UNIT- 4

Monte Carlo Prediction, Monte Carlo Estimation of Action Values , Monte Carlo Control , Off-policy Prediction via Importance Sampling, Incremental Implementation , Off-Policy Monte Carlo Control , Temporal-Difference Learning: TD Prediction , Advantages of TD Prediction Methods, Optimality of TD(O), TD(l) , Sarsa: On-Policy TD Control , Q-Learning: Off-Policy TD Control, unified view of DP, MC and TD evaluation methods

MODEL FREE LEARNING

- In model-free setting we do not have the full knowledge of the MDP
- **Model-free prediction**: Estimate the value function of an unknown MDP
- **Model-free control**: Optimise the value function of an unknown MDP
- Model-free methods require only *experience* - sample sequences of states, actions, and rewards $(S_1, A_1, R_2, \cdots)$ from **actual** or **simulated** interaction with an environment.
- Actual experince requires no knowledge of the environment's dynamics.
- Simulated experience 'requires' models to generate samples only. No knowledge of the complete probability distributions of state transitions is required. In many cases this is easy to do.

Monte Carlo Prediction

- Each occurrence of state s in an episode is called a visit to s.
- The first-visit MC method estimates v̂(s) as the average of the returns following first visits to s, whereas the every-visit MC method averages the returns following all visits to s.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

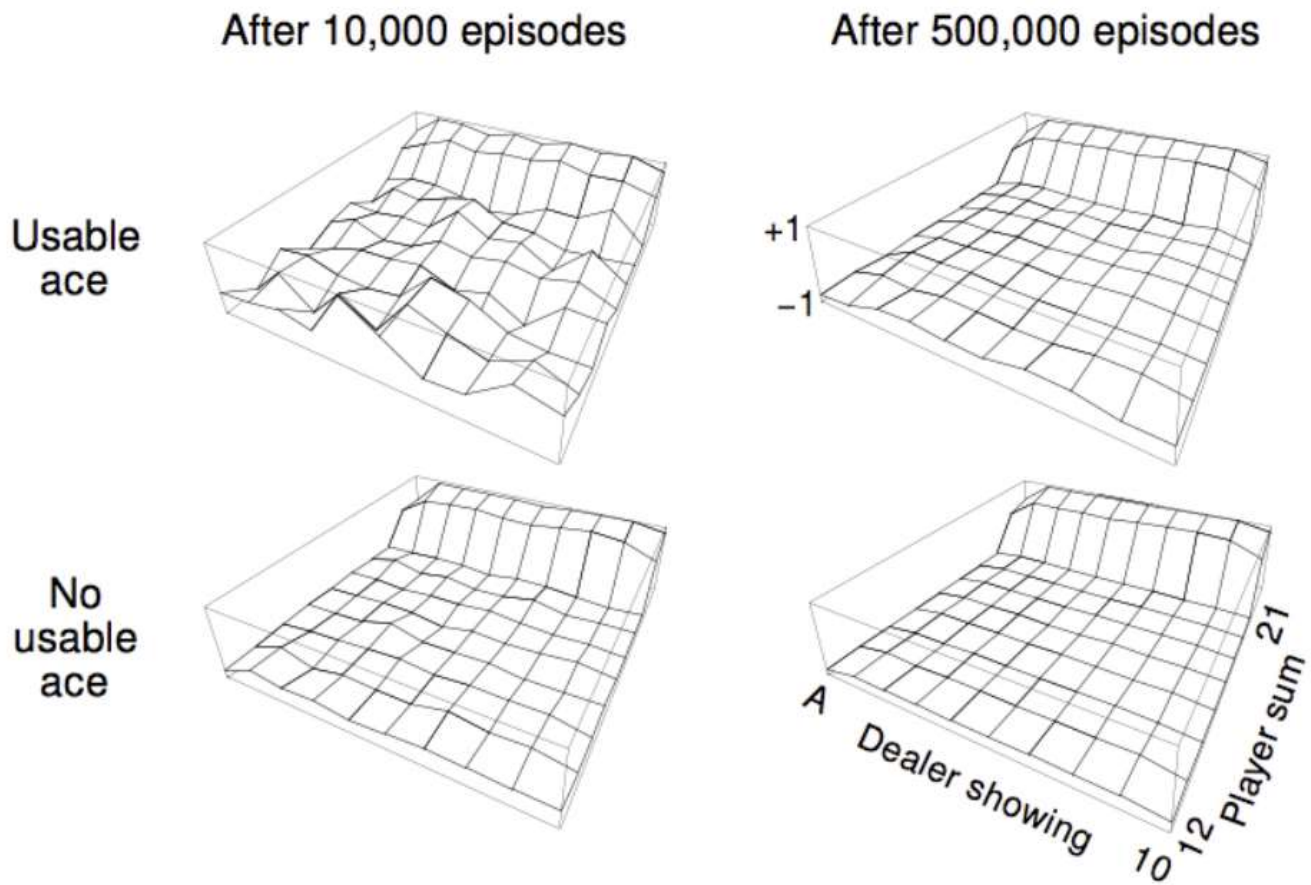
Blackjack Example

- States (200 of them):
  - Current sum (12-21)
  - Dealer's showing card (ace-10)
  - Do I have a "useable" ace? (yes-no)
- Action stick: Stop receiving cards (and terminate)
- Action twist: Take another card (no replacement)
- Reward for stick:
  - +1 if sum of cards $>$ sum of dealer cards
  - 0 if sum of cards $=$ sum of dealer cards
  - -1 if sum of cards $<$ sum of dealer cards
- Reward for twist:
  - -1 if sum of cards $>$ 21 (and terminate)
  - 0 otherwise
- Transitions: automatically twist if sum of cards $<$ 12

Slide courtesy: *David Silver [Deepmind]*


How to Play
BLACKJACK
wikiHow

Blackjack Example



Policy: stick if sum of cards ≥ 20, otherwise twist

Slide courtesy: *David Silver [Deepmind]*

Monte Carlo Estimation of Action Values

1.Estimating Action Values: When a model is not available, it's crucial to estimate action values (q-values) rather than just state values. This is because action values help in suggesting a policy when there's uncertainty about the environment.

2.Policy Evaluation for Action Values: The goal is to estimate q-values (expected return for a state-action pair) under a given policy. Monte Carlo methods are used for this purpose.

3.Visiting State-Action Pairs: In Monte Carlo methods, a state-action pair is considered visited if the state is visited and the corresponding action is taken in an episode.

4.Every-Visit vs. First-Visit MC Methods: Every-visit MC method averages returns over all visits to a state-action pair, while first-visit MC method averages returns only after the first visit in each episode.

5.Exploration Problem: If a deterministic policy is followed, many state-action pairs may never be visited, leading to inaccurate estimates of action values. This is because learning requires exploration of all possible actions from each state.

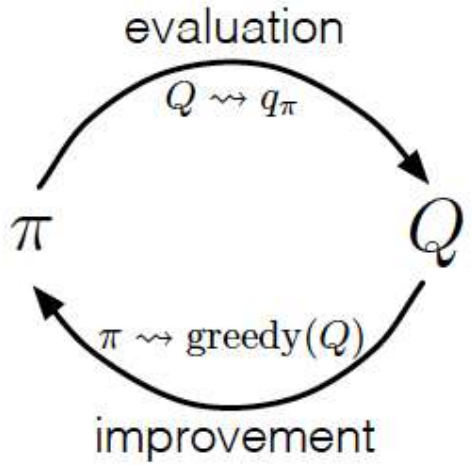
6.Maintaining Exploration: To ensure exploration, one approach is to start episodes in a state-action pair with a nonzero probability, guaranteeing that all state-action pairs are visited infinitely over time. This is known as the assumption of exploring starts.

7.Alternative Approaches: In practical scenarios, exploring starts may not be feasible. Instead, stochastic policies that have a nonzero probability of selecting all actions in each state can be used to ensure all state-action pairs are encountered.

Monte Carlo Control

•Maintains both an approximate policy and an approximate value function, iteratively improving towards optimality.
•Repeatedly updates the value function to better approximate the current policy's value function.
•Repeatedly improves the policy based on the current value function to enhance its effectiveness.
•Changes in the value function and policy create moving targets for each other but collectively contribute to approaching optimality.
•Involves alternating complete steps of policy evaluation and improvement, starting with an initial policy ($\pi 0$) and ending with the optimal policy and action-value function.

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$



- Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an action-value function, and therefore no model is needed to construct the greedy policy.

$$\pi(s) \doteq \arg\max_a q(s,a).$$

Policy improvement theorem is

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

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Off-policy Prediction via Importance Sampling

- Learning Control Dilemma: Methods aim to learn action values based on optimal behavior but need to explore non-optimal actions to find the best ones. This raises the question of learning about the optimal policy while behaving exploratively.
- On-Policy Learning: Involves a compromise, learning action values for a near-optimal policy that still explores. Two policies are used: the target policy (to be learned) and the behavior policy (for exploration), termed as off-policy learning.
- On-Policy vs. Off-Policy: On-policy methods are simpler but less powerful, while off-policy methods are more complex but more versatile, applicable even with different behavior policies.
- Off-Policy Methods: Include on-policy methods as a special case and have various applications, like learning from non-learning controllers or human experts.
- Assumption of Coverage: Off-policy learning requires that every action under the target policy is occasionally taken under the behavior policy, ensuring coverage.
- Importance Sampling: Key to off-policy methods, it weights returns based on the relative probabilities of trajectories under the target and behavior policies, known as the importance-sampling ratio.

Given a starting state St, the probability of the subsequent state–action trajectory, A_t, S_{t+1},A_{t+1}, . . . ,ST , occurring under any policv $\boldsymbol{\pi}$ is

$$\begin{aligned}
&\Pr\{A_t, S_{t+1}, A_{t+1}, \ldots, S_T \mid S_t, A_{t:T-1} \sim \pi\} \\
&\quad = \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\
&\quad = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k),
\end{aligned}$$

where p here is the state-transition probability function defined by (3.4). Thus, the relative probability of the trajectory under the target and behavior policies (the importance-sampling ratio) is

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{5.3}$$

Although the trajectory probabilities depend on the MDP's transition probabilities, which are generally unknown, they appear identically in both the numerator and denominator, and thus cancel. The importance sampling ratio ends up depending only on the two policies and the sequence, not on the MDP.

Recall that we wish to estimate the expected returns (values) under the target policy, but all we have are returns G_t due to the behavior policy. These returns have the wrong expectation $\mathbb{E}[G_t|S_t=s] = v_b(s)$ and so cannot be averaged to obtain v_π. This is where importance sampling comes in. The ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value:

$$\mathbb{E}[\rho_{t:T-1}G_t \mid S_t=s] = v_\pi(s). \tag{5.4}$$

To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}G_t}{|\mathcal{T}(s)|}.$$

When importance sampling is done as a simple average in this way it is called *ordinary importance sampling*.

An important alternative is *weighted importance sampling*, which uses a *weighted* average, defined as

$$V(s) \doteq \frac{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}G_t}{\sum_{t\in\mathcal{T}(s)} \rho_{t:T(t)-1}}, \tag{5.6}$$

Incremental Implementation

•Monte Carlo Prediction Methods: Can be implemented incrementally, episode-by-episode, using extensions of techniques. Instead of averaging rewards, Monte Carlo methods average returns.
•On-Policy Monte Carlo Methods: Use the same methods but average returns instead of rewards.
•Off-Policy Monte Carlo Methods: Use ordinary importance sampling or weighted importance sampling.
•Ordinary Importance Sampling: Scales returns by the importance sampling ratio and then averages them. Incremental methods frcan be adapted by using scaled returns instead of rewards.
•Weighted Importance Sampling: Requires forming a weighted average of returns, needing a slightly different incremental algorithm compared to ordinary importance sampling

Suppose we have a sequence of returns $G_1, G_2, \ldots, G_{n-1}$, all starting in the same state and each with a corresponding random weight W_i (e.g., $W_i = \rho_{t_i:T(t_i)-1}$). We wish to form the estimate

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \qquad n \geq 2, \tag{5.7}$$

and keep it up-to-date as we obtain a single additional return G_n. In addition to keeping track of V_n, we must maintain for each state the cumulative sum C_n of the weights given to the first n returns. The update rule for V_n is

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\Big[G_n - V_n\Big], \qquad n \geq 1, \tag{5.8}$$

and

$$C_{n+1} \doteq C_n + W_{n+1},$$

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of π
    Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

Off-policy Monte Carlo Control

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \in \mathbb{R}$ (arbitrarily)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \arg\max_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

Temporal-Difference Learning

Monte Carlo (MC) Methods:

- **Update Based on Full Episode:** MC methods wait until the end of an episode to update the value estimates.
- **No Bootstrapping:** They do not use bootstrapping, meaning they rely solely on actual returns obtained during an episode.
- **Higher Variance:** MC methods typically have higher variance, especially when dealing with long episodes or high stochasticity.
- **Suitable for Episodic Tasks:** They are well-suited for episodic tasks where the agent interacts with the environment until termination.
- **Example:** MC methods calculate returns by averaging the actual rewards obtained from the start of an episode until the end.

Temporal Difference (TD) Methods:

- **Update at Each Time Step:** TD methods update value estimates at each time step based on the immediate reward and the estimated value of the next state.
- **Bootstrapping:** They use bootstrapping, incorporating estimates from subsequent states into the value updates.
- **Lower Variance:** TD methods typically have lower variance compared to MC methods, especially in environments with long episodes or high stochasticity.
- **Suitable for Continuous Tasks:** They are suitable for continuous tasks where the agent interacts with the environment continuously.
- **Example:** TD methods update value estimates using a combination of the current reward and the estimated value of the next state, using techniques like TD(0) or TD(λ).

TD Prediction

A simple every-visit Monte Carlo method suitable for nonstationary environments is

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

The simplest TD method makes the update

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

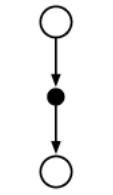
Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Because TD(0) bases its update in part on an existing estimate, we say that it is a bootstrapping method, like DP.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]. \end{aligned}$$



TD(0)

Finally, note that the quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This quantity, called the *TD error*, arises in various forms throughout reinforcement learning:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{6.5}$$

Monte Carlo error can be written as a sum of TD errors

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \qquad \text{(from (3.9))} \\ &= \delta_t + \gamma\big(G_{t+1} - V(S_{t+1})\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\big(G_{t+2} - V(S_{t+2})\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(G_T - V(S_T)\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0 - 0) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. \end{aligned} \tag{6.6}$$

Driving Home Example

- Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant.
- Say on this Friday you are leaving at exactly 6 o'clock, and you estimate that it will take 30 minutes to get home.
- As you reach your car it is 6:05, and you notice it is starting to rain. Traffic is often slower in the rain, so you reestimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time.
- As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn onto the side street where you live at 6:40. Three minutes later you are home.
- The sequence of states, times, and predictions is thus as follows

State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

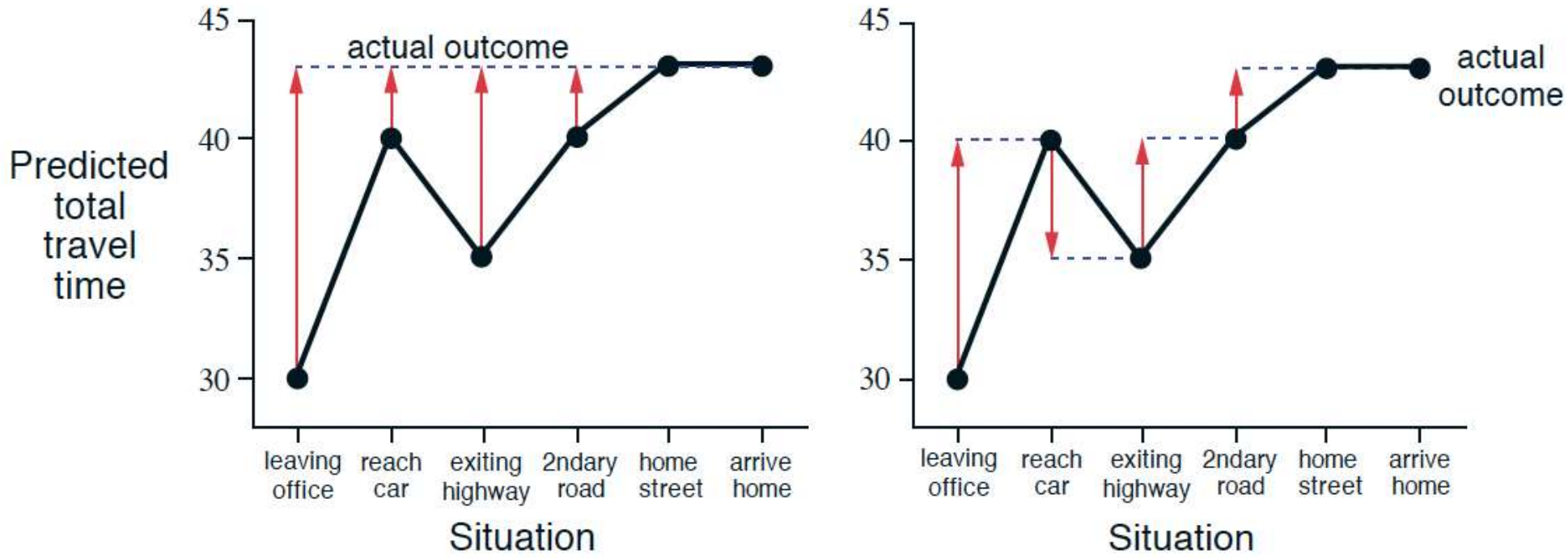


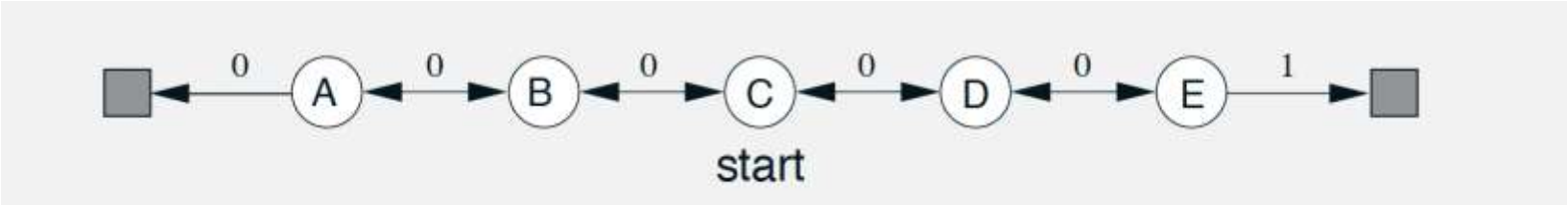
Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

Advantages of TD Prediction methods

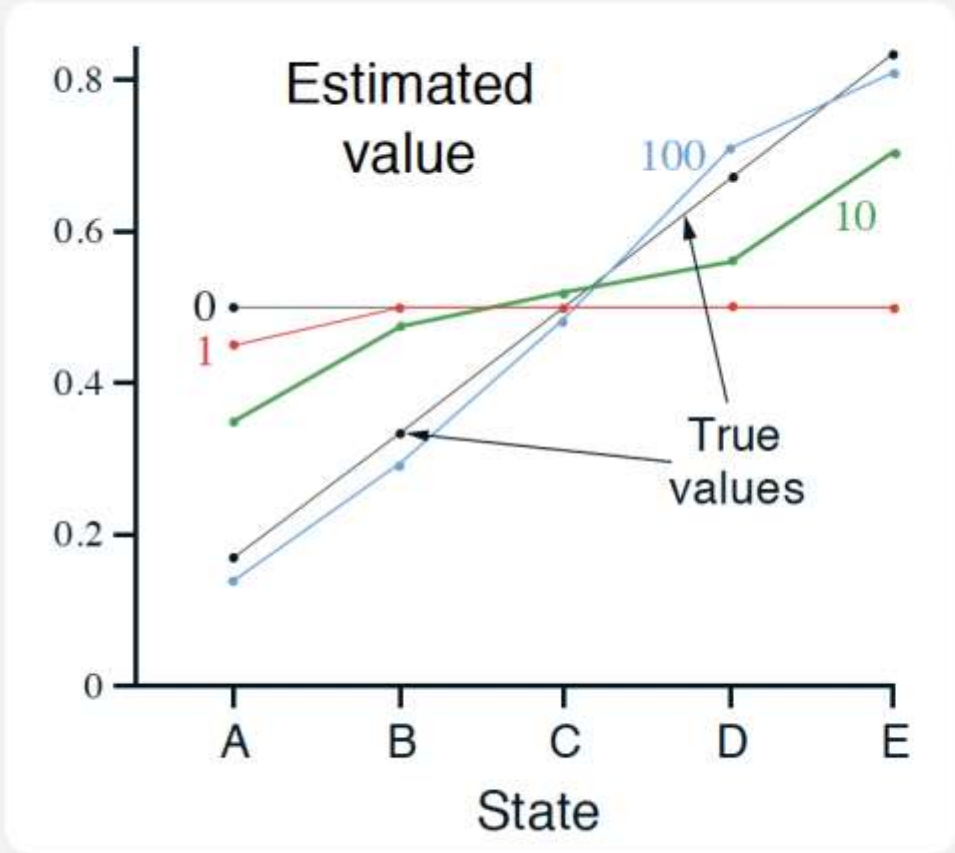
- vs. Dynamic Programming
  - No model required
- vs. Monte Carlo
  - Allows online incremental learning
  - Does not need to ignore episodes with experimental actions

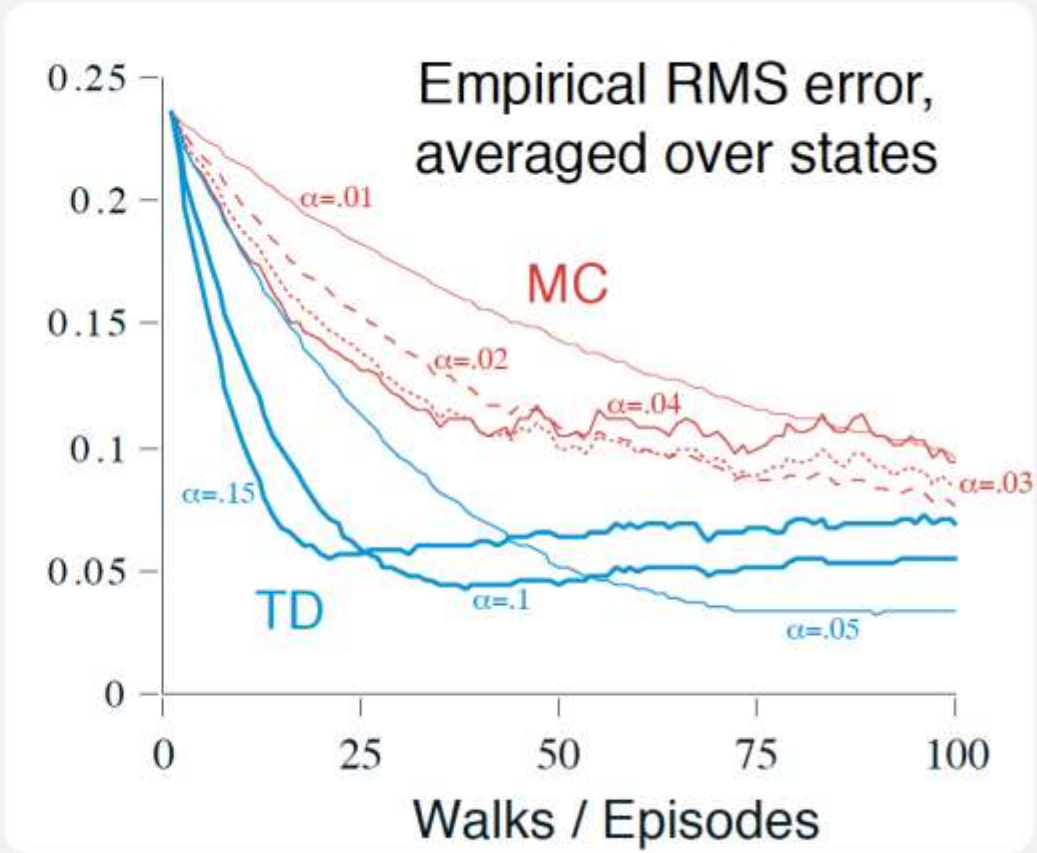
- Still guarantees convergence
- Converges faster than MC *in practice*
  - ex) Random Walk
  - No *theoretical* results yet

Random Walk-Example



- A Markov reward process, or MRP, is a Markov decision process without actions.
- In this MRP, all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability.
- Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero.
- For example, a typical episode might consist of the following state-and-reward sequence: C, 0,B, 0, C, 0,D, 0, E, 1.
- Because this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state.
- Thus, the true value of the center state is $v_{\pi}(C) = 0.5$.
- The true values of all the states, A through E, are1/6 , 2/6, 3/6, 4/6, and 5/6


Estimated value
0.8
0.6
0.4
0.2
0
100
10
0
1
True values
A
B
C
D
E
State

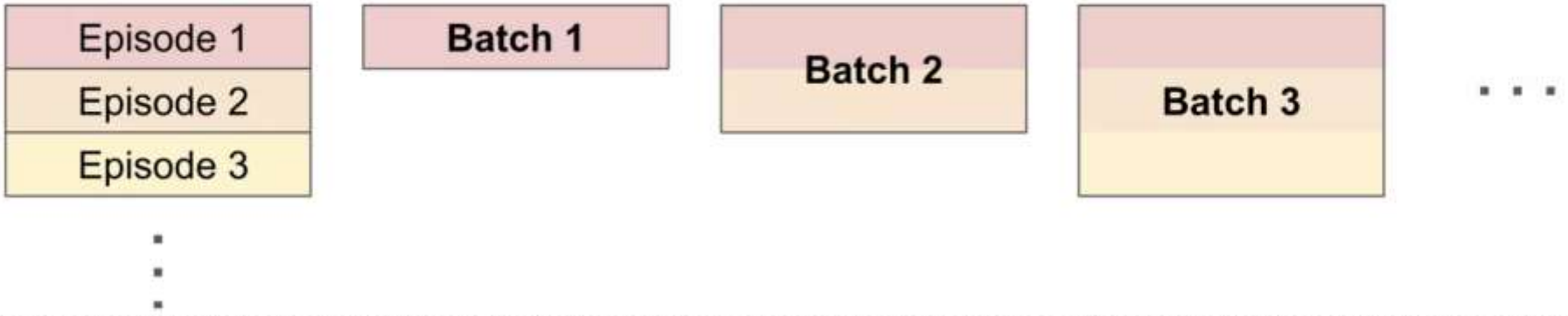

Empirical RMS error, averaged over states
0.25
0.2
0.15
0.1
0.05
0
α=.01
MC
α=.02
α=.04
α=.03
α=.15
TD
α=.1
α=.05
0
25
50
75
100
Walks / Episodes

Optimality of TD(0)

Batch Updating

- Repeat learning from same experience until convergence
- Useful when finite amount of experience is available
- Convergence guaranteed with small step-size parameter
- **MC and TD converge to different answers**

ex)



Sarsa: On-policy TD Control

- Learn action-value function $Q(S_t, A_t)$ with TD(0)
- Use transition $(S_t, A_t, R_{t+1}, S_{t+1}. A_{t+1})$ for updates

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)}_{\text{TD error}} \Big].$$

- Change policy π greedily with q_π
- Converges if:
  - all (s, a) is visited infinitely many times
  - policy converges to greedy policy

Sarsa: On-Policy TD Control Pseudocode

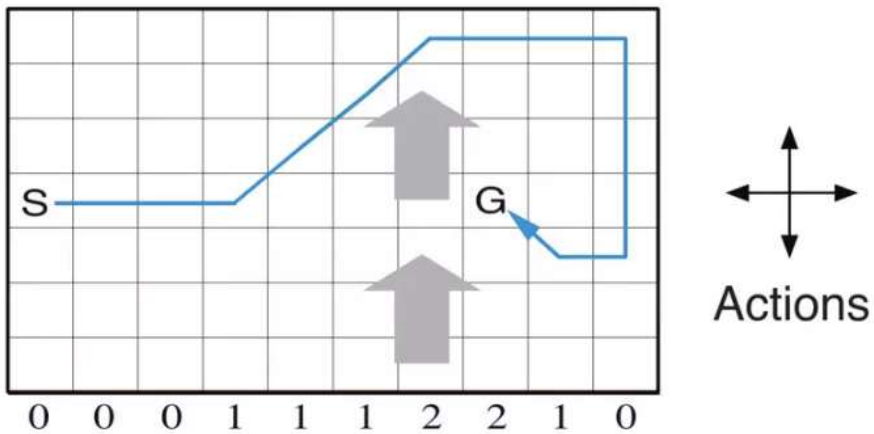
Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Windy Gridworld Example

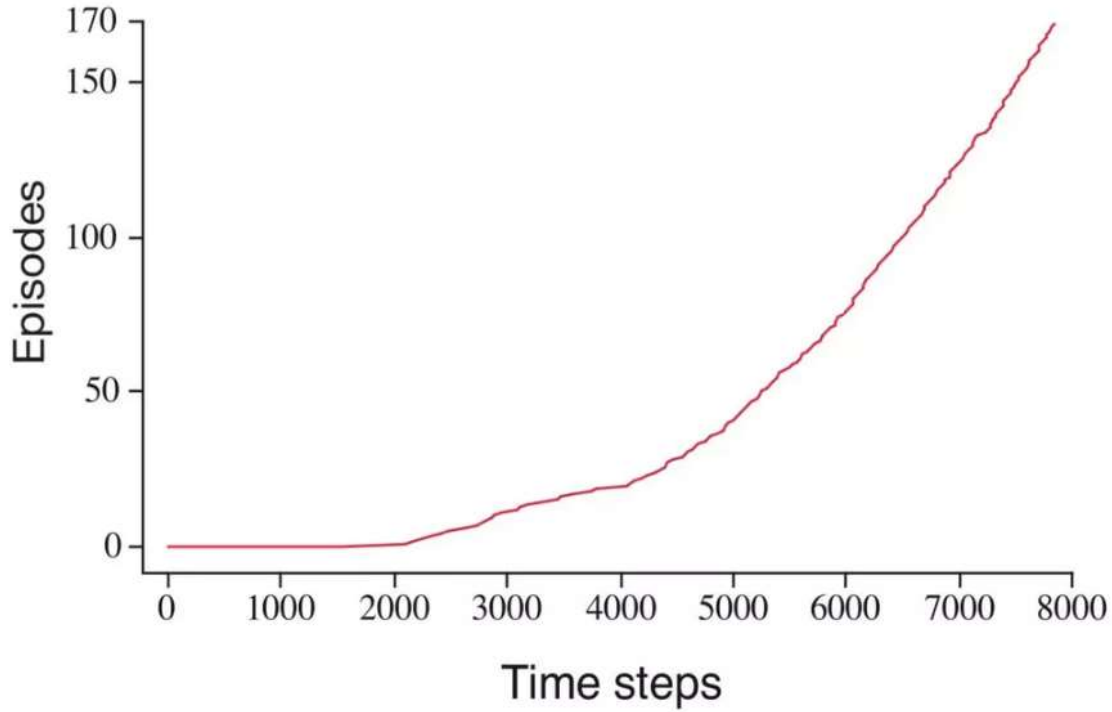
- Gridworld with “Wind”
  - Actions: 4 directions
  - Reward: -1 until goal
  - “Wind” at each column shifts agent upward
  - “Wind” strength varies by column



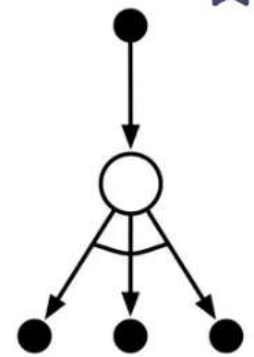
- Termination not guaranteed for all policies
- Monte Carlo cannot be used easily

Windy Gridworld Example

- Converges at 17 steps (instead of optimal 15) due to exploring policy



Q-learning: Off-policy TD Control



- Q directly approximates q_* independent of behavior policy

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

- Converges if all (s, a) is visited infinitely many times

Q-learning: Off-policy TD Control: Pseudocode

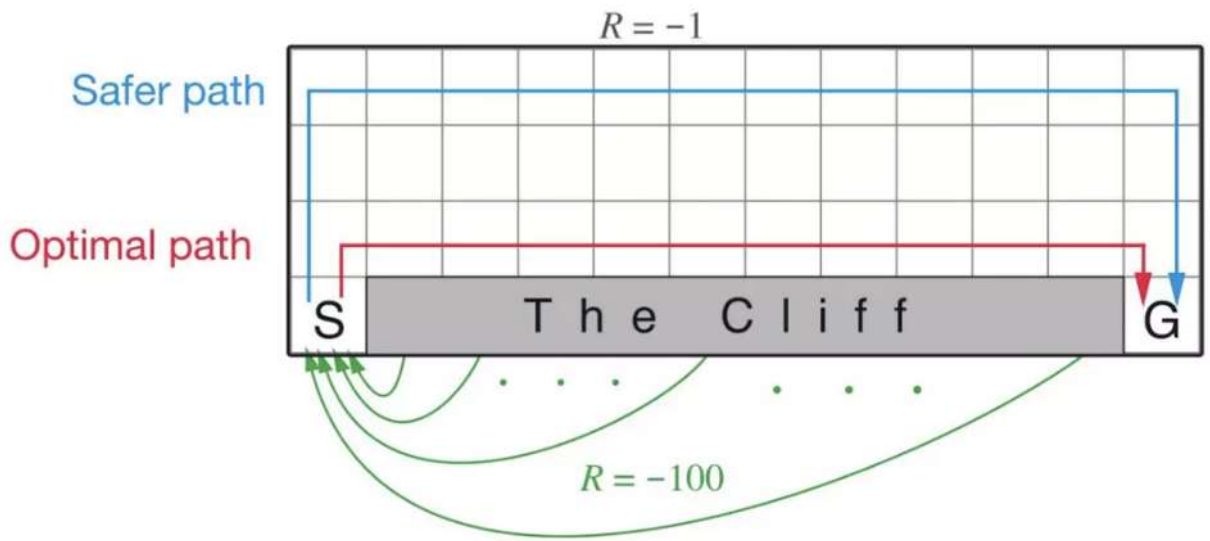
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

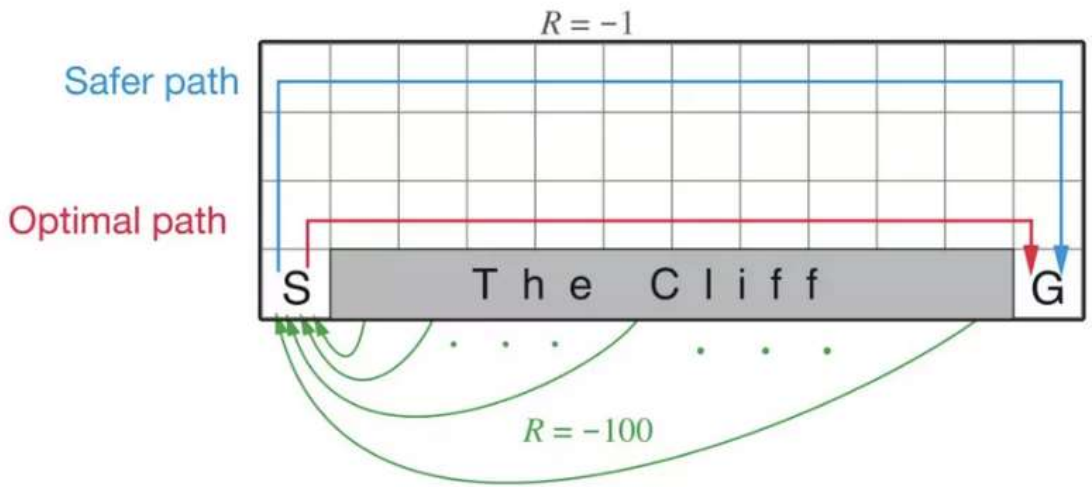
Cliff Walking Example

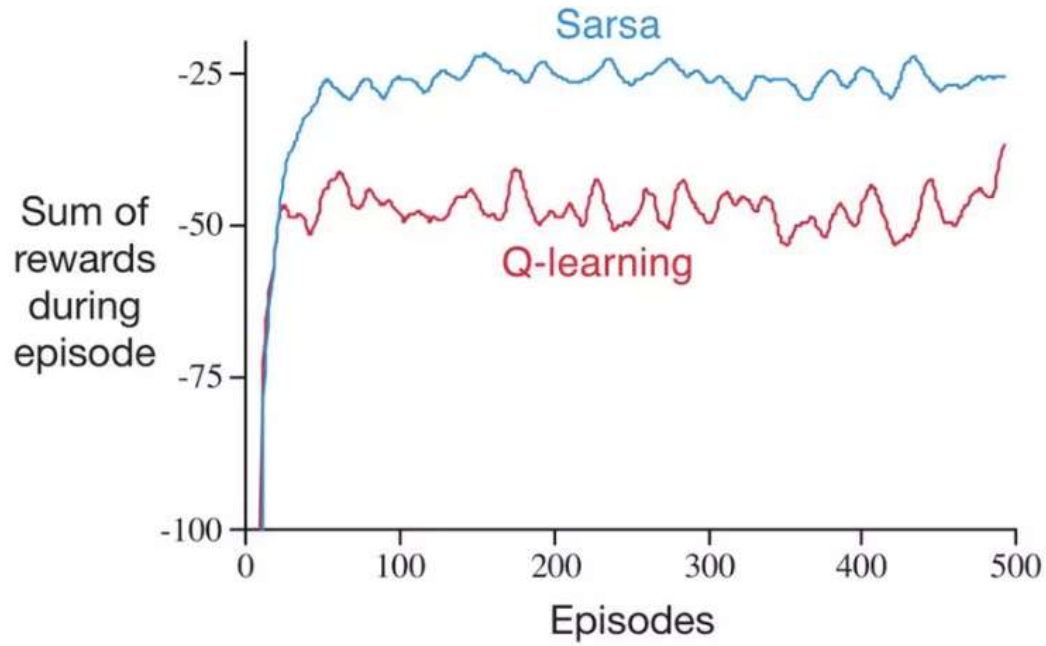
- Gridworld with “cliff” with high negative reward
- ε-greedy (behavior) policy for both Sarsa and Q-learning (ε = 0.1)



Cliff Walking Example: Sarsa vs. Q-learning

- Q-learning learns optimal policy
- Sarsa learns safe policy
- Q-learning has worse online performance
- Both reach optimal policy with ε-decay





Eligibility Traces

- Eligibility traces unify and generalize TD and Monte Carlo methods.
- When TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end ($\boldsymbol{\lambda}$=1) and one-step TD methods at the other ($\boldsymbol{\lambda}$=0).
- Eligibility traces also provide a way of implementing Monte Carlo methods online and on continuing problems without episodes.
- Another way of unifying TD and Monte Carlo methods is the n-step TD methods.
- What eligibility traces offer beyond these is an elegant algorithmic mechanism with significant computational advantages. The mechanism is a short-term memory vector, the eligibility trace $\mathbf{z}_t \in \mathbb{R}^d$, that parallels the long-term weight vector $\mathbf{w}_t \in \mathbb{R}^d$.
- *The idea is that when a component of w_t participates in*

producing an estimated value, then the corresponding component of $\mathbf{z}_t$ is bumped up and then begins to fade away. Learning will then occur in that component of $\mathbf{w}_t$ if a nonzero TD error occurs before the trace falls back to zero. The trace-decay parameter $\lambda \in [0, 1]$ determines the rate at which the trace falls.

Advantages of eligibility traces

- The primary computational advantage of eligibility traces over n-step methods is that only a single trace vector is required rather than a store of the last n feature vectors.
- Learning also occurs continually and uniformly in time rather than being delayed and then catching up at the end of the episode.
- In addition learning can occur and affect behavior immediately after a state is encountered rather than being delayed n steps.

Forward Views:

- Monte Carlo methods -update a state based on all the future rewards
- n-step TD methods - update based on the next n rewards and state n steps in the future.
- Such formulations, based on looking forward from the updated state, are called forward views.
- Forward views are always somewhat complex to implement because the update depends on later things that are not available at the time.

Backward Views:

- It is often possible to achieve nearly the same updates—and sometimes exactly the same updates—with an algorithm that uses the current TD error, looking backward to recently visited states using an eligibility trace.
- These alternate ways of looking at and implementing learning algorithms are called backward views.

The λ-return

- n-step return as the sum of the first n rewards plus the estimated value of the state reached in n steps, each appropriately discounted.
- The general form of that equation, for any parameterized function approximator, is

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^{n} \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T - n,$$

The TD(λ) algorithm can be understood as one particular way of averaging n-step updates. This average contains all the n-step updates, each weighted proportionally to λ^{n-1} (where $\lambda \in [0, 1]$), and is normalized by a factor of $1 - \lambda$ to ensure that the weights sum to 1 (Figure 12.1). The resulting update is toward a return, called the λ-*return*, defined in its state-based form by

$$G_t^{\lambda} \doteq (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}. \tag{12.2}$$

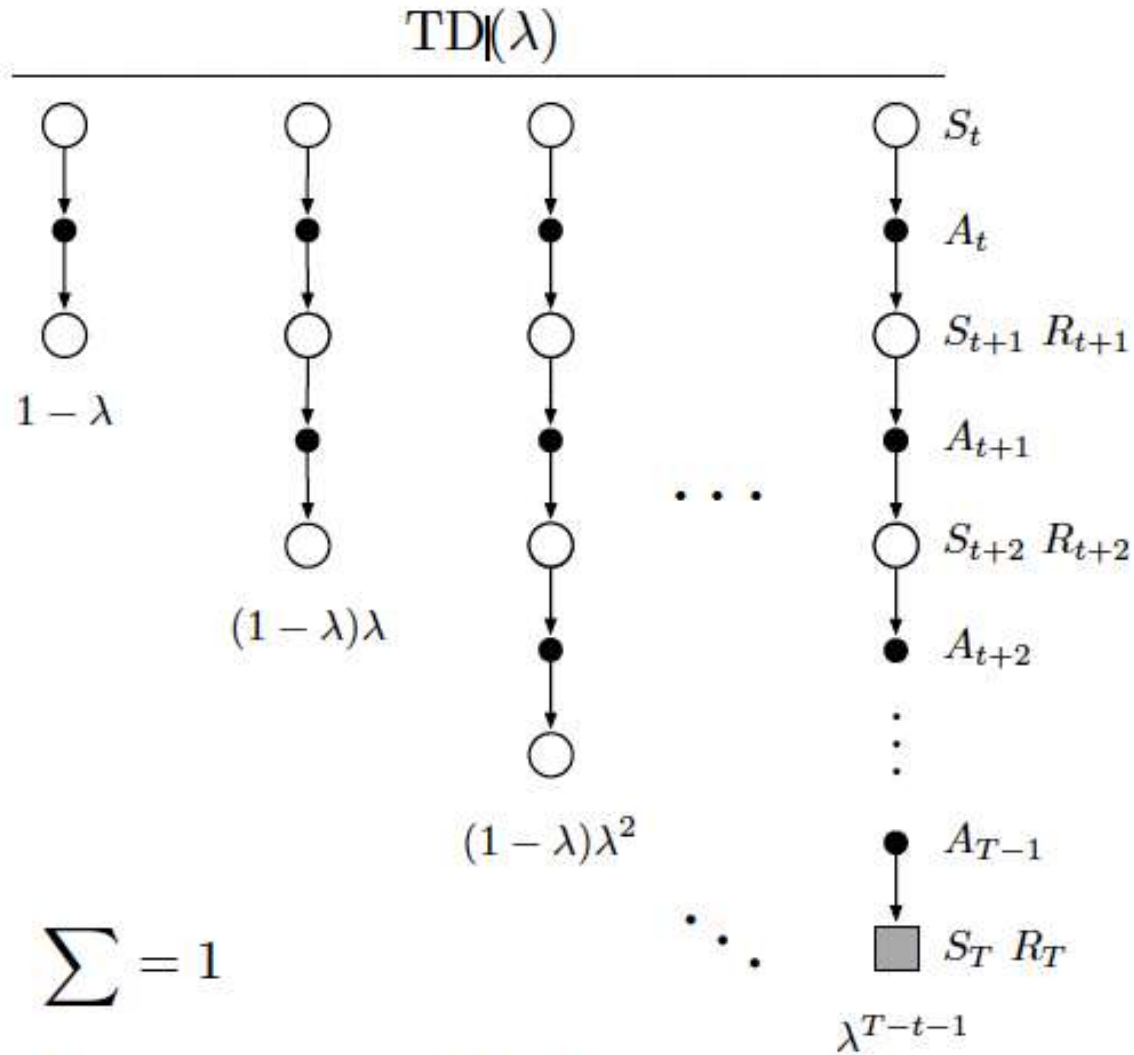


Figure 12.1: The backup digram for TD(λ). If $\lambda = 0$, then the overall update reduces to its first component, the one-step TD update, whereas if $\lambda = 1$, then the overall update reduces to its last component, the Monte Carlo update.

Figure 12.2 further illustrates the weighting on the sequence of n-step returns in the λ-return. The one-step return is given the largest weight, $1-\lambda$; the two-step return is given the next largest weight, $(1-\lambda)\lambda$; the three-step return is given the weight $(1-\lambda)\lambda^2$; and so on. The weight fades by λ with each additional step. After a terminal state has been reached, all subsequent n-step returns are equal to the conventional return, G_t. If

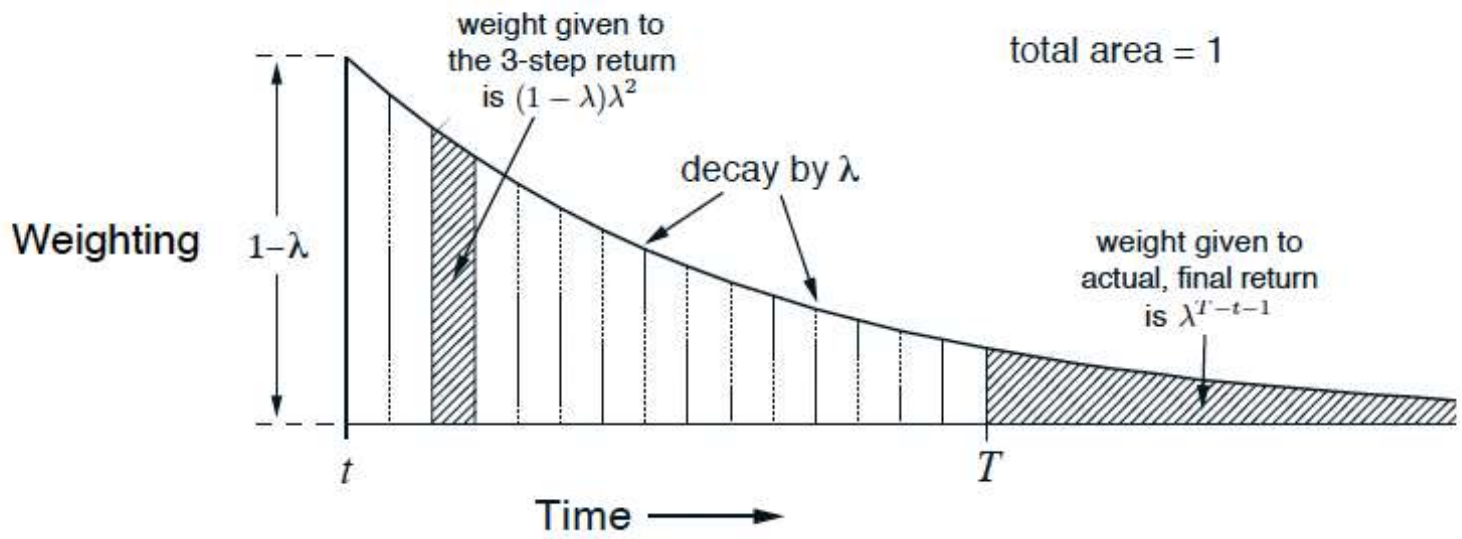


Figure 12.2: Weighting given in the λ-return to each of the n-step returns.

$$G_t^\lambda \;=\; (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} \;+\; \lambda^{T-t-1}G_t,$$

As an offline algorithm, it makes no changes to the weight vector during the episode. Then, at the end of the episode, a whole sequence of offline updates are made according to our usual semi-gradient rule, using the $\boldsymbol{\lambda}$-return as the target:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[G_t^\lambda - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t), \quad t=0,\ldots,T-1.$$

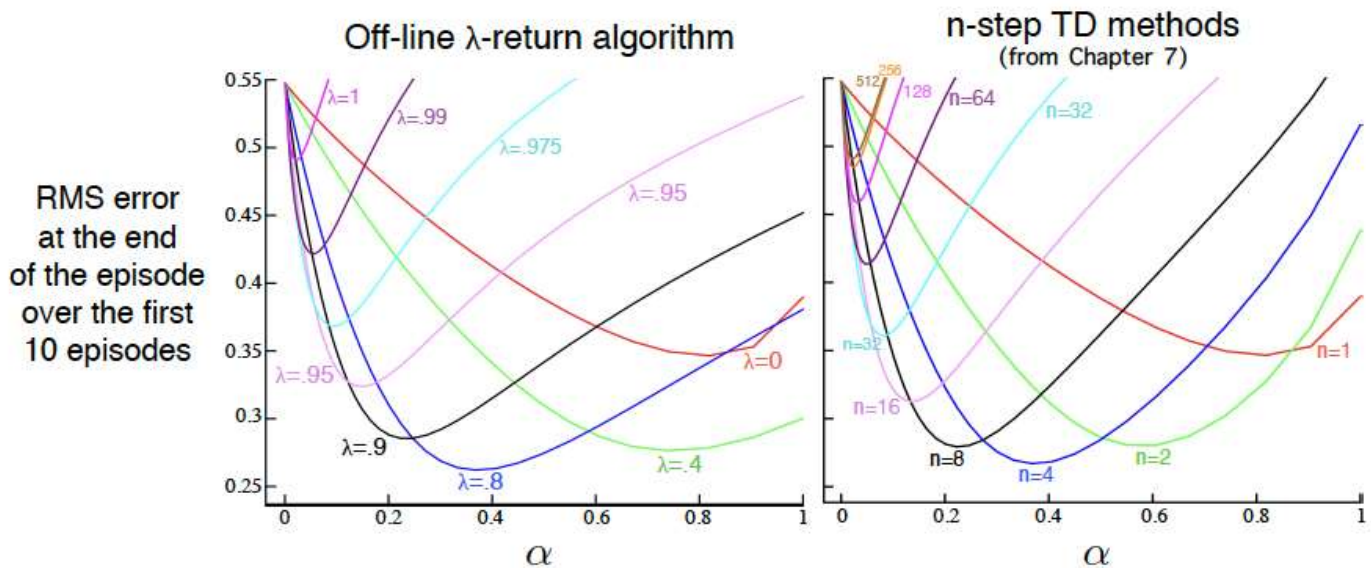


Figure 12.3: 19-state Random walk results (Example 7.1): Performance of the offline λ-return algorithm alongside that of the n-step TD methods. In both case, intermediate values of the bootstrapping parameter (λ or n) performed best. The results with the offline λ-return algorithm are slightly better at the best values of α and λ, and at high α.

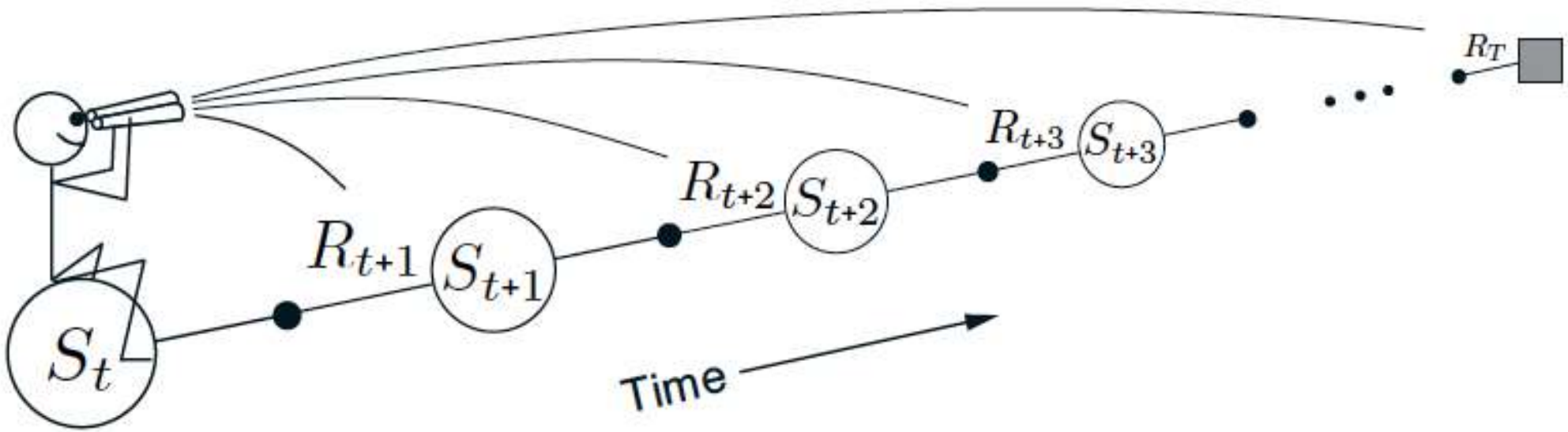


Figure 12.4: The forward view. We decide how to update each state by looking forward to future rewards and states.

TD($\boldsymbol{\lambda}$)

- TD($\boldsymbol{\lambda}$) improves over the offline "-return algorithm in three ways.
- First it updates the weight vector on every step of an episode rather than only at the end, and thus its estimates may be better sooner.
- Second, its computations are equally distributed in time rather than all at the end of the episode.
- And third, it can be applied to continuing problems rather than just to episodic problems. In this section we present the semi-gradient version of TD($\boldsymbol{\lambda}$) with function approximation.
- With function approximation, the eligibility trace is a vector z_t with the same number of components as the weight vector w_t.
- Whereas the weight vector is a long-term memory, accumulating over the lifetime of the system, the eligibility trace is a short-term memory, typically lasting less time than the length of an episode.
- Eligibility traces assist in the learning process; their only consequence is that they affect the weight vector, and then the weight vector determines the estimated value.

In TD(λ), the eligibility trace vector is initialized to zero at the beginning of the episode, is incremented on each time step by the value gradient, and then fades away by $\gamma\lambda$:

$$\begin{aligned}\mathbf{z}_{-1} &\doteq \mathbf{0},\\ \mathbf{z}_t &\doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t,\mathbf{w}_t), \quad 0 \le t \le T,\end{aligned} \tag{12.5}$$

The trace is said to indicate the eligibility of each component of the weight vector for undergoing learning changes should a reinforcing event occur. The reinforcing events we are concerned with are the moment-by-moment one-step TD errors. The TD error for state-value prediction is

$$\delta_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t). \tag{12.6}$$

In TD(λ), the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t. \tag{12.7}$$

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

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize S
    $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
    Loop for each step of episode:
    | Choose $A \sim \pi(\cdot|S)$
    | Take action A, observe R, S'
    | $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S,\mathbf{w})$
    | $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
    | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
    | $S \leftarrow S'$
    until S' is terminal

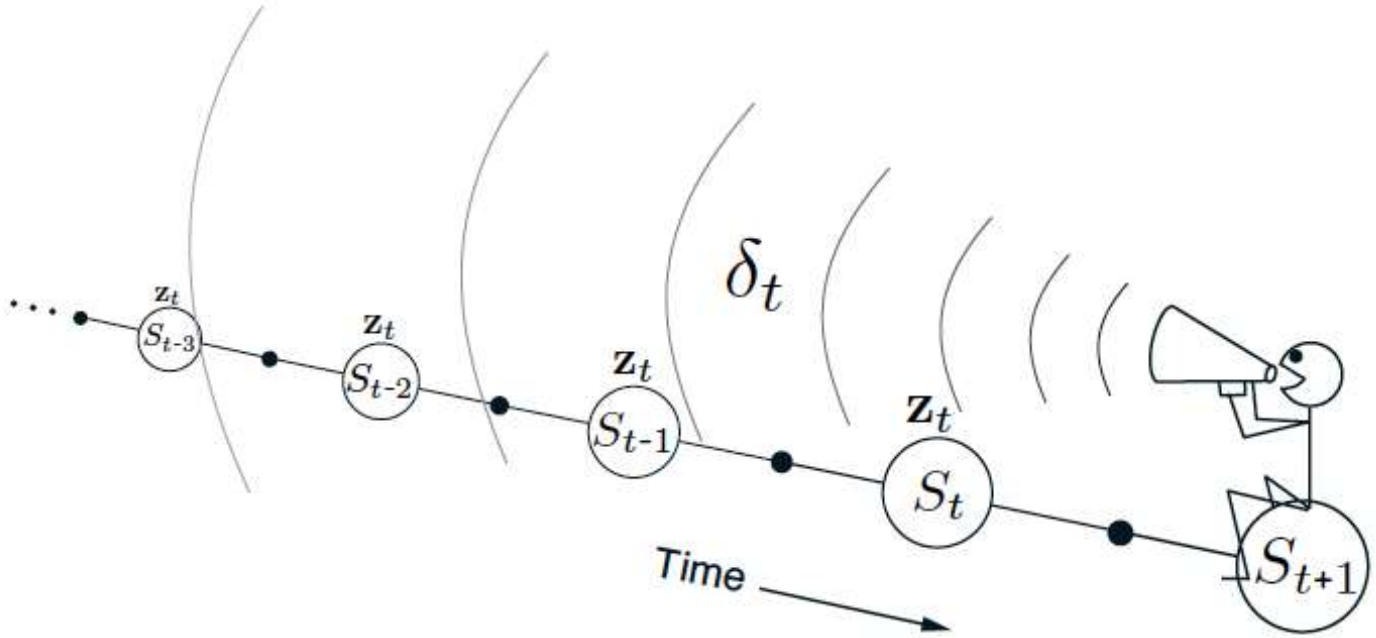


Figure 12.5: The backward or mechanistic view of TD(λ). Each update depends on the current TD error combined with the current eligibility traces of past events.

TD($\boldsymbol{\lambda}$) is oriented backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to how much that state contributed to the current eligibility trace at that time. We might imagine ourselves riding along the stream of states, computing TD errors, and shouting them back to the previously visited states,

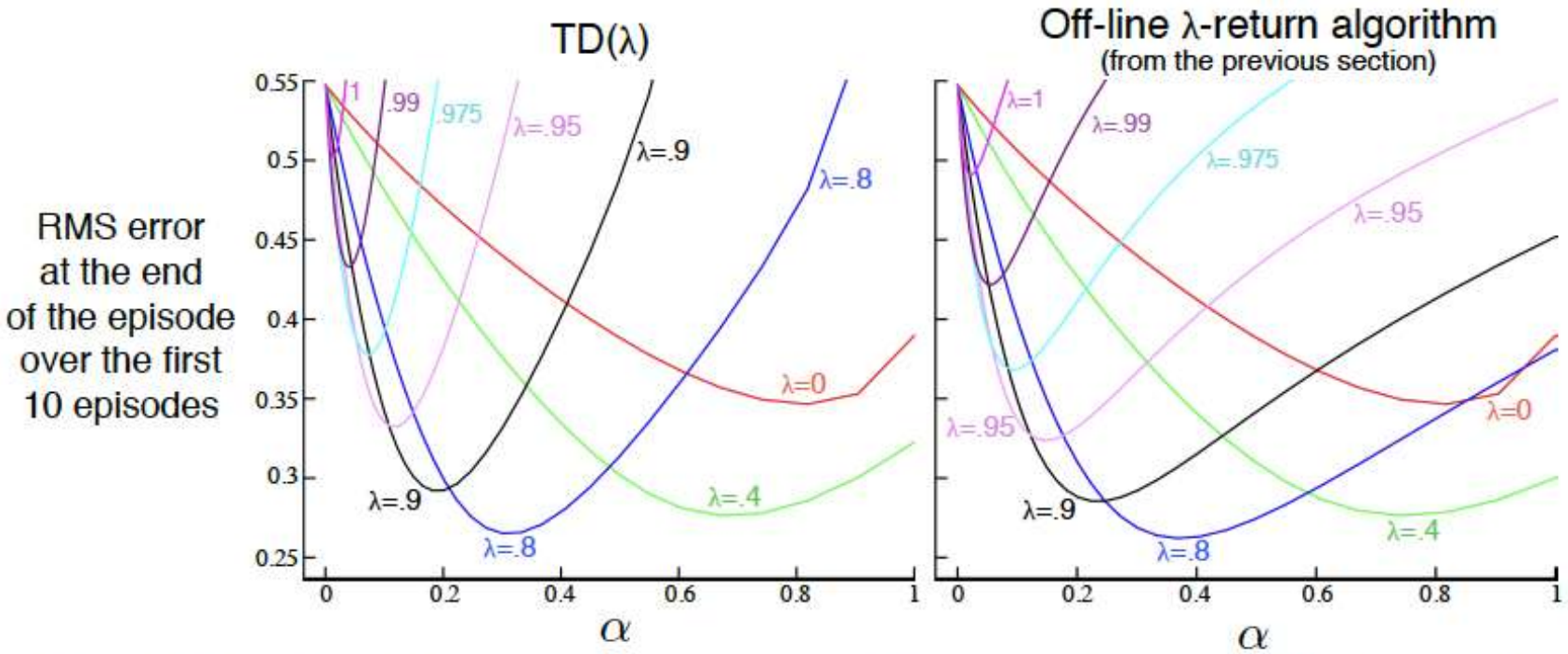


Figure 12.6: 19-state Random walk results (Example 7.1): Performance of TD(λ) alongside that of the offline λ-return algorithm. The two algorithms performed virtually identically at low (less than optimal) α values, but TD(λ) was worse at high α values.

$$\overline{\text{VE}}(\mathbf{w}_\infty) \quad \leq \quad \frac{1-\gamma\lambda}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \tag{12.8}$$

That is, the asymptotic error is no more than $\frac{1-\gamma\lambda}{1-\gamma}$ times the smallest possible error. As λ approaches 1, the bound approaches the minimum error (and it is loosest at $\lambda = 0$).

Unified View: Temporal-Difference Backup

$$V(s_t) \leftarrow V(s_t) + \alpha_T \left(R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right)$$

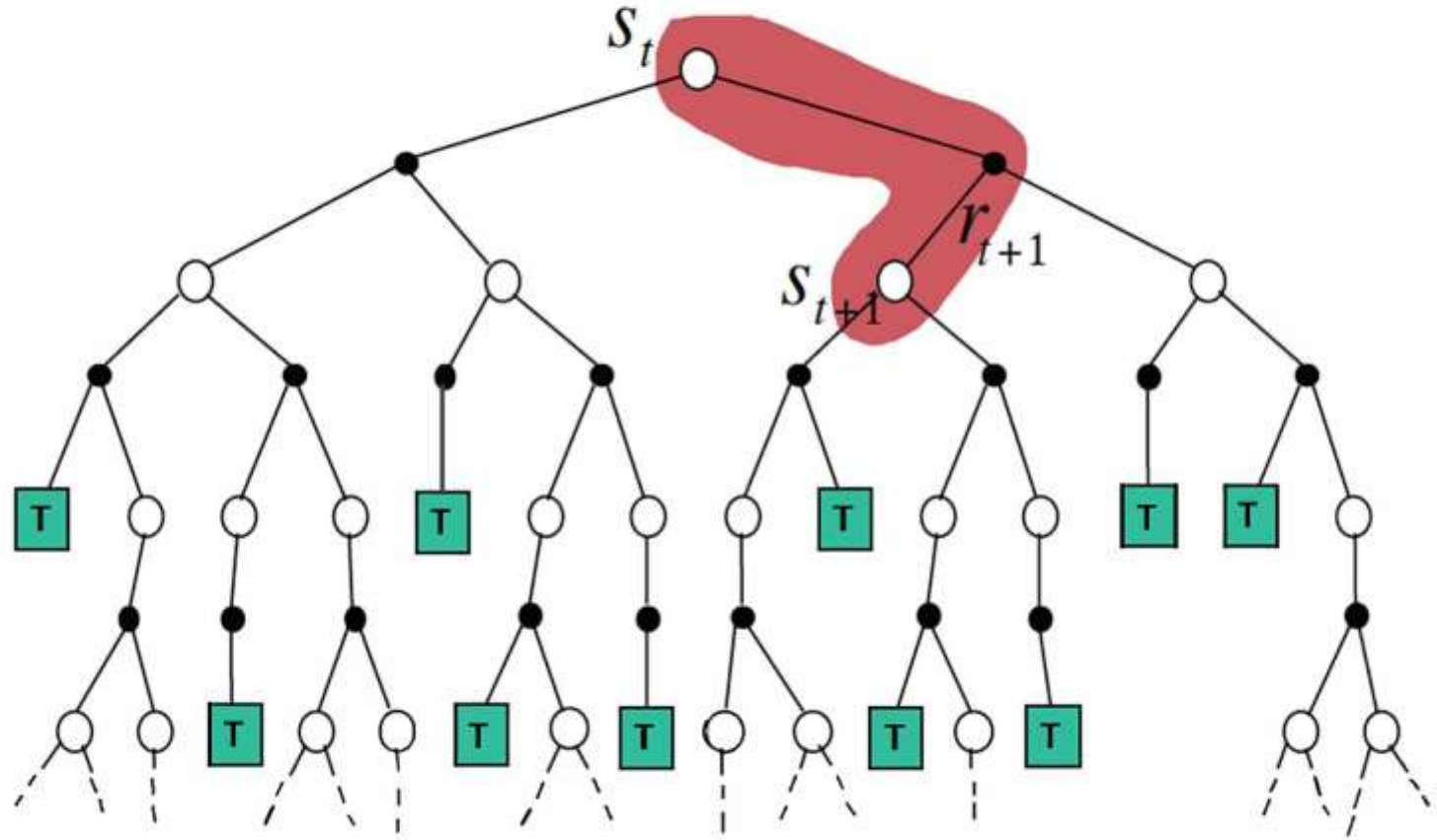


Figure credit: *David Silver, DeepMind*

- Use of 'sample backups' and 'bootstrapping'.

Unified View: Dynamic Programing Backup

$$v_\pi \doteq v^{(k+1)}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^{(k)}(s') \right\}$$

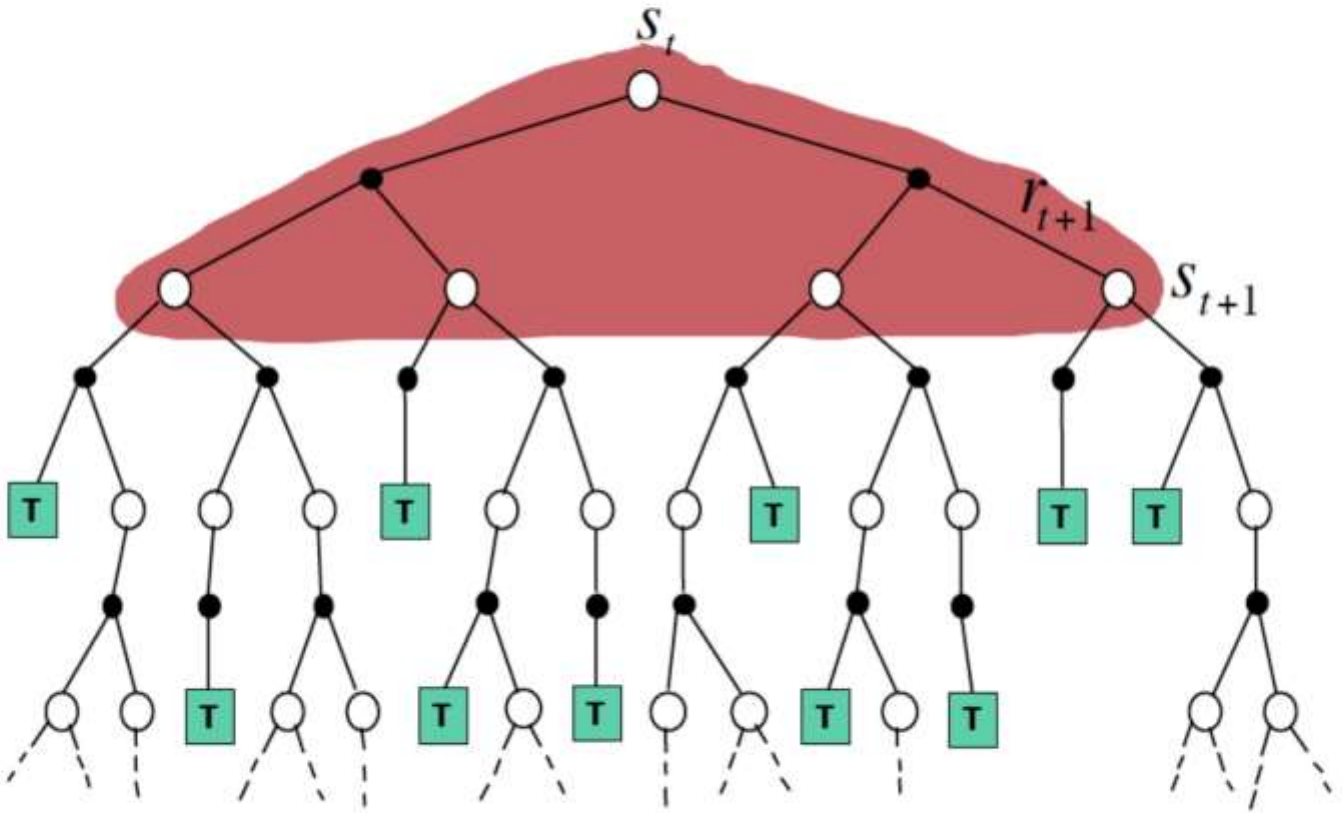


Figure credit: *David Silver, DeepMind*

- Use of 'full backups' and no 'bootstrapping'.

Unified View: Monte-Carlo Backup

$$V(s_t) \leftarrow V(s_t) + \alpha_T \left(G_t - V(s_t)\right)$$

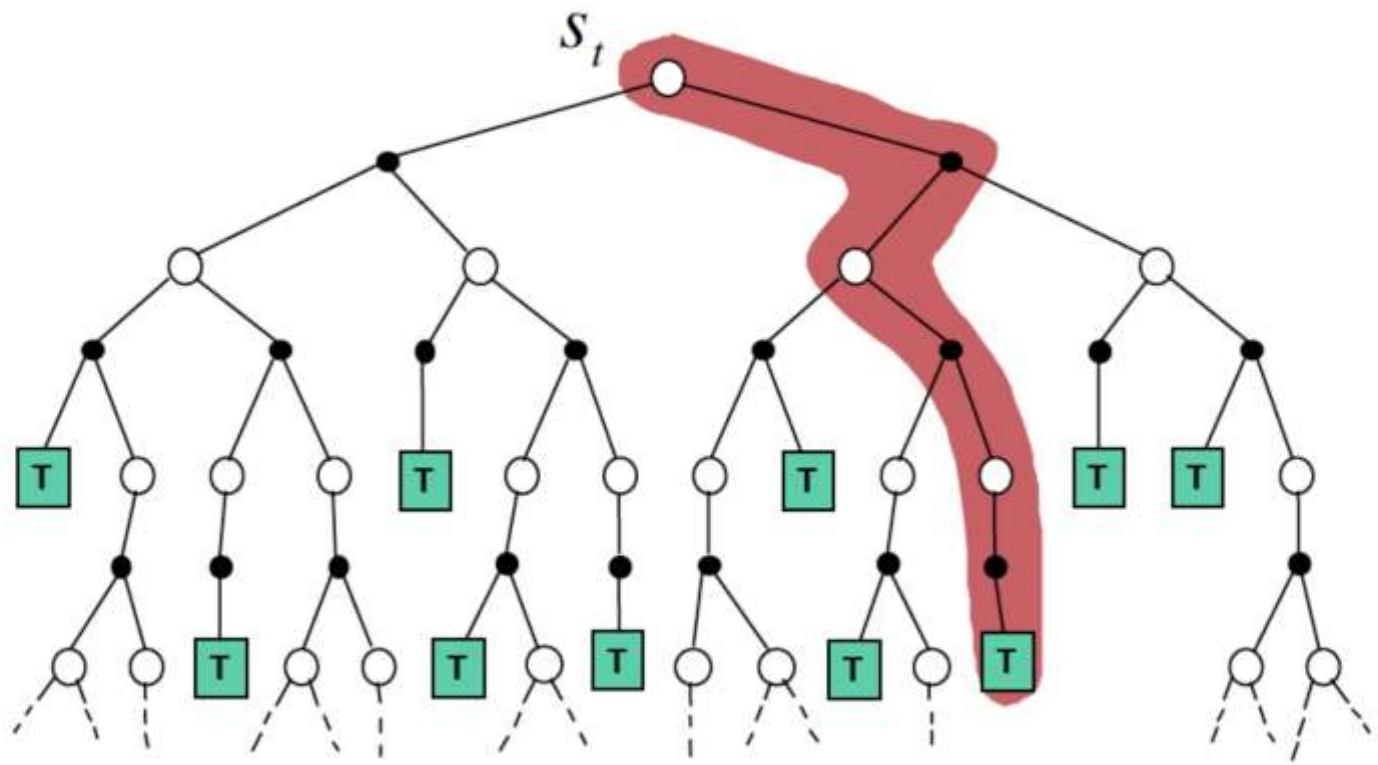


Figure credit: *David Silver, DeepMind*

- Use of 'sample backups' and no 'bootstrapping'.