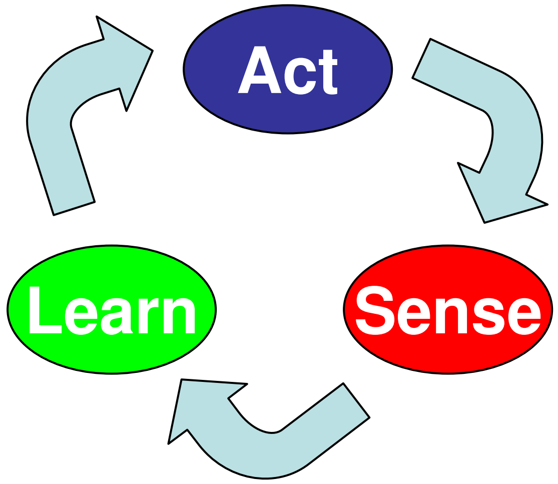
Introduction

Reinforcement Learning



Scott Sanner
NICTA / ANU
First.Last@nicta.com.au

Lecture Goals

1) To understand formal models for decision-making under uncertainty and their properties

- *Unknown models* (reinforcement learning)

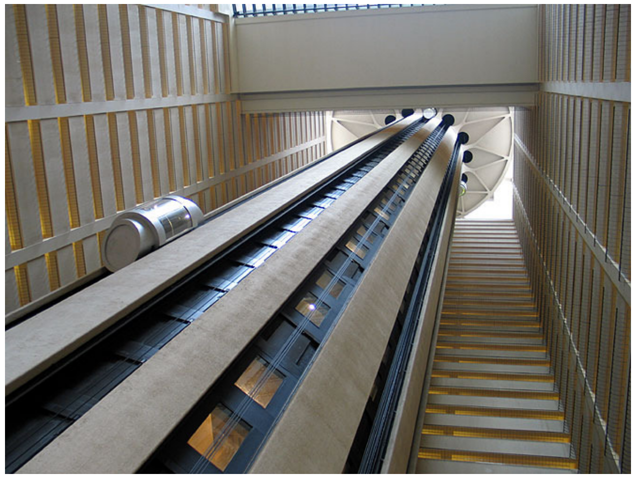
- *Known models* (planning under uncertainty)

2) To understand efficient solution algorithms for these models

Applications

Elevator Control

- **Concurrent Actions**
  - Elevator: up/down/stay
  - 6 elevators: 3^6 actions
- **Dynamics**:
  - Random arrivals (e.g., Poisson)
- **Objective:**
  - Minimize total wait
  - (Requires being proactive about future arrivals)
- **Constraints:**
  - People might get annoyed if elevator reverses direction



Two-player Games

- **Othello / Reversi**
  - Solved by Logistello!
  - Monte Carlo RL (self-play) + Logistic regression + Search

- **Backgammon**
  - Solved by TD-Gammon!
  - Temporal Difference (self-play) + Artificial Neural Net + Search

- **Go**
  - Learning + Search?
  - Unsolved!

Multi-player Games: Poker

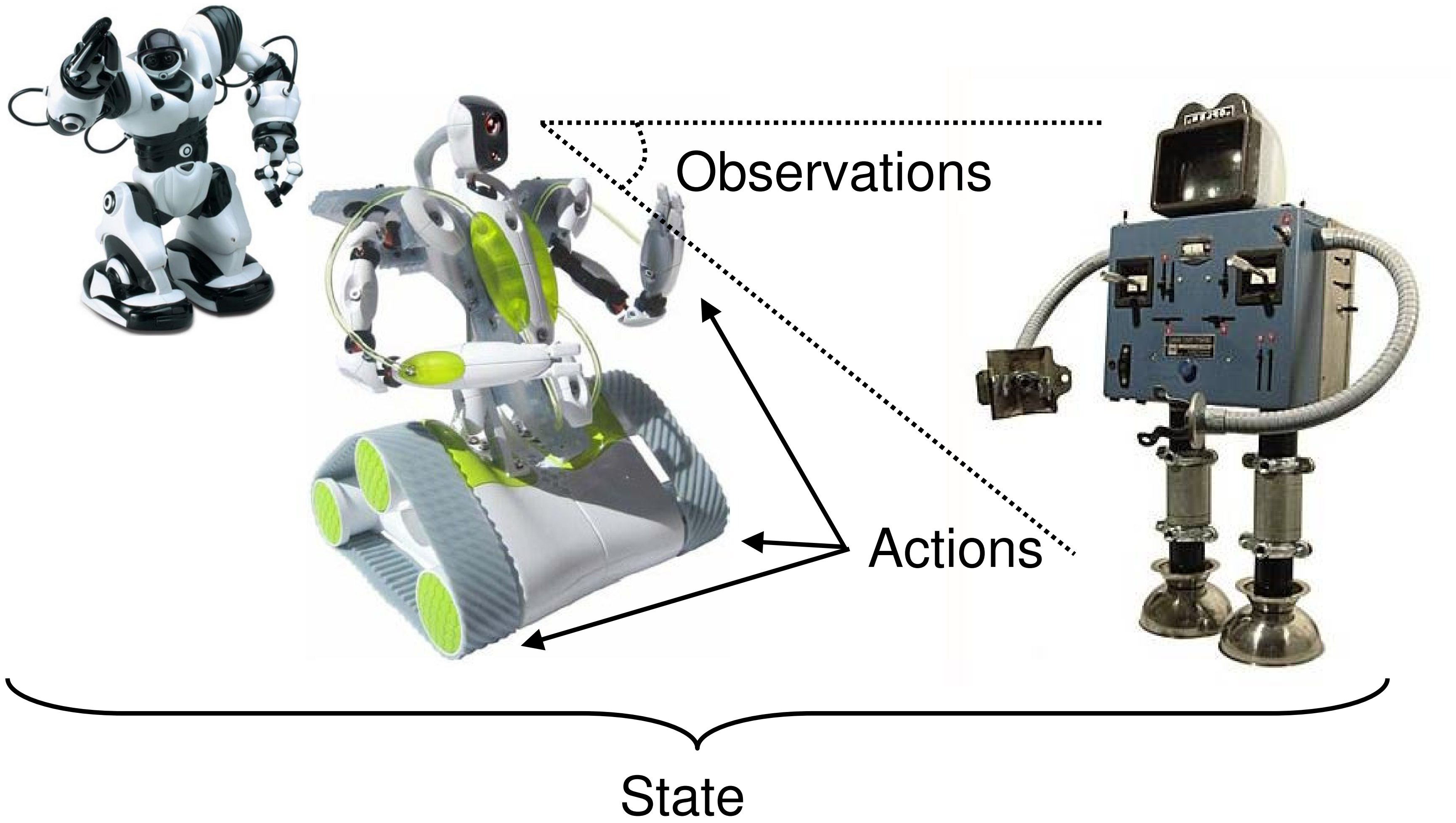
- **Multiagent (adversarial)**
- **Strict uncertainty**
  - Opponent may abruptly change strategy
  - Might prefer best outcome for *any* opponent strategy
- **Multiple rounds (sequential)**
- **Partially observable!**
  - Earlier actions may reveal information
  - Or they may not (bluff)

DARPA Grand Challenge

- **Autonomous mobile robotics**
  - Extremely complex task, requires expertise in vision, sensors, real-time operating systems
- **Partially observable**
  - e.g., only get noisy sensor readings
- **Model unknown**
  - e.g., steering response in different terrain

How to formalize these problems?

Observations, States, & Actions



Observations and States

- **Observation set O**
  - Perceptions, e.g.,
    - Distance from car to edge of road
    - My opponent's bet in Poker
- **State set S**
  - At any point in time, system is in some state, e.g.,
    - Actual distance to edge of road
    - My opponent's hand of cards in Poker
  - State set description varies between problems
    - Could be all observation histories
    - Could be infinite (e.g., continuous state description)

Agent Actions

- **Action set A**
  - Actions could be *concurrent*
  - If k actions, $\mathbf{A} = \mathbf{A_1} \times \ldots \times \mathbf{A_k}$
    - Schedule all deliveries to be made at 10am
  - All actions need not be under agent control
    - Other agents, e.g.,
      - Alternating turns: Poker, Othello
      - Concurrent turns: Highway Driving, Soccer
    - *Exogenous events* due to *Nature*, e.g.,
      - Random arrival of person waiting for elevator
      - Random failure of equipment
- **{history}** encodes all prev. observations, actions

Observation Function

- Z: **{history}** $\times$ **O** $\rightarrow$ **[0,1]**
  - ***Not observable*****:** (used in *conformant planning*)
    - **O =** $\varnothing$
    - e.g., heaven vs. hell
      - » only get feedback once you meet St. Pete
  - ***Fully observable*****:**
    - **S** $\leftrightarrow$ **O** … the case we focus on!
    - e.g., many board games,
      - » Othello, Backgammon, Go
  - ***Partially observable*****:**
    - all remaining cases
    - also called *incomplete information* in game theory
    - e.g., driving a car, Poker

Transition Function

- **T: {history}$\times$A$\times$S$\rightarrow$[0,1]**
  - Some properties
    - *Stationary*: **T** does not change over time
    - *Markovian*: **T**: **S** $\times$ **A** $\times$ **S** $\rightarrow$ [0,1]
      - Next state dependent only upon previous state / action
      - If not Markovian, can always augment state description
        - » e.g., elevator traffic model differs throughout day;
          so encode time in **S** to make **T** Markovian!

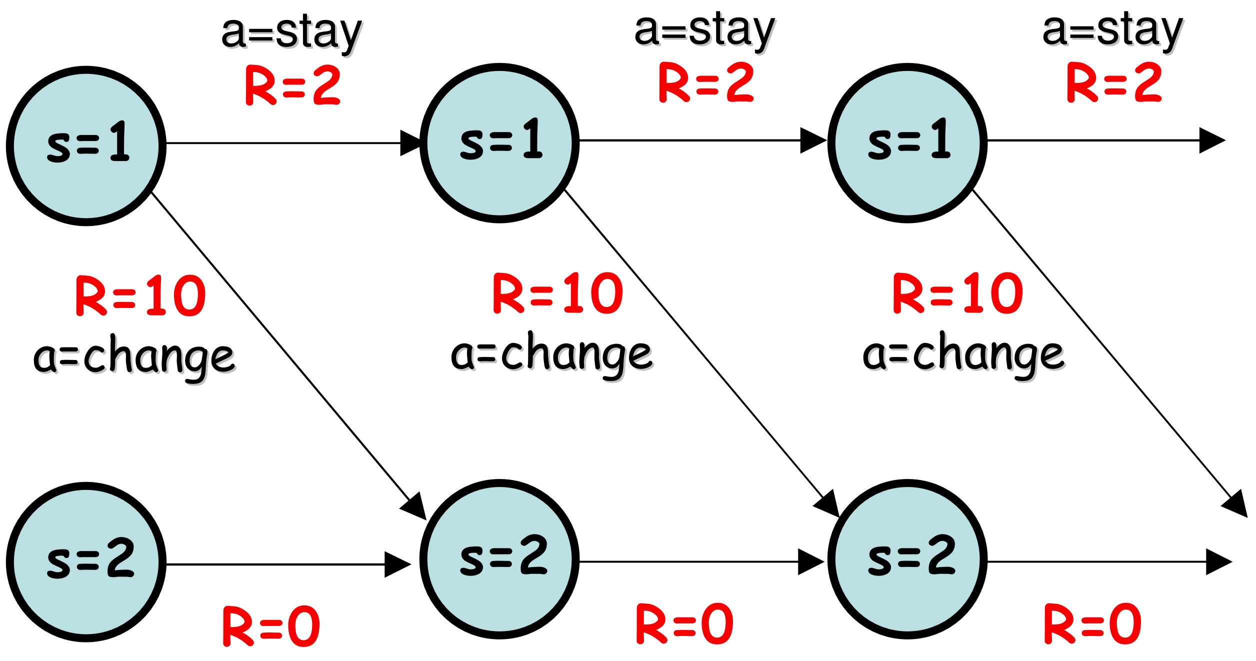
Goals and Rewards

- Goal-oriented rewards
  - Assign any reward value s.t. *R(success) > R(fail)*
  - Can have negative costs *C(a)* for action *a*
    - Known as *stochastic shortest path* problems

- What if multiple (or no) goals?
  - How to specify preferences?
  - *R(s,a)* assigns utilities to each state *s* and action *a*
    - Then *maximize expected utility*

… but how to trade off rewards over time?

Optimization: Best Action when s=1?



- Must define objective criterion to optimize!
  - How to trade off immediate vs. future reward?
  - E.g., use discount factor γ (try γ=.9 vs. γ=.1)

Trading Off Sequential Rewards

- **Sequential-decision making objective**
  - **Horizon**
    - *Finite*: Only care about h-steps into future
    - Infinite: Literally; will act same today as tomorrow
  - **How to trade off reward over time?**
    - *Expected average cumulative return*
    - *Expected discounted cumulative return*
      - Use discount factor γ
        - » Reward t time steps in future discounted by γ^t
      - Many interpretations
        - » Future reward worth less than immediate reward
        - » $(1-\gamma)$ chance of termination at each time step
    - Important property:
        - » cumulative reward finite

Knowledge of Environment

- ***Model-known*:**
  - Know **Z**, **T**, **R**
  - Called: *Planning (under uncertainty)*
    - Planning generally assumed to be goal-oriented
    - *Decision-theoretic* if maximizing expected utility

- ***Model-free*:**
  - At least one of **Z**, **T**, **R** unknown
  - Called: *Reinforcement learning*
    - Have to interact with environment to obtain samples of **Z**, **T**, **R**
    - Use **R** samples as reward *reinforcement* to optimize actions

- **Can still approximate model in model-free case**
  - Permits hybrid planning and learning

Saves expensive interaction!

Finally a Formal Model

- **Environment model: $\langle$S,O,A,Z,T,R$\rangle$**
  - General characteristics
    - Stationary and Markov transitions?
    - Number of agents (1, 2, 3+)
    - Observability (full, partial, none)
  - Objective
    - Horizon
    - Average vs. discounted (γ)
  - Model-based or model-free

Can you provide this description for five previous examples?

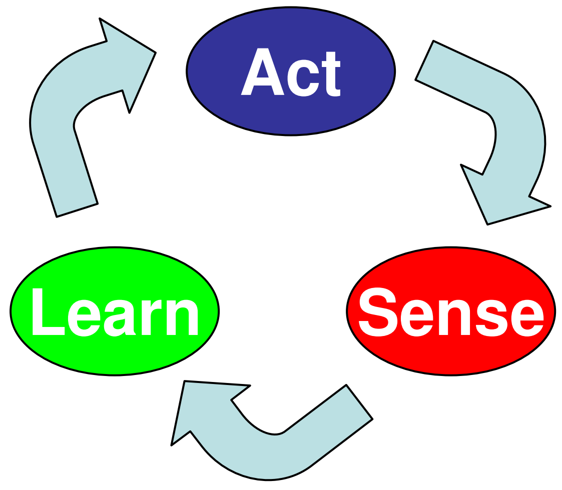
Note: Don't worry about solution just yet, just formalize problem.

- **Different assumptions yield well-studied models**
  - Markovian assumption on **T** frequently made (MDP)

- **Different properties of solutions for each model**
  - That's what this lecture is about!

MDPs and Model-based Solutions

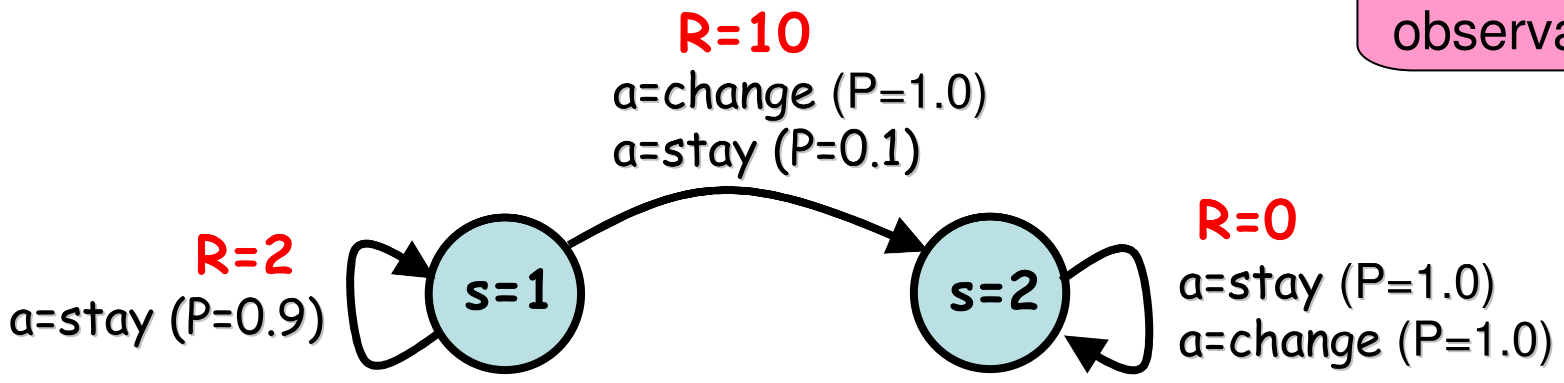
Reinforcement Learning



Scott Sanner
NICTA / ANU
First.Last@nicta.com.au

MDPs $\langle S,A,T,R,\gamma\rangle$

Note: fully observable

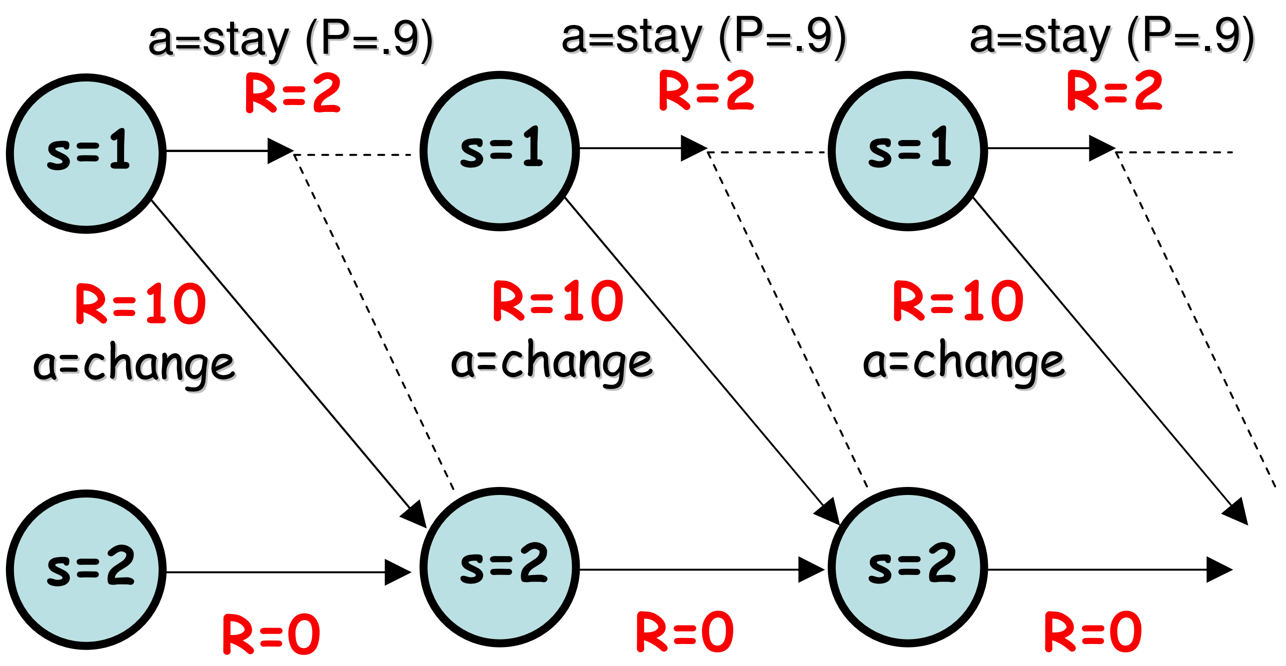


- S = {1,2}; A = {stay, change}
- Reward
  - R(s=1,a=stay) = 2
  - …
- Transitions
  - T(s=1,a=stay,s'=1) = P(s'=1 | s=1, a=stay) = .9
  - …
- Discount γ

How to act in an MDP?

Define policy $\pi: S \rightarrow A$

What's the best Policy?



- Must define reward criterion to optimize!
  - Discount factor γ important (γ=.9 vs. γ=.1)

MDP Policy, Value, & Solution

- Define *value of a policy* π:

$$V_\pi(s) = E_\pi\left[\sum_{t=0}^{\infty} \gamma^t \cdot r_t \middle| s = s_0\right]$$

- Tells how much value you expect to get by following π starting from state *s*

- Allows us to define optimal solution:
  - Find optimal policy π* that maximizes value
  - Surprisingly: $\exists\pi^*. \forall s, \pi.\ V_{\pi^*}(s) \geq V_\pi(s)$
  - Furthermore: always a deterministic π*

Value Function $\rightarrow$ Policy

- Given arbitrary value V (optimal or not)…
  - A *greedy policy* π_V takes action in each state that maximizes expected value w.r.t. *V*:

$$\pi_V(s) = \arg\max_a \left\{ R(s,a) + \gamma \sum_{s'} T(s,a,s')V(s') \right\}$$

  - If can act so as to obtain *V* after doing action *a* in state *s*, π_V guarantees *V(s)* in expectation

If *V* not optimal, but a *lower bound* on *V**, π_V guarantees at least that much value!

Value Iteration: from finite to ∞ decisions

- Given optimal *t-1-stage-to-go* value function
- How to act optimally with *t* decisions?
  - Take action *a* then act so as to achieve V^{t-1} thereafter

$$Q^t(s,a) := R(s,a) + \gamma \cdot \sum_{s' \in S} T(s,a,s') \cdot V^{t-1}(s')$$

  - What is expected value of best action *a* at decision stage *t*?

$$V^t(s) := \max_{a \in A} \left\{ Q^t(s,a) \right\}$$

  - At ∞ horizon, converges to V*

$$\lim_{t \to \infty} \max_s |V^t(s) - V^{t-1}(s)| = 0$$

  - This *value iteration* solution know as *dynamic programming (DP)*

Make sure you can derive these equations from first principles!

Bellman Fixed Point

- Define *Bellman backup* operator *B*:

$$\overbrace{(B\,\underbrace{V}_{V^{t-1}})}^{V^{t}}(s) = \max_a \left\{ R(s,a) + \gamma \sum_{s'} T(s,a,s')V(s') \right\}$$

- ∃ an optimal value function V* and an optimal deterministic greedy policy $\pi^* = \pi_{V^*}$ satisfying:

$$\forall s.\ V^*(s) = (B\,V^*)(s)$$

Bellman Error and Properties

- Define *Bellman error BE*:

$$(BE\,V) = \max_s |(B\,V)(s) - V(s)|$$

- Clearly:

$$(BE\,V^*) = 0$$

- Can prove *B* is a contraction operator for *BE*:

$$(BE\,(B\,V)) \leq \gamma(BE\,V)$$

Hmmm…. Does this suggest a solution?

Value Iteration: in search of fixed-point

- Start with arbitrary value function V^0
- Iteratively apply Bellman backup

$$V^t(s) = (B\,V^{t-1})(s)$$

Look familiar? Same DP solution as before.

- Bellman error decreases on each iteration
  - Terminate when

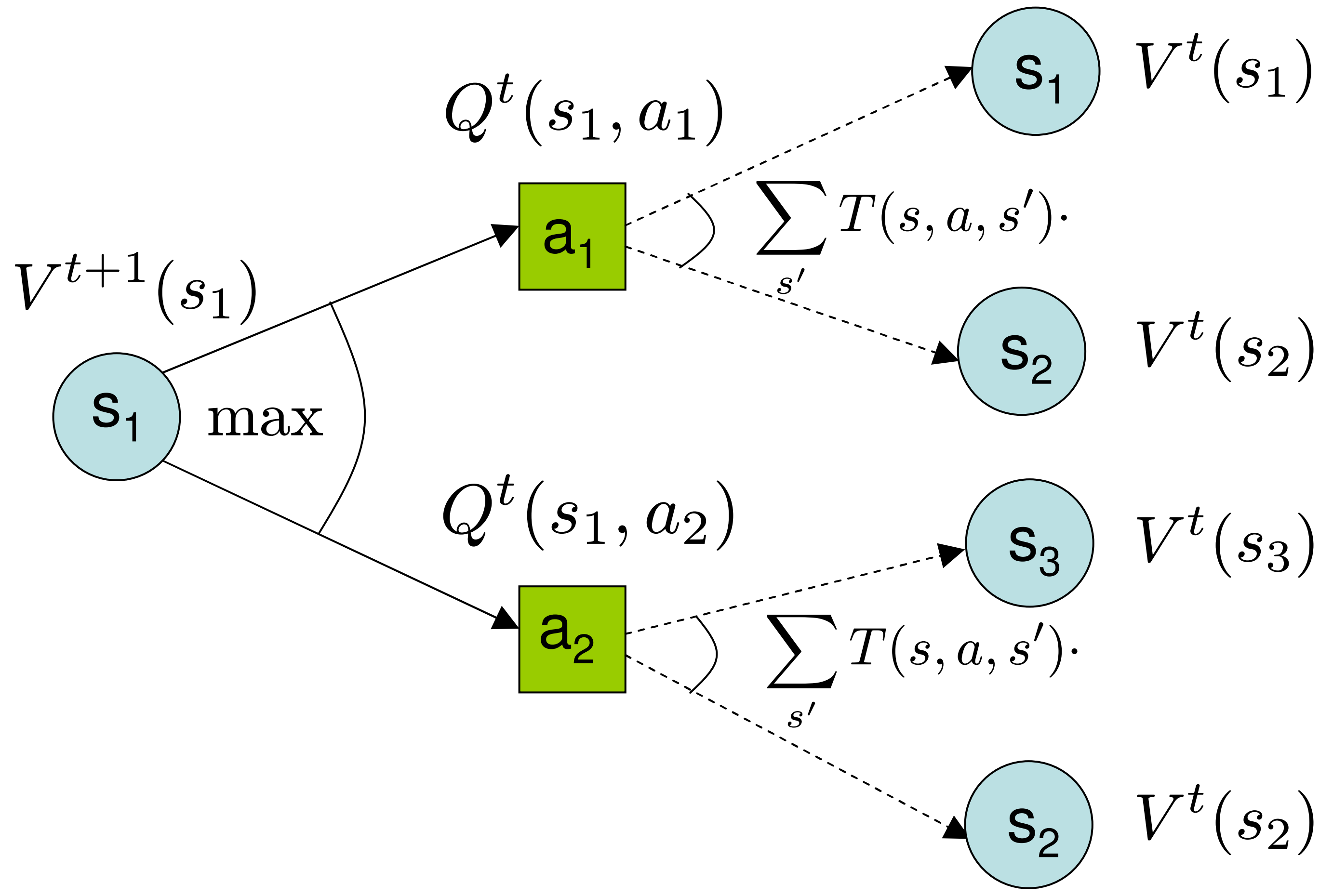
$$\max_s |V^t(s) - V^{t-1}(s)| < \frac{\epsilon(1-\gamma)}{2\gamma}$$

  - Guarantees ε-optimal value function
    - i.e., V^t within ε of V^* for all states

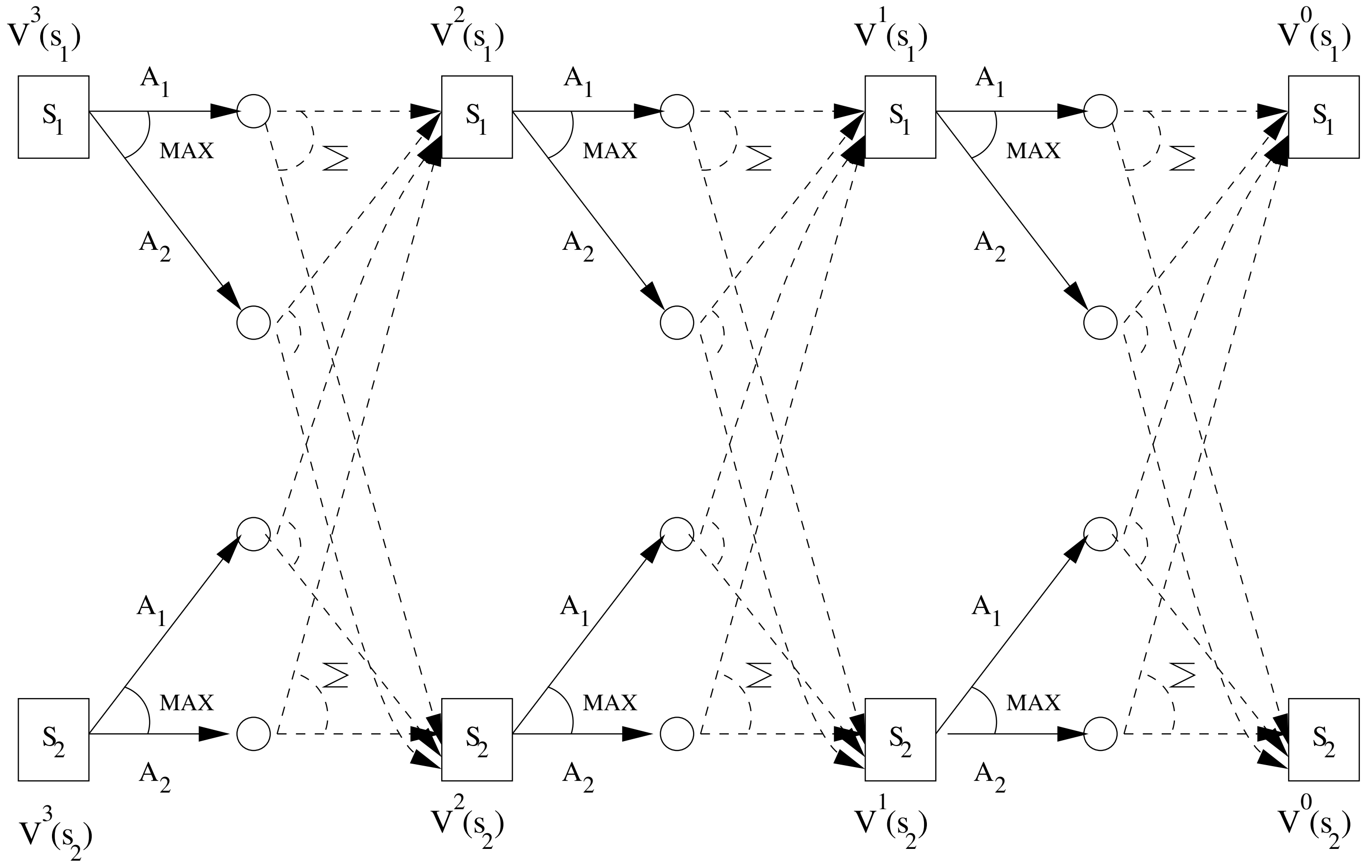
Precompute maximum number of steps for ε?

Single Dynamic Programming (DP) Step

- Graphical view:



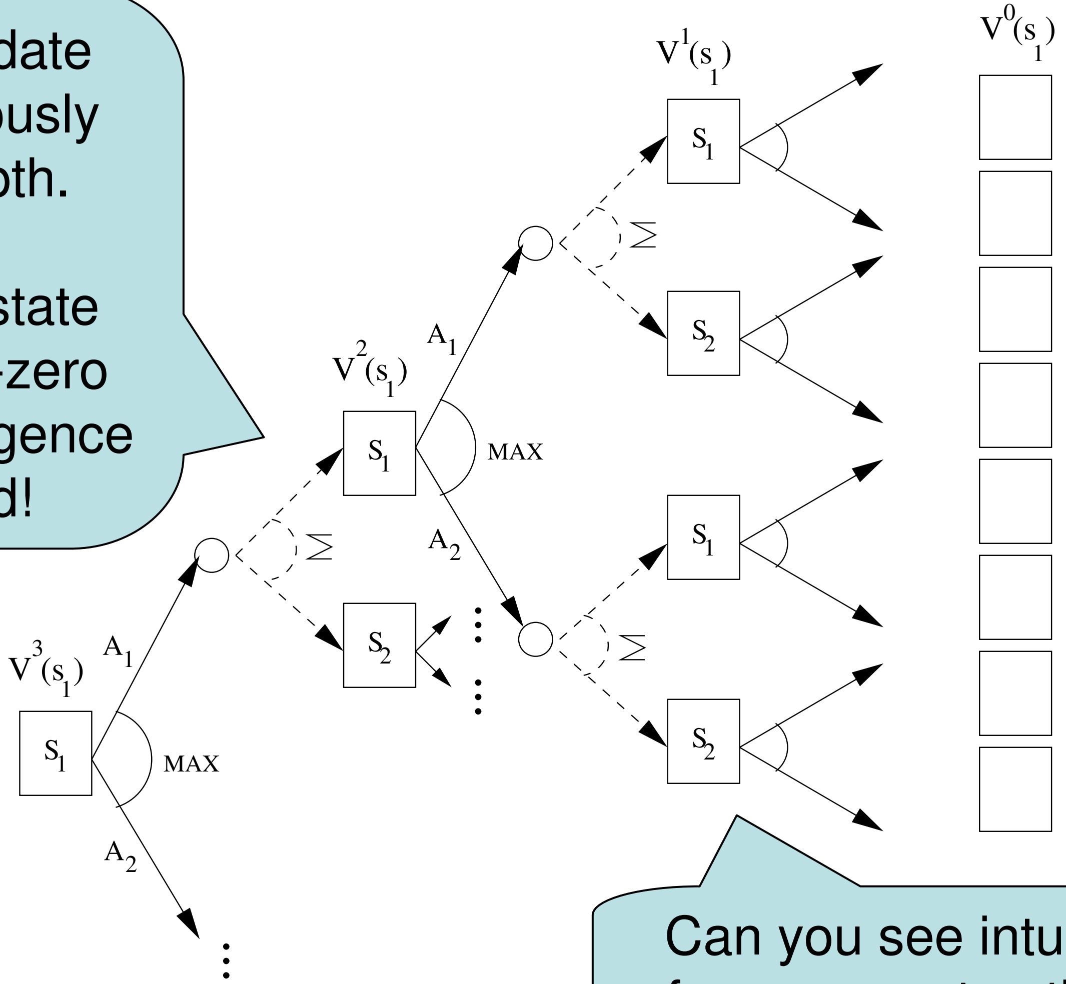
Synchronous DP Updates (Value Iteration)



Asynchronous DP Updates (Asynchronous Value Iteration)

Don't need to update values synchronously with uniform depth.

As long as each state updated with non-zero probability, convergence still guaranteed!



Can you see intuition for error contraction?

Real-time DP (RTDP)

- Async. DP guaranteed to converge over *relevant states*
  - *relevant states*: states reachable from initial states under π*
  - may converge without visiting all states!

Algorithm 1: RTDP (for stochastic shortest path problems)

```
begin
    // Initialize $\hat{V}_h$ with admissible value function
    $\hat{V}_h := V_h$;
    while not converged and not out of time do
        Draw $s$ from initial state distribution at random;
        while $s \notin \{terminal\ states\}$ do
            $\hat{V}_h(s) = \textsc{BellmanUpdate}(\hat{V}_h, s)$;
            $a = \textsc{GreedyAction}(\hat{V}_h, s)$;
            $s := s' \sim T(s, a, s')$;
    return $\hat{V}_h$;
end
```

Prioritized Sweeping (PS)

- Simple asynchronous DP idea
  - Focus backups on high error states
  - Can use in conjunction with other focused methods, e.g., RTDP

Where do RTDP and PS each focus?

- Every time state visited:
  - Record Bellman error of state
  - Push state onto queue with priority = Bellman error

- In between simulations / experience, repeat:
  - Withdraw maximal priority state from queue
  - Perform Bellman backup on state
    - Record Bellman error of predecessor states
    - Push predecessor states onto queue with priority = Bellman error

Which approach is better?

- Synchronous DP Updates
  - Good when you need a policy for every state
  - OR transitions are dense
- Asynchronous DP Updates
  - Know best states to update
    - e.g., reachable states, e.g. RTDP
    - e.g., high error states, e.g. PS
  - Know how to order updates
    - e.g., from goal back to initial state if DAG

Policy Evaluation

- Given π, how to derive V_π?
- *Matrix inversion*
  - Set up linear equality (no max!) for each state

$$\forall s.\ V_\pi(s) = \left\{ R(s,\pi(s)) + \gamma \sum_{s'} T(s,\pi(s),s') V_\pi(s') \right\}$$

  - Can solve linear system in vector form as follows

$$V_\pi = R_\pi (I - \gamma T_\pi)^{-1}$$

Guaranteed invertible.

- *Successive approximation*
  - Essentially value iteration with fixed policy
  - Initialize V_π^0 arbitrarily

$$V_\pi^t(s) := \left\{ R(s,\pi(s)) + \gamma \sum_{s'} T(s,\pi(s),s') V_\pi^{t-1}(s') \right\}$$

  - Guaranteed to converge to V_π

Policy Iteration

1. *Initialization:* Pick an arbitrary initial decision policy $\pi_0 \in \Pi$ and set $i = 0$.

2. *Policy Evaluation:* Solve for V_{π_i} (previous slide).

3. *Policy Improvement:* Find a new policy π_{i+1} that is a greedy policy w.r.t. V_{π_i}

   (i.e., $\pi_{i+1} \in \arg\max_{\pi \in \Pi} \{R_\pi + \gamma T_\pi V_{\pi_i}\}$ with ties resolved via a total precedence order over actions).

4. *Termination Check:* If $\pi_{i+1} \neq \pi_i$ then increment i and go to step 2 else return π_{i+1}.

Modified Policy Iteration

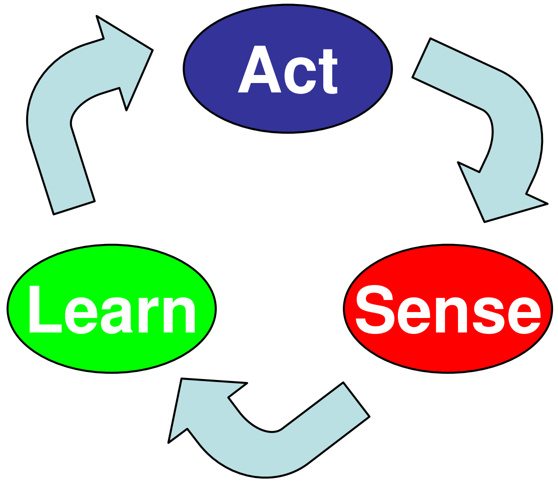
- *Value iteration*
  - Each iteration seen as doing 1-step of policy evaluation for current greedy policy
  - Bootstrap with value estimate of previous policy
- *Policy iteration*
  - Each iteration is full evaluation of V_{π} for current policy π
  - Then do greedy policy update
- *Modified policy iteration*
  - Like policy iteration, but $V_{\pi i}$ need only be closer to V* than $V_{\pi i-1}$
    - Fixed number of steps of successive approximation for $V_{\pi i}$ suffices when bootstrapped with $V_{\pi i-1}$
  - Typically faster than VI & PI in practice

Conclusion

- Basic introduction to MDPs
  - Bellman equations from first principles
  - Solution via various algorithms
- Should be familiar with *model-based* solutions
  - Value Iteration
    - Synchronous DP
    - Asynchronous DP (RTDP, PS)
  - (Modified) Policy Iteration
    - Policy evaluation
- Model-free solutions just sample from above

Model-free MDP Solutions

Reinforcement Learning



Scott Sanner
NICTA / ANU
First.Last@nicta.com.au

Chapter 5: Monte Carlo Methods

Reinforcement Learning, Sutton & Barto, 1998. Online.

- Monte Carlo methods learn from sample returns
  - Sample from
    - experience in real application, or
    - simulations of known model
  - Only defined for episodic (terminating) tasks
  - *On-line:* Learn while acting

Slides from Rich Sutton's course CMP499/609
Reinforcement Learning in AI
http://rlai.cs.ualberta.ca/RLAI/RLAIcourse/RLAIcourse2007.html

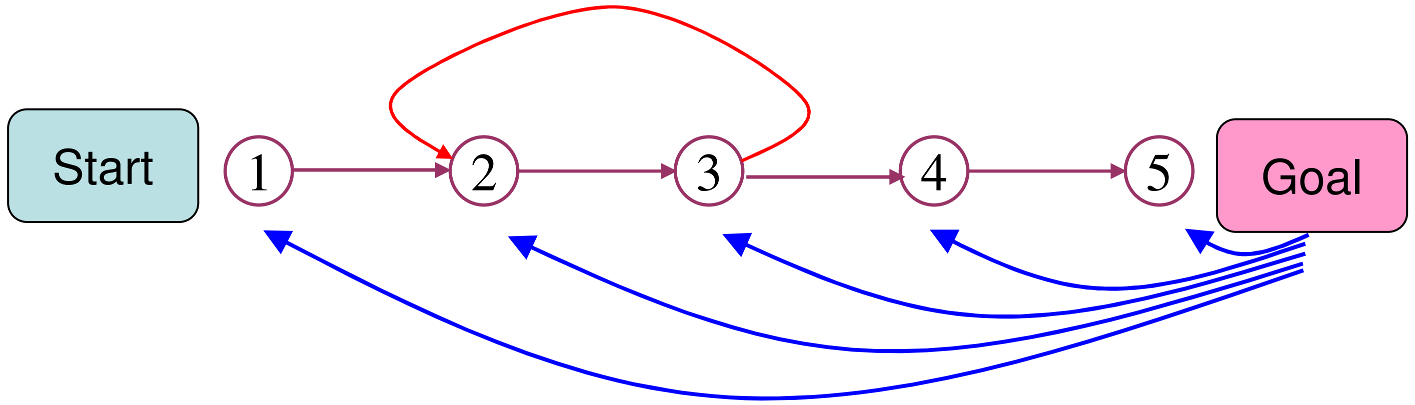
Essence of Monte Carlo (MC)

- MC samples directly from *value expectation* for each state given π

$$V_\pi(s) = E_\pi\left[\sum_{t=0}^{\infty} \gamma^t \cdot r_t \middle| s_0 = s\right]$$

Monte Carlo Policy Evaluation

- *Goal:* learn $V^{\pi}(s)$
- *Given:* some number of episodes under π which contain *s*
- *Idea:* Average returns observed after visits to s



update each state with final discounted return

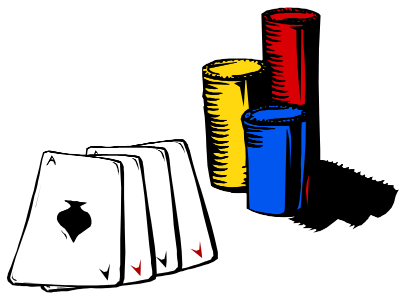
Monte Carlo policy evaluation

```
Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
    (a) Generate an episode using $\pi$
    (b) For each state $s$ appearing in the episode:
            $R \leftarrow$ return following the first occurrence of $s$
            Append $R$ to $Returns(s)$
            $V(s) \leftarrow$ average$(Returns(s))$
```

Blackjack example

- *Object:* Have your card sum be greater than the dealer's without exceeding 21.
- *States* (200 of them):
  - current sum (12-21)
  - dealer's showing card (ace-10)
  - do I have a useable ace?

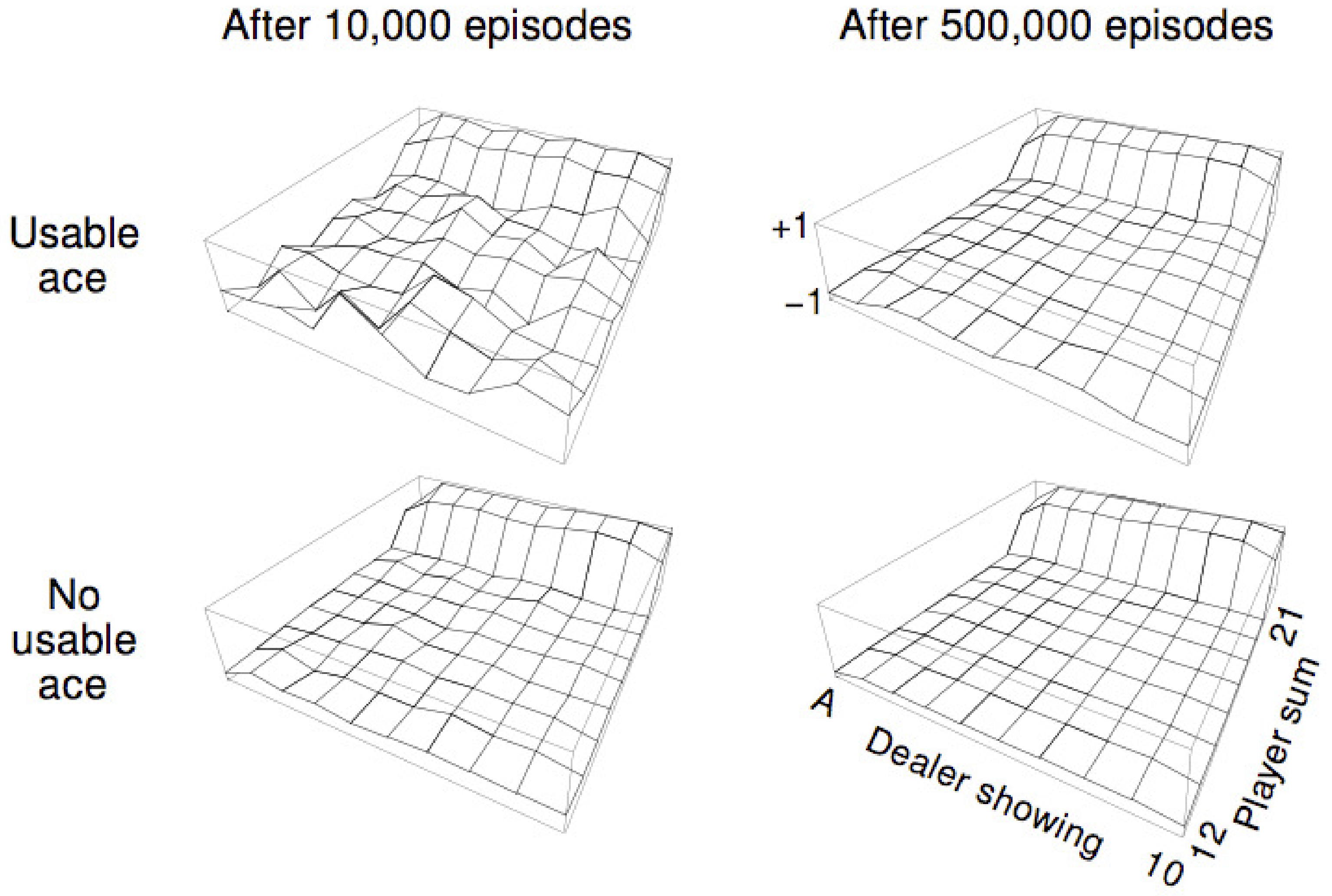


- *Reward:* +1 win, 0 draw, -1 lose
- *Actions:* stick (stop receiving cards), hit (receive another card)

Assuming fixed policy for now.

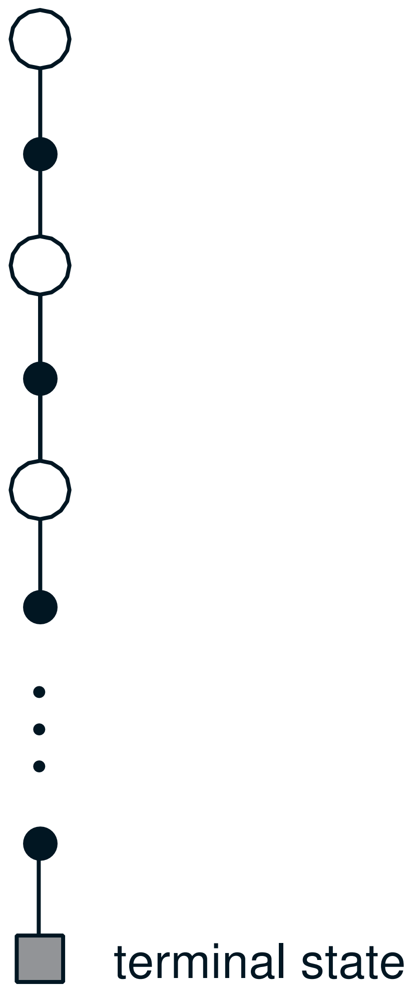
- *Policy:* Stick if my sum is 20 or 21, else hit

Blackjack value functions



Backup diagram for Monte Carlo

- Entire episode included
- Only one choice at each state (unlike DP)
- MC does not bootstrap
- Time required to estimate one state does not depend on the total number of states



MC Control: Need for Q-values

- Control: want to learn a good policy
  - Not just evaluate a given policy
- If no model available
  - Cannot execute policy based on *V(s)*
  - Instead, want to learn *$Q^*(s,a)$*
- *$Q^\pi(s,a)$* - average return starting from state *s* and action *a* following *π*

$$Q^\pi(s,a) = E_\pi\left[\sum_{t=0}^{\infty}\gamma^t \cdot r_t \middle| s_0 = s, a_0 = a\right]$$

Monte Carlo Control

$$\begin{tikzcd}[column sep=huge]
\pi \arrow[r, bend left, "\text{evaluation}", "Q \to Q^{\pi}"'] & Q \arrow[l, bend left, "\pi \to \text{greedy}(Q)"', "\text{improvement}"]
\end{tikzcd}$$

Instance of **Generalized Policy Iteration.**

- MC policy iteration: Policy evaluation using MC methods followed by policy improvement
- Policy improvement step: Greedy $\pi'(s)$ is action a maximizing $Q^{\pi}(s,a)$

Convergence of MC Control

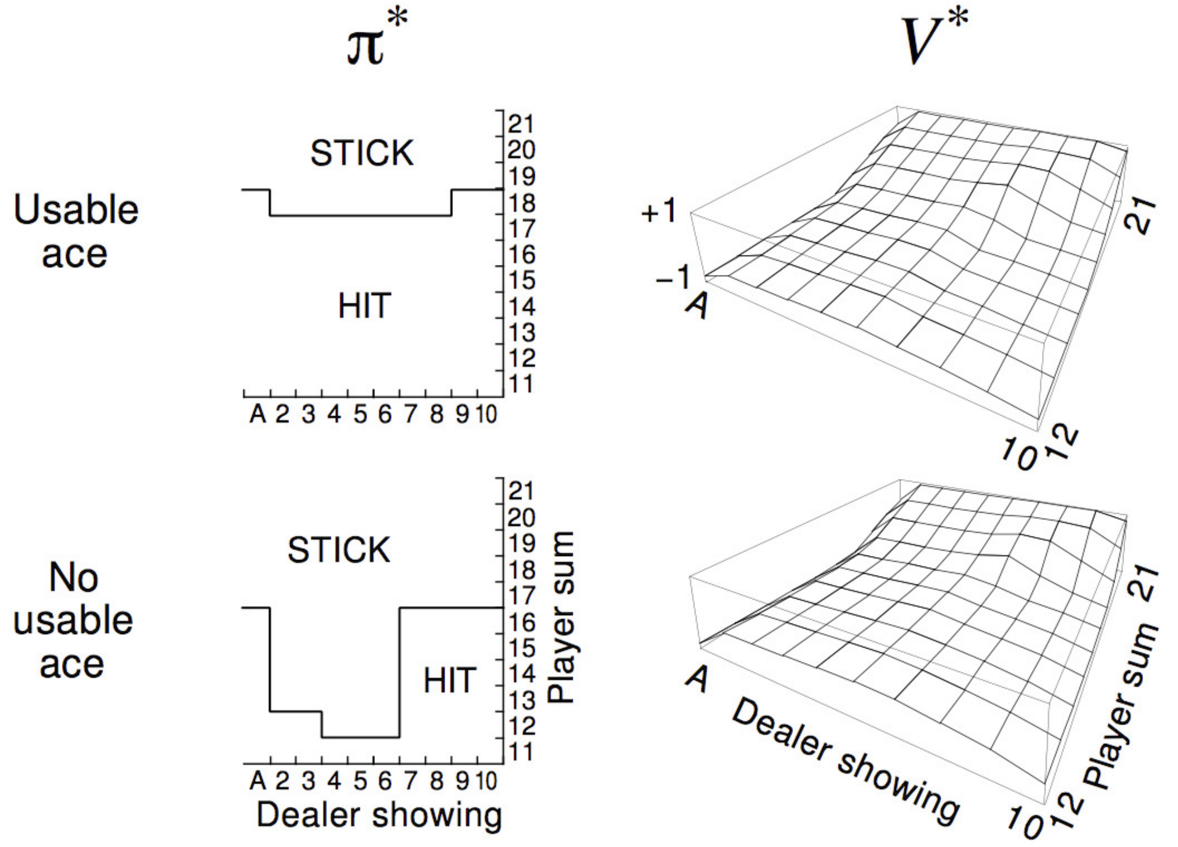
- Greedy policy update improves or keeps value:

$$\begin{aligned}
Q^{\pi_k}(s, \pi_{k+1}(s)) &= Q^{\pi_k}(s, \arg\max_a Q^{\pi_k}(s,a)) \\
&= \max_a Q^{\pi_k}(s,a) \\
&\geq Q^{\pi_k}(s, \pi_k(s)) \\
&= V^{\pi_k}(s).
\end{aligned}$$

- This assumes all Q(s,a) visited an infinite number of times
  - Requires exploration, not just exploitation
- In practice, update policy after finite iterations

Blackjack Example Continued

- MC Control with exploring starts…
- Start with random (s,a) then follow π



Monte Carlo Control

- How do we get rid of exploring starts?
  - Need *soft* policies: $\pi(s,a) > 0$ for all *s* and *a*
  - e.g. ε-soft policy:

$$\frac{\varepsilon}{|A(s)|} \qquad\qquad 1-\varepsilon+\frac{\varepsilon}{|A(s)|}$$

non-max greedy

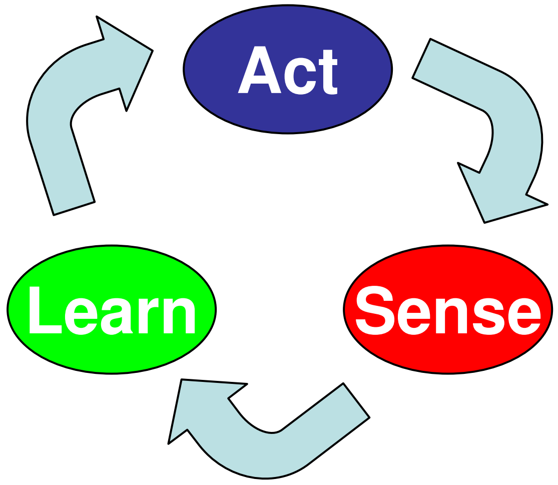
- Similar to GPI: move policy *towards* greedy policy (i.e. ε-soft)
- Converges to best ε-soft policy

Summary

- MC has several advantages over DP:
  - Learn from direct interaction with environment
  - No need for full models
  - Less harm by Markovian violations
- MC methods provide an alternate policy evaluation process
- No bootstrapping (as opposed to DP)

Temporal Difference Methods

Reinforcement Learning



Scott Sanner
NICTA / ANU
First.Last@nicta.com.au

Chapter 6: Temporal Difference (TD) Learning

Reinforcement Learning, Sutton & Barto, 1998. Online.

- Rather than sample full returns as in Monte Carlo…

  TD methods sample Bellman backup

Slides from Rich Sutton's course CMP499/609
Reinforcement Learning in AI
http://rlai.cs.ualberta.ca/RLAI/RLAIcourse/RLAIcourse2007.html

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 :

$$V(s_t) \leftarrow V(s_t) + \alpha \big[\underset{\uparrow}{R_t} - V(s_t) \big]$$

target: the actual return after time *t*

The simplest TD method, TD(0) :

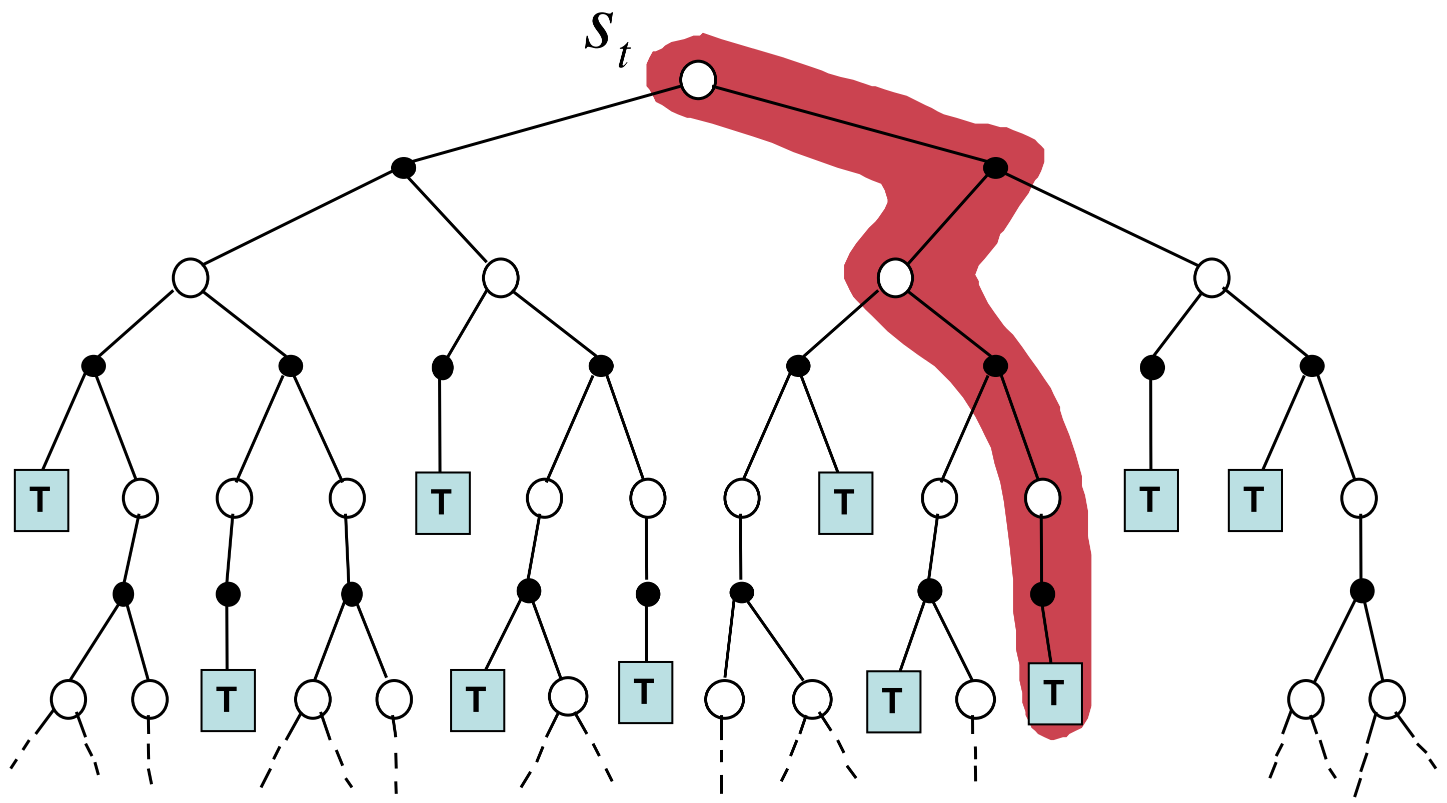
$$V(s_t) \leftarrow V(s_t) + \alpha \big[\underbrace{r_{t+1} + \gamma V(s_{t+1})} - V(s_t) \big]$$

target: an estimate of the return

Simple Monte Carlo

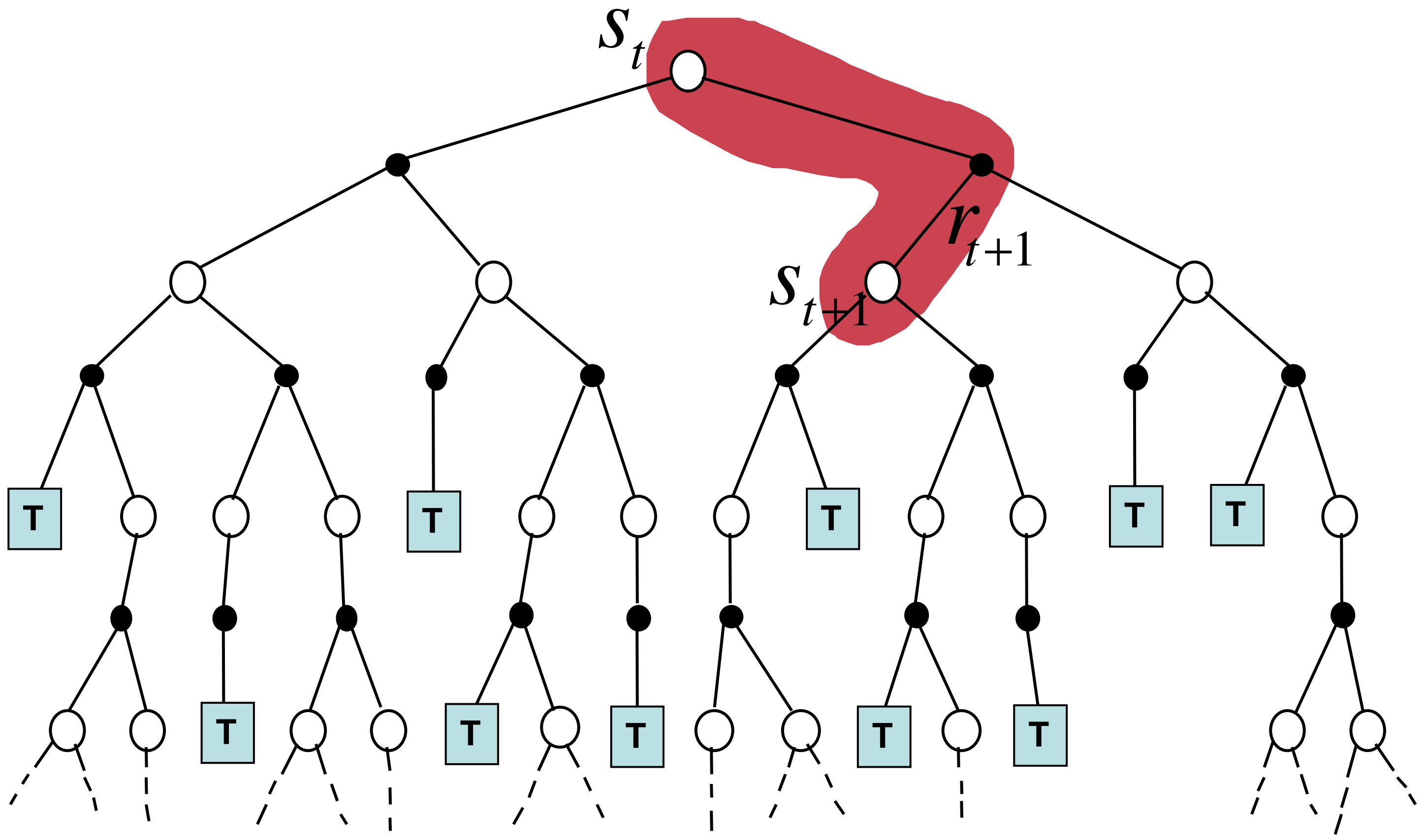
$$V(s_t) \leftarrow V(s_t) + \alpha\left[R_t - V(s_t)\right]$$

where R_t is the actual return following state s_t.



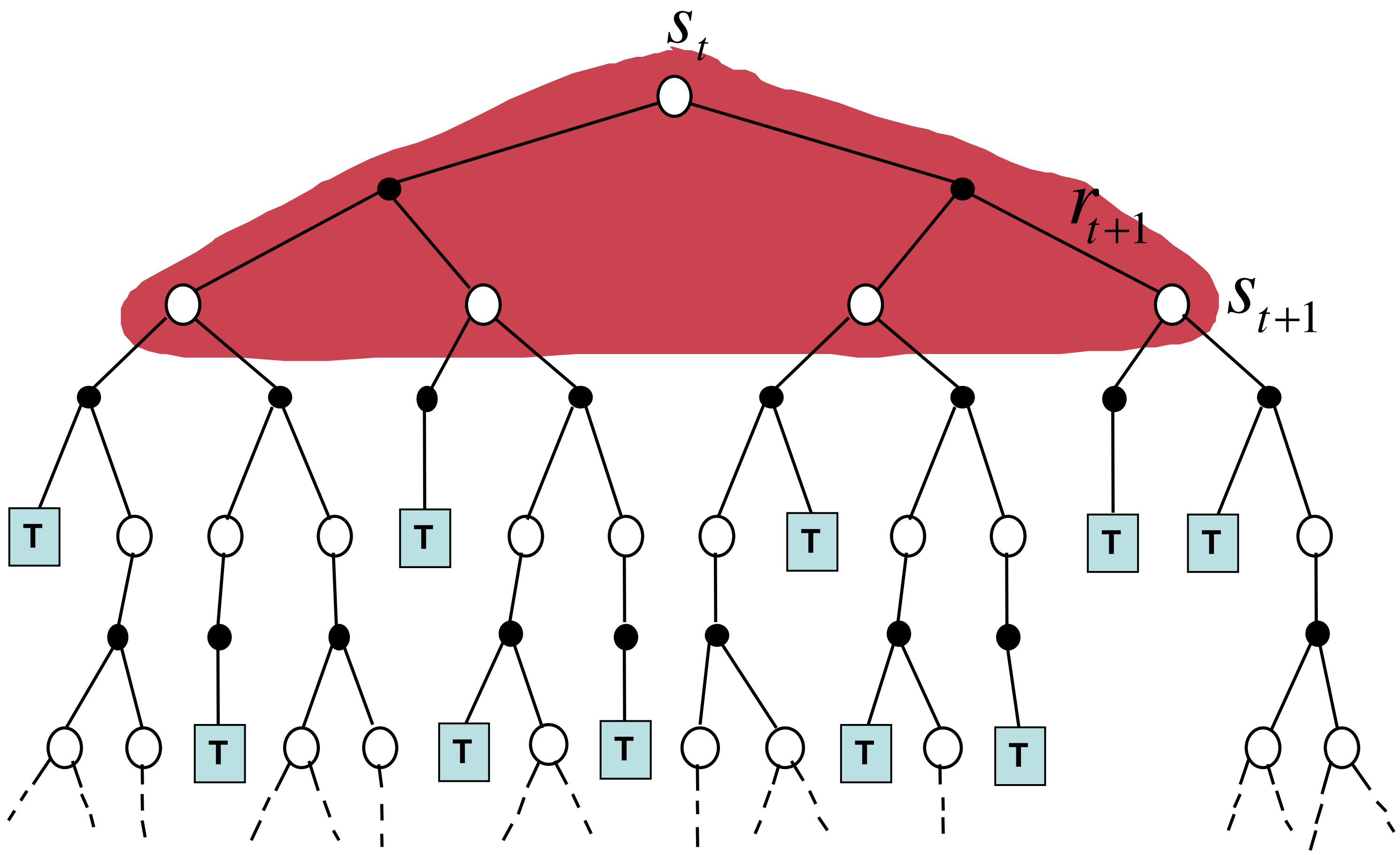
Simplest TD Method

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$



cf. Dynamic Programming

$$V(s_t) \leftarrow E_\pi \{r_{t+1} + \gamma V(s_t)\}$$



TD methods bootstrap and sample

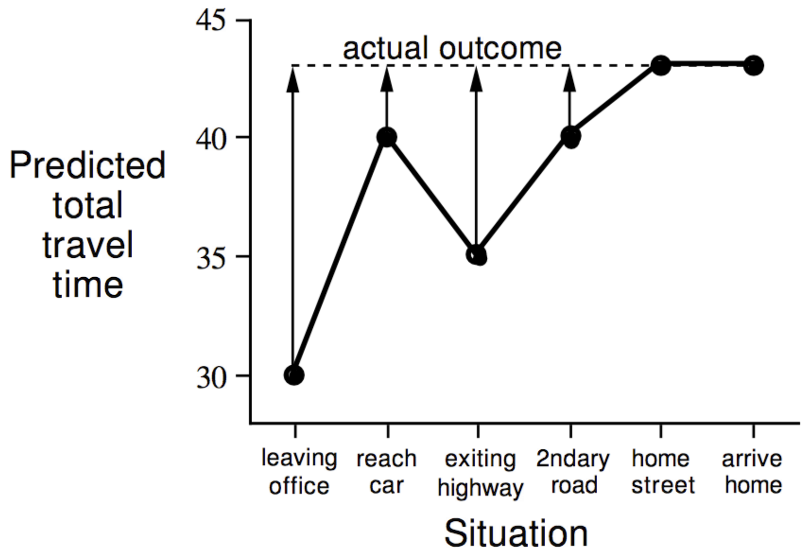
- Bootstrapping: update with *estimate*
  - MC does not bootstrap
  - DP bootstraps
  - TD bootstraps
- Sampling:
  - MC samples
  - DP does not sample
  - TD samples

Example: Driving Home

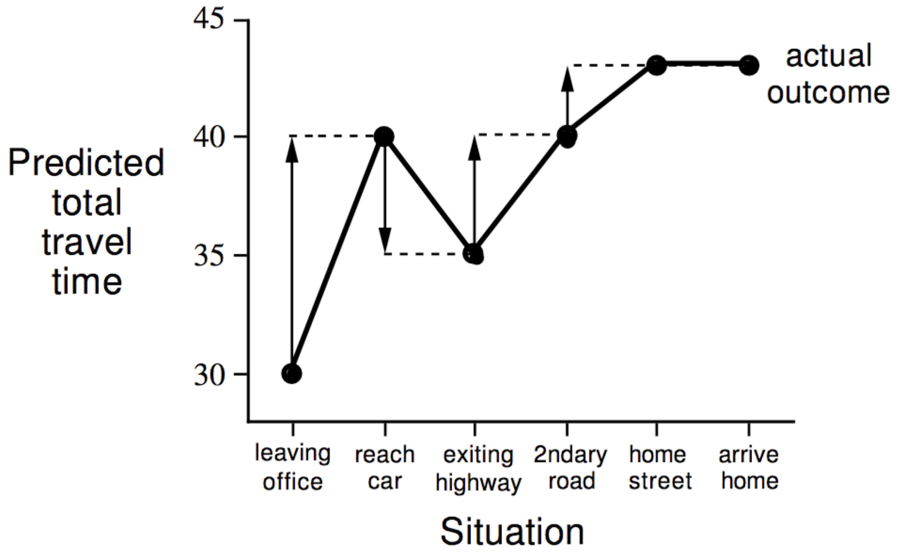
State	Elapsed Time (minutes)		Predicted Time to Go	Predicted Total Time
leaving office	0		30	30
reach car, raining	5	(5)	35	40
exit highway	20	(15)	15	35
behind truck	30	(10)	10	40
home street	40	(10)	3	43
arrive home	43	(3)	0	43

Driving Home

Changes recommended by Monte Carlo methods (α=1)



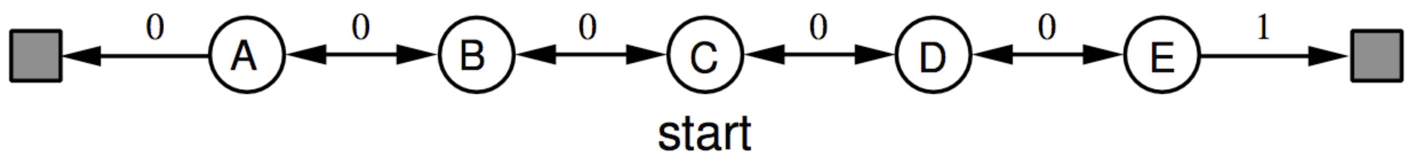
Changes recommended by TD methods (α=1)

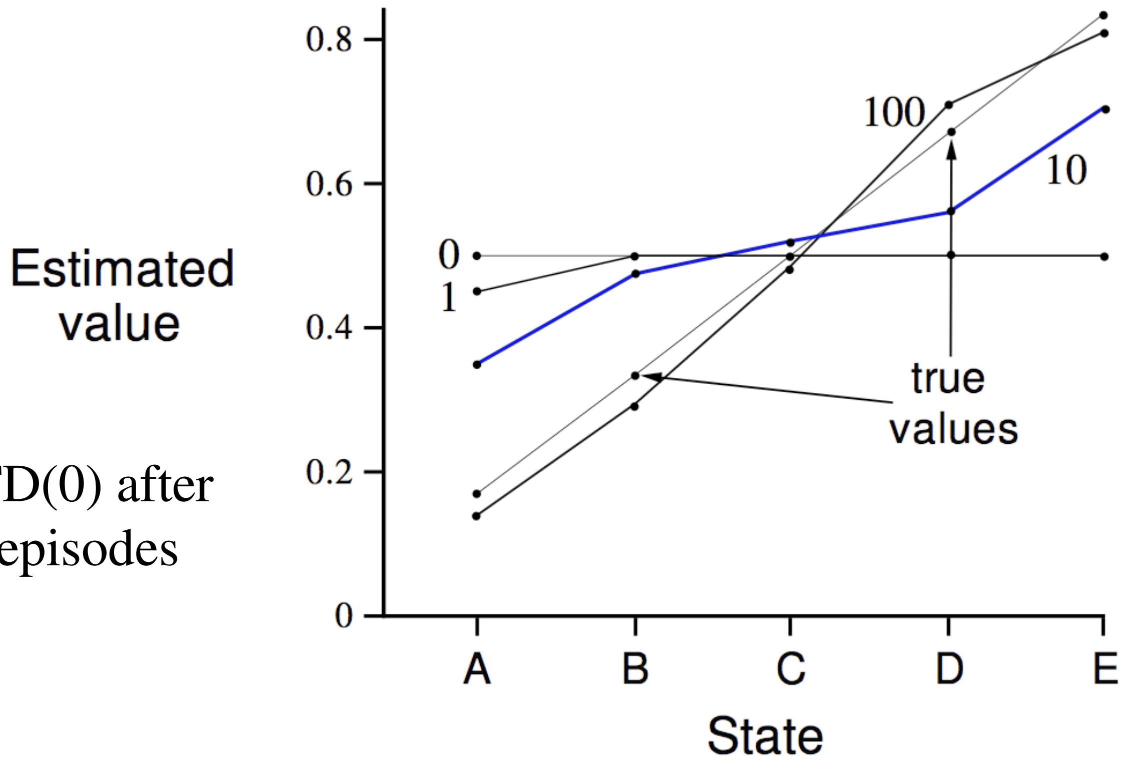


Advantages of TD Learning

- TD methods do not require a model of the environment, only experience
- TD, but not MC, methods can be fully incremental
  - You can learn before knowing the final outcome
    - Less memory
    - Less peak computation
  - You can learn without the final outcome
    - From incomplete sequences
- Both MC and TD converge, but which is faster?

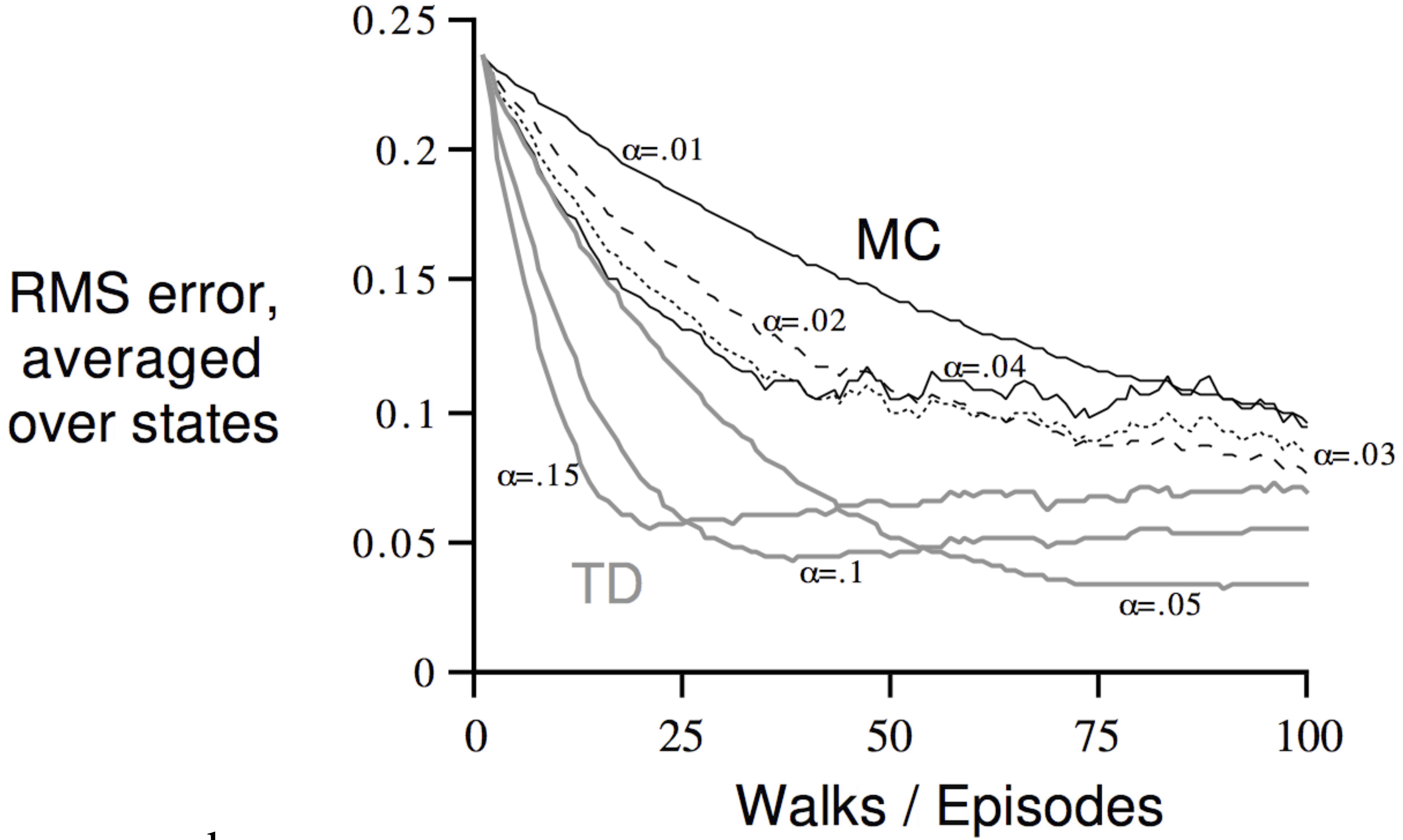
Random Walk Example





Values learned by TD(0) after various numbers of episodes

TD and MC on the Random Walk



Data averaged over
100 sequences of episodes

Optimality of TD(0)

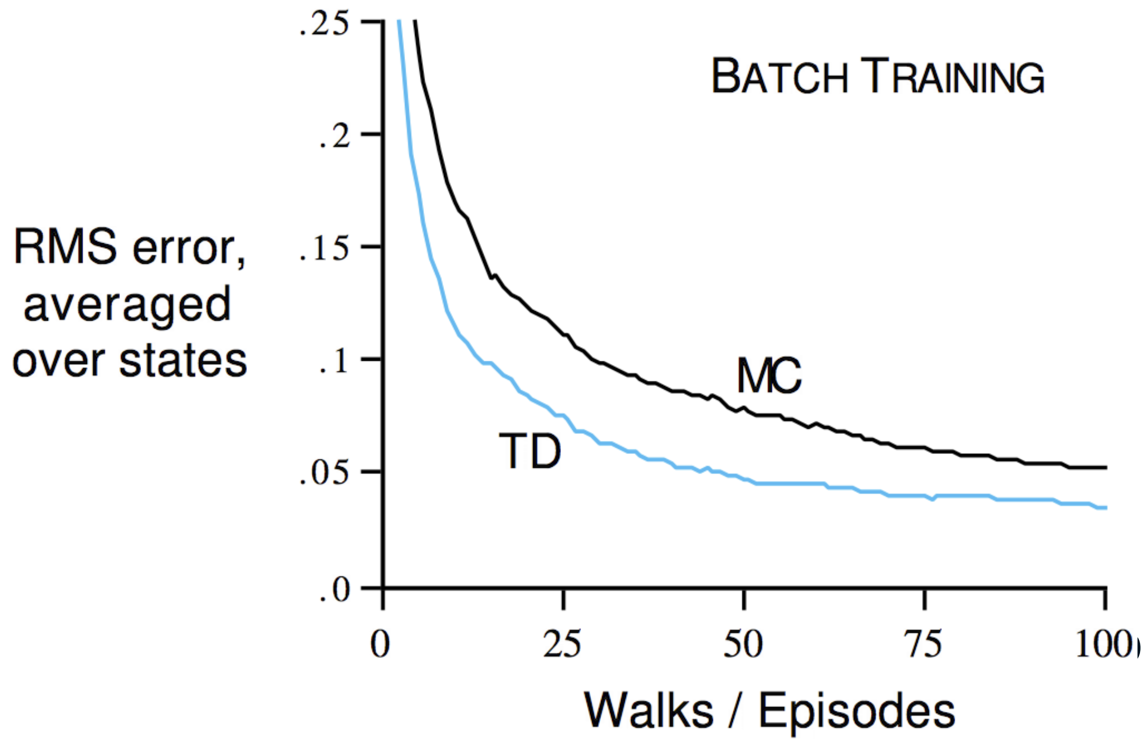
Batch Updating: train completely on a finite amount of data, e.g., train repeatedly on 10 episodes until convergence.

Only update estimates after complete pass through the data.

For any finite Markov prediction task, under batch updating, TD(0) converges for sufficiently small α.

Constant-α MC also converges under these conditions, but to a different answer!

Random Walk under Batch Updating



After each new episode, all previous episodes were treated as a batch, and algorithm was trained until convergence. All repeated 100 times.

You are the Predictor

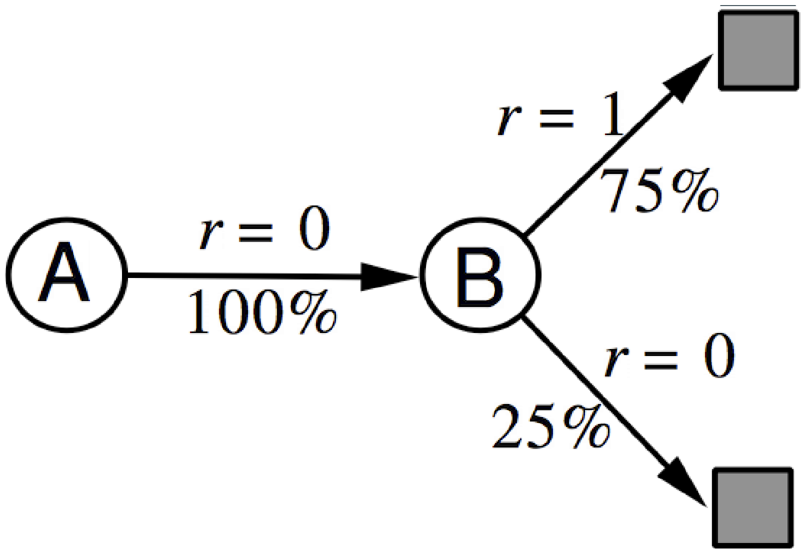
Suppose you observe the following 8 episodes:

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0

$V(\text{A})?$

$V(\text{B})?$

You are the Predictor



$V(\mathrm{A})$?

You are the Predictor

- The prediction that best matches the training data is V(A)=0
  - This minimizes the mean-square-error
  - This is what a batch Monte Carlo method gets

- If we consider the sequential aspect 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
  - This is what TD(0) gets

MC and TD results are same in ∞ limit of data.

But what if data < ∞?

Sarsa: On-Policy TD Control

Turn this into a control method by always updating the policy to be greedy with respect to the current estimate:

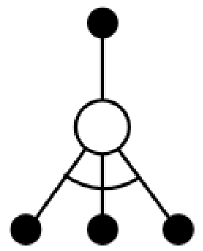
```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until $s$ is terminal
```

SARSA = TD(0) for Q functions.

Q-Learning: Off-Policy TD Control

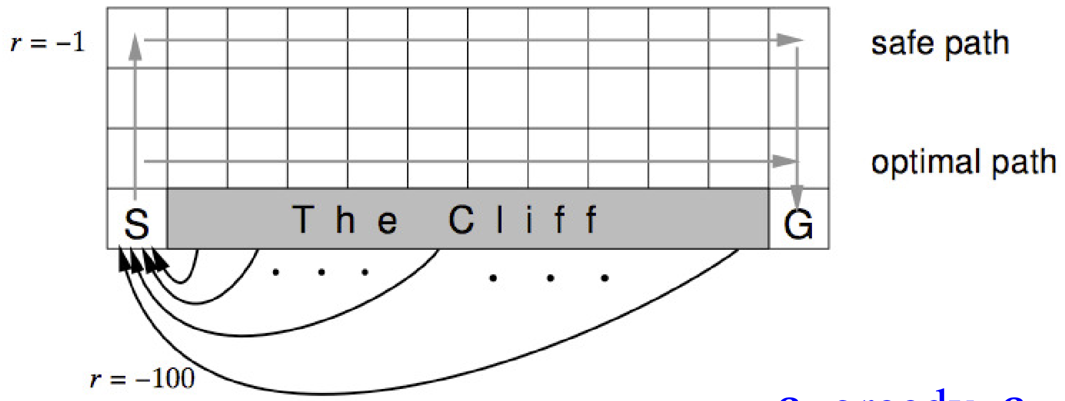
One - step Q - learning :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

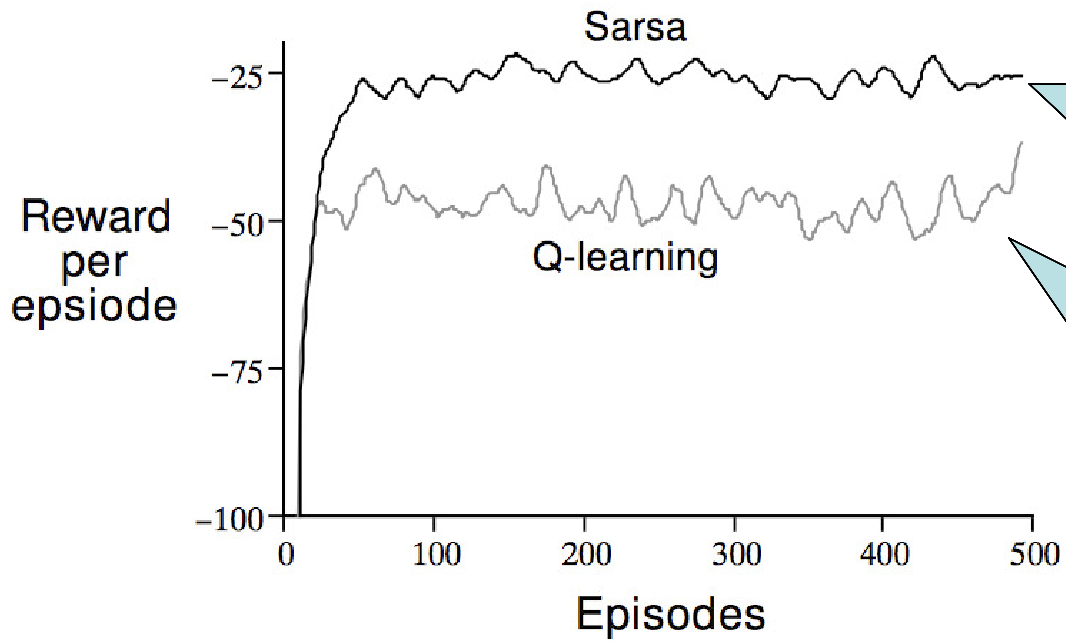


```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Cliffwalking



ε−greedy, ε = 0.1

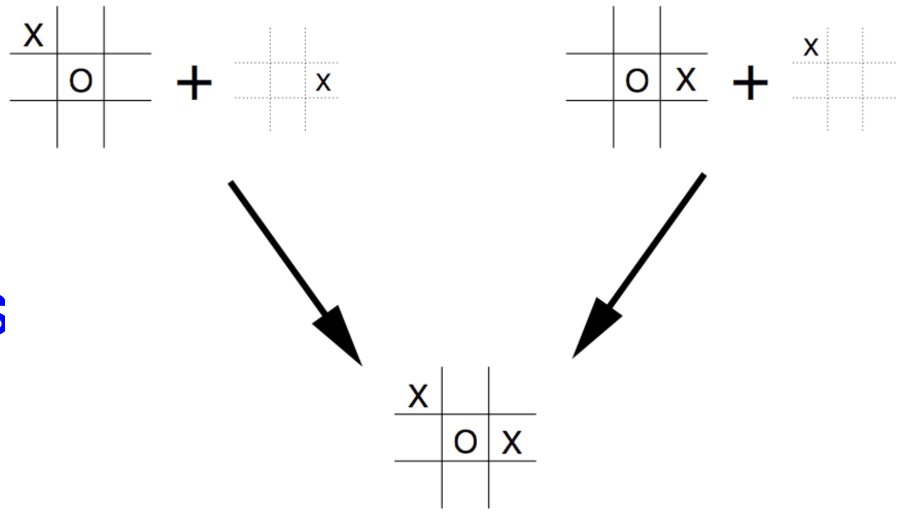


Optimal *exploring* policy.

Optimal policy, but exploration hurts more here.

Practical Modeling: Afterstates

- Usually, a state-value function evaluates states in which the agent can take an action.
- But sometimes it is useful to evaluate states after agent has acted, as in tic-tac-toe.
- Why is this useful?
- An afterstate is really just an action that looks like a state



Summary

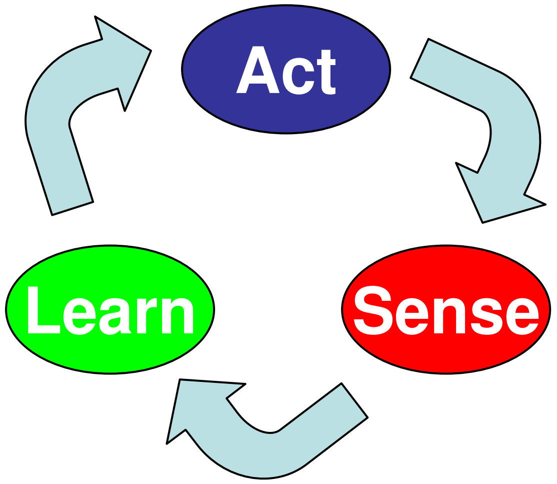
- TD prediction
- Introduced *one-step tabular model-free TD methods*
- Extend prediction to control
  - On-policy control: Sarsa (instance of GPI)
  - Off-policy control: Q-learning

a.k.a. bootstrapping

- These methods sample from Bellman backup, combining aspects of DP and MC methods

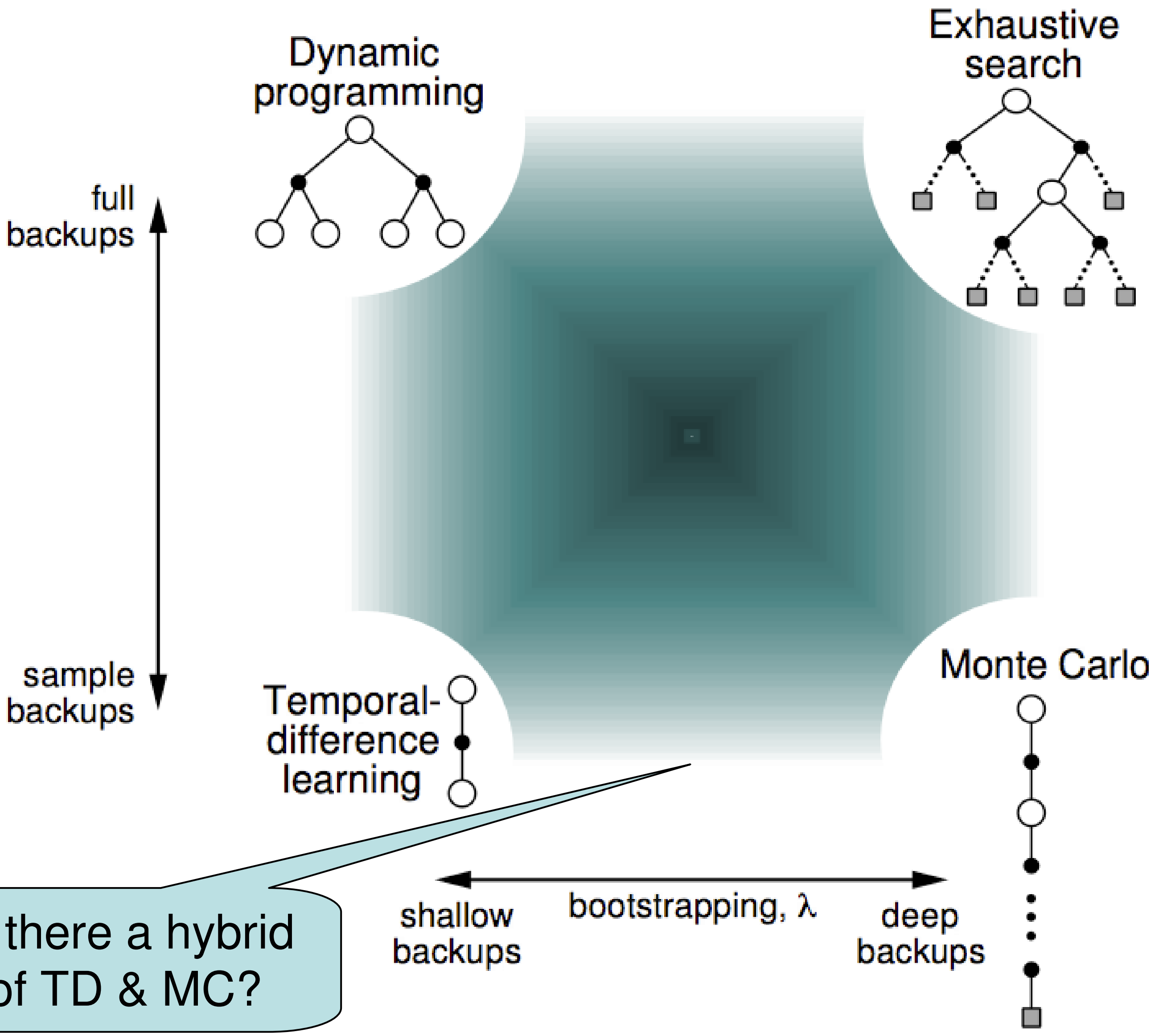
TD(λ): Between TD and MC

Reinforcement Learning



Scott Sanner
NICTA / ANU
First.Last@nicta.com.au

Unified View



Is there a hybrid of TD & MC?

Chapter 7: TD(λ)

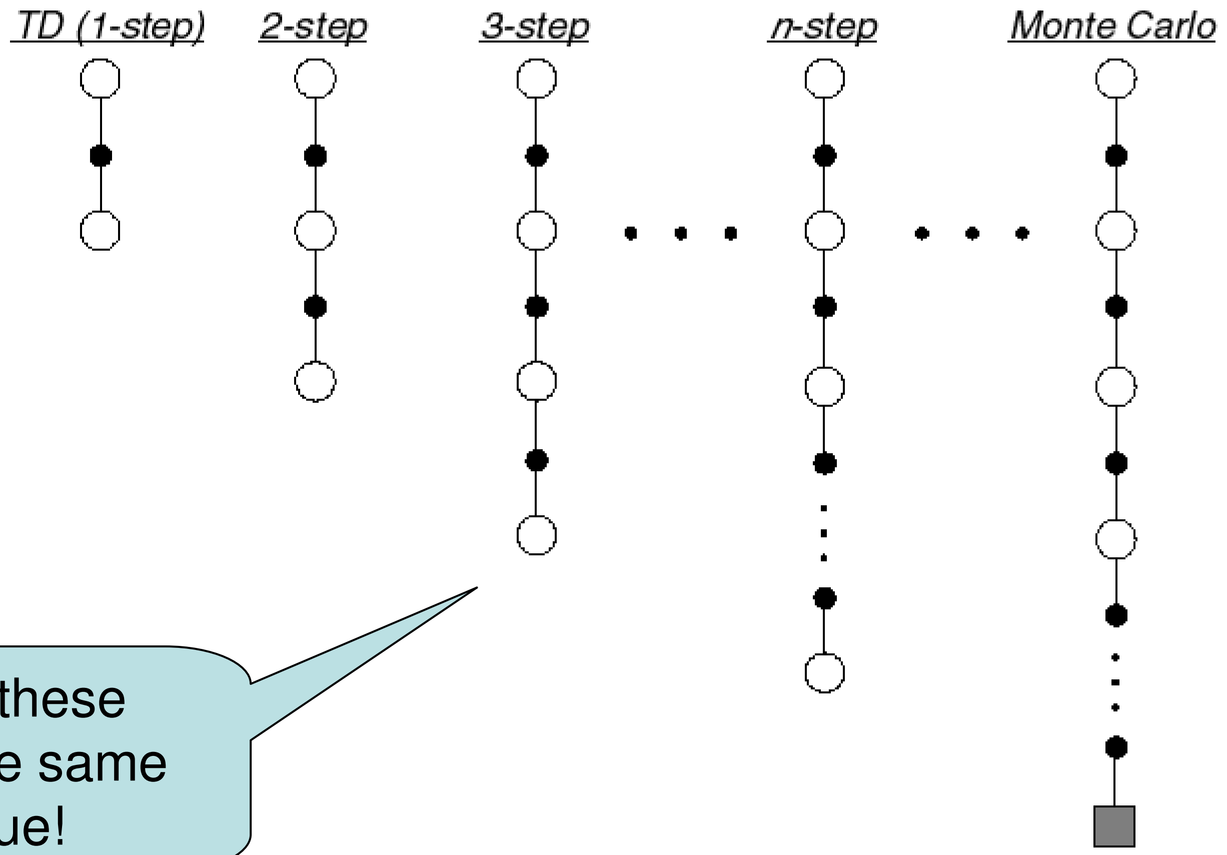
Reinforcement Learning, Sutton & Barto, 1998. Online.

- MC and TD estimate same value
  - More estimators between two extremes
- Idea: average multiple estimators
  - Yields lower variance
  - Leads to faster learning

Slides from Rich Sutton's course CMP499/609
Reinforcement Learning in AI
http://rlai.cs.ualberta.ca/RLAI/RLAIcourse/RLAIcourse2007.html

N-step TD Prediction

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

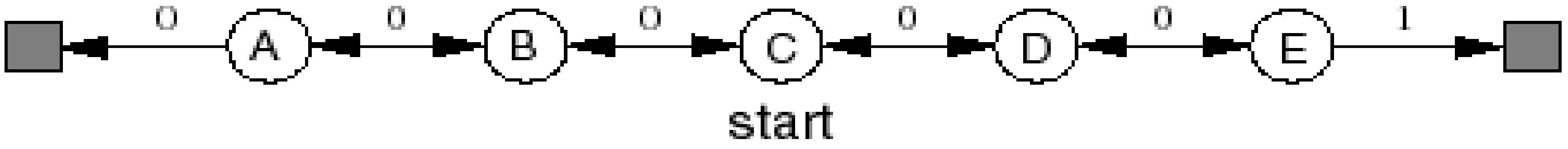


All of these estimate same value!

N-step Prediction

- Monte Carlo: $R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T$
- TD: $R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1})$
  - Use V to estimate remaining return
- n-step TD:
  - 2 step return: $R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})$
  - n-step return: $R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$

Random Walk Examples



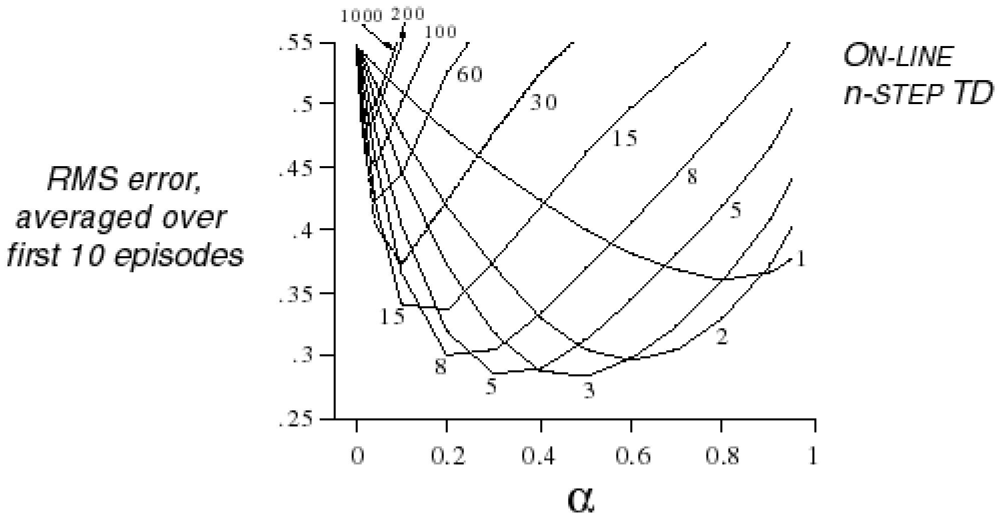
- How does 2-step TD work here?
- How about 3-step TD?

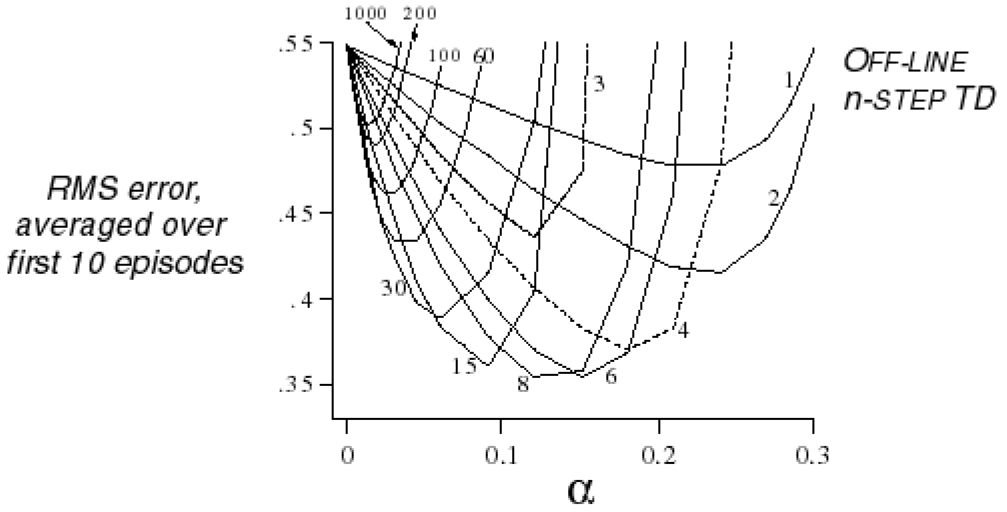
Hint: TD(0) is 1-step return… update previous state on each time step.

A Larger Example

- Task: 19 state random walk

- Do you think there is an optimal n? for everything?



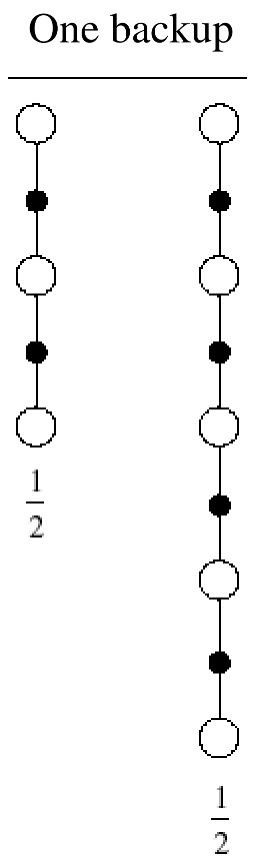


Averaging N-step Returns

- n-step methods were introduced to help with TD(λ) understanding
- Idea: backup an average of several returns
  - e.g. backup half of 2-step & 4-step

$$R_t^{avg} = \frac{1}{2} R_t^{(2)} + \frac{1}{2} R_t^{(4)}$$

- Called a complex backup
  - Draw each component
  - Label with the weights for that component



Forward View of TD(λ)

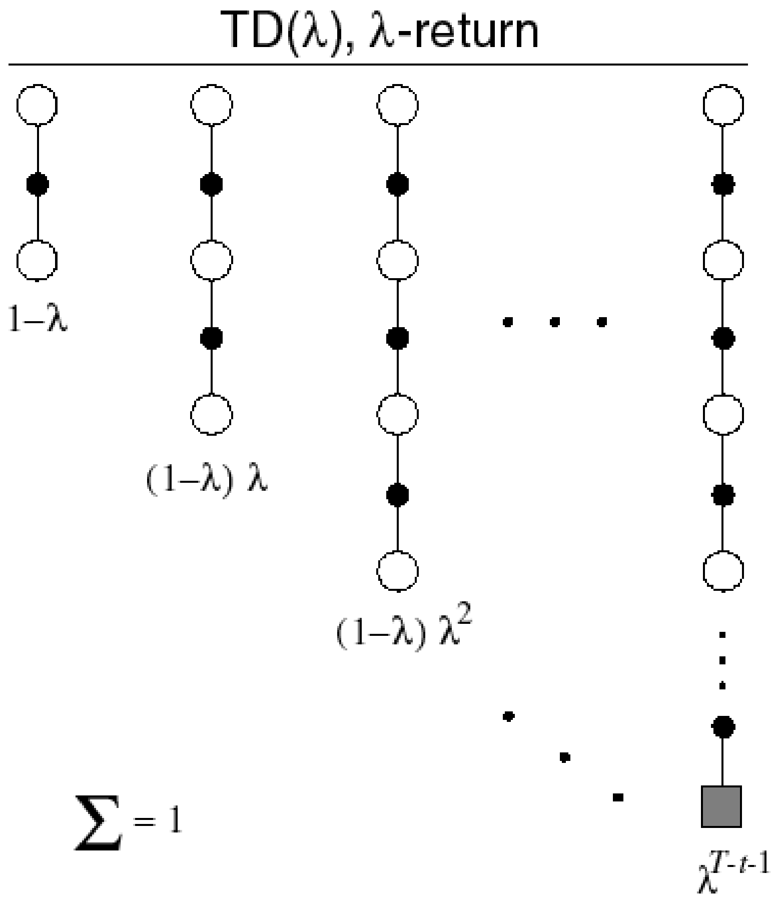
- TD(λ) is a method for averaging all n-step backups
  - weight by λ^{n-1} (time since visitation)

    λ-return:

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

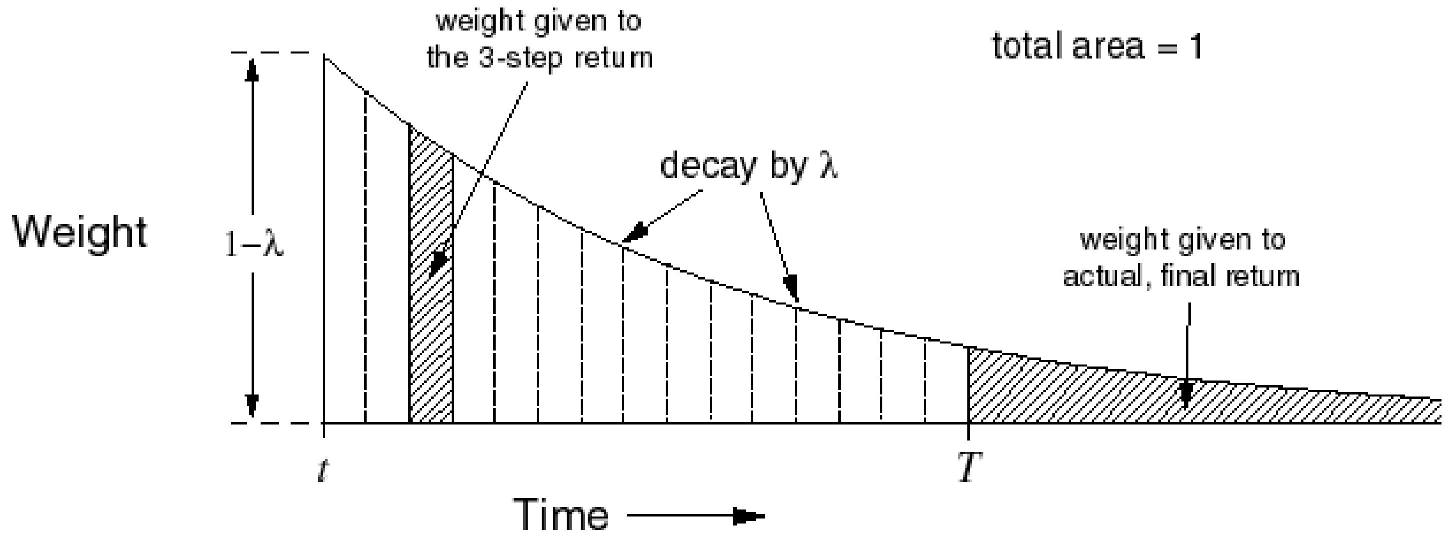
- Backup using λ-return:

$$\Delta V_t(s_t) = \alpha\left[R_t^\lambda - V_t(s_t)\right]$$



What happens when λ=1, λ= 0?

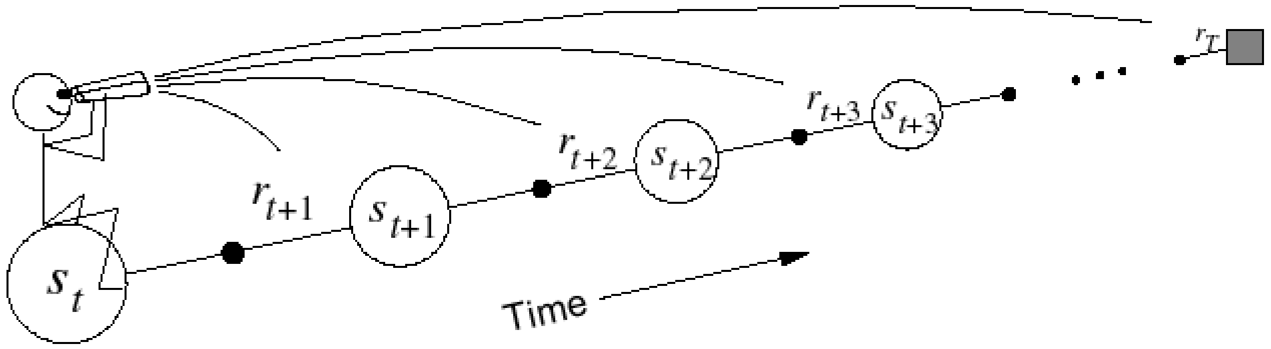
λ-return Weighting Function



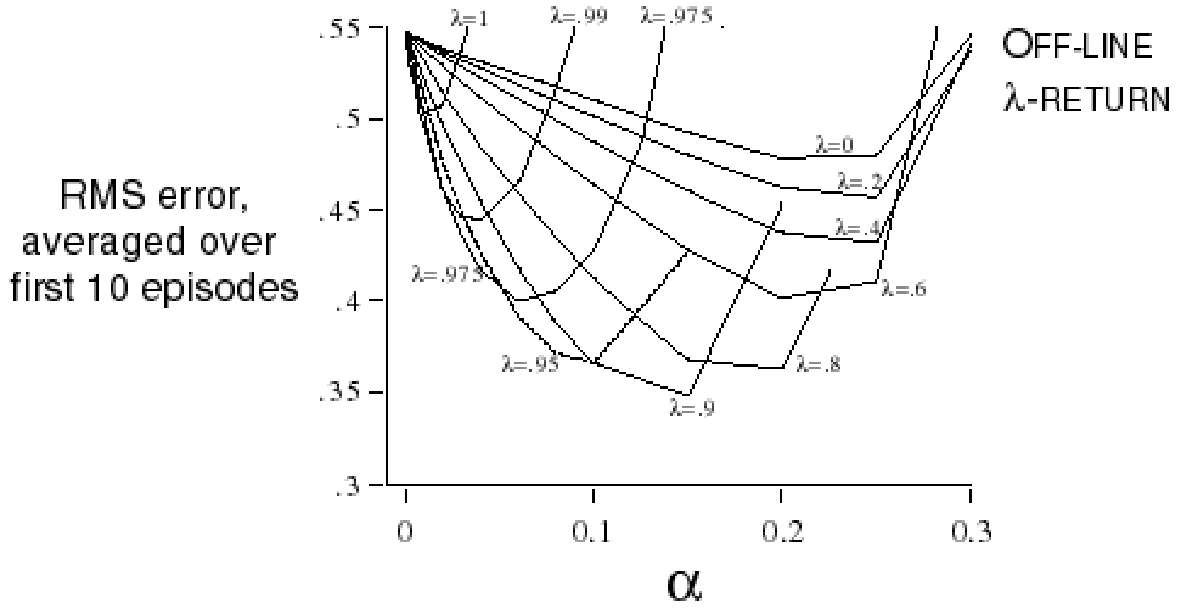
$$R_t^\lambda = \underbrace{(1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}R_t^{(n)}}_{\text{Until termination}} + \underbrace{\lambda^{T-t-1}R_t}_{\text{After termination}}$$

Forward View of TD(λ)

- Look forward from each state to determine update from future states and rewards:

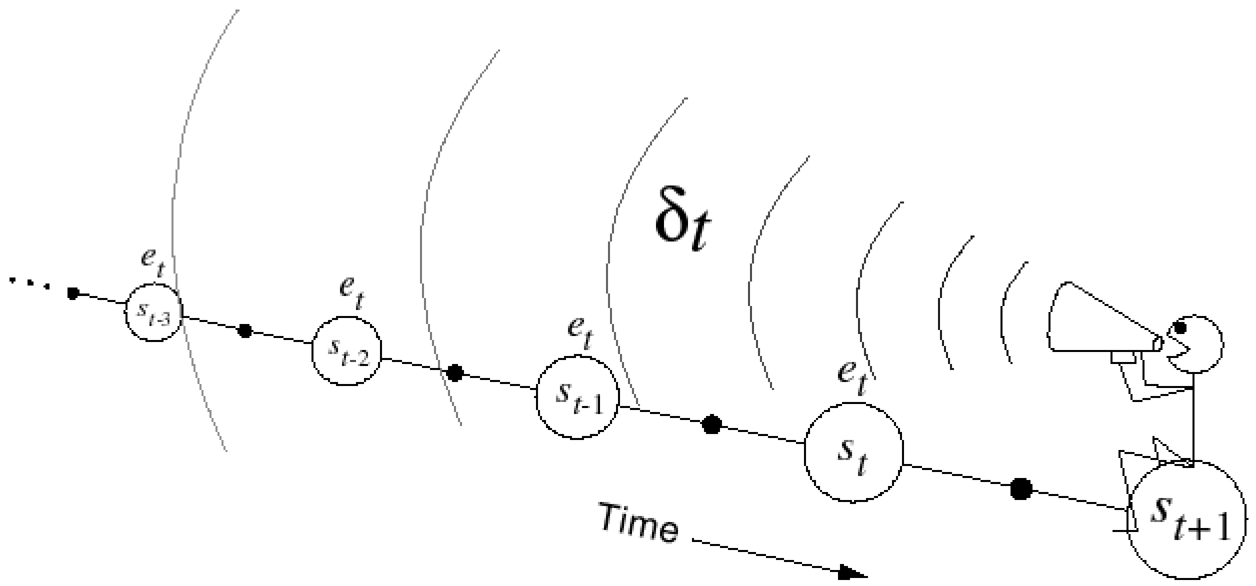


λ-return on the Random Walk



- Same 19 state random walk as before
- Why do you think intermediate values of λ are best?

Backward View



$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

- Shout δ_t backwards over time
- The strength of your voice decreases with temporal distance by $\gamma\lambda$

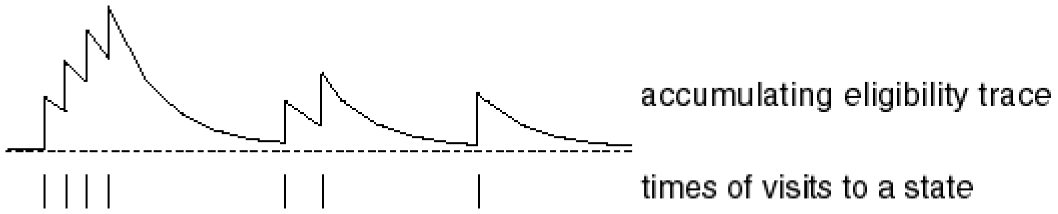
Backward View of TD(λ)

- The forward view was for theory
- The backward view is for mechanism

$$e_t(s) \in \Re^+$$

- New variable called *eligibility trace*
  - On each step, decay all traces by $\gamma\lambda$ and increment the trace for the current state by 1
  - Accumulating trace

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$



On-line Tabular TD(λ)

```
Initialize $V(s)$ arbitrarily
Repeat (for each episode):
    $e(s) = 0$, for all $s \in S$
    Initialize $s$
    Repeat (for each step of episode):
        $a \leftarrow$ action given by $\pi$ for $s$
        Take action $a$, observe reward, $r$, and next state $s'$
        $\delta \leftarrow r + \gamma V(s') - V(s)$
        $\boxed{e(s) \leftarrow e(s) + 1}$
        For all s:
            $V(s) \leftarrow V(s) + \alpha\delta \overset{\downarrow}{e}(s)$
            $\boxed{e(s) \leftarrow \gamma\lambda e(s)}$
        $s \leftarrow s'$
    Until $s$ is terminal
```

Forward View = Backward View

- The forward (theoretical) view of averaging returns in TD(λ) is equivalent to the backward (mechanistic) view for off-line updating
- The book shows:

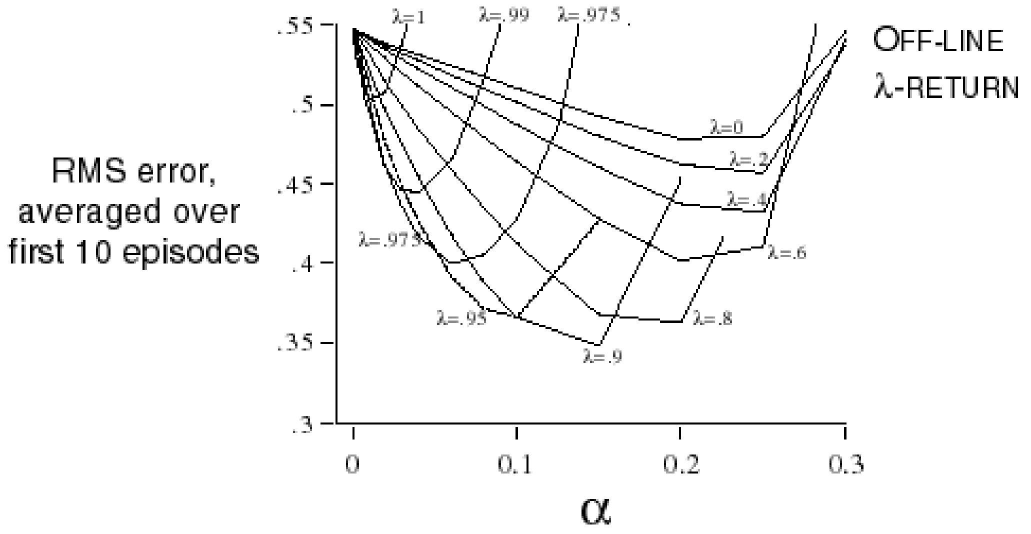
$$\underbrace{\sum_{t=0}^{T-1}\Delta V_t^{TD}(s)}_{\text{Backward updates}} = \underbrace{\sum_{t=0}^{T-1}\Delta V_t^{\lambda}(s_t) I_{ss_t}}_{\text{Forward updates}}$$

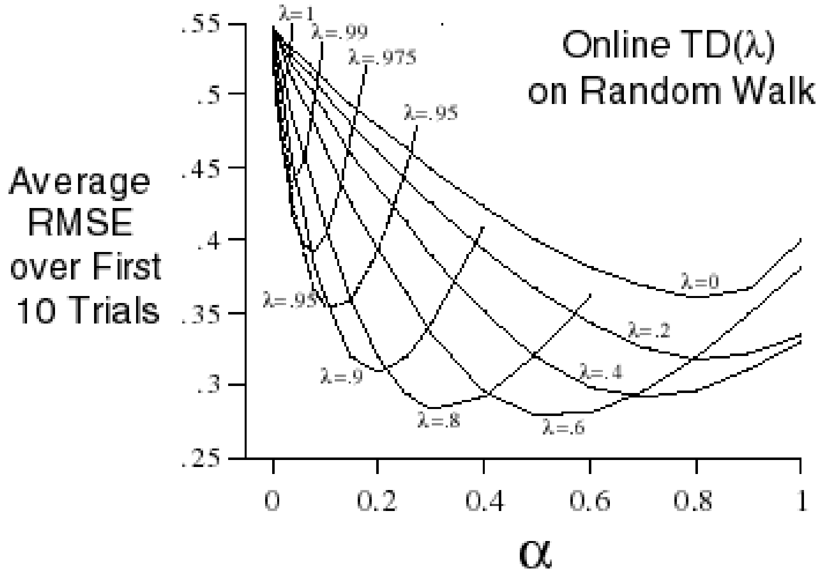
$$\Big\Downarrow \quad \text{algebra shown in book}$$

$$\sum_{t=0}^{T-1}\Delta V_t^{TD}(s) = \sum_{t=0}^{T-1}\alpha I_{ss_t}\sum_{k=t}^{T-1}(\gamma\lambda)^{k-t}\delta_k \qquad \sum_{t=0}^{T-1}\Delta V_t^{\lambda}(s_t) I_{ss_t} = \sum_{t=0}^{T-1}\alpha I_{ss_t}\sum_{k=t}^{T-1}(\gamma\lambda)^{k-t}\delta_k$$

On-line versus Off-line on Random Walk

Save all updates for end of episode.





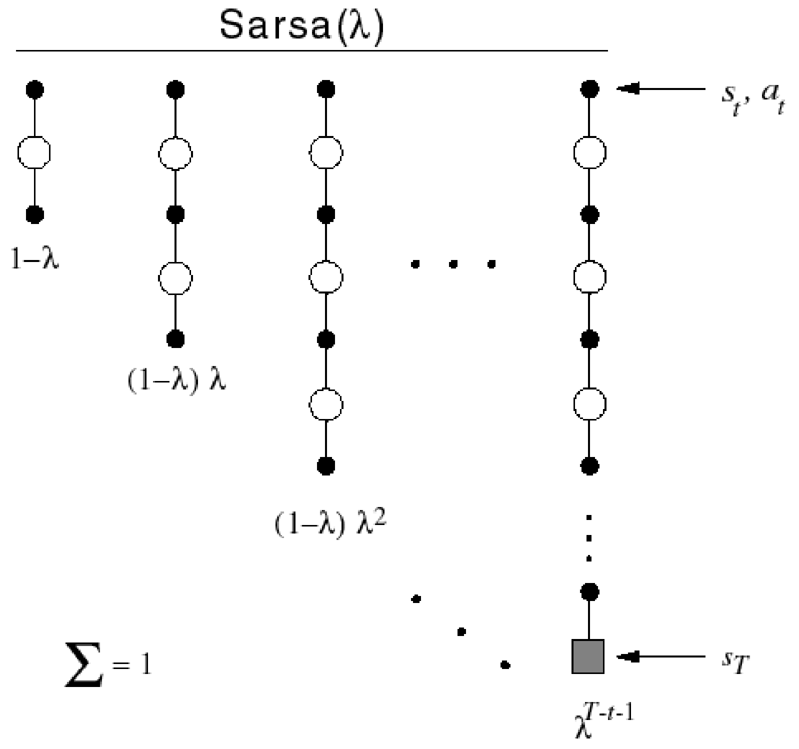
- Same 19 state random walk
- On-line better over broader range of params
  - Updates used immediately

Control: Sarsa(λ)

- Save eligibility for state-action pairs instead of just states

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases}$$

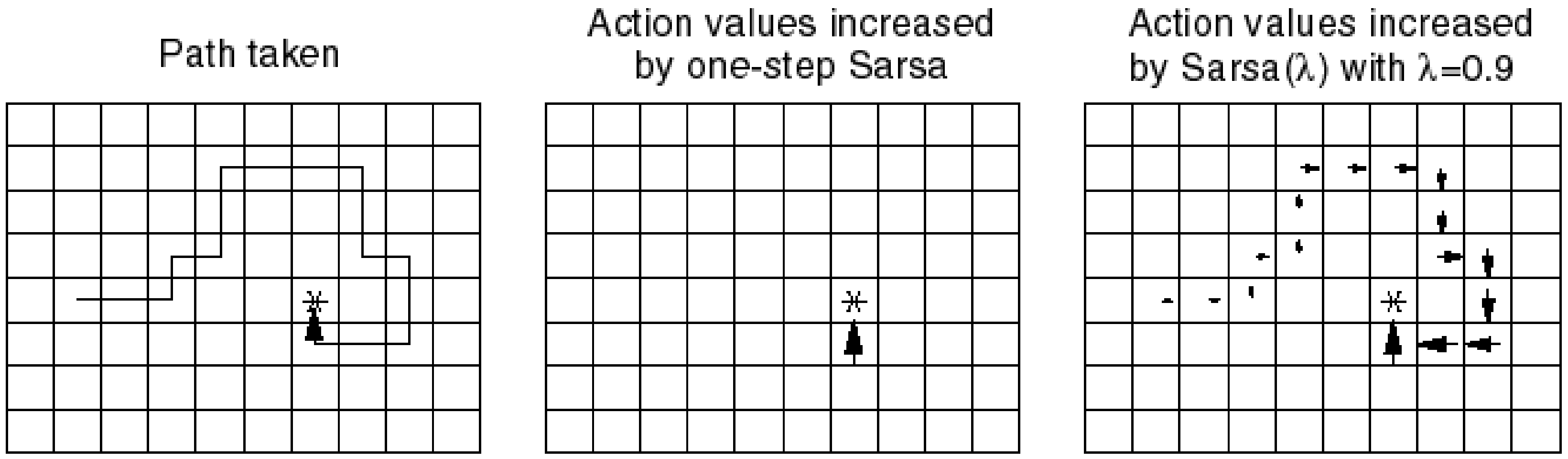
$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a)$$
$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_t)$$



Sarsa(λ) Algorithm

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode) :
    $e(s,a) = 0$, for all $s,a$
    Initialize $s,a$
    Repeat (for each step of episode) :
        Take action $a$, observe $r,s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g. $\varepsilon$ - greedy)
        $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
        $e(s,a) \leftarrow e(s,a) + 1$
        For all $s,a$ :
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
            $e(s,a) \leftarrow \gamma\lambda e(s,a)$
        $s \leftarrow s'; a \leftarrow a'$
    Until $s$ is terminal
```

Sarsa(λ) Gridworld Example


Path taken
Action values increased by one-step Sarsa
Action values increased by Sarsa(λ) with λ=0.9

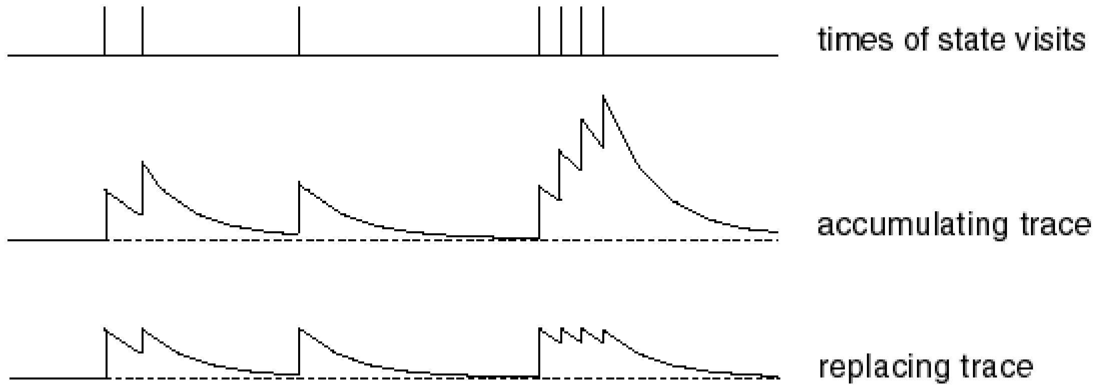
- With one trial, the agent has much more information about how to get to the goal
  - not necessarily the *best* way
- Can considerably accelerate learning

Replacing Traces

- Using accumulating traces, frequently visited states can have eligibilities greater than 1
  - This can be a problem for convergence

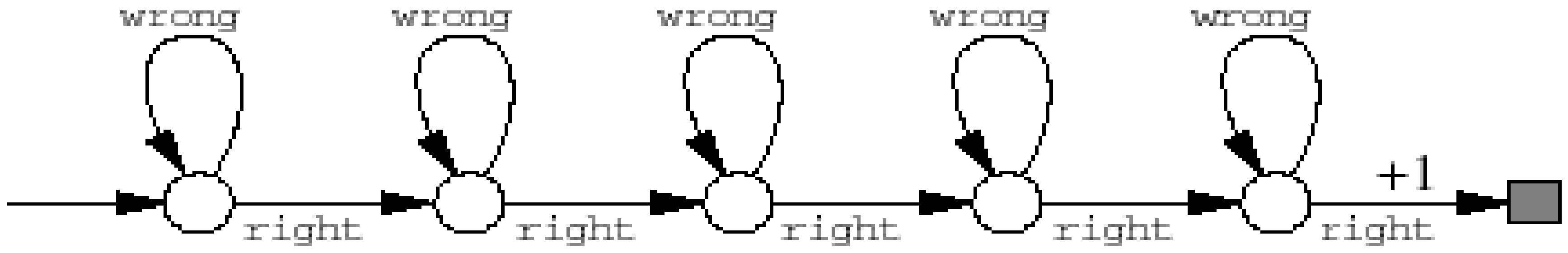
- *Replacing traces:* Instead of adding 1 when you visit a state, set that trace to 1

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases}$$



Why Replacing Traces?

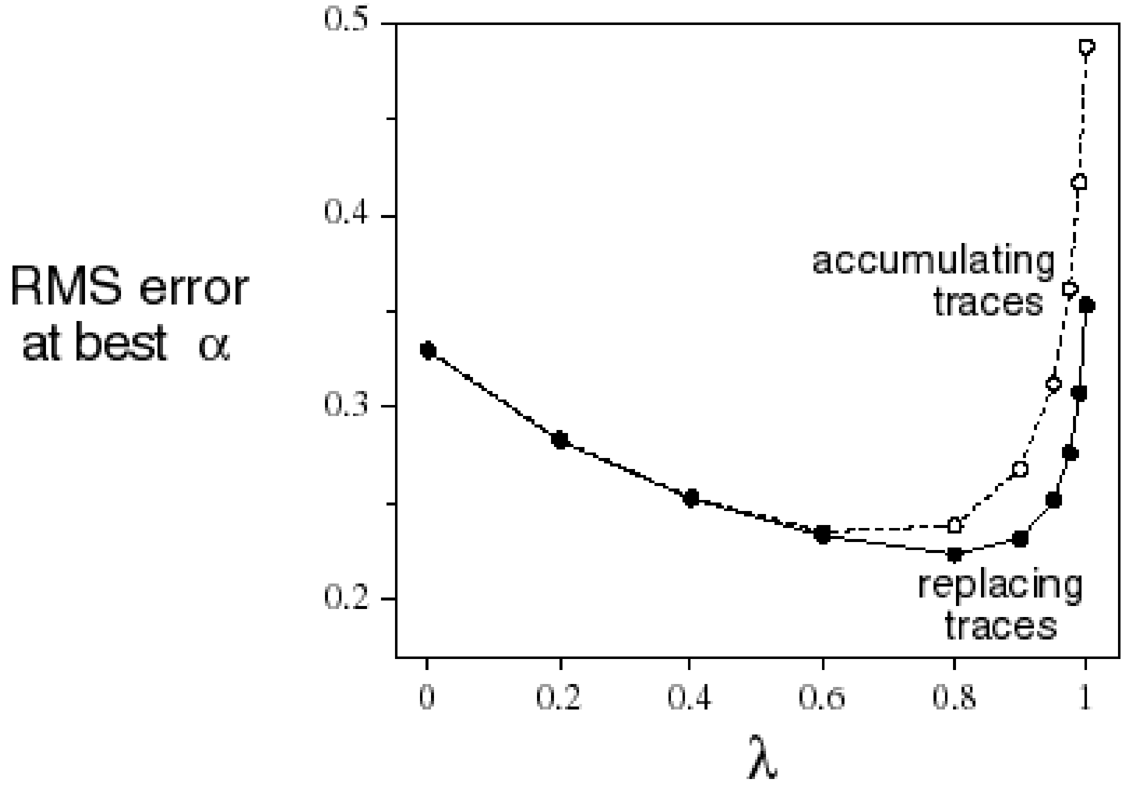
- Replacing traces can significantly speed learning
- Perform well for a broader set of parameters
- Accumulating traces poor for certain types of tasks:



Why is this task particularly
onerous for accumulating traces?

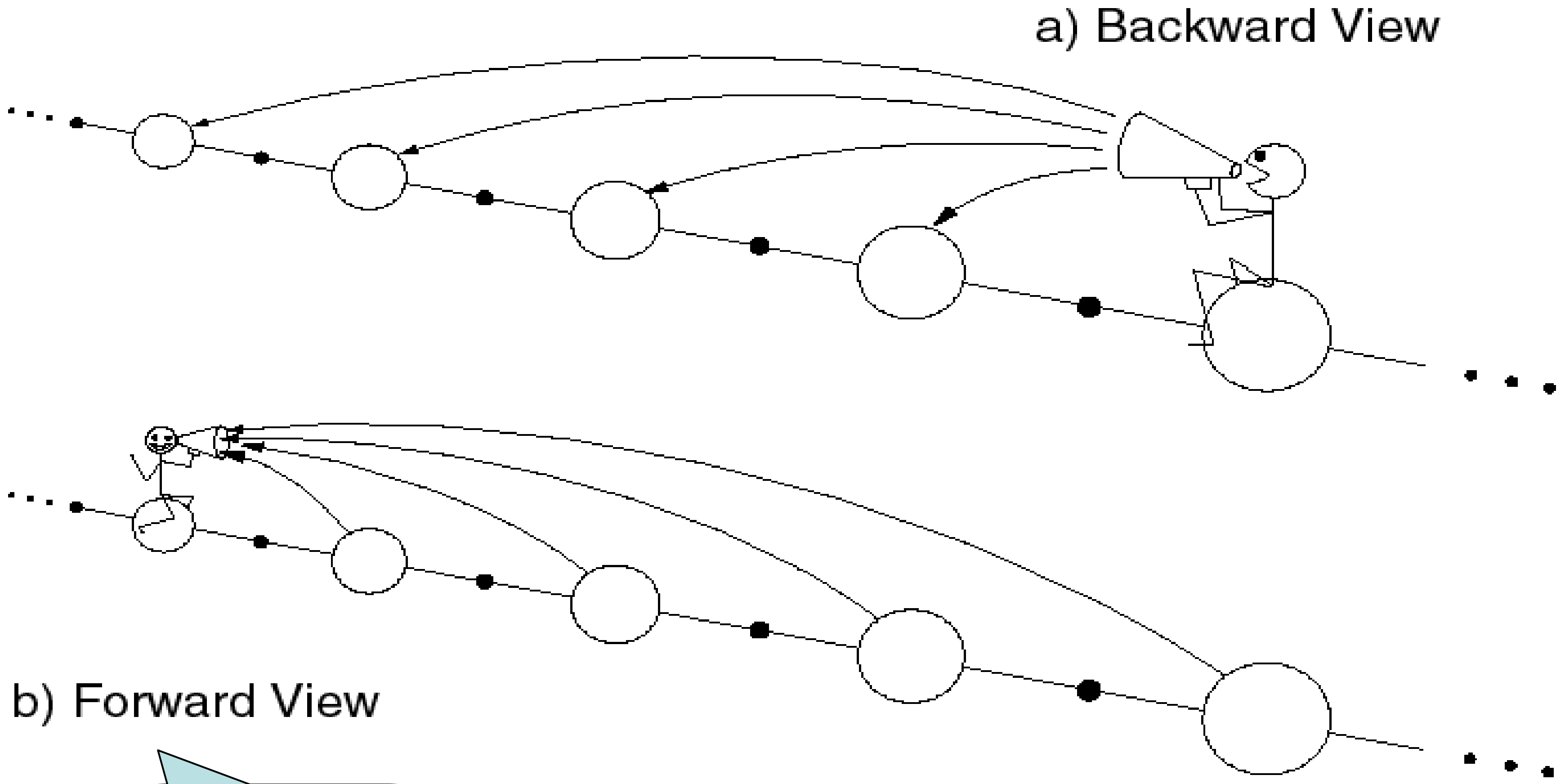
Replacing Traces Example

- Same 19 state random walk task as before
- Replacing traces perform better than accumulating traces over more values of λ



The Two Views

Efficient implementation



Averaging estimators.

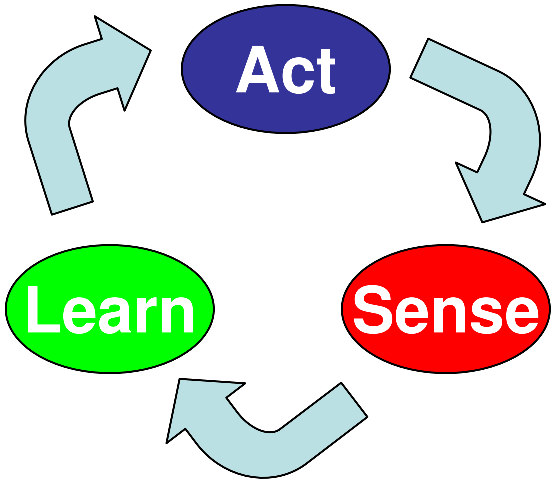
Advantage of backward view for continuing tasks?

Conclusions

- TD(λ) and eligibilities
  - efficient, incremental way to interpolate between MC and TD
- Averages multiple noisy estimators
  - Lower variance
  - Faster learning
- Can significantly speed learning
- Does have a cost in computation

Practical Issues and Discussion

Reinforcement Learning



Scott Sanner
NICTA / ANU
First.Last@nicta.com.au

Need for Exploration in RL

- For *model-based* (known) MDP solutions
  - Get convergence with *deterministic policies*
- But for *model-free* RL…
  - Need *exploration*
  - Usually use *stochastic policies* for this
    - Choose exploration action with small probability
  - Then get *convergence* to optimality

Why Explore?

- Why do we *need* exploration in RL?
  - Convergence requires all state/action values updated
    - Easy when model-based
      - Update any state as needed
    - Harder when model-free (RL)
      - Must be in a state to take a sample from it
- Current best policy may not explore all states…
  - Must occasionally *divert from exploiting* best policy
  - Exploration ensures all reachable states/actions *updated with non-zero probability*

Key property, cannot guarantee convergence to π^* otherwise!

Two Types of Exploration (of many)

- ε-greedy
  - Select random action ε of the time
    - Can decrease ε over time for convergence
    - But should we explore *all* actions with same probability?

- Gibbs / Boltzmann Softmax
    - Still selects all actions with non-zero probability
    - Draw actions from

$$P(a|Q) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_b e^{\frac{Q(s,b)}{\tau}}}$$

    - More uniform as "temperature" $\tau \to \infty$
    - More greedy as $\tau \to 0$

DP Updates vs. Sample Returns

- How to do updates?
  - Another major dimension of RL methods
    - **MC** uses full sample return

$$V_\pi(s_t) = E_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k}\middle|s_t = s\right]$$

    - **TD(0)** uses sampled DP backup (bootstrapping)

$$\begin{aligned} V_\pi(s_t) &= \sum_{s'} P(s'|s_t,\pi(s_t))\left[R(s_t,\pi(s_t),s') + \gamma V_\pi(s')\right] \\ &= E_{s'\sim P(\cdot|s_t,\pi(s_t))}\left[r_{t+1} + \gamma V_\pi(s')\middle|s_t = s\right] \end{aligned}$$

    - **TD(λ)** interpolates between TD(0) and MC=TD(1)

The Great MC vs. TD(λ) Debate

- As we will see…
  - TD(λ) methods generally learn faster than MC…
    - Because TD(λ) updates value throughout episode
    - MC has to wait until termination of episode

- But MC methods are robust to MDP violations
  - non-Markovian models:
    - MC methods can also be used to evaluate semi-MDPs
  - Partially observable:
    - MC methods can also be used to evaluate POMDP controllers
    - Technically includes value *function approximation* methods

Why partially observable? B/c FA aliases states to achieve generalization.

Proven that TD(λ) may not converge

Policy Evaluation vs. Control

- If learning V_π (policy evaluation)
  - Just use plain MC or TD(λ); always on-policy!
- For *control* case, where learning Q_π
  - Terminology for off- vs. on-policy…

Sampling Method

On- or off-policy

	MC	**TD(λ)**
On-policy	MC On-policy Control (GPI)	SARSA (GPI)
Off-policy	MC Off-policy Control	Q-learning *if* λ=0

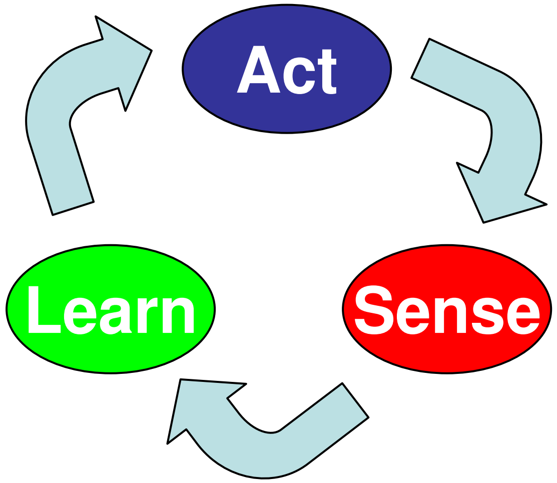
Summary: Concepts you should know

- Policy evaluation vs. control
- Exploration
  - Where needed for RL.. which of above cases?
  - Why needed for convergence?
  - ε-greedy vs. softmax
    - Advantage of latter?
- MC vs. TD(λ) methods
  - Differences in sampling approach?
  - (Dis)advantages of each?
- Control in RL
  - Have to learn Q-values, why?
  - On-policy vs. off-policy exploration methods

This is main web of ideas. From here, it's largely just implementation tricks of the trade.

RL with Function Approximation

Reinforcement Learning



Scott Sanner
NICTA / ANU
First.Last@nicta.com.au

General Function Approximation

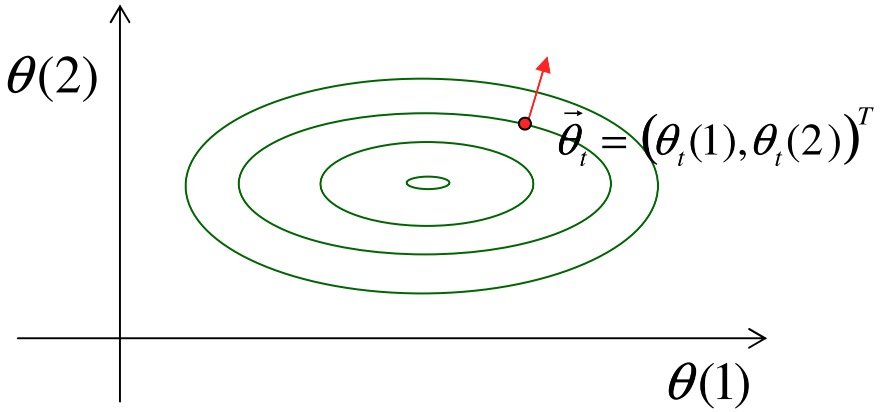
- Posit f(θ,s)
  - Can be linear, e.g., $V_{\vec{\theta}}(s) = \sum_{i=0}^{m} \theta_i f_i(s)$
  - Or non-linear, e.g., $V_{\vec{\theta}}(s) = \sum_{i=0}^{m} \sum_{j=0}^{n} \theta_i \theta_j f_{ij}(s)$
  - Cover details in a moment…
- All we need is gradient $\nabla_{\vec{\theta}} V_{\vec{\theta}}(s)$
  - In order to train weights via gradient descent

Gradient Descent

Let f be any function of the parameter space.

Its gradient at any point $\vec{\theta}_t$ in this space is:

$$\nabla_{\vec{\theta}} f(\vec{\theta}_t) = \left(\frac{\partial f(\vec{\theta}_t)}{\partial \theta(1)}, \frac{\partial f(\vec{\theta}_t)}{\partial \theta(2)}, \ldots, \frac{\partial f(\vec{\theta}_t)}{\partial \theta(n)} \right)^T .$$



Iteratively move down the gradient:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \alpha \nabla_{\vec{\theta}} f(\vec{\theta}_t)$$

Gradient Descent for RL

- Update parameters to minimize error
  - Use mean squared error $(\delta^t)^2$ where

$$\delta^t = (v^t - V_{\vec{\theta}^t}(s^t))$$

  - v^t can be MC return
  - v^t can be TD(0) 1-step sample

- At every sample, update is

Make sure you can derive this!

$$\vec{\theta}^{t+1} = \vec{\theta}^t + \alpha\delta^t \nabla_{\vec{\theta}^t} V_{\vec{\theta}^t}(s^t)$$

Gradient Descent for TD(λ)

- Eligibility now over *parameters*, not states
  - So eligibility vector has same dimension as $\vec{\theta}$
  - Eligibility is proportional to gradient

$$\vec{e}^{t+1} = \gamma\lambda\vec{e}^{t} + \nabla_{\vec{\theta}^t} V_{\vec{\theta}^t}(s^t)$$

Can you justify this?

  - TD error as usual, e.g. TD(0):

$$\delta^t = r^{t+1} + \gamma V_{\vec{\theta}^t}(s^{t+1}) - V_{\vec{\theta}^t}(s^t)$$

- Update now becomes

$$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \alpha\delta^t\vec{e}^{t}$$

Linear-value Approximation

- Most popular form of function approximation

$$V_{\vec{\theta}}(s) = \sum_{i=0}^{m} \theta_i f_i(s)$$

  – Just have to learn weights (<< # states)

- Gradient computation?

$$\frac{\partial}{\partial \theta_i} V_{\vec{\theta}}(s) = \frac{\partial}{\partial \theta_i} \sum_{i=0}^{m} \theta_i f_i(s)$$

$$= f_i(s)$$

Linear-value Approximation

- Warning, as we'll see later…
  - May be too limited if don't choose right features
- But can add in new features on the fly
  - Initialize parameter to zero
    - Does not change value function
    - Parameter will be learned with new experience
  - Can even use overlapping (or hierarchical) features
    - Function approximation will learn to trade off weights
      - Automatic bias-variance tradeoff!
    - Always use a regularizer (even when not adding features)
      - Means: add parameter penalty to error, e.g., $\|\vec{\theta}\|_2^2$

Especially important when redundant features… why?

Nice Properties of Linear FA Methods

- The gradient for MSE is simple: $\nabla_{\vec{\theta}} V_t(s) = \vec{\phi}_s$
  - For MSE, the error surface is simple:
- Linear gradient descent TD(λ) converges:
  - Step size decreases appropriately
  - On-line sampling (states sampled from the on-policy distribution)
  - Converges to parameter vector $\vec{\theta}_\infty$ with property:

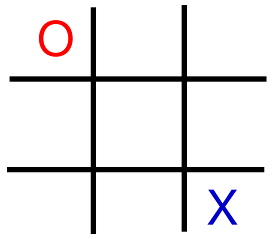
Not control!

$$MSE(\vec{\theta}_\infty) \leq \frac{1-\gamma\lambda}{1-\gamma} MSE(\vec{\theta}^*)$$

best parameter vector

(Tsitsiklis & Van Roy, 1997)

TicTacToe and Function Approximation



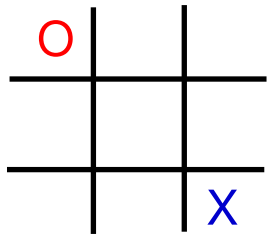
- Tic-Tac-Toe
  - Approximate value function with

$$V_{\vec{\theta}}(s) = \sum_{i=0}^{m} \theta_i f_i(s)$$

  - Let each f_i be “an O in 1”, “an X in 9”
  - Will never learn optimal value
    - Need conjunctions of multiple positions
    - Linear function can do disjunction at best

TicTacToe and Function Approximation

- How to improve performance?

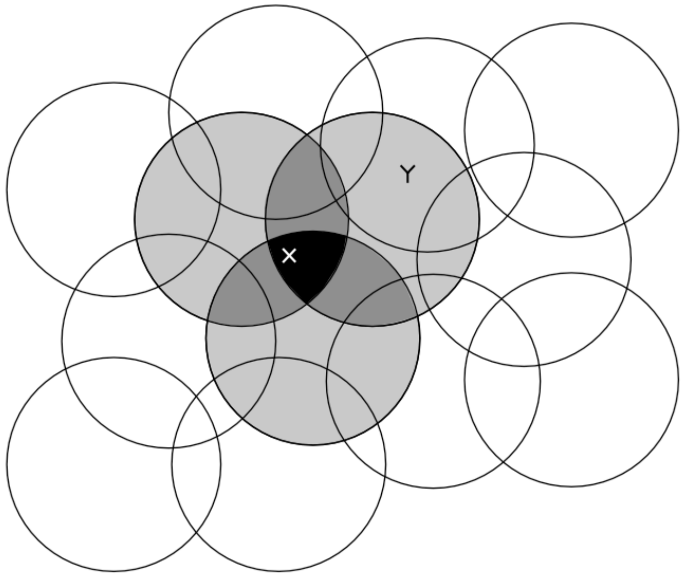


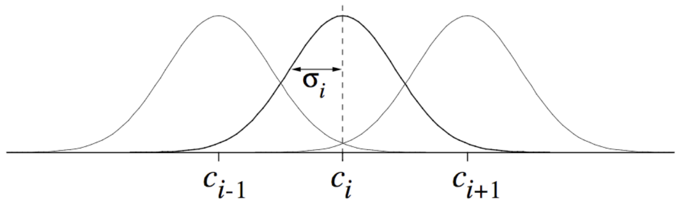
  - Better features
    - adapt or generate them as you go along
    - E.g., conjunctions of other features

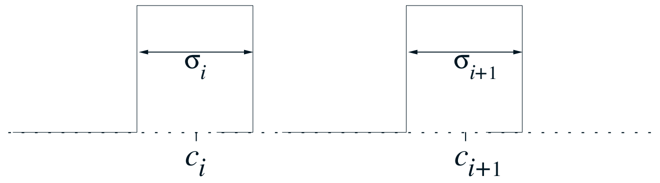
  - Use a more powerful function approximator
    - E.g., non-linear such as neural network
    - Latent variables learned at hidden nodes are boolean function expressive
      - » encodes new complex features of input space

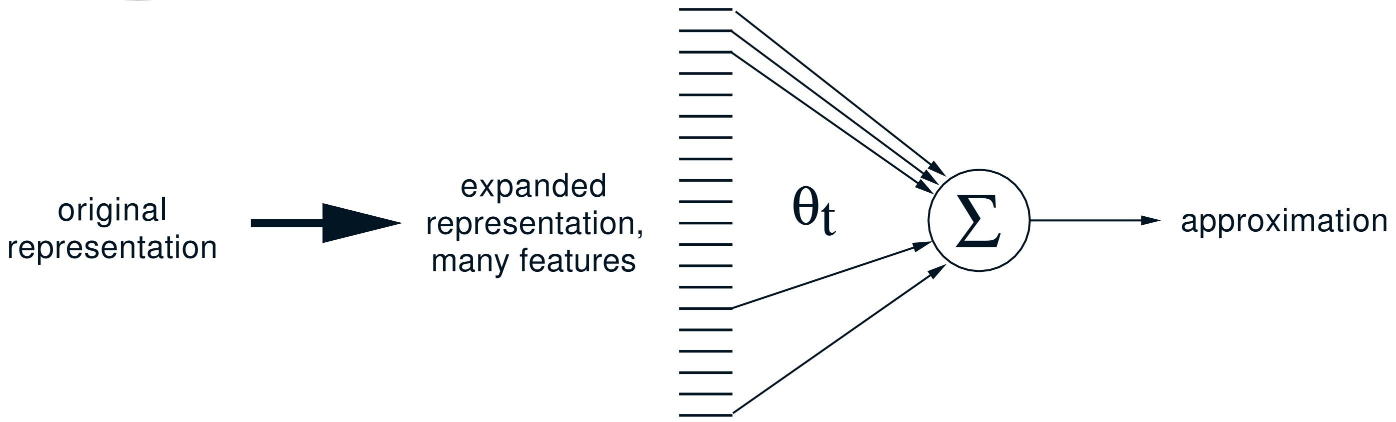
Some Common Feature Classes…

Coarse Coding

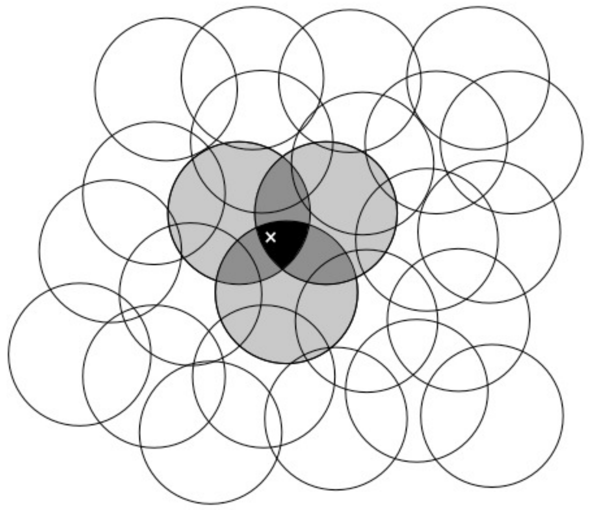


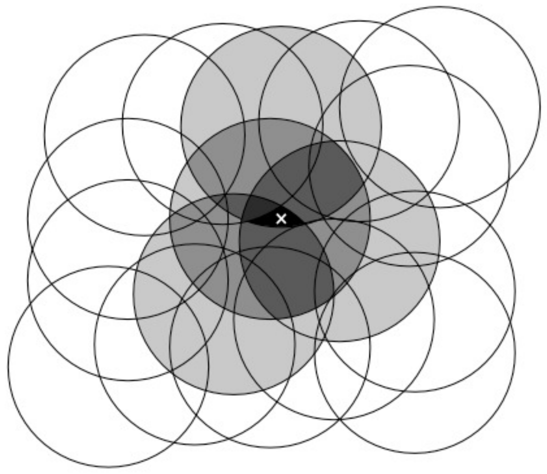


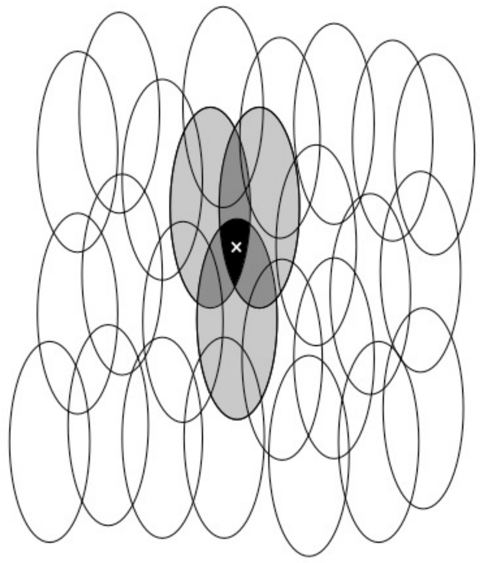




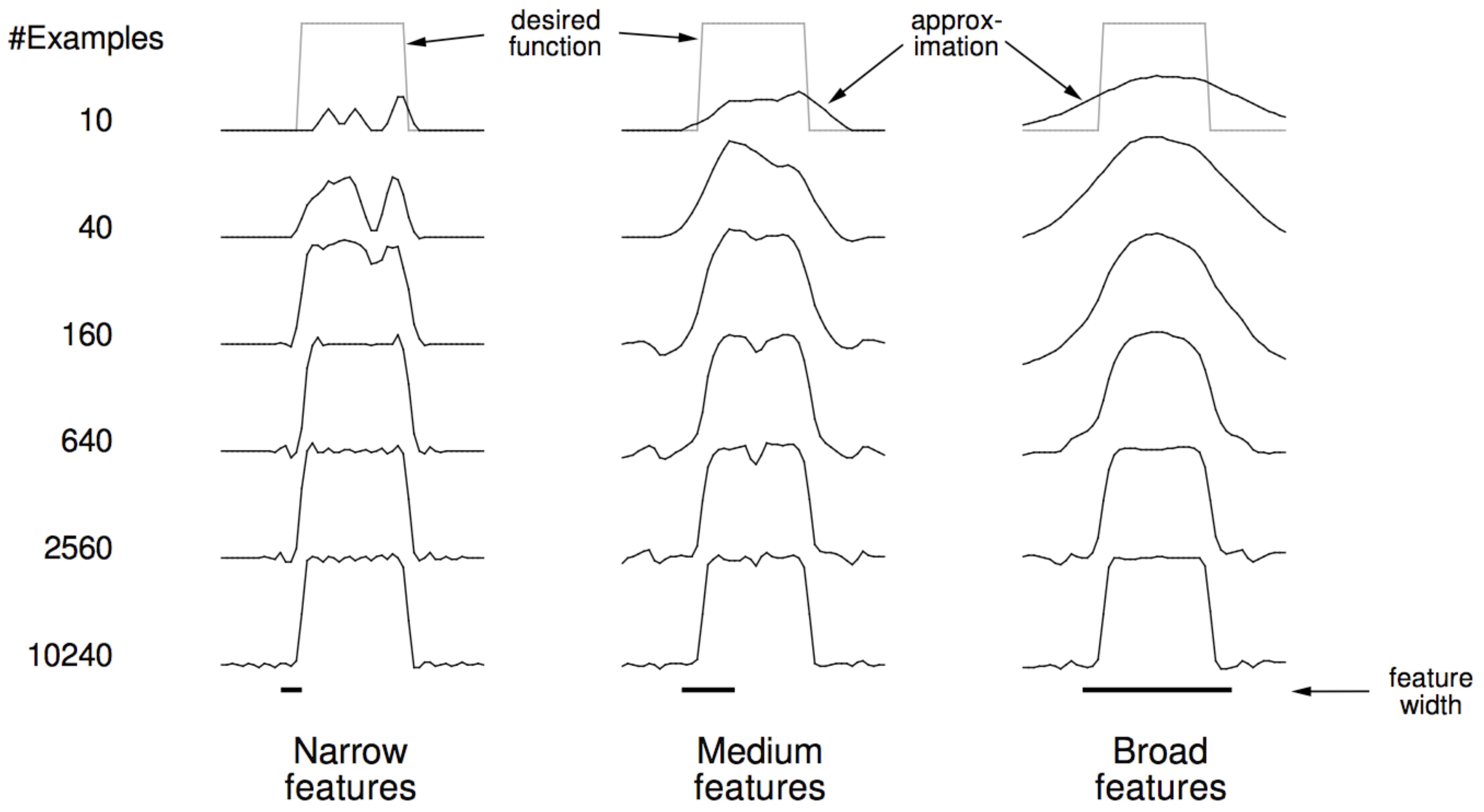
Shaping Generalization in Coarse Coding


a) Narrow generalization


b) Broad generalization

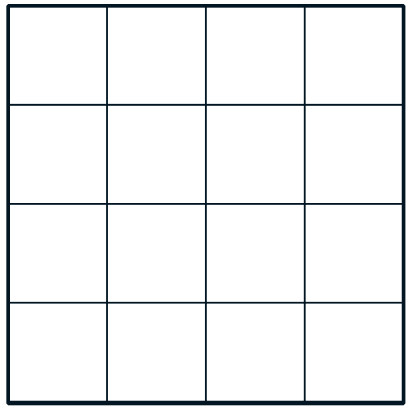

c) Asymmetric generalization

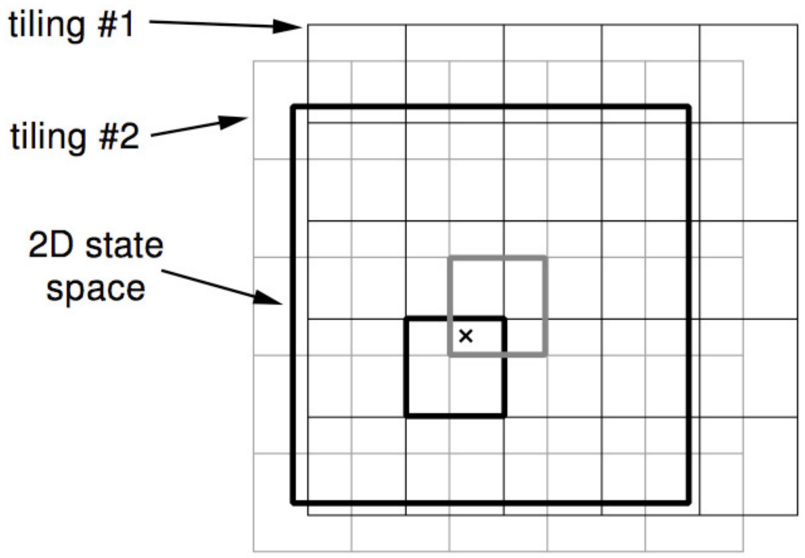
Learning and Coarse Coding



Tile Coding

But if state continuous… use continuous FA, not discrete tiling!





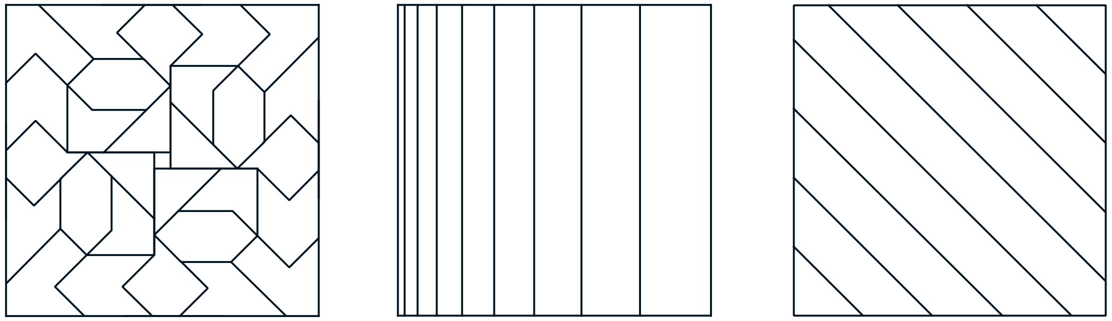
- Binary feature for each tile
- Number of features present at any one time is constant
- Binary features means weighted sum easy to compute
- Easy to compute indices of the features present

Shape of tiles ⇒ Generalization

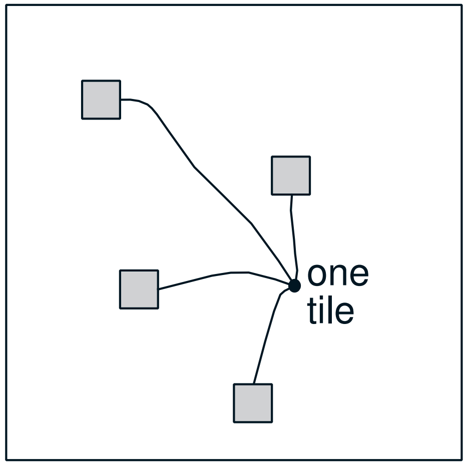
#Tilings ⇒ Resolution of final approximation

Tile Coding Cont.

Irregular tilings



Hashing

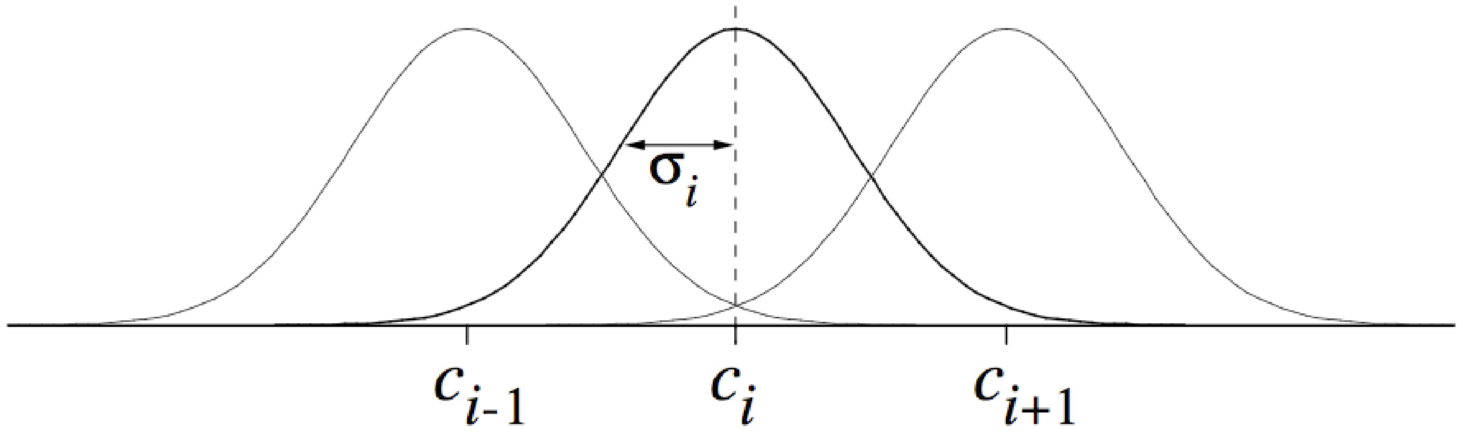


CMAC
"Cerebellar model
arithmetic computer"
Albus 1971

Radial Basis Functions (RBFs)

e.g., Gaussians

$$\phi_s(i) = \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right)$$



Nonlinear Value-approximation

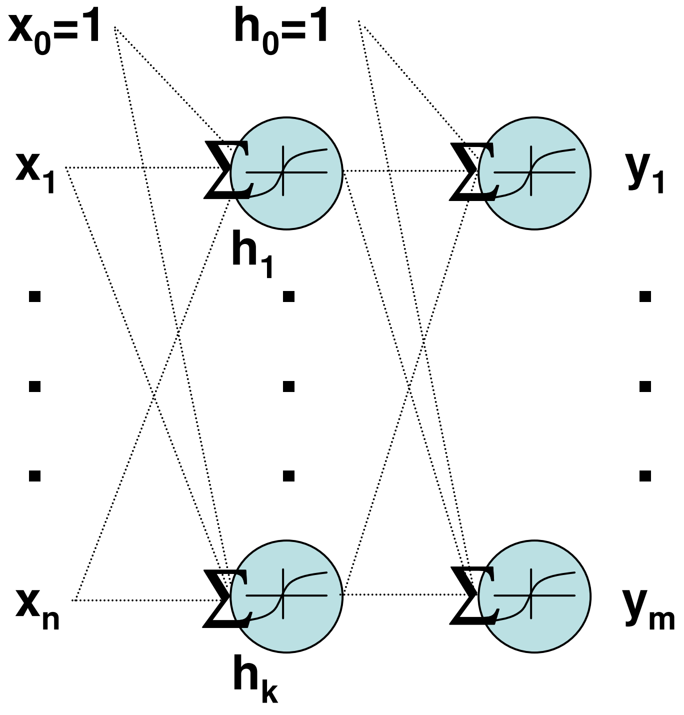
- Can use an artificial neural network (ANN)…
  - Not convex, but good methods for training
- *Learns* latent features at hidden nodes
  - Good if you don’t know what features to specify
- Easy to implement
  - Just need derivative of parameters
  - Can derive via backpropagation
    - Fancy name for the chain rule

TD-Gammon
= TD(λ) + Function
Approx. with ANNs

ANNs in a Nutshell

- **Neural Net:** non-linear weighted combination of *shared* sub-functions

- **Backpropagation:** to minimize SSE, train weights using *gradient descent* and *chain rule*



all edges ········ have weight $\mathbf{w}_{j,i}$

Where the *Real* Problem Lies

- Function approximation often a necessity, which to use?
  - MC
  - TD(λ)
  - TD *xyz* with adaptive lambda, etc…

Note: TD(λ) may diverge for control! MC robust for FA, PO, Semi-MDPs.

- Algorithm choice can help (speed up) convergence
  - But *if* features are inadequate
  - *Or* function approximation method is too restricted
    - May never come close to optimal value (e.g., TicTacToe)

- **Concerns for RL w/ FA:**
  - **Primary: Good features, approximation architecture**
  - Secondary: then (but also important) is convergence rate

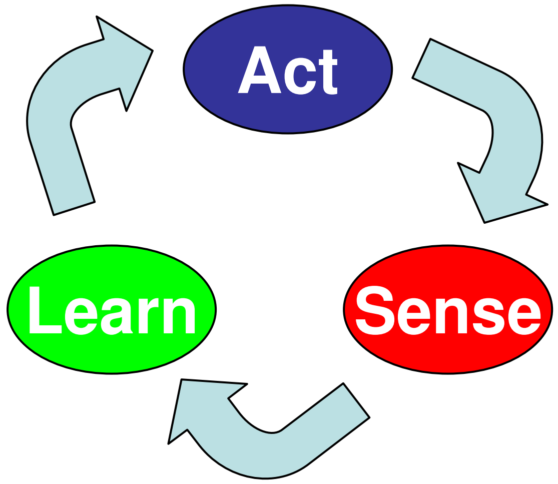
Recap: FA in RL

- FA a necessity for most real-world problems
  - Too large to solve exactly
    - Space (curse of dimensionality)
    - Time
  - Utilize power of generalization!

- Introduces additional issue
  - Not just speed of convergence
    - As for exact methods
  - But *also* features and approximation architecture
    - Again: for real-world applications, this is arguably most important issue to be resolved!

Conclusion

Reinforcement Learning



Scott Sanner
NICTA / ANU
First.Last@nicta.com.au

Lecture Goals

1) To understand formal models for decision-making under uncertainty and their properties

   - *Unknown models* (reinforcement learning)

   - *Known models* (planning under uncertainty)

2) To understand efficient solution algorithms for these models

Summary

- Lecture contents
  - Modeling sequential decision making
  - Model-based solutions
    - Value iteration, Policy iteration
  - Model-free solutions
    - Exploration, Control
    - Monte Carlo, TD(λ)
    - Function Approximation

- This is just the tip of the iceberg
  - But a large chunk of the tip