CS395T: Foundations of Machine Learning for Systems Researchers

Fall 2025

Lecture 7:

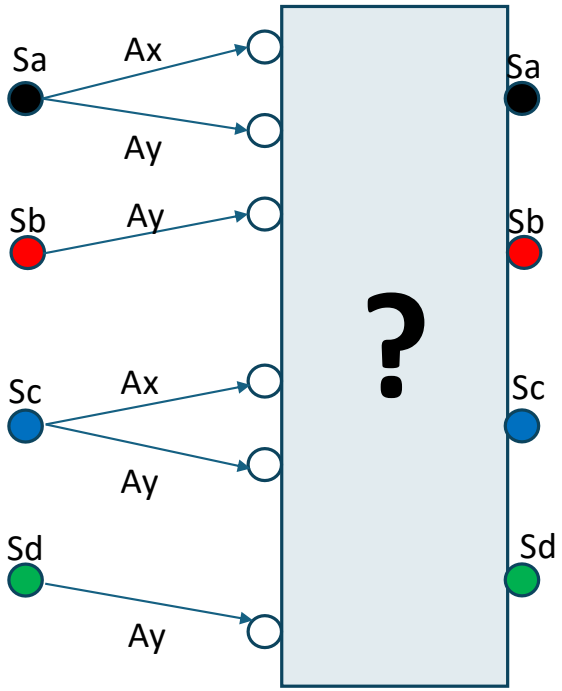
Model-free (Sampling) Methods for MDPs

Problem

In practice, we usually have incomplete knowledge of environment

- Rewards?
- Probabilities?

How do we do policy optimization when we have incomplete knowledge of environment?



Model-free methods

Based on sampling: determine optimal policy from estimates of rewards and probabilities

Issues

- What are samples?
- How do we learn from a sample?
- When do we learn from samples?
- How to update policy when you get new samples?

Goal is to determine optimal policy and not to learn full MDP, which is usually impossible

Organization

Background:

- Incremental & exponential recency-weighted averaging of sequences

Offline method

- Collect samples and post-process them to find V/Q estimates
- Like batching in NN training

Online methods

- Boot-strapping methods: start with initial approximations for V/Q values and update when you get sample
  - Temporal-difference (TD) methods: TD(n), TD(λ), SARSA, Q-learning
- Non-bootstrapping method: no initial approximation to V/Q values needed
  - Monte Carlo (RL terminology)

Background

Incremental averaging of sequence of values

- Problem: given
  - Sequence of (possibly noisy) measurements of some quantity: $V_1, V_2, V_3, \ldots$
  - Maintain a running average of measurements: $\hat{V}_1, \hat{V}_2, \hat{V}_3, \ldots$

$$
\begin{aligned}
\hat{V}_1 &= V_1 \\
\hat{V}_i &= \frac{V_1 + \ldots + V_i}{i} \quad (i \geq 1) \\
&= \frac{V_1 + \ldots + V_{i-1}}{i-1} * \frac{i-1}{i} + \frac{V_i}{i} \\
&= (1 - \frac{1}{i})\hat{V}_{i-1} + \frac{1}{i} * V_i
\end{aligned}
$$

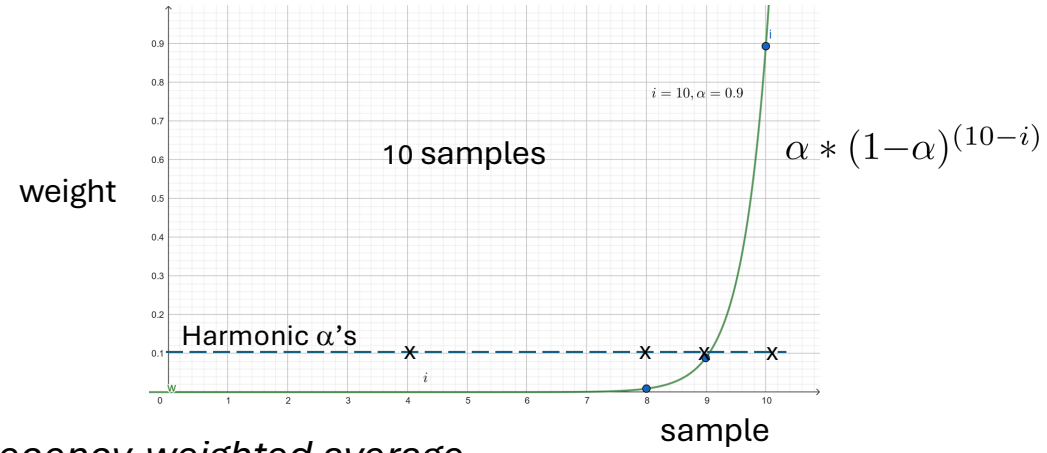
Target

$$\hat{V}_i = \hat{V}_{i-1} + \alpha_i(V_i - \hat{V}_{i-1}) \quad \text{where } \alpha_i = \frac{1}{i}$$

Error

Harmonic α's

Non-stationary problems



- *Exponential recency-weighted average*
  - For non-stationary problems, better to give more weight to more recent measurements
  - Maintain a **weighted** average of measurements

$$\hat{V}_1 = V_1$$

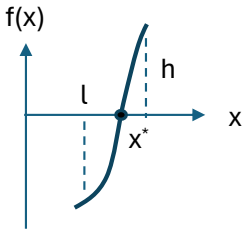
$$\boxed{\hat{V}_i = \hat{V}_{i-1} + \alpha(V_i - \hat{V}_{i-1}) \quad \text{where } 0 \leq \alpha \leq 1 \text{ is constant}}$$

$$\hat{V}_i = \alpha V_i + \alpha(1-\alpha)V_{i-1} + \alpha(1-\alpha)^2 V_{i-2} + \ldots + \alpha(1-\alpha)^{i-2} V_2 + (1-\alpha)^{i-1} V_1$$

$\alpha \to 1$ gives more weight to recent measurements
$\alpha \to 0$ gives more weight to older measurements

Stochastic Approximation Theory

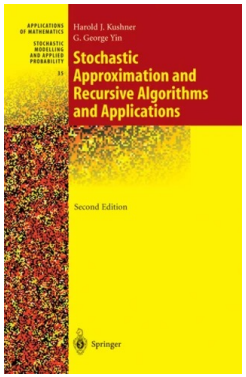
- Given: continuous function f(x), non-decreasing in interval (l,h), and f(l) < 0 and f(h) > 0
- Intermediate value theorem: $\exists x^*$: $l < x^* < h$ such that $f(x^*) = 0$
- Iterative method for finding x^*
  - x_1 = any point in interval (l,h)
  - if $f(x_1) < 0$ try point to right of x_1 otherwise try point to left of x_1
    - Iterative scheme: $x_i = x_{i-1} - \alpha * f(x_{i-1})$ where α is some constant > 0



- Often f cannot be measured exactly, and we can only measure g(x) = f(x) + d where d is a zero-mean noise term
- One solution: measure $g(x_i)$ many times and take average
  - Problem: you may spend a lot of effort in estimating $f(x_i)$ even if x_i is far from x^*
- Robbins-Monro[1951]: use iterative scheme with adaptive $\alpha_i > 0$

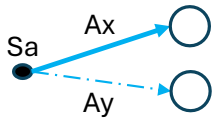
$$x_i = x_{i-1} - \alpha_i * g(x_{i-1}) \text{ where } \sum_{i=1}^{\infty} \alpha_i = \infty \text{ and } \sum_{i=1}^{\infty} \alpha_i^2 < \infty$$
$$x_i \text{ converges to } x^*$$

- Stochastic gradient-descent: f is the derivative of loss function $\mathcal{L}$



Sampling Methods

High-level idea



$\pi(Sa) = Ax$

Focus on *policy evaluation* for policy π, H = ∞: need to solve linear system

$$V^{\pi}(s) = \Sigma_{s'} P(s, \pi(s), s') * (R(s, \pi(s), s') + \gamma * V^{\pi}(s'))$$

Since π is fixed, simplify notation by dropping π from equations

$$V(s) = \Sigma_{s'} P(s, s') * (R(s, s') + \gamma * V(s'))$$

We do not know *P* and *R*, but assume

$\hat{P}$: estimate for *P* and row-stochastic matrix

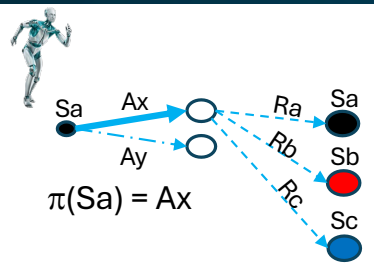
$\hat{R}$: estimate for *R*

Compute $\hat{V}$, an estimate for state valuations, by solving linear system

$$\hat{V}(s) = \Sigma_{s'} \hat{P}(s, s') * (\hat{R}(s, s') + \gamma * \hat{V}(s'))$$

If $\hat{P} \approx$ P and $\hat{R} \approx$ R, then $\hat{V} \approx$ V

First attempt: offline method



Collect a multiset of samples

- At state Sa, agent takes action π(Sa) and sees what happens
  - Observation SARS = <Start**S**tate, **A**ction, Observed **R**eward, Observed **S**tate>
- Repeat to obtain multiset of observations starting at various states
  - O = [$SARS_1$, $SARS_2$, $SARS_3$,.....,$SARS_n$]

Estimate V by offline processing of observations

- n(Sx) = number of times agent started in state Sx in O (assume > 0)
- n(Sx,Sy) = number of times agent ended up in state Sy when it started in state Sx
- $\hat{P}(Sx,Sy) = \frac{n(Sx,Sy)}{n(Sx)}$ ($\hat{P}$ is a row-stochastic matrix)
- $\hat{R}$(Sx,Sy) = observed reward in any Sx $\rightarrow$Sy sample
- Solve linear system to find $\hat{V}$

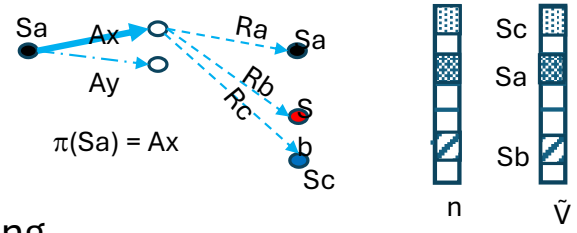
$$\hat{V}(s) = \frac{1}{n(s)} \Sigma_{s'} n(s,s') * (\hat{R}(s,s') + \gamma * \hat{V}(s'))$$

Drawbacks of offline method

1. We do not learn valuations until all samples have been collected.
   - **Online methods**: pipeline sample collection and valuation estimation
   - *Exploitation: use valuations to guide where to sample*
2. Inefficient to jump around between starting states while collecting samples
   - Solution: agent should **follow paths** in the MDP graph while collecting samples rather than jumping around
   - **Episodes**

For now, collect samples, store on disk and process them in streaming fashion iteratively until convergence.
Like “batching” in NN training

TD(0): simplest online method



- Recall: offline processing

$$\hat{V}(s) = \frac{1}{n(s)} \sum_{s'} n(s,s') * (\hat{R}(s,s') + \gamma * \hat{V}(s'))$$

- Online updates: maintain two arrays while processing samples
  - n(s) = number of times state s has been visited (initialized to 0)
  - Ṽ(s)= current estimate of V (initialized arbitrarily)|
- New sample <Sa, Ax, R(Sa,S'), S'> comes in

$$\tilde{V}(Sa) \leftarrow \frac{n(Sa)*\tilde{V}(Sa) + (R(Sa,S') + \gamma * \tilde{V}(S'))}{n(Sa)+1} \text{ which is equivalent to}$$

$$\tilde{V}(Sa) \leftarrow \tilde{V}(Sa) + \frac{1}{\alpha}(R(Sa,S') + \gamma * \tilde{V}(S') - \tilde{V}(Sa)) \text{ where } \alpha = n(Sa)+1$$

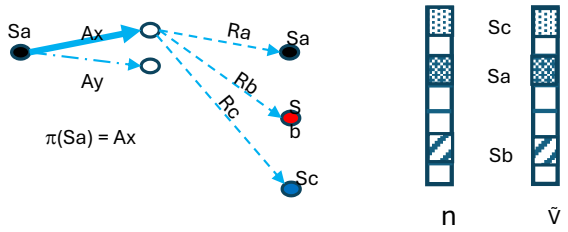
$$n(Sa) \leftarrow n(Sa)+1$$

TD-target

TD-error

*Intuition: average over **multiset** of samples = **weighted** average over sample **set**, weighted by frequency*

TD(0) program




```
Array n:1..|S| = 0; //number of visits to states
Array Ṽ:1..|S| = arbitrary; //V estimates

while (not converged) do
 for each sample <Sa,π(Sa),R(Sa,S'),S'> do {
    increment n(Sa);
    Ṽ(Sa) ← Ṽ(Sa) + 1/n(Sa) (R(Sa,S')+γ*Ṽ(S') - Ṽ(Sa));
  }
```

Tabular method: maintain table to map states to values

Comments

Convergence (Balachander, Chatterjee *et al.*): Given a sequence of one-hop samples O, the $\tilde{V}$ values computed by the TD(0) program will converge to the $\hat{V}$ values in the fixpoint equation

Caveat: need to iterate over sample sequence O many times like in batching

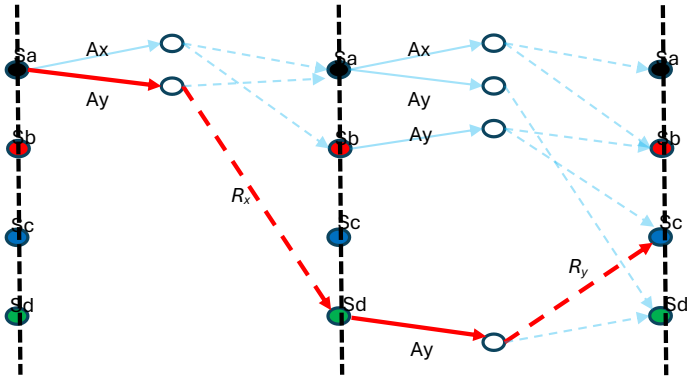
Convergence: $\tilde{V}$ values computed by TD(0) program will converge to V^{π} if each state is visited unboundedly often in O (called *exploring starts*).

In practice: throw away sample after it is processed and regenerate later

Boot-strapping method: $\tilde{V}$ is initialized arbitrarily

Biased estimator until convergence

TD(1)

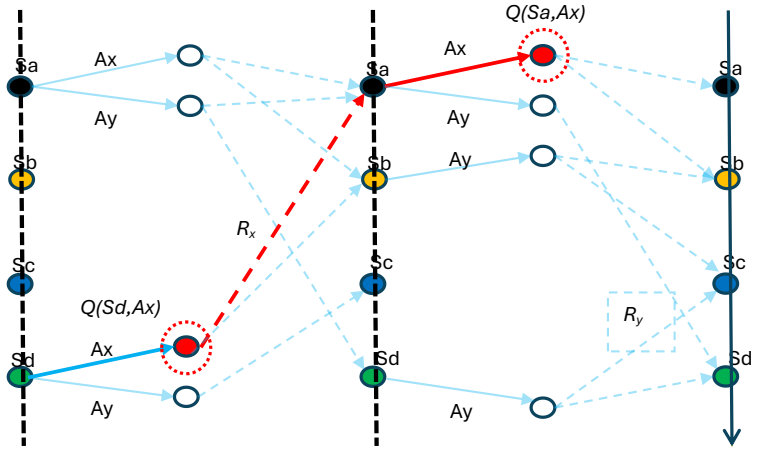


- TD(1) two-hop sample: <S_0,A_0,R_1,S_1,A_1,R_2,S_2> using policy π
- Update valuations by computing total discounted reward

$$V(Sa) \leftarrow V(Sa) + \alpha[Rx + \gamma * Ry + \gamma^2 V(Sc) - V(Sa)] \quad (\text{where } \alpha{=}n(Sa))$$

- Generalize to TD(n): paths with (n-1) hops
- Convergence arguments are similar to TD(0) case

SARSA



- Maintain table of Q-values rather than V values, using policy π

$$Q(Sd, Ax) \leftarrow Q(Sd, Ax) + \alpha * [Rx + \gamma * Q(Sa, \pi(Sa)) - Q(Sd, Ax)]$$

- Generalize to multiple hops

On-policy and off-policy methods

Terminology

- Target policy: policy you are optimizing
- Behavior policy: policy that generates actions

On-policy methods

- Target policy = Behavior policy
- “Learn about the policy you are following” (alternate policy evaluation and improvement)
- Can be less efficient than off-policy methods
- Example: TD(n), SARSA

Off-policy methods

- Behavior policy ≠ Target policy
- “Learn about one policy while behaving according to different policy” (fuse policy evaluation & improvement)
- May find optimal policy faster than on-policy methods
- Examples: Q-learning

Good informal explanation

Exploratory policies

Pure exploration: pick an action at random. Does not exploit Q-value estimates.

ε-greedy: use exploration ε (~5-10%) fraction of time, and exploitation rest of time. For exploration, choose action at random. Most popular method.

Boltzmann exploration: like ε-greedy but uses softmax over current Q-value estimates to select action for exploration

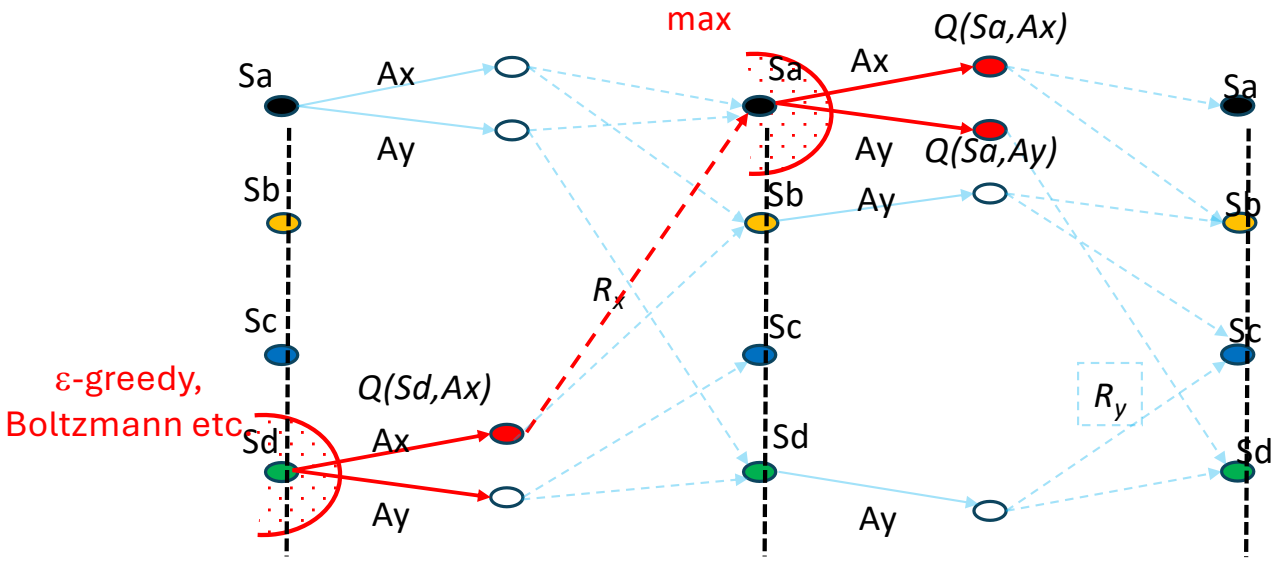
Pure exploitation: pick action with highest Q-value. May get stuck in local minimum.

Policy network: DNN maps state to distribution over actions, and sampler picks action

Exploration-exploitation tradeoff

Good explanation of off-policy methods

Q-learning



- Off-policy method: needs exploratory policy
- Compute Q-values like SARSA but optimize over actions like value iteration

$$Q(Sd, Ax) \quad \leftarrow \quad Q(Sd, Ax) + \alpha * [Rx + \gamma * max_{As} Q(Sa, As) - Q(Sd, Ax)]$$

- Generalize to multiple hops
- Convergence of Q-learning

Monte Carlo Method

Expectations over Trajectories

Unrolled MDP DAG for fixed π, rooted at start state S0, H = T

Each edge is labeled with probability and reward
Sum of probabilities on outgoing edges of node = 1

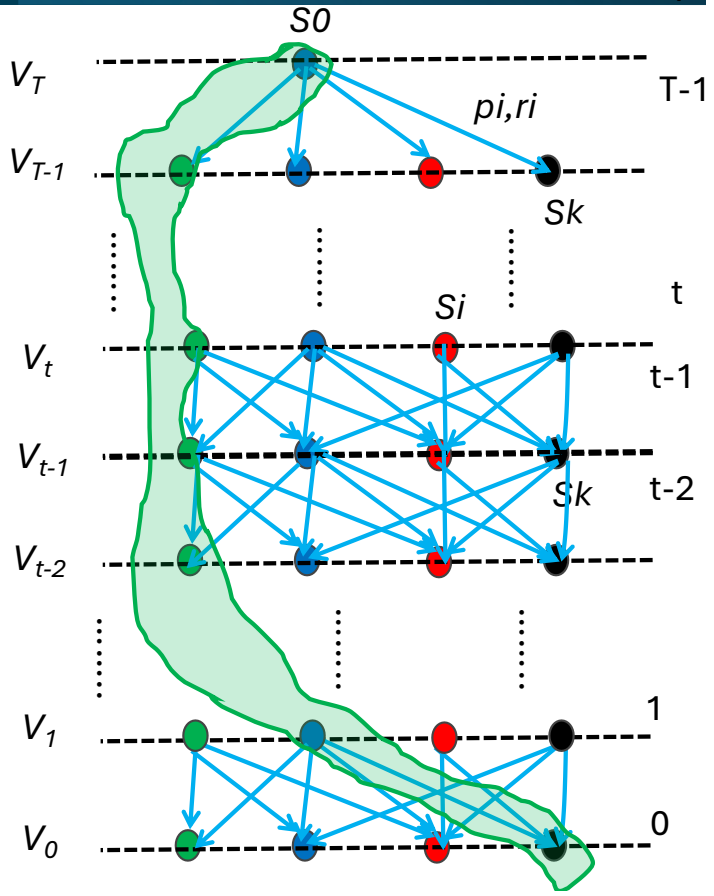
Trajectory: $\tau \sim \pi$ path from root to a leaf

Probability of trajectory τ : $\mathbb{P}(\tau)$ = product of edge probabilities

Reward of trajectory τ : R(τ) = sum of rewards on trajectory

Sum of trajectory probabilities = 1

Meaningful to talk about $\mathbb{E}_{\tau\sim\pi}\left[\text{property of } \tau\right]\left(= \sum_{\tau\sim\pi} \mathbb{P}(\tau)(\text{property of } \tau)\right)$



V/Q Valuations as Expectations over Trajectories: Continuous tasks

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim D^{\pi}(s)}\Big[R(\tau)\Big] \quad \text{(where } D^{\pi}(s) \text{ is set of trajectories starting at s)}$$

Proof for continous tasks (episodic task proof is similar)

$$V^{\pi}(s) = \sum_{s'} p(s,s')[\ \boxed{R(s,s')} + \gamma \boxed{V^{\pi}(s')}\ ] \quad \text{(fixpoint equation)}$$

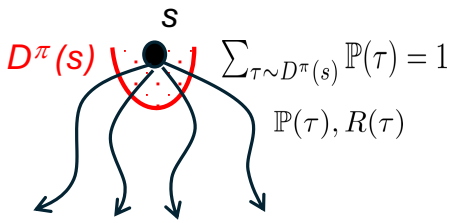
$$= \sum_{s'} p(s,s')[\ \boxed{R(s,s')} + \gamma \boxed{\sum_{\tau \sim D^{\pi}(s')} \mathbb{P}(\tau) * R(\tau)}\ ] \quad \text{(assumption about } V^{\pi}(s'))$$

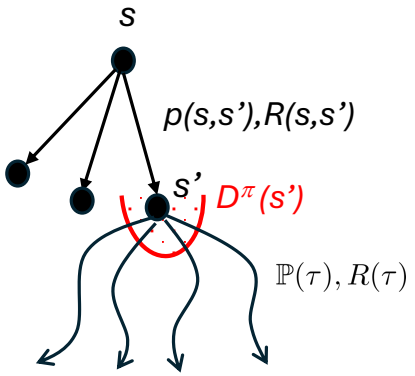
$$= \sum_{s'} p(s,s')[\ \boxed{\sum_{\tau \sim D^{\pi}(s')} \mathbb{P}(\tau) * R(s,s')} + \gamma \boxed{\sum_{\tau \sim D^{\pi}(s')} \mathbb{P}(\tau) * R(\tau)}\ ] \quad \text{(because } \boxed{\sum_{\tau \sim D^{\pi}(s')} \mathbb{P}(\tau)} = 1)$$

$$= \sum_{s'} p(s,s') \sum_{\tau \sim D^{\pi}(s')} \mathbb{P}(\tau) * \boxed{(R(s,s') + \gamma * R(\tau))}$$

$$= \sum_{s'} \sum_{\tau \sim D^{\pi}(s')} p(s,s') * \mathbb{P}(\tau) * \boxed{(R(s,s') + \gamma * R(\tau))}$$

$$= \sum_{\tau \sim D^{\pi}(s)} \mathbb{P}(\tau) * (R(\tau))$$





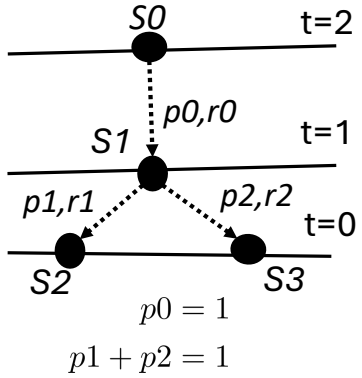
Example

Bellman:

$$V(S0) = \boxed{p0{*}(r0 + (p1{*}r1 + p2{*}r2))}$$
$$= p0{*}r0 + p0{*}p1{*}r1 + p0{*}p2{*}r2$$

Trajectories:

$$V(S0) = \boxed{p0{*}p1{*}(r0{+}r1) + p0{*}p2{*}(r0{+}r2)}$$
$$= p0{*}p1{*}r0 + p0{*}p1{*}r1 + p0{*}p2{*}r0 + p0{*}p2{*}r2$$
$$= p0{*}r0{*}\underbrace{(p1{+}p2)}_{=1} + p0{*}p1{*}r1 + p0{*}p2{*}r2$$
$$= p0{*}r0 + p0{*}p1{*}r1 + p0{*}p2{*}r2$$

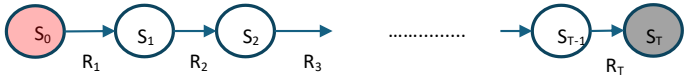


Unrolled MDP for H = 2, policy π

Intuition: rooted DAG → set of paths starting at root

Contribution of each edge to expected reward at *S0* is distributed over trajectories that contain it

Episode



From a mathematical perspective, multiset of samples can be obtained by jumping randomly between states

Practical view: more economical to follow **paths in the state space**, recording states and rewards, optionally updating V/Q values on the fly

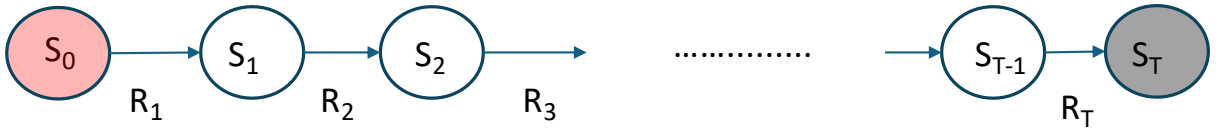
Terminal/absorbing states

- States with no outgoing edges in MDP graph
- (E.g.) win/lose states in two-person games

Episode: $<S_0,A_0,R_1,S_1,A_1,R_2,S_2,\ldots,S_{T-1},A_{T-1},R_T,S_T>$

- Sequence of state transitions that form a path in the MDP graph
- S_T is terminal state

Monte Carlo Method



Problem

- MDP with **absorbing/terminal** states w/fixed valuations
- Policy is fixed: π

Monte Carlo method

- Sample multiset of episodes
  - At each state, sum discounted rewards from all samples starting at that state
  - At the end, divide sum by number of episodes starting at that state
- Intuition: expected number of times path is sampled is proportional to its probability
- *Non-boot-strapping* method: we are propagating updates to state valuations back from terminal states with fixed valuations

Details

- For given episode, update valuations of *all* intermediate nodes on path
  - Implementation: propagate (discounted) rewards backwards along path
- Repeated occurrences of node on path
  - First-visit MC vs. every-visit MC

Monte Carlo (no discount): Barto & Sutton

Algorithm 1: First-Visit MC Prediction

Input: policy π, positive integer *num_episodes*
Output: value function V ($\approx v_\pi$, if *num_episodes* is large enough)
Initialize $N(s) = 0$ for all $s \in \mathcal{S}$
Initialize Returns$(s) = 0$ for all $s \in \mathcal{S}$
for *episode* $e \leftarrow 1$ ***to*** $e \leftarrow num\_episodes$ **do**
  Generate, using π, an episode $S_0, A_0, R_1, S_1, A_1, R_2 \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for *time step* $t = T - 1$ ***to*** $t = 0$ *(of the episode e)* **do**
    $G \leftarrow G + R_{t+1}$
    if *state* S_t *is* ***not*** *in the sequence* $S_0, S_1, \ldots, S_{t-1}$ **then**
      Returns$(S_t) \leftarrow$ Returns$(S_t) + G_t$
      $N(S_t) \leftarrow N(S_t) + 1$
  end
end
$V(s) \leftarrow \frac{\text{Returns}(s)}{N(s)}$ for all $s \in \mathcal{S}$
return V

Algorithm 2: Every-Visit MC Prediction

Input: policy π, positive integer *num_episodes*
Output: value function V ($\approx v_\pi$, if *num_episodes* is large enough)
Initialize $N(s) = 0$ for all $s \in \mathcal{S}$
Initialize Returns$(s) = 0$ for all $s \in \mathcal{S}$
for *episode* $e \leftarrow 1$ ***to*** $e \leftarrow num\_episodes$ **do**
  Generate, using π, an episode $S_0, A_0, R_1, S_1, A_1, R_2 \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for *time step* $t = T - 1$ ***to*** $t = 0$ *(of the episode e)* **do**
    $G \leftarrow G + R_{t+1}$
    Returns$(S_t) \leftarrow$ Returns$(S_t) + G_t$
    $N(S_t) \leftarrow N(S_t) + 1$
  end
end
$V(s) \leftarrow \frac{\text{Returns}(s)}{N(s)}$ for all $s \in \mathcal{S}$
return V

Pros and cons of MC vs. TD

TD can learn before knowing terminal state in episode

- TD can learn after every step of episode
- MC must wait till the terminal state is known

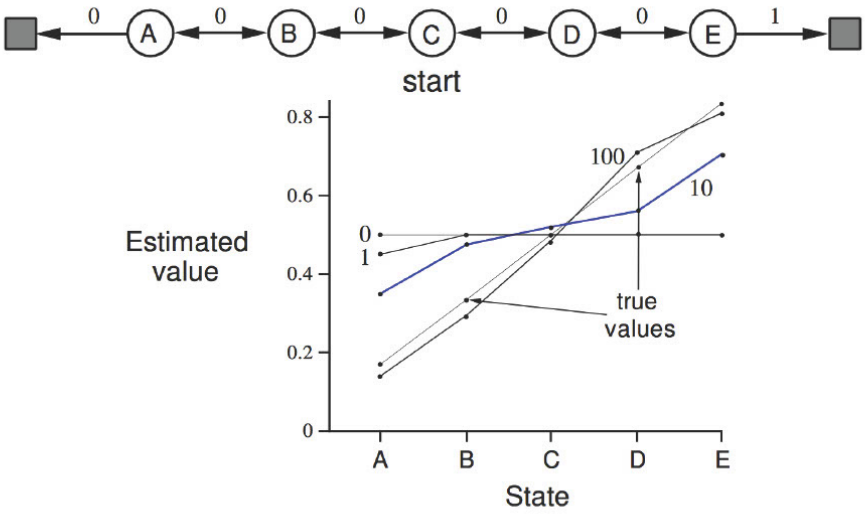
TD can learn even if there is no terminal state in episode

- TD can learn even in continuing environments where there are no terminal states
- MC only works for episodic (terminating) environments

Bias/variance tradeoff

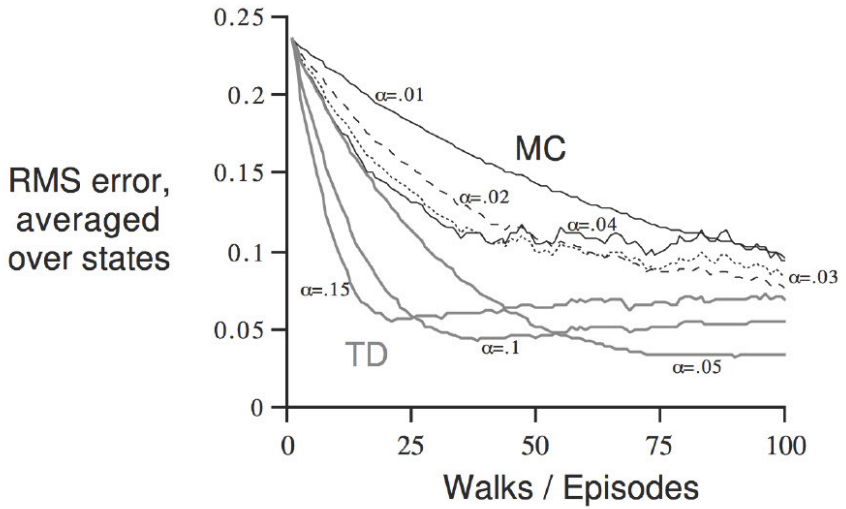
- Bias
  - MC+ : Returns from terminating paths: unbiased estimator of V^{π}
  - TD- : TD targets $R_{t+1} + \gamma * V_i^{\pi}(S_t)$: biased estimator (until convergence)
- Variance
  - MC-: episodes may have long and variable length paths, so high variance
  - TD+: short paths, so lower variance

Random Walk Example



Convergence

Random Walk: MC vs. TD



Implementation: Eligibility Traces & TD(λ)

TD(0) Pseudocode with Episodes (Barto & Sutton)

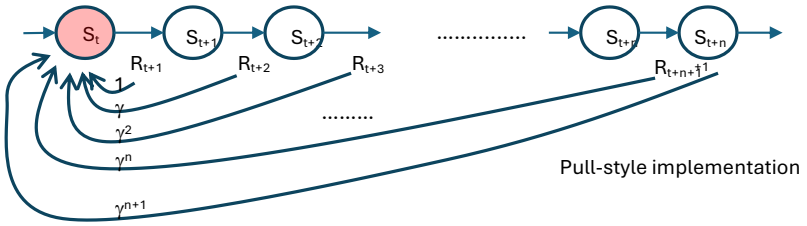
Algorithm 1 Tabular TD(0) for estimating v_π

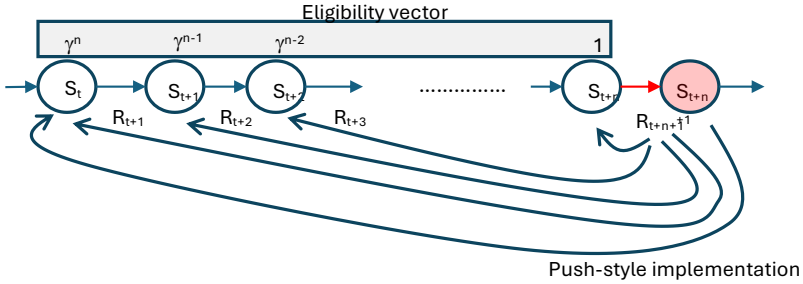
Input: Policy π to be evaluated **Parameters:** Learning rate $\alpha \in (0, 1]$

for each episode: **do**
    Initialize S
    while S is not terminal: **do**
        Take action A given by $\pi(a|S)$
        Observe R, S'
        Update $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
        $S \leftarrow S'$
    end while
end for

Implementing TD(n): push vs. pull

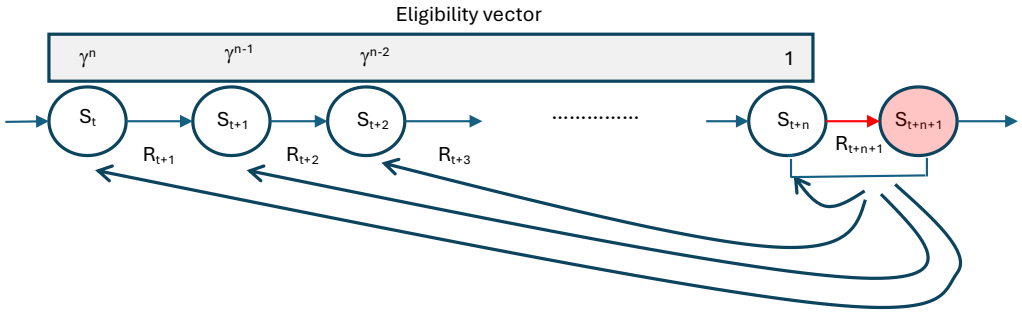
$$G_t^n = R_{t+1} + \gamma R_{t+2} ..... \gamma^n R_{t+n+1} + \gamma^{n+1} V^n(S_{t+n+1})$$
$$V^n(S_t) = (1-\alpha) V^n(S_t) + \alpha(G_t^n)$$





Detail: updates actually performed using TD(0) Errors

$$
\begin{aligned}
G_t^n &= R_{t+1} + \gamma R_{t+2} ..... \gamma^n R_{t+n+1} + \gamma^{n+1} V^n(S_{t+n+1}) \\
&= \Sigma_{i=0}^n (\gamma^i R_{t+i+1} + \gamma^{i+1} V^n(S_{t+i+1}^n) - \gamma^i V^n(S_{t+i})) \\
&= \Sigma_{i=0}^n \gamma^i (R_{t+i+1} + \gamma V^n(S_{t+i+1}) - V^n(S_{t+i})) \\
&= \Sigma_{i=0}^n \gamma^i G_{t+i}^0 \\
V^n(S_t) &= (1-\alpha) V^n(S_t) + \alpha(G_t^n)
\end{aligned}
$$

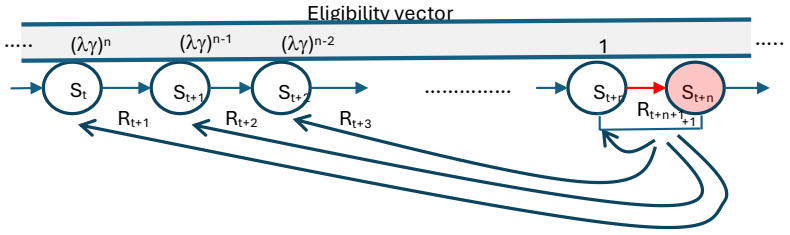


TD(λ): get the effect of TD(n) without having to pick an n

Intuition: rather than drop old states from sliding window, use exponential recency-weighted averaging trick

Sliding window for TD(n):
- Multiply Eligibility vector by γ
- Eligibility[S_{t+n+1}] ← 1
- Eligibility[S_t] ← 0

Sliding window for TD(λ): replacing traces version
- New hyper-parameter λ
- Multiply Eligibility vector by $\gamma\lambda$
- Eligibility[St+n+1] ← 1



Implementation of TD(λ)

Algorithm 1: TD(λ) - estimating state-value function with eligibility traces.

```
import numpy as np

state_values = np.zeros(n_states) # initial guess = 0 value
eligibility = np.zeros(n_states)

lamb = 0.95 # the lambda weighting factor
state = env.reset() # start the environment, get the initial state
# Run the algorithm for some episodes
for t in range(n_steps):
  # act according to policy
  action = policy(state)
  new_state, reward, done = env.step(action)
  # Update eligibilities
  eligibility *= lamb * gamma
  eligibility[state] += 1.0

  # get the td-error and update every state's value estimate
  # according to their eligibilities.
  td_error = reward + gamma * state_values[new_state] - state_values[state]
  state_values = state_values + alpha * td_error * eligibility

  if done:
    state = env.reset()
  else:
    state = new_state
```

Alistair Ries blog:
https://amreis.github.io/ml/reinf-learn/2017/11/02/reinforcement-learning-eligibility-traces.html

Summary: concepts and keywords

Model-free methods

Deterministic vs. stochastic policies

Offline vs. online methods

Boot-strapping methods: TD(0), TD(n), TD(λ), SARSA, Q-learning

Non-boot-strapping methods: Monte Carlo

Bias-variance tradeoff between TD and Monte Carlo

Eligibility traces: implementation