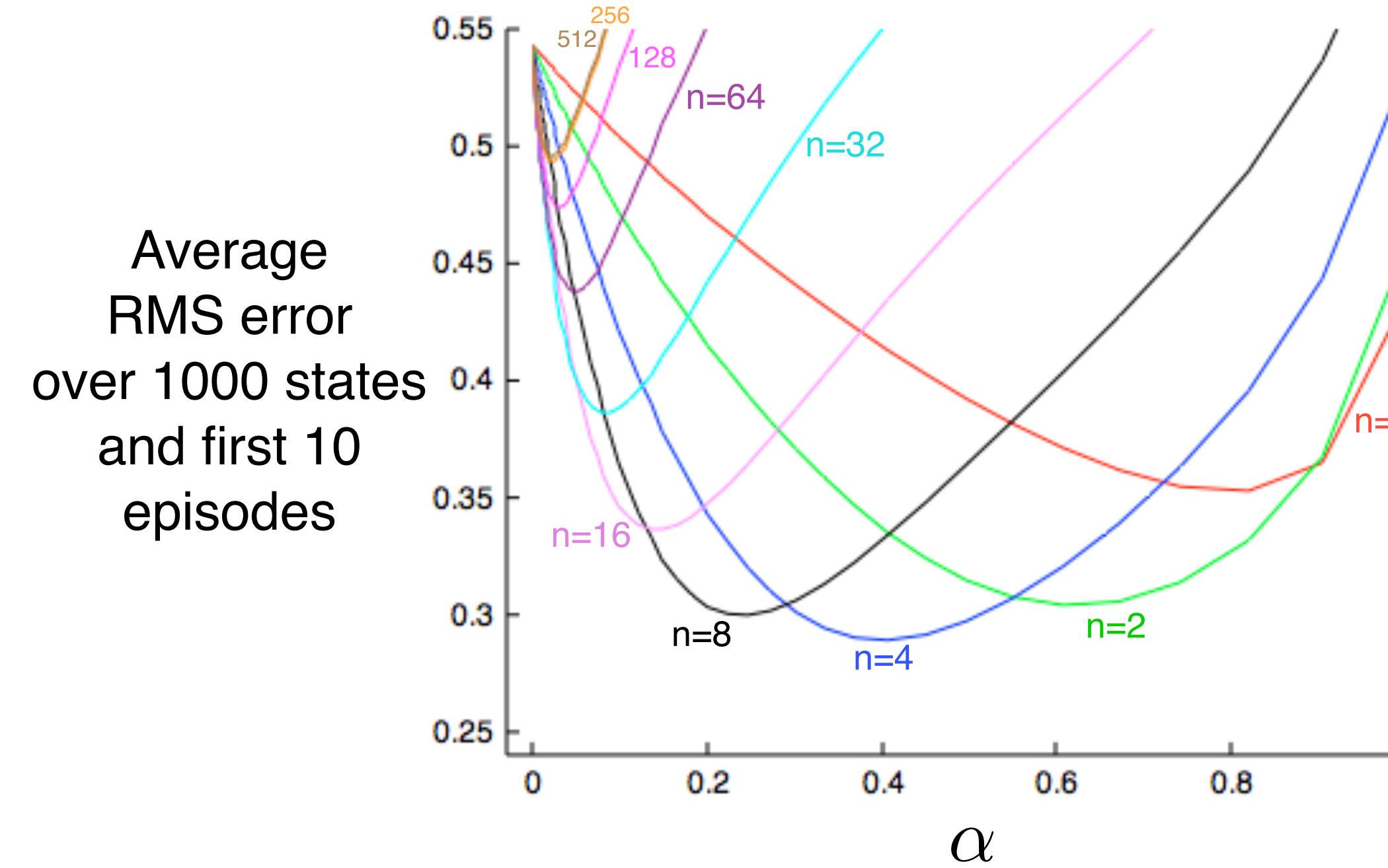
where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1} \phi_t] \in \mathbb{R}^n \quad \text{and} \quad \mathbf{A} \doteq \mathbb{E}[\phi_t (\phi_t - \gamma \phi_{t+1})^\top] \in \mathbb{R}^n \times \mathbb{R}^n$$

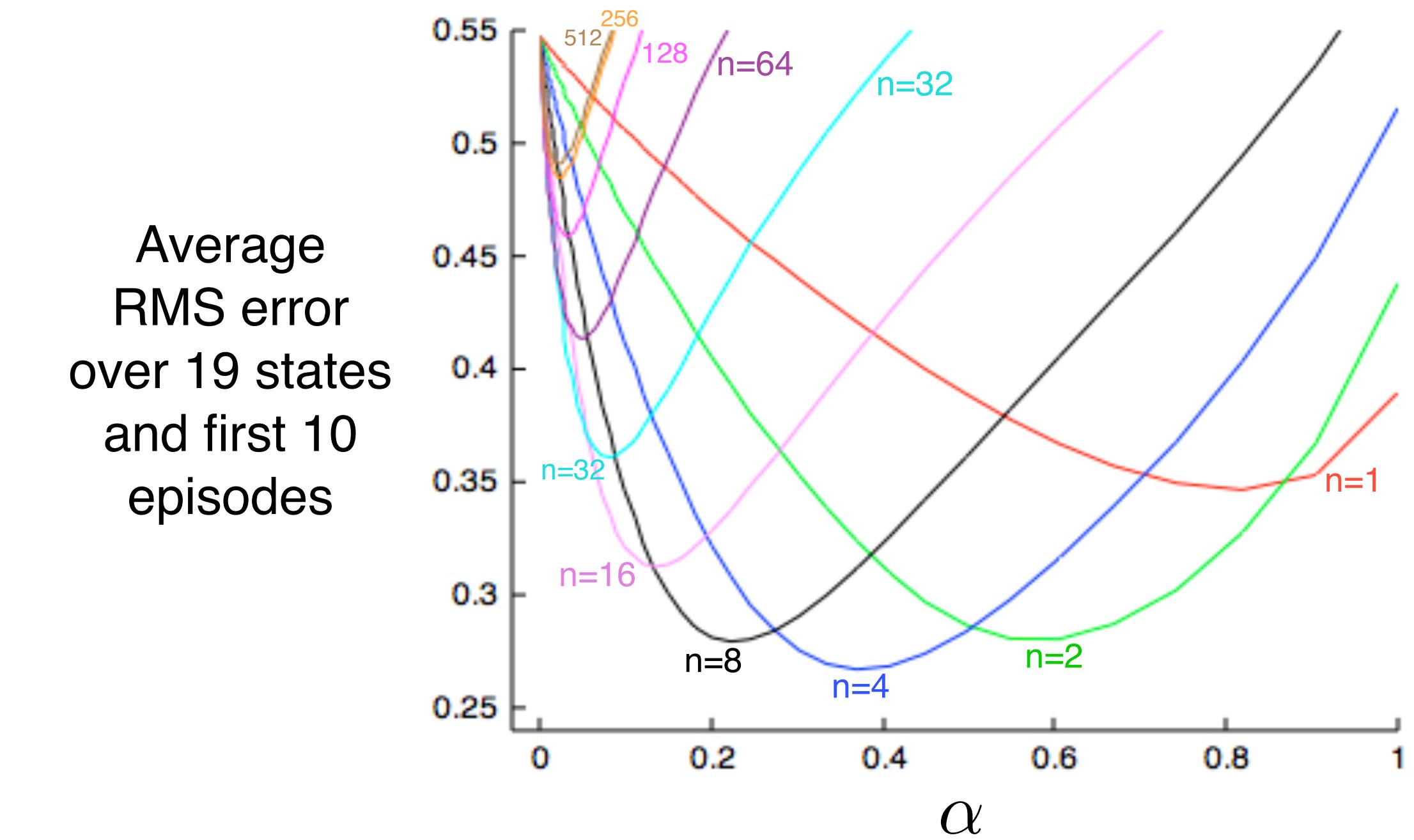
Guarantee:

$$\text{MSVE}(\theta_{TD}) \leq \frac{1}{1-\gamma} \min_{\theta} \text{MSVE}(\theta)$$

Bootstrapping greatly speeds learning very much like the tabular case



1000 states aggregated
into 20 groups of 50

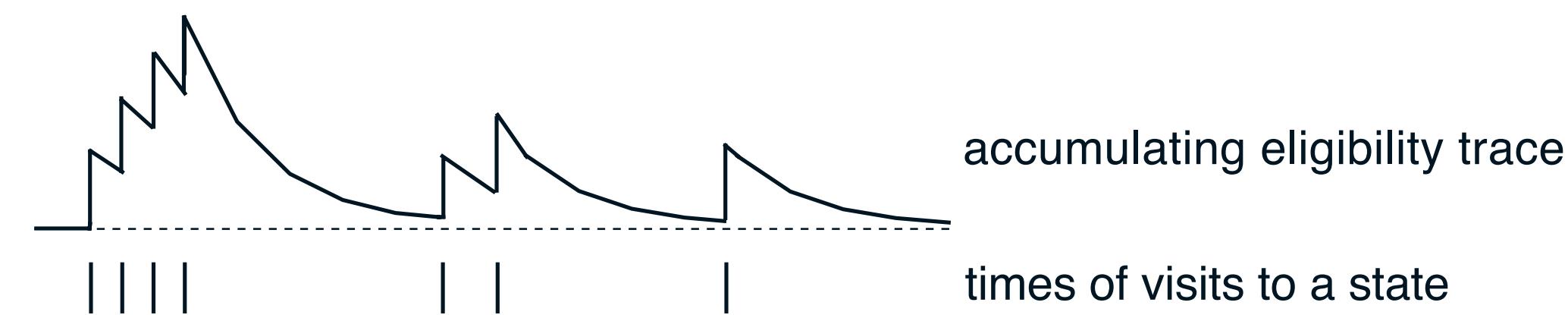


19 states tabular

Eligibility traces (mechanism)

- The forward view was for theory
- The backward view is for *mechanism*
- New memory vector called *eligibility trace* $\mathbf{e}_t \in \mathbb{R}^n$
- On each step, decay each component by $\gamma\lambda$ and increment the trace for the current state by 1
- *Accumulating trace*

$$\begin{aligned}\mathbf{e}_0 &\doteq \mathbf{0}, \\ \mathbf{e}_t &\doteq \nabla \hat{v}(S_t, \theta_t) + \gamma \lambda \mathbf{e}_{t-1}\end{aligned}$$



- *Replacing trace*: trace becomes 1 when state is visited

The Semi-gradient TD(λ) algorithm

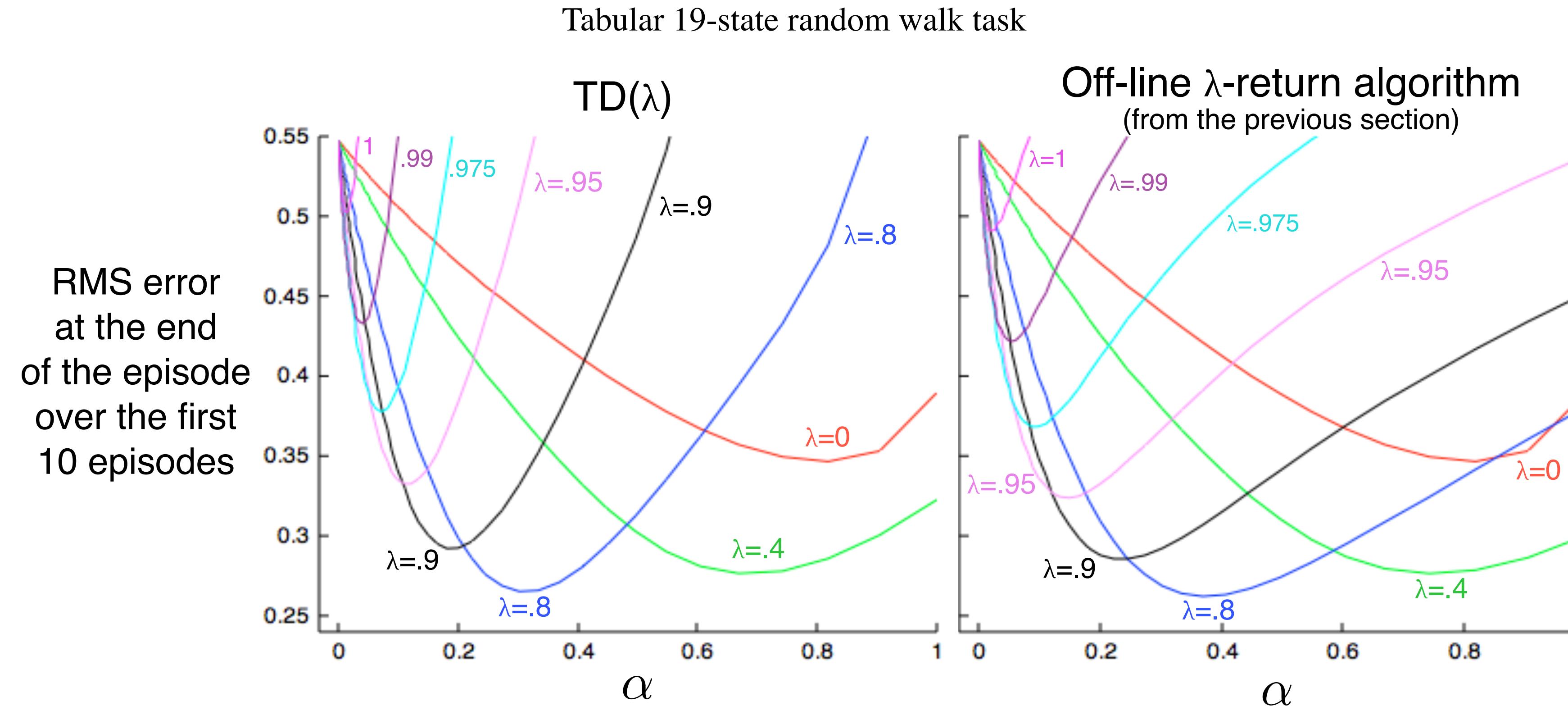
$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \delta_t \mathbf{e}_t$$

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{\theta}_t) - \hat{v}(S_t, \boldsymbol{\theta}_t)$$

$$\mathbf{e}_0 \doteq \mathbf{0},$$

$$\mathbf{e}_t \doteq \nabla \hat{v}(S_t, \boldsymbol{\theta}_t) + \gamma \lambda \mathbf{e}_{t-1}$$

TD(λ) performs similarly to offline λ -return alg. but slightly worse, particularly at high α

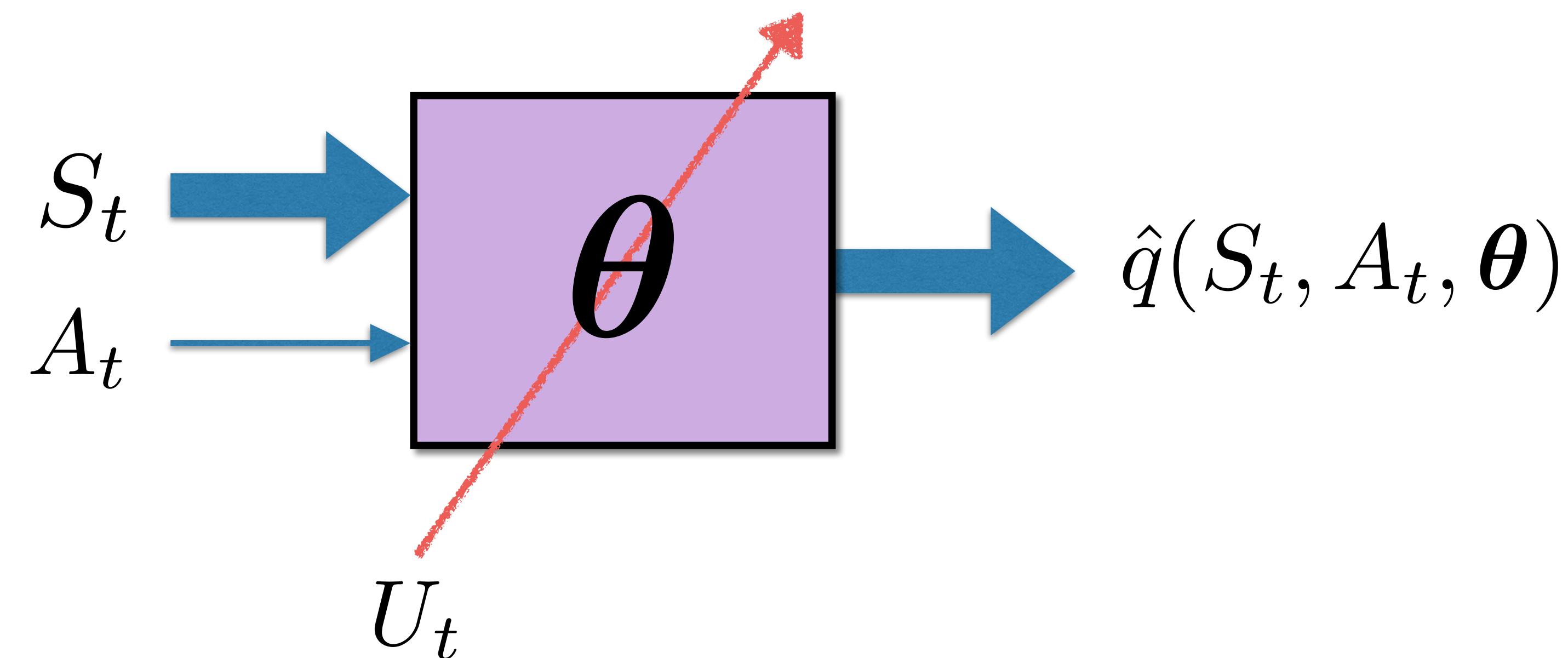


Can we do better? Can we update online?

Conclusions

- Value-function approximation by stochastic gradient descent enables RL to be applied to arbitrarily large state spaces
- Most algorithms just carry over the Targets from the tabular case
- With bootstrapping (TD), we don't get true gradient descent methods
 - this complicates the analysis
 - but the linear, on-policy case is still guaranteed convergent
 - and learning is still *much faster*

Value function approximation (VFA) for control



(Semi-)gradient methods carry over to control in the usual on-policy GPI way

- Always learn the action-value function of the current policy
- Always act near-greedily wrt the current action-value estimates
- The learning rule is:

$$\theta_{t+1} \doteq \theta_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \theta_t) \right] \nabla \hat{q}(S_t, A_t, \theta_t)$$

update target, e.g., $U_t = G_t$ (MC)

$U_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \theta_t)$ (Sarsa)

(Expected Sarsa) $U_t = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) \hat{q}(S_{t+1}, a, \theta_t)$ $U_t = \sum_{s', r} p(s', r|S_t, A_t) \left[r + \gamma \sum_{a'} \pi(a'|s') \hat{q}(s', a', \theta_t) \right]$ (DP)

(Semi-)gradient methods carry over to control

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t) \right] \nabla \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)$$

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^n \rightarrow \mathbb{R}$

Initialize value-function weights $\boldsymbol{\theta} \in \mathbb{R}^n$ arbitrarily (e.g., $\boldsymbol{\theta} = \mathbf{0}$)

Repeat (for each episode):

$S, A \leftarrow$ initial state and action of episode (e.g., ε -greedy)

Repeat (for each step of episode):

Take action A , observe R, S'

If S' is terminal:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha [R - \hat{q}(S, A, \boldsymbol{\theta})] \nabla \hat{q}(S, A, \boldsymbol{\theta})$$

Go to next episode

Choose A' as a function of $\hat{q}(S', \cdot, \boldsymbol{\theta})$ (e.g., ε -greedy)

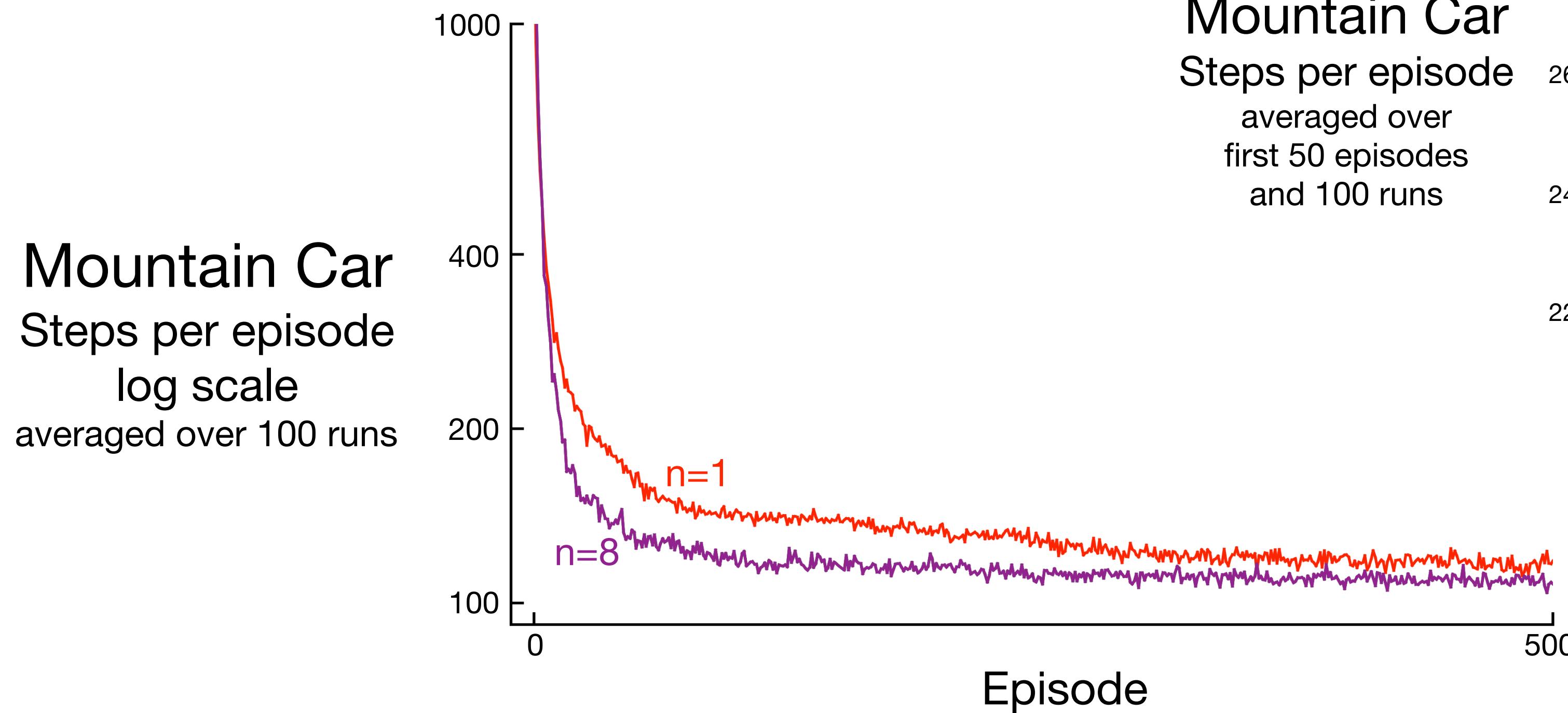
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha [R + \gamma \hat{q}(S', A', \boldsymbol{\theta}) - \hat{q}(S, A, \boldsymbol{\theta})] \nabla \hat{q}(S, A, \boldsymbol{\theta})$$

$$S \leftarrow S'$$

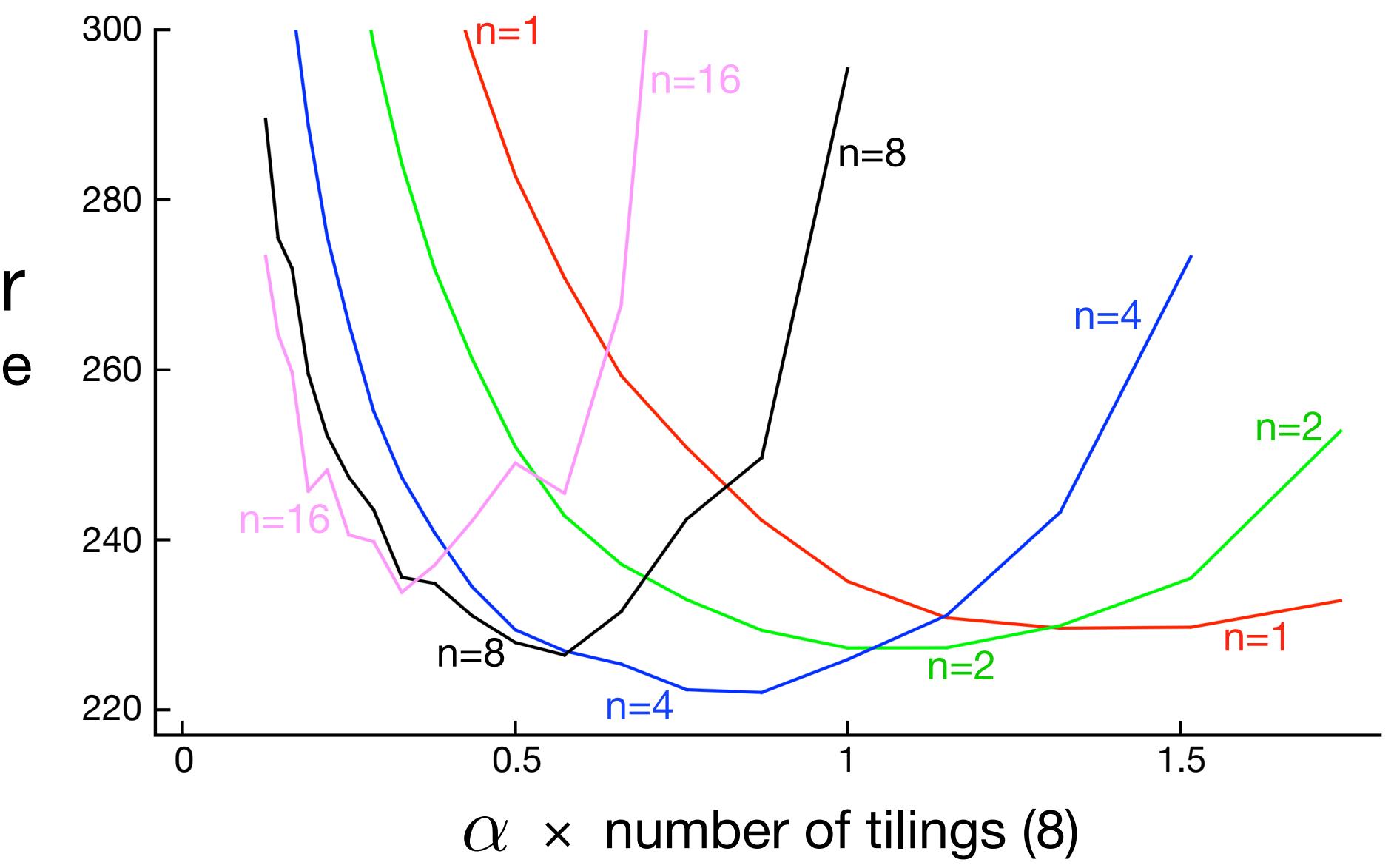
$$A \leftarrow A'$$

n -step semi-gradient Sarsa is better for $n > 1$

$$\theta_{t+n} \doteq \theta_{t+n-1} + \alpha \left[G_t^{(n)} - \hat{q}(S_t, A_t, \theta_{t+n-1}) \right] \nabla \hat{q}(S_t, A_t, \theta_{t+n-1}), \quad 0 \leq t < T$$



Mountain Car
Steps per episode
averaged over
first 50 episodes
and 100 runs



Conclusions

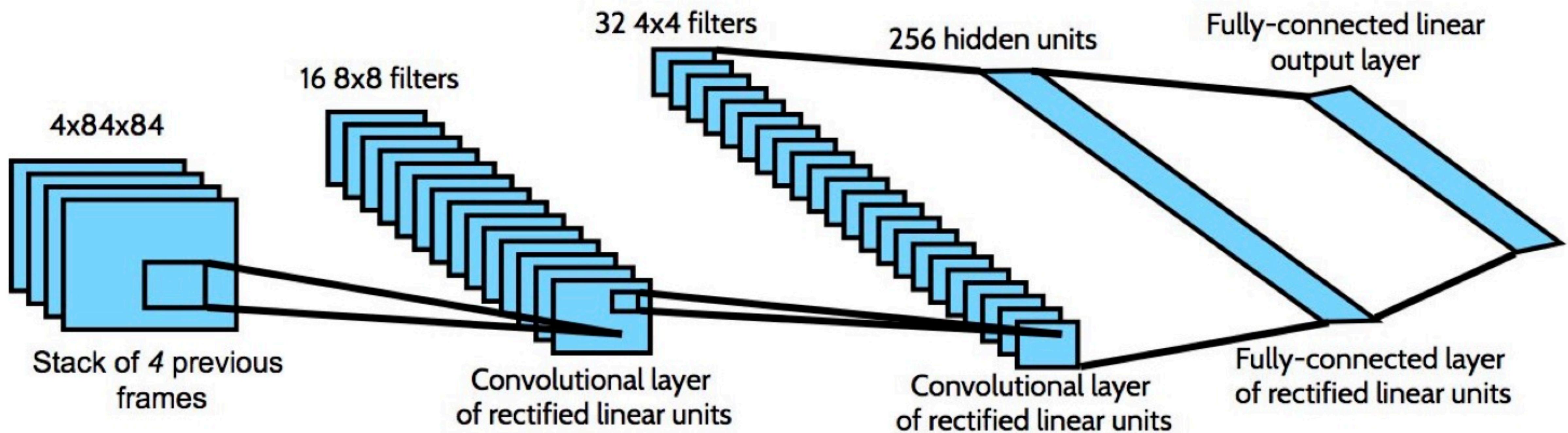
- Control is straightforward in the on-policy case
- Formal results (bounds) exist for the linear, on-policy case (eg. Gordon, 2000, Perkins & Precup, 2003 and follow-up work)
 - we get chattering near a good solution, not convergence

DQN

(Mnih, Kavukcuoglu, Silver, et al., Nature 2015)

- Learns to play video games **from raw pixels**, simply by playing
- Can learn Q function by Q-learning

$$\Delta \mathbf{w} = \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \mathbf{w}) - Q(S_t, A_t; \mathbf{w}) \right) \nabla_{\mathbf{w}} Q(S_t, A_t; \mathbf{w})$$



DQN

(Mnih, Kavukcuoglu, Silver, et al., Nature 2015)

- Learns to play video games **from raw pixels**, simply by playing
- Can learn Q function by Q-learning

$$\Delta \mathbf{w} = \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \mathbf{w}) - Q(S_t, A_t; \mathbf{w}) \right) \nabla_{\mathbf{w}} Q(S_t, A_t; \mathbf{w})$$

- Core components of DQN include:

- Target networks (Mnih et al. 2015)

$$\Delta \mathbf{w} = \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \mathbf{w}^-) - Q(S_t, A_t; \mathbf{w}) \right) \nabla_{\mathbf{w}} Q(S_t, A_t; \mathbf{w})$$

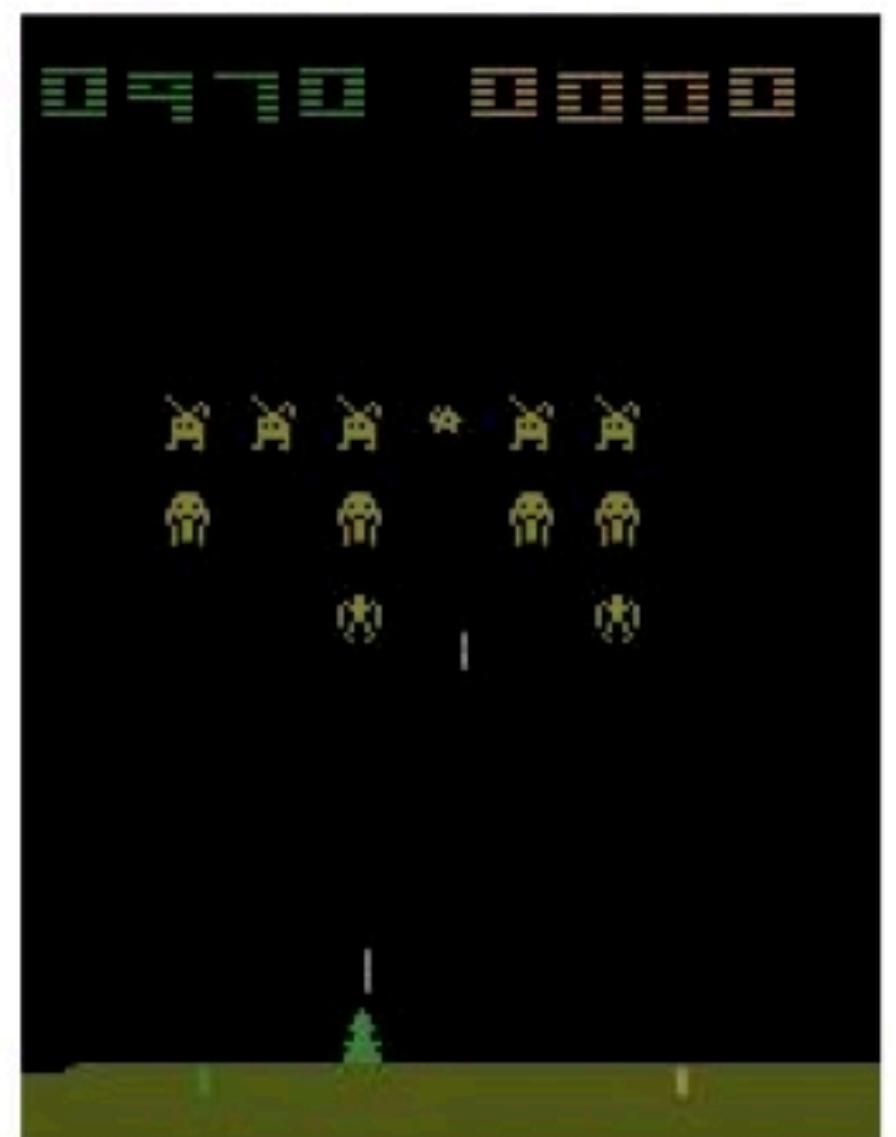
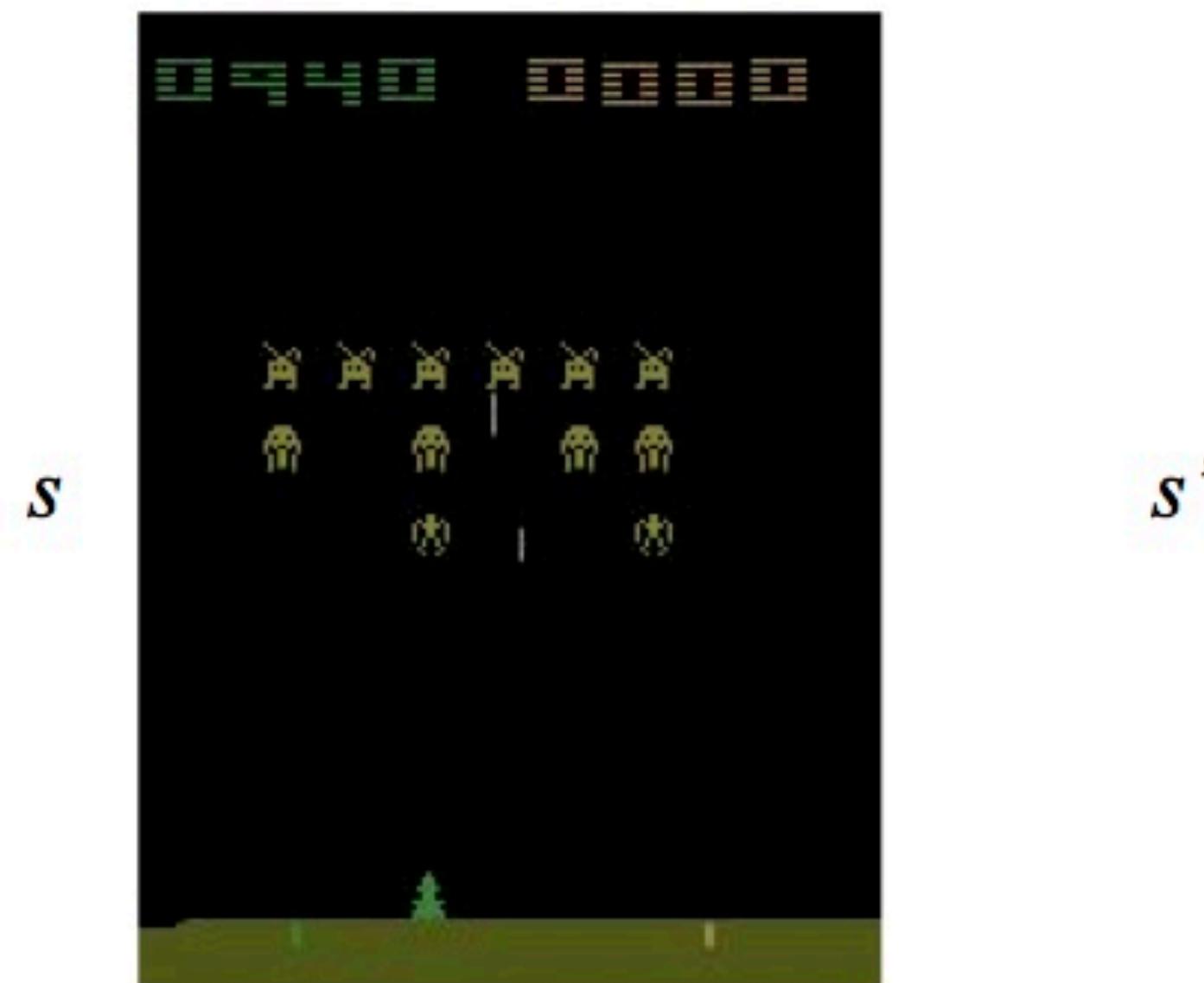
- Experience replay (Lin 1992): replay previous tuples (s, a, r, s')

Target Network Intuition

(Slide credit: Vlad Mnih)

- Changing the value of one action will change the value of other actions and similar states.
- The network can end up chasing its own tail because of bootstrapping.
- Somewhat surprising fact - bigger networks are less prone to this because they alias less.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r \sim D} \left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i) \right)^2$$



DQN

(Mnih, Kavukcuoglu, Silver, et al., Nature 2015)

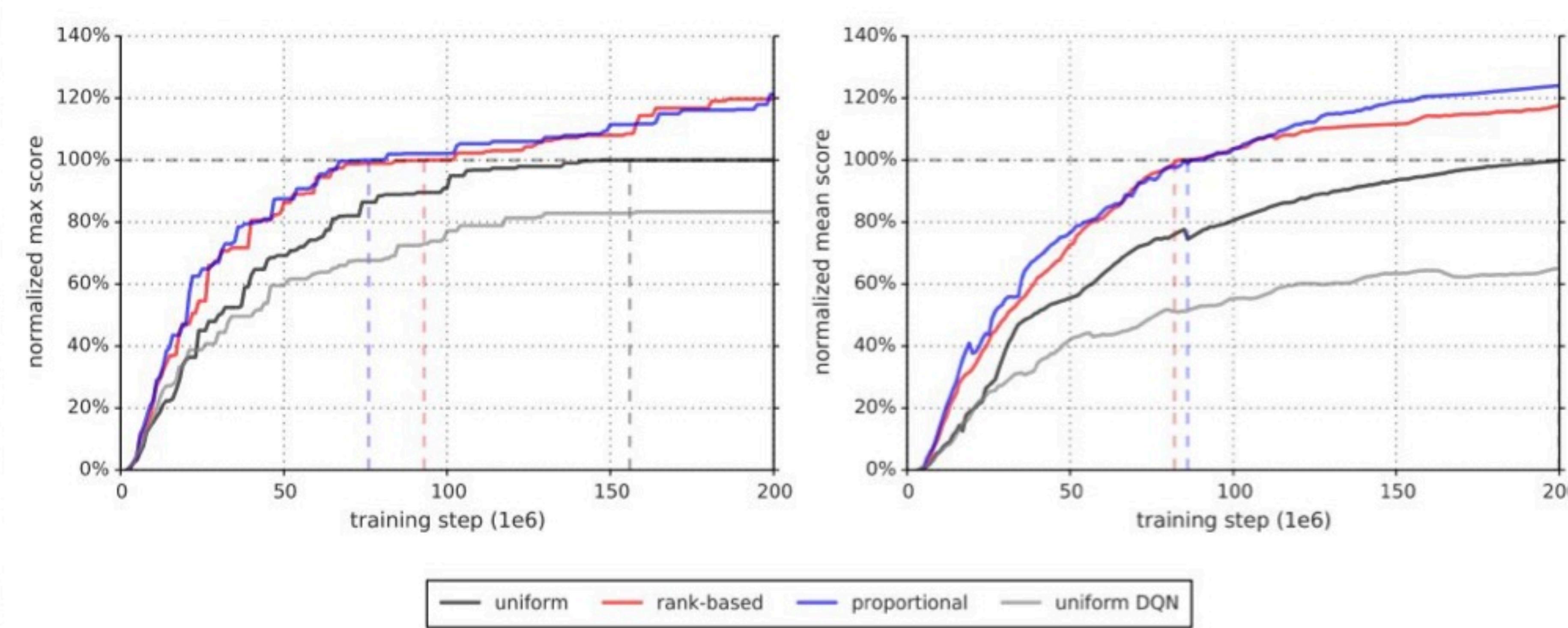
- Many later improvements to DQN
 - Double Q-learning (van Hasselt 2010, van Hasselt et al. 2015)
 - Prioritized replay (Schaul et al. 2016)
 - Dueling networks (Wang et al. 2016)
 - Asynchronous learning (Mnih et al. 2016)
 - Adaptive normalization of values (van Hasselt et al. 2016)
 - Better exploration (Bellemare et al. 2016, Ostrovski et al., 2017, Fortunato, Azar, Piot et al. 2017)
 - Distributional losses (Bellemare et al. 2017)
 - Multi-step returns (Mnih et al. 2016, Hessel et al. 2017)
 - ... many more ...

Prioritized Experience Replay

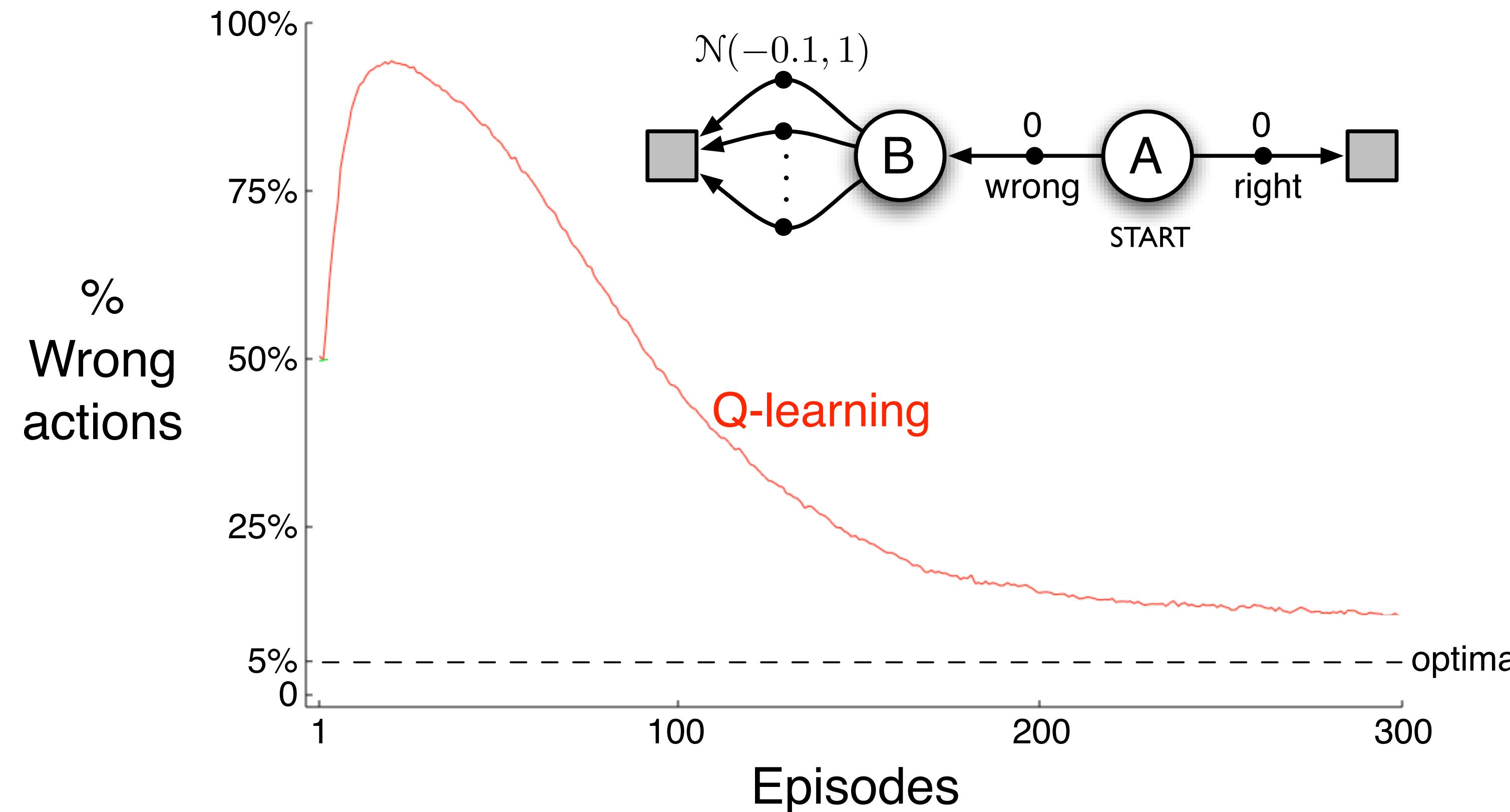
"Prioritized Experience Replay", Schaul et al. (2016)

- Idea: Replay transitions in proportion to TD error:

$$\left| r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right|$$



Maximization Bias Example



Tabular Q-learning:
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

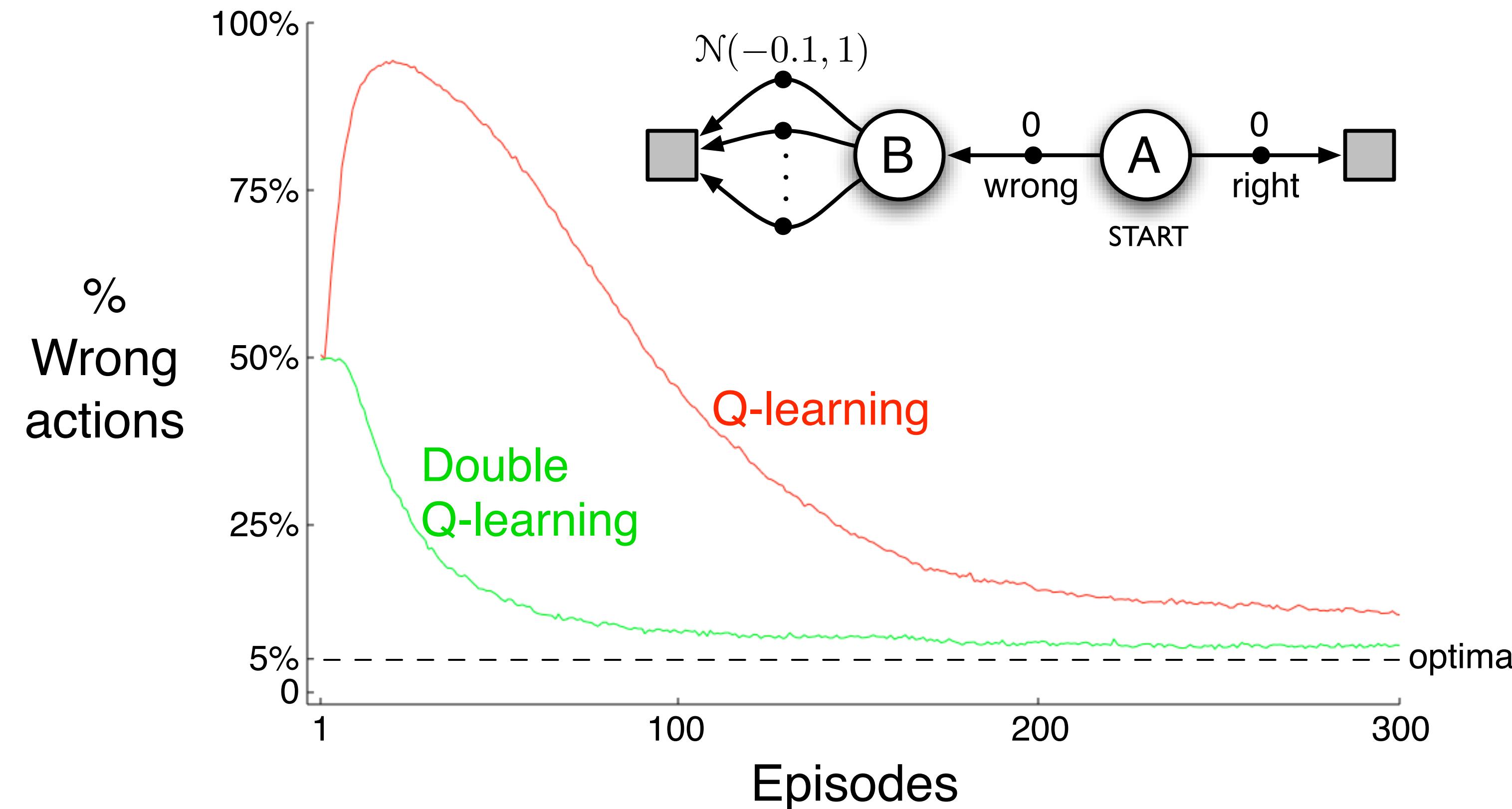
Double Q-Learning

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
 - never on the same time steps (Q_1 and Q_2 are indep.)
 - pick Q_1 or Q_2 at random to be updated on each step
- If updating Q_1 , use Q_2 for the value of the next state:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left(R_{t+1} + Q_2(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right)$$

- Action selections are (say) ε -greedy with respect to the sum of Q_1 and Q_2

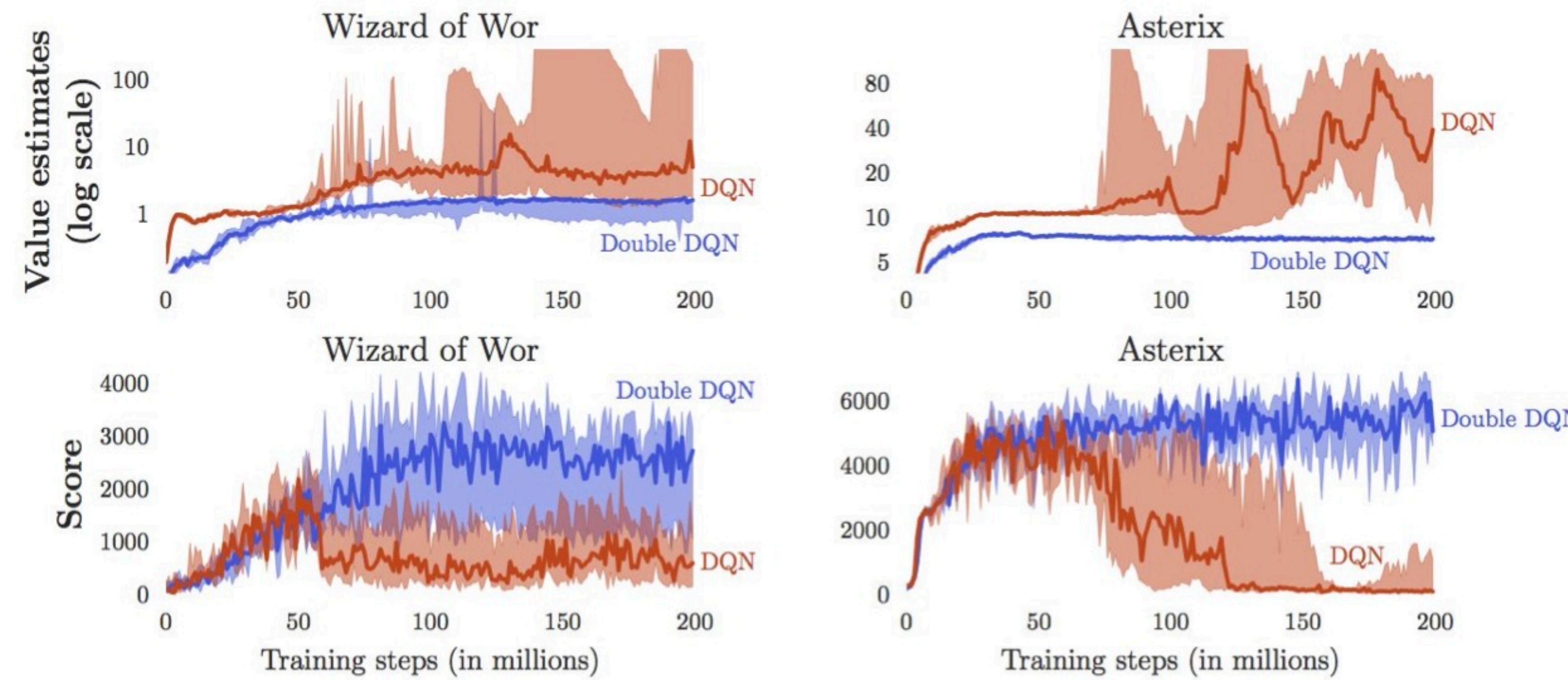
Example of Maximization Bias



Double Q-learning:

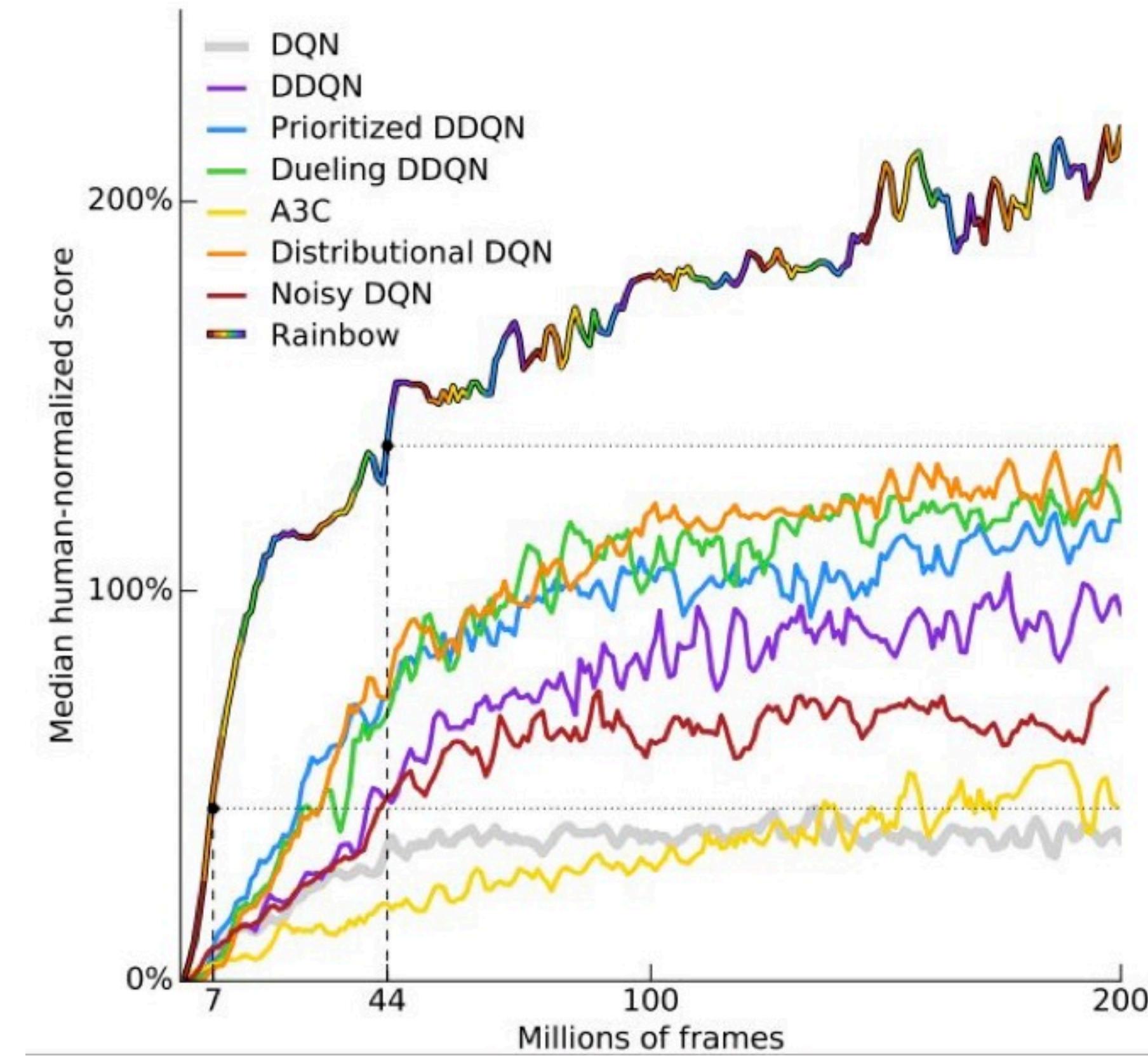
$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right]$$

Double DQN



cf. van Hasselt et al, 2015)

Which DQN improvements matter?

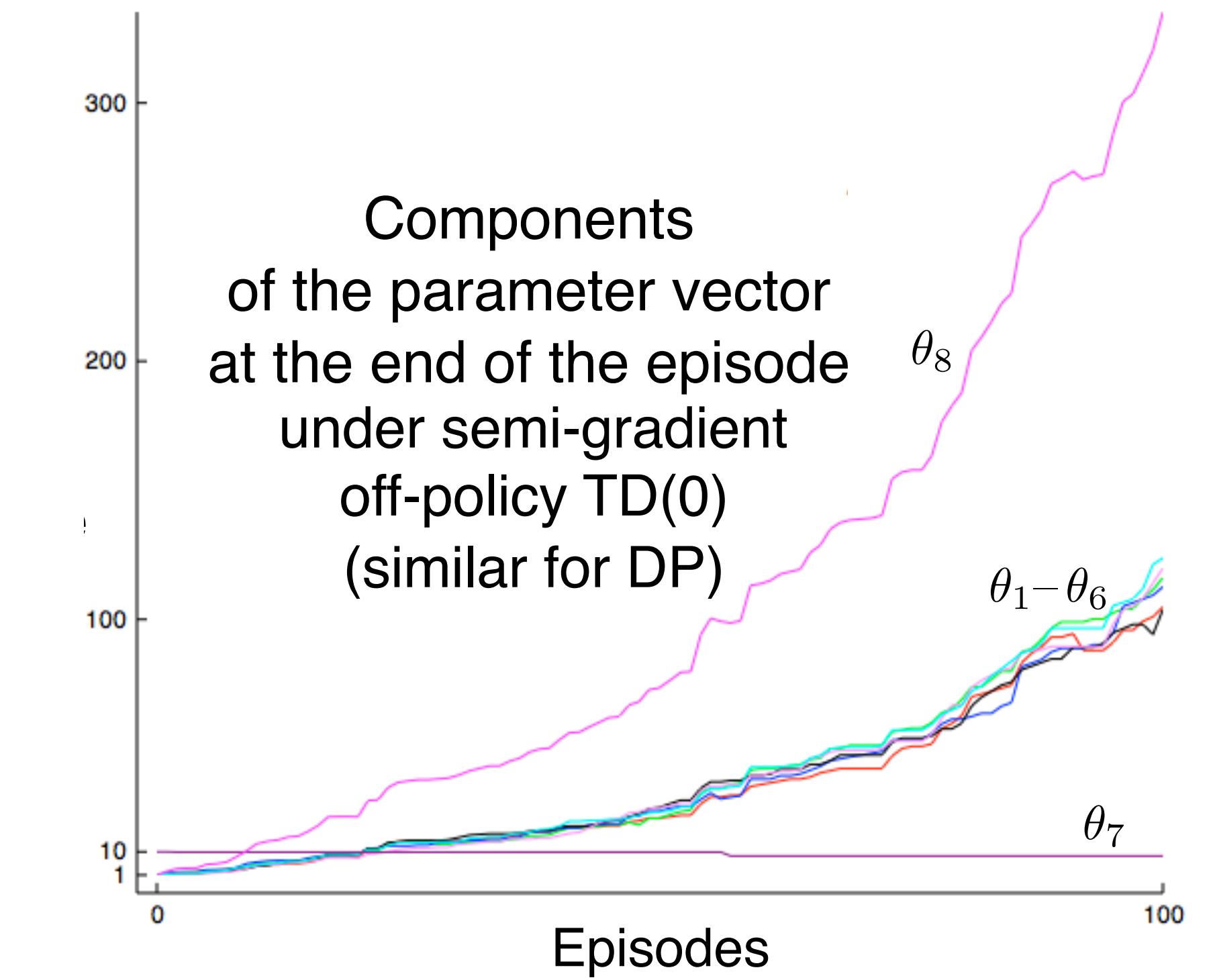
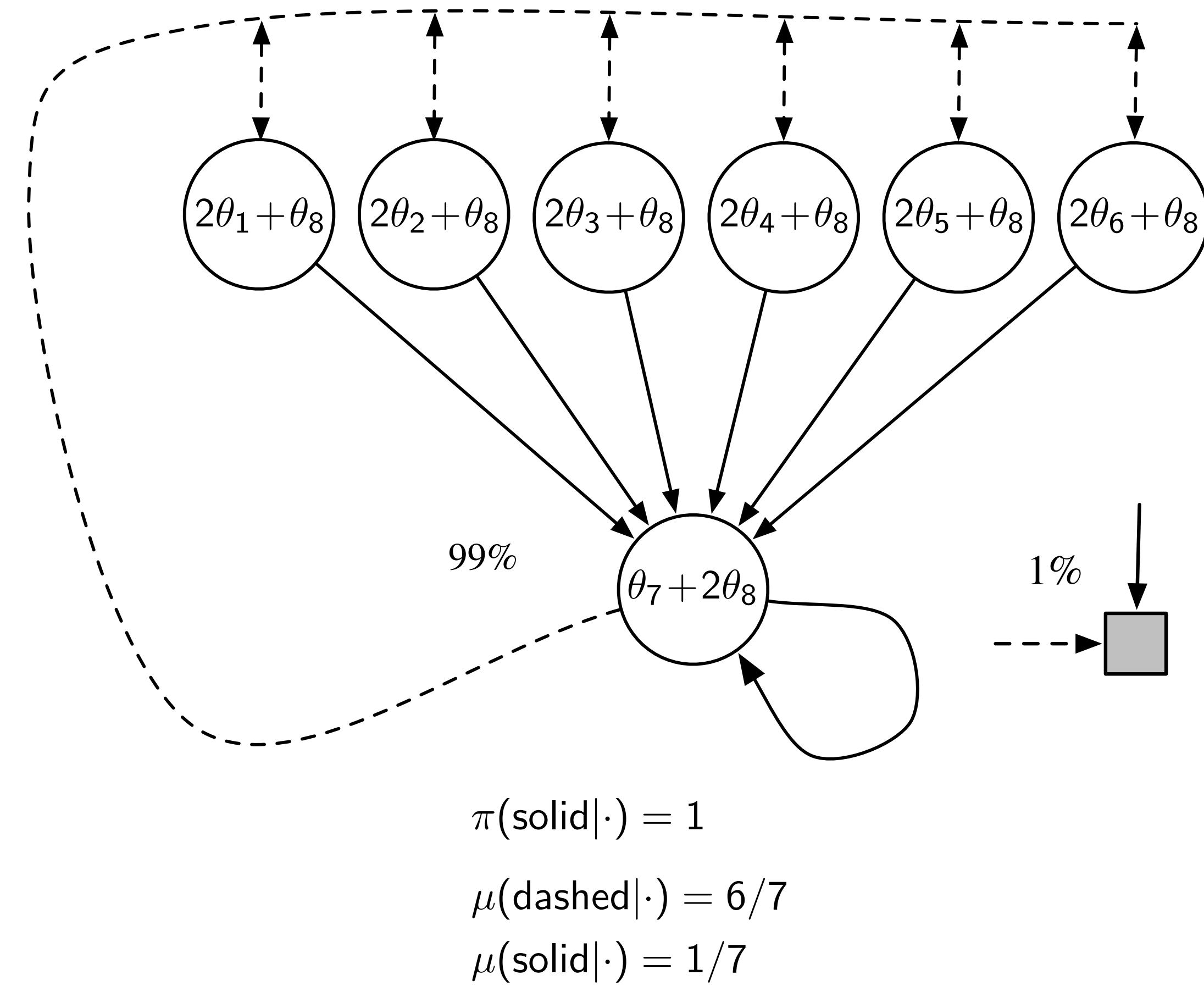


Rainbow model, Hessel et al, 2017)

Off-policy with Function Approximation can be very hard!

- Even linear FA
- Even for prediction (two fixed policies π and μ)
- Even for Dynamic Programming
- The deadly triad: FA, TD, off-policy
 - Any two are OK, but not all three
 - With all three, we may get instability
(elements of θ may increase to $\pm\infty$)

Baird's counterexample illustrates the instability



What causes the instability?

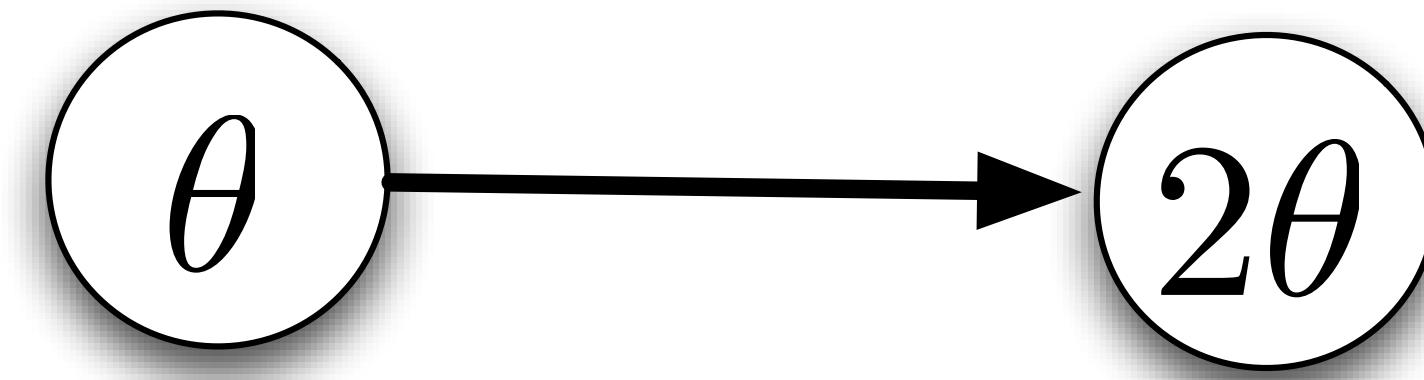
- It has nothing to do with learning or sampling
 - Even dynamic programming suffers from divergence with FA
- It has nothing to do with exploration, greedification, or control
 - Even prediction alone can diverge
- It has nothing to do with local minima or complex non-linear approximators
 - Even simple linear approximators can produce instability

The deadly triad

- The risk of divergence arises whenever we combine three things:
 1. Function approximation
 - significantly generalizing from large numbers of examples
 2. Bootstrapping
 - learning value estimates from other value estimates, as in dynamic programming and temporal-difference learning
 3. Off-policy learning
 - learning about a policy from data not due to that policy, as in Q-learning, where we learn about the greedy policy from data with a necessarily more exploratory policy

Any 2 Ok

TD(0) can diverge: A simple example



$$\begin{aligned}\delta &= r + \gamma\theta^\top\phi' - \theta^\top\phi \\ &= 0 + 2\theta - \theta \\ &= \theta\end{aligned}$$

TD update: $\Delta\theta = \alpha\delta\phi$

$= \alpha\theta$ **Diverges!**

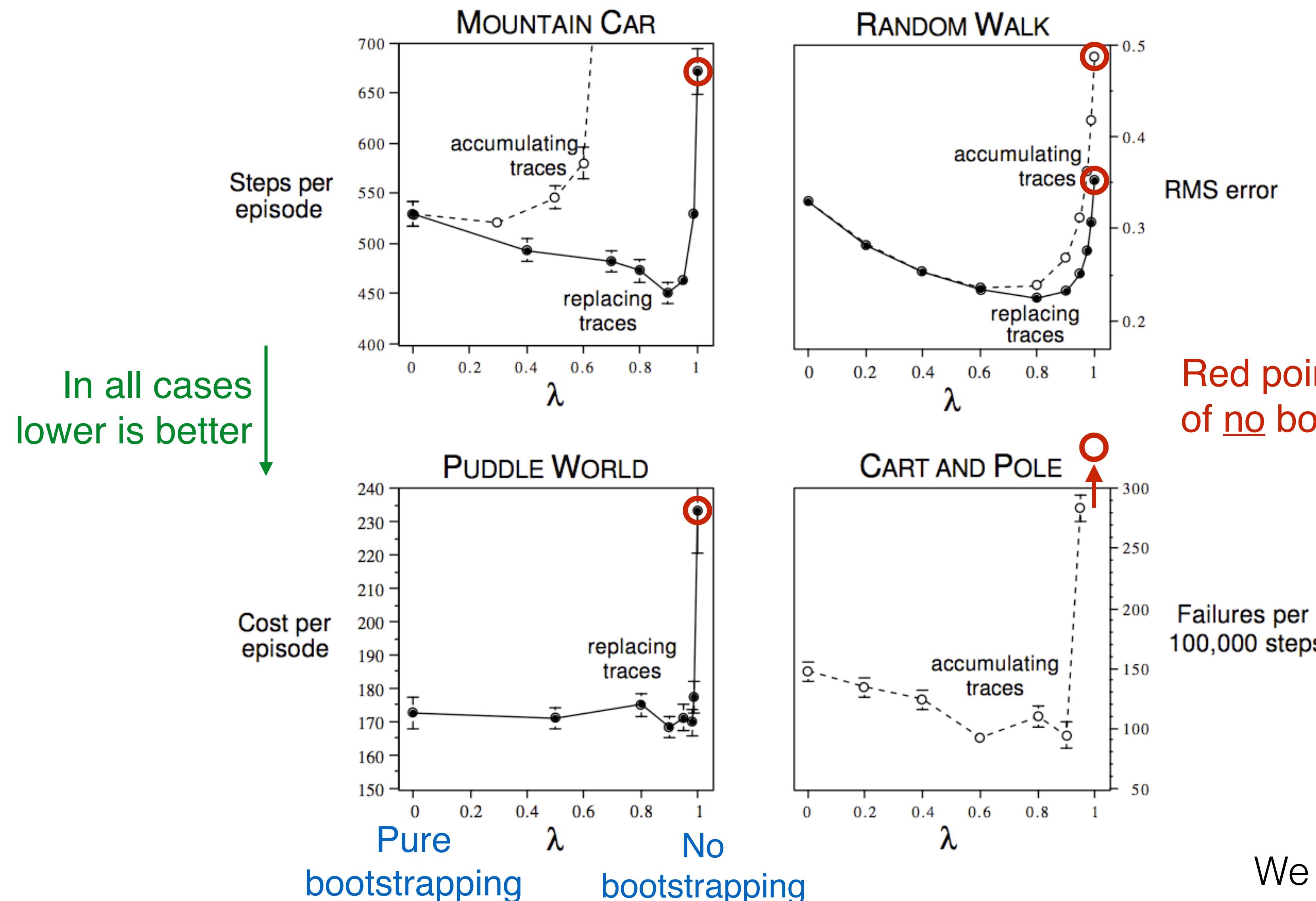
TD fixpoint: $\theta^* = 0$

Can we do without bootstrapping?

- Bootstrapping is critical to the computational efficiency of DP
- Bootstrapping is critical to the data efficiency of TD methods
- On the other hand, bootstrapping introduces bias, which harms the asymptotic performance of approximate methods
- The degree of bootstrapping can be finely controlled via the λ parameter, from $\lambda=0$ (full bootstrapping) to $\lambda=1$ (no bootstrapping)

4 examples of the effect of bootstrapping

suggest that $\lambda=1$ (no bootstrapping) is a very poor choice



Desiderata: We want a TD algorithm that

- Bootstraps (genuine TD)
- Works with linear function approximation
(stable, reliably convergent)
- Is simple, like linear TD — $O(n)$
- Learns fast, like linear TD
- Can learn off-policy
- Learns from online causal trajectories
(no repeat sampling from the same state)

4 easy steps to stochastic gradient descent

1. Pick an objective function $J(\theta)$,
a parameterized function to be minimized
2. Use calculus to analytically compute the gradient $\nabla_{\theta} J(\theta)$
3. Find a “sample gradient” $\nabla_{\theta} J_t(\theta)$ that you can sample on
every time step and whose expected value equals the gradient
4. Take small steps in θ proportional to the sample gradient:

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} J_t(\theta)$$

Conventional TD is not the gradient of anything

TD(0) algorithm:

$$\Delta\theta = \alpha\delta\phi$$

$$\delta = r + \gamma\theta^\top\phi' - \theta^\top\phi$$

Assume there is a J such that:

$$\frac{\partial J}{\partial\theta_i} = \delta\phi_i$$

Then look at the second derivative:

$$\left. \begin{aligned} \frac{\partial^2 J}{\partial\theta_j\partial\theta_i} &= \frac{\partial(\delta\phi_i)}{\partial\theta_j} = (\gamma\phi'_j - \phi_j)\phi_i \\ \frac{\partial^2 J}{\partial\theta_i\partial\theta_j} &= \frac{\partial(\delta\phi_j)}{\partial\theta_i} = (\gamma\phi'_i - \phi_i)\phi_j \end{aligned} \right\} \quad \frac{\partial^2 J}{\partial\theta_j\partial\theta_i} \neq \frac{\partial^2 J}{\partial\theta_i\partial\theta_j}$$

Contradiction!

Real 2nd derivatives must be symmetric

Gradient descent for TD: What should the objective function be?

Mean-Square
Value Error

$$\begin{aligned} \text{MSE}(\theta) &= \sum_s d_s (V_\theta(s) - V(s))^2 \\ &= \| V_\theta - V \|_D^2 \end{aligned}$$

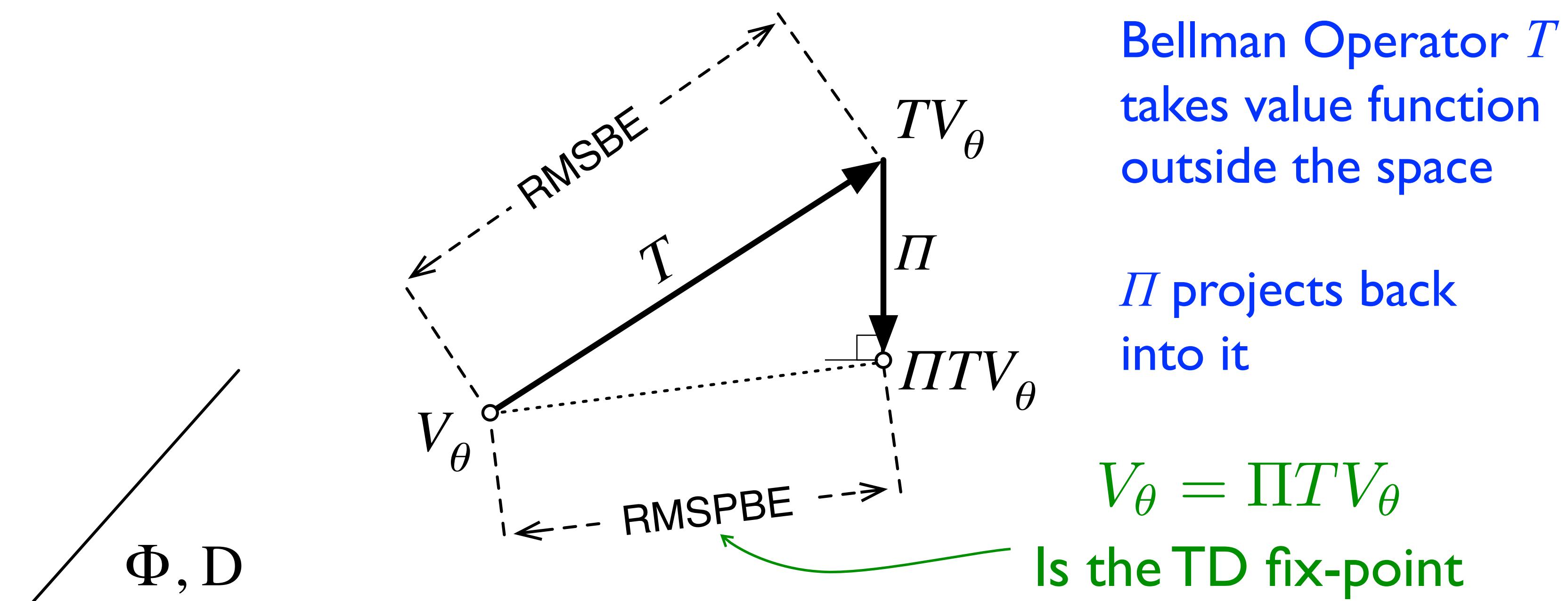
True value
function

Mean-Square
Bellman Error

$$\text{MSBE}(\theta) = \| V_\theta - TV_\theta \|_D^2$$

$$\begin{aligned} V &= r + \gamma PV \\ &= TV \end{aligned}$$

Value function geometry



The space spanned by the feature vectors,
weighted by the state visitation distribution

$$D = \text{diag}(d)$$

Mean Square Projected Bellman Error (MSPBE)

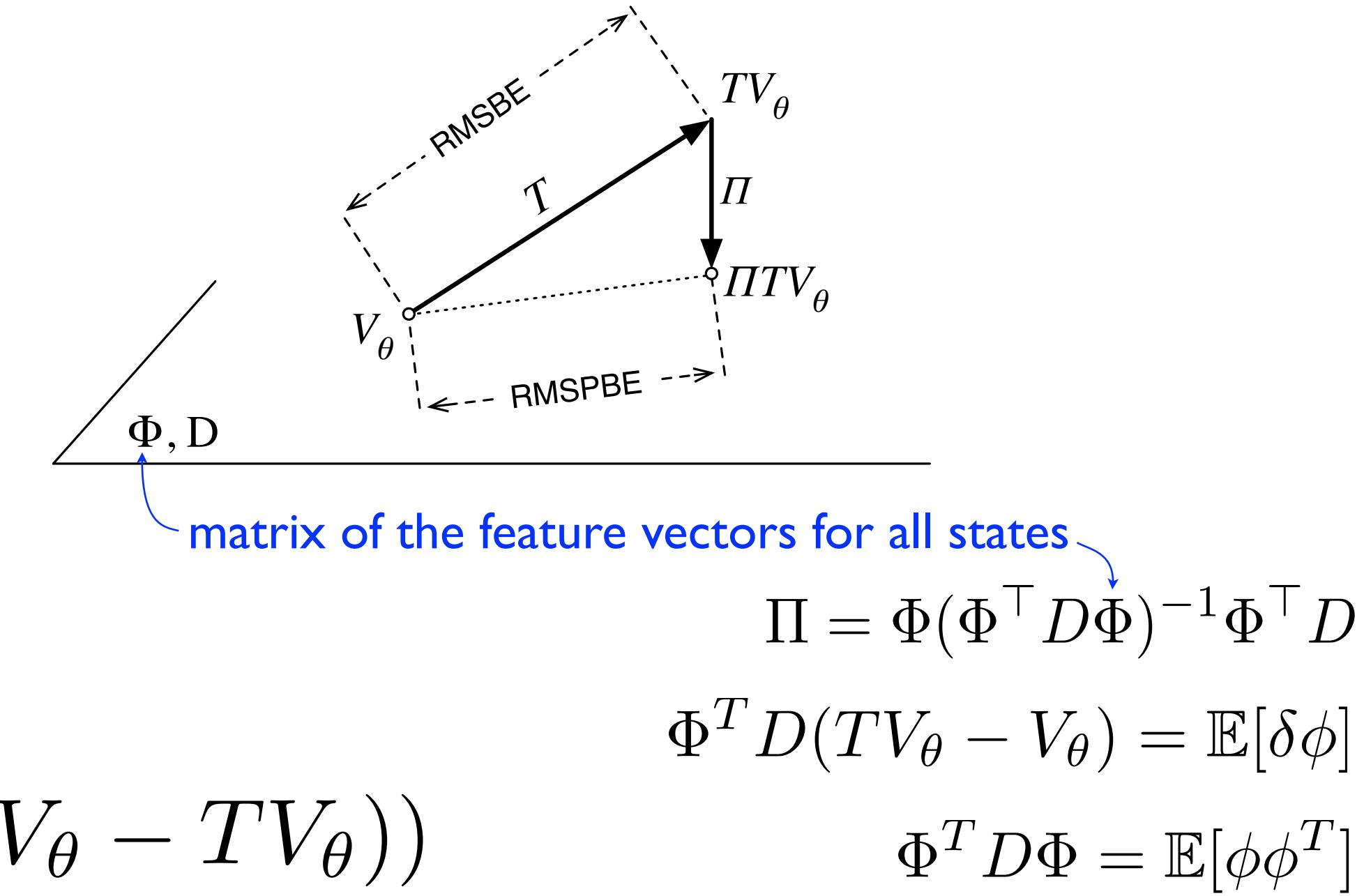
The Gradient-TD Family of Algorithms

- True gradient-descent algorithms in the Projected Bellman Error
- GTD(λ) and GQ(λ), for learning V and Q
- Solve two open problems:
 - convergent linear-complexity off-policy TD learning
 - convergent non-linear TD
- Extended to control variate, proximal forms by Mahadevan et al.

First relate the geometry to the iid statistics

$$\text{MSPBE}(\theta)$$

$$\begin{aligned}
 &= \| V_\theta - \Pi TV_\theta \|_D^2 \\
 &= \| \Pi(V_\theta - TV_\theta) \|_D^2 \\
 &= (\Pi(V_\theta - TV_\theta))^\top D(\Pi(V_\theta - TV_\theta)) \\
 &= (V_\theta - TV_\theta)^\top \Pi^\top D \Pi (V_\theta - TV_\theta) \\
 &= (V_\theta - TV_\theta)^\top D^\top \Phi (\Phi^\top D \Phi)^{-1} \Phi^\top D (V_\theta - TV_\theta) \\
 &= (\Phi^\top D(TV_\theta - V_\theta))^\top (\Phi^\top D \Phi)^{-1} \Phi^\top D (TV_\theta - V_\theta) \\
 &= \mathbb{E}[\delta\phi]^\top \mathbb{E}[\phi\phi^\top]^{-1} \mathbb{E}[\delta\phi].
 \end{aligned}$$



Derivation of the TDC algorithm

$$\begin{aligned}
 \Delta\theta &= -\frac{1}{2}\alpha\nabla_\theta J(\theta) & = & -\frac{1}{2}\alpha\nabla_\theta \| V_\theta - \Pi TV_\theta \|_D^2 \\
 & & = & -\frac{1}{2}\alpha\nabla_\theta \left(\mathbb{E}[\delta\phi] \mathbb{E}[\phi\phi^\top]^{-1} \mathbb{E}[\delta\phi] \right) \\
 & & = & -\alpha (\nabla_\theta \mathbb{E}[\delta\phi]) \mathbb{E}[\phi\phi^\top]^{-1} \mathbb{E}[\delta\phi] \\
 & & = & -\alpha \mathbb{E}[\nabla_\theta [\phi(r + \gamma\phi'^\top\theta - \phi^\top\theta)]] \mathbb{E}[\phi\phi^\top]^{-1} \mathbb{E}[\delta\phi] \\
 & & = & -\alpha \mathbb{E}[\phi(\gamma\phi' - \phi)^\top]^\top \mathbb{E}[\phi\phi^\top]^{-1} \mathbb{E}[\delta\phi] \\
 & & = & -\alpha (\gamma\mathbb{E}[\phi'\phi^\top] - \mathbb{E}[\phi\phi^\top]) \mathbb{E}[\phi\phi^\top]^{-1} \mathbb{E}[\delta\phi] \\
 & & = & \alpha\mathbb{E}[\delta\phi] - \alpha\gamma\mathbb{E}[\phi'\phi^\top] \mathbb{E}[\phi\phi^\top]^{-1} \mathbb{E}[\delta\phi] \\
 & & \approx & \alpha\mathbb{E}[\delta\phi] - \alpha\gamma\mathbb{E}[\phi'\phi^\top] w
 \end{aligned}$$

$s \xrightarrow{r} s'$
 $\downarrow \quad \downarrow$
 $\phi \quad \phi'$

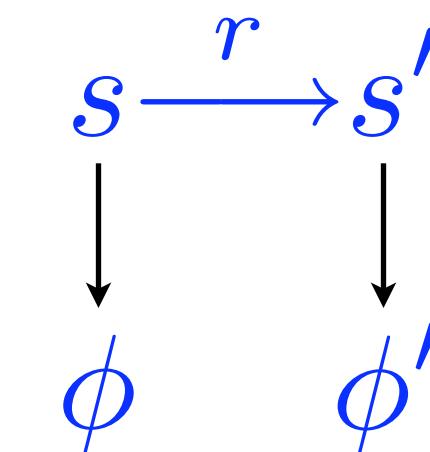
(sampling) $\approx \alpha\delta\phi - \alpha\gamma\phi'\phi^\top w$

This is the trick!
 $w \in \Re^n$ is a second
set of weights

TD with gradient correction (TDC) algorithm

aka GTD(0)

- on each transition



- update two parameters TD(0)

$$\theta \leftarrow \theta + \alpha \delta \phi - \alpha \gamma \phi' (\phi^\top w)$$

with gradient
correction

$$w \leftarrow w + \beta (\delta - \phi^\top w) \phi$$

estimate of the
TD error (δ) for
the current state ϕ

- where, as usual

$$\delta = r + \gamma \theta^\top \phi' - \theta^\top \phi$$

Convergence theorems

- All algorithms converge w.p.1 to the TD fix-point:

$$\mathbb{E}[\delta\phi] \longrightarrow 0$$

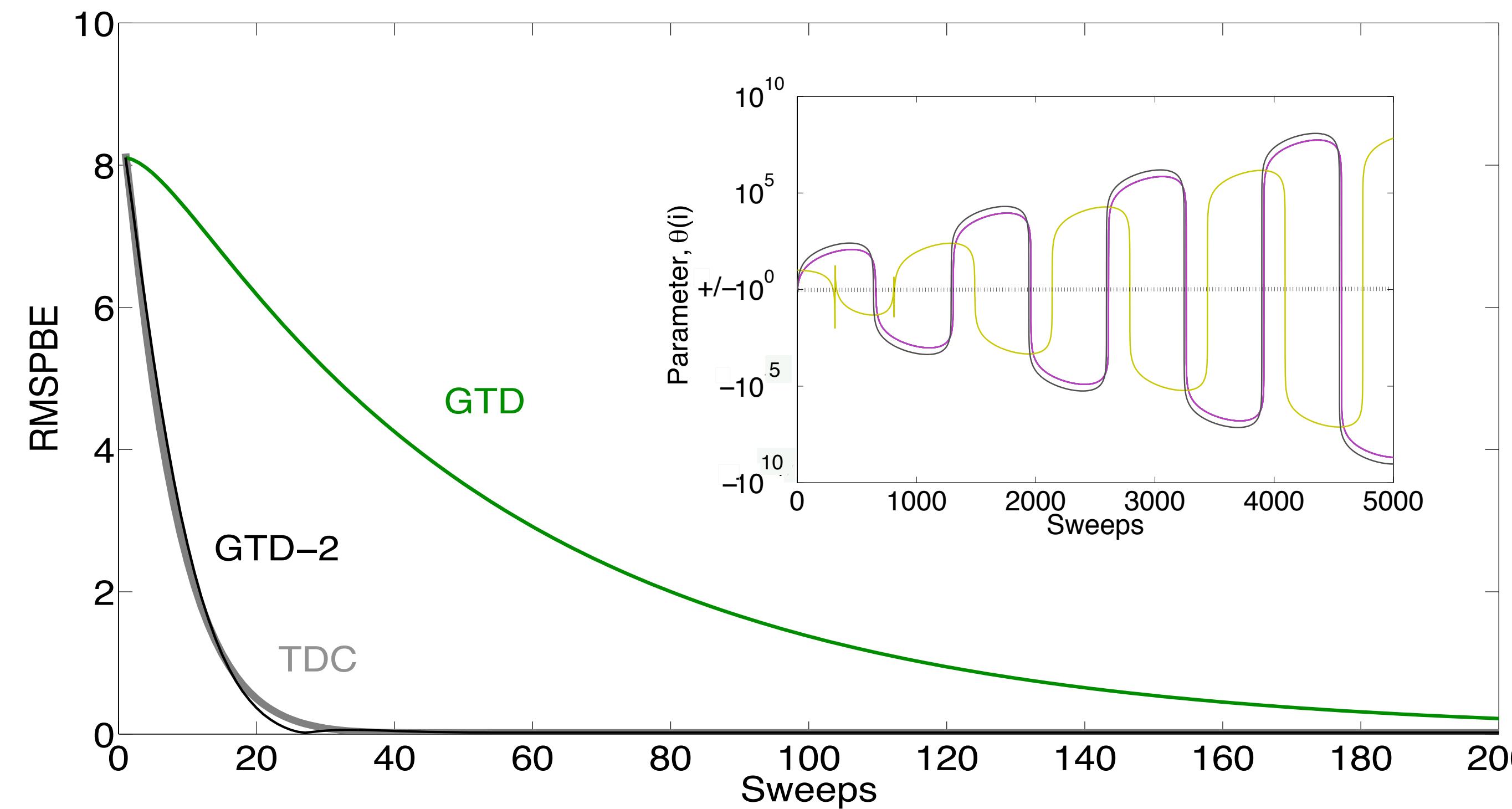
- GTD, GTD-2 converges at one time scale

$$\alpha = \beta \longrightarrow 0$$

- TD-C converges in a two-time-scale sense

$$\alpha, \beta \longrightarrow 0 \quad \frac{\alpha}{\beta} \longrightarrow 0$$

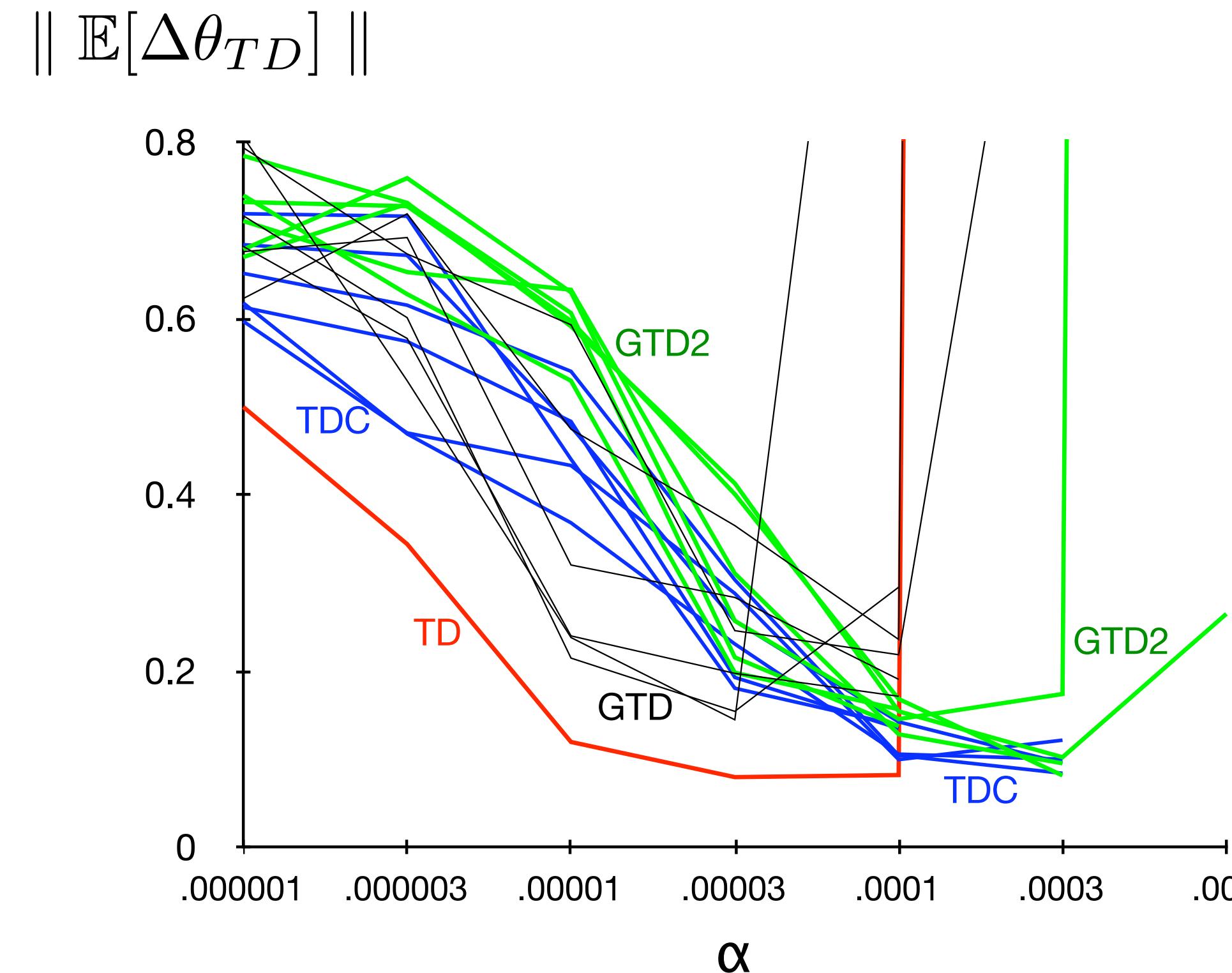
Off-policy result: Baird's counter-example



Gradient algorithms converge. TD diverges.

Computer Go experiment

- Learn a linear value function (probability of winning) for 9x9 Go from self play
- One million features, each corresponding to a template on a part of the Go board



Off-policy RL with FA and TD remains challenging; but there are multiple possible solutions

- Gradient TD, proximal gradient TD, and hybrids
 - Emphatic TD
 - Higher λ (less TD)
 - Recognizers (less off-policy)
 - LSTD ($O(n^2)$ methods)
-
- More work needed
on these novel algs!

Value-based or policy-based? DQN or A3C?

- This is an application-dependent choice!
- If policy space is simple to parameterize, policy search/AC work very well
- Eg. powerplant control
- If policy space is complicated, value-based is better
- Using a value function can greatly reduce variance

Open questions

- Huge gap between theory and practice!
- Is there a natural way to exploit more stable function approximators? Eg kernels, averages...
- Improve stability of deep RL
- Planning with approximate models
- Exploration, exploration, exploration....