[S&B Book] Chapter 6: Temporal-Difference Learning

Tags

- Temporal-Difference (TD) learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas
  - Like MC, TD can learn directly from raw experience without a model of the environment's dynamics;
  - Like DP, TD updates estimates based in part on other learned estimates without waiting for a final outcome (they bootstrap).

▼ 6.1 TD Prediction

- TD method:

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

  The simplest TD method makes the update immediately on the transition to S_{t+1} and receiving R_{t+1};

  This method is also called ***one-step*** **TD, or TD(0)**, because it is a special case of the TD(λ) and n-step TD methods.

- Algorithm:

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

- *sample updates* and *expected updates*
  - TD and MC are ***sample updates*** because they involve looking ahead to a sample successor state (or state-action pair), using the value of successor and the reward along the way to compute a back-up value, and then updating the value of the original state (or state-action pair) accordingly;
  - DP methods are ***expected updates*** because they are based on complete distribution of all possible successors;
- **TD error:**

  The difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$; the TD error arises in various forms throughout reinforcement learning;

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

- If the array V does not change during the episode (as it does not change in Monte Carlo methods), then the Monte Carlo error can be written as a sum of TD errors:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S) + \gamma V(S_{t+1}) + -\gamma V(S_{t+1}) \\ &= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \\ &= \delta_t + \gamma\delta_{t+1} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(G_T - V(S_T)) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k \end{aligned}$$

Exercise 6.1:

If V changes during the episode, then (6.6) only holds approximately; what would the difference be between the two sides? Let Vt denote the array of state values used at time t in the TD error (6.5) and in the TD update (6.2). Redo the derivation above to determine the additional amount that must be added to the sum of TD errors in order to equal the Monte Carlo error.

Solution:

$$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$$

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

$$\begin{aligned}
G_t - V_t(S_t) &= R_{t+1} + \gamma G_{t+1} - V_t(S_t) + \gamma V_t(S_{t+1}) - \gamma V_t(S_{t+1}) \\
&= \delta_t + \gamma \left(G_{t+1} - V_t(S_{t+1})\right) \\
&= \delta_t + \gamma(G_{t+1} - V_{t+1}(S_{t+1}) - \alpha[R_t + \gamma V_t(S_{t+1}) - V_t(S_t)]) \\
&= \delta_t + \gamma(G_{t+1} - V_{t+1}(S_{t+1})) + \gamma \Delta V_{t+1}(S_t) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - V_{t+2}(S_{t+2})) + \gamma \Delta V_{t+1}(S_t) + \gamma^2 \Delta V_{t+2}(S_{t+1}) \\
&= \cdots \\
&= \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k + \gamma^{k-t+1} \Delta V_{k+1}(S_k)
\end{aligned}$$

▼ 6.2 Advantages of TD Prediction Methods

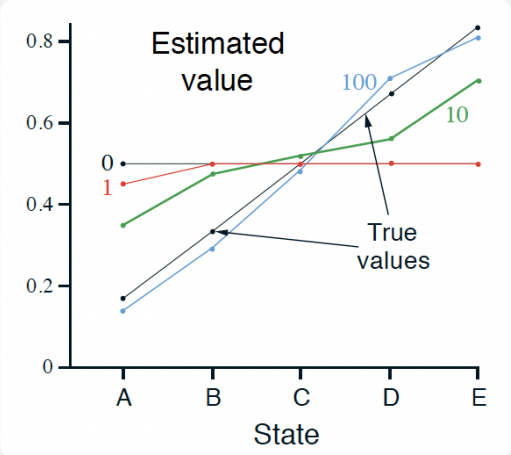
1. TD methods over DP methods: **TD methods do not require a model of the environment**, of its reward and next-state probability distributions.
2. TD methods over MC methods: **TD methods are naturally implemented in an online, fully incremental fashion.** The advantage becomes obvious when:
   - Some applications have very long episodes so that delaying all learning until the end of the episode is too slow.
   - Some applications are continuing tasks and have no episodes at all.
3. TD methods over MC methods: MC methods must ignore or discount episodes on which experimental actions are taken, which can generally slow learning. **TD methods learn from each transition regardless of what subsequent actions are taken.**

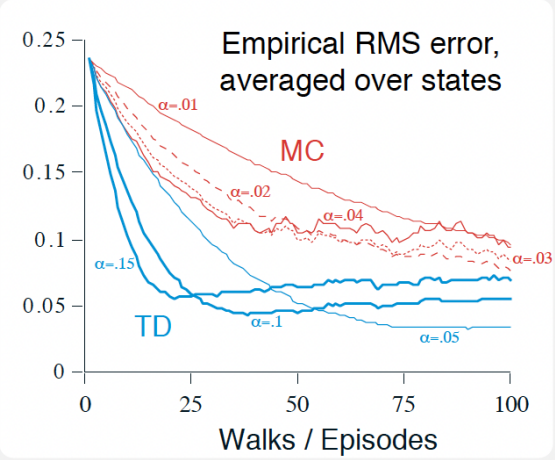
TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks as illustrated in the following example.

Example 6.2 Random Walk

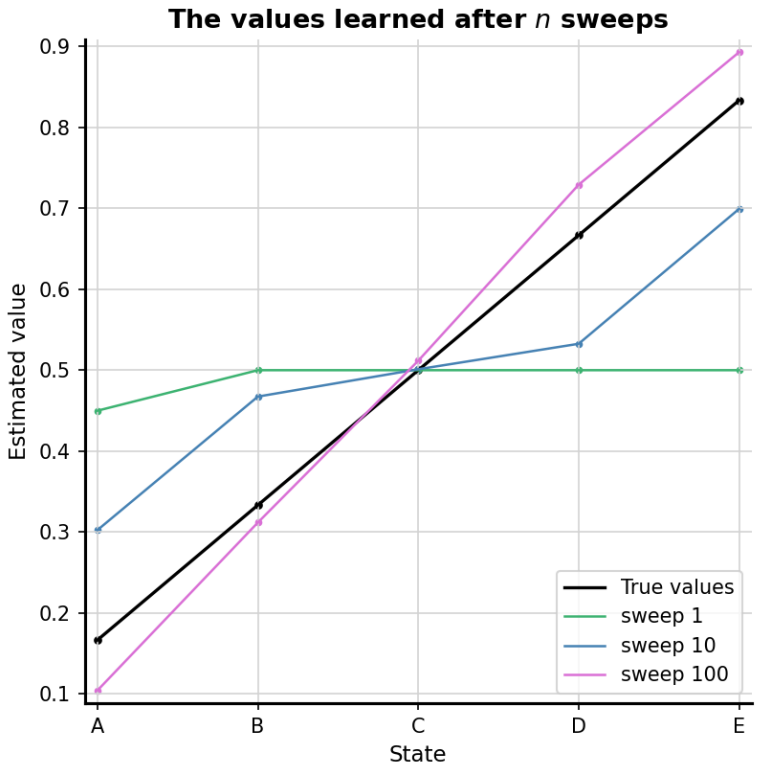
In this example we empirically compare the prediction abilities of TD(0) and constant-α MC when applied to the following Markov reward process:

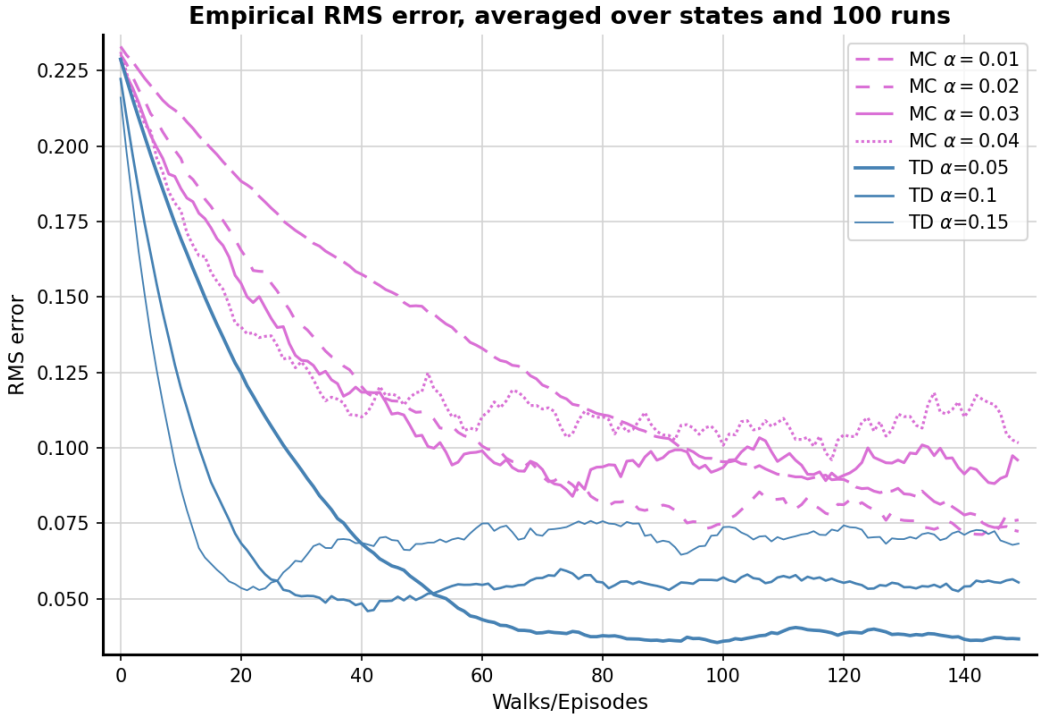
A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which there is no need to distinguish the dynamics due to the environment from those due to the agent. In this MRP, all episodes start in the center state, 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 $v_\pi(\mathsf{C}) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$.





Implementation


The values learned after n sweeps
Estimated value
State
A
B
C
D
E
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
True values
sweep 1
sweep 10
sweep 100


Empirical RMS error, averaged over states and 100 runs
RMS error
Walks/Episodes
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0.225
0
20
40
60
80
100
120
140
MC $\alpha = 0.01$
MC $\alpha = 0.02$
MC $\alpha = 0.03$
MC $\alpha = 0.04$
TD α=0.05
TD α=0.1
TD α=0.15

https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_06_temporal_difference_learning/example_6_2_random_walk.py

▼ 6.3 Optimality of TD(0)

- **Batch updating**

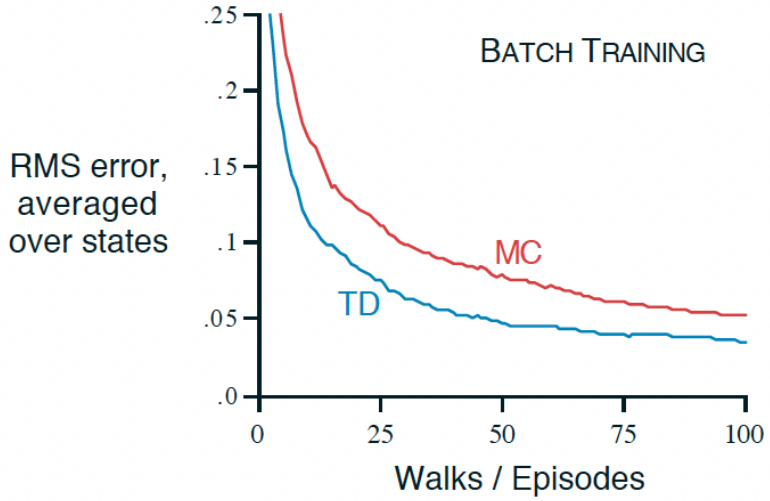
> Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function, V , the increments specified by (6.1) or (6.2) are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges.

  Updates are made only after processing each complete *batch* of training data

- With a sufficiently small α, TD(0) with batch updating converges deterministically to a single answer. The same rule applies to the constant- α MC method, but to a different answer.

Example 6.3:

Batch-updating versions of TD(0) and constant- α MC were applied as follows to the random walk prediction example. After each new episode, all episodes seen so far were treated as a batch. They were repeatedly presented to the algorithm, either TD(0) or constant- α MC, with α sufficiently small that the value function converged.



Implementation:

https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_06_temporal_difference_learning/example_6_3_batch_updating.py

- Batch MC find the estimates that minimize the mean-squared error on the training set, whereas batch TD(0) always find the estimates that would be exactly correct for the maximum-likelihood model of the Markov process;
- *Maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest;
- *Certainty-equivalence estimate* - MLE is the model of the Markov process formed in the obvious way from the observed episodes. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. **TD(0) converges to the certainty-equivalence estimate.**

▼ 6.4 Sarsa: On-policy TD Control

This method considers transitions from state-action pair to state-action pair, and learn the values of state-action pairs.

- **Sarsa:**

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right]$$

If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, which give rise to the name ***Sarsa*** for the algorithm.

- Convergence properties:

  Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ε-greedy policies by setting $\varepsilon = 1/t$).

- Algorithm:

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

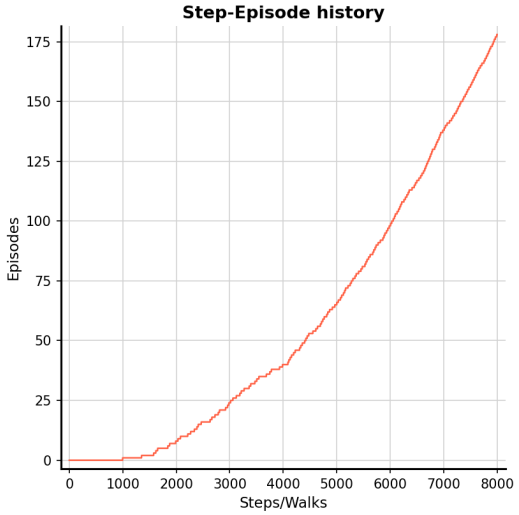
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

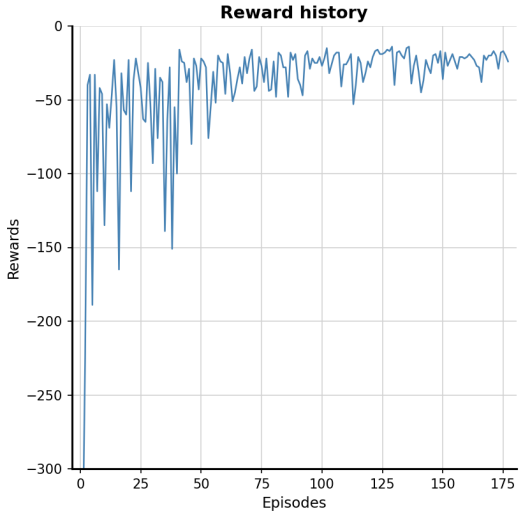
Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') -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

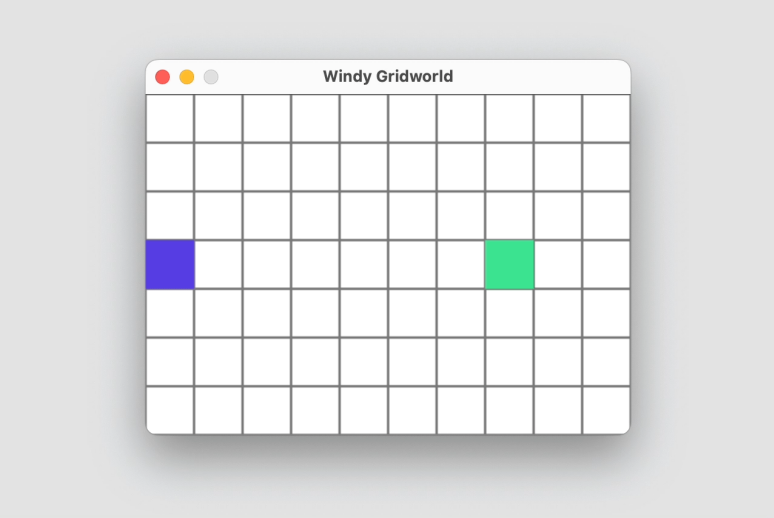
Example 6.5

A standard gridworld with start goal states, and a crosswind running upward through the middle of the grid. The actions are the standard four — *up, down, right, and left* — in the middle region the resultant next states are shifted upward by a "wind," the strength of which varies from column to column.

Implementation:



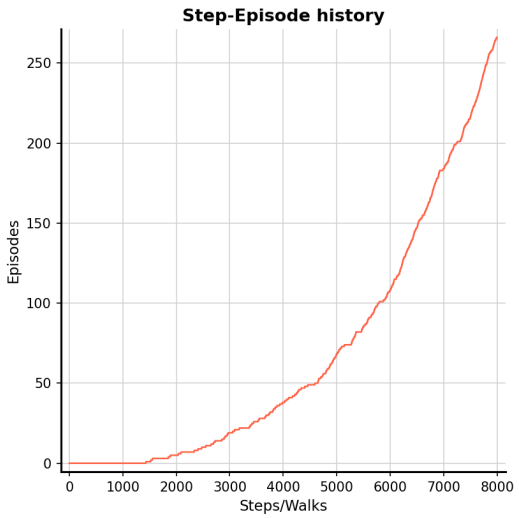


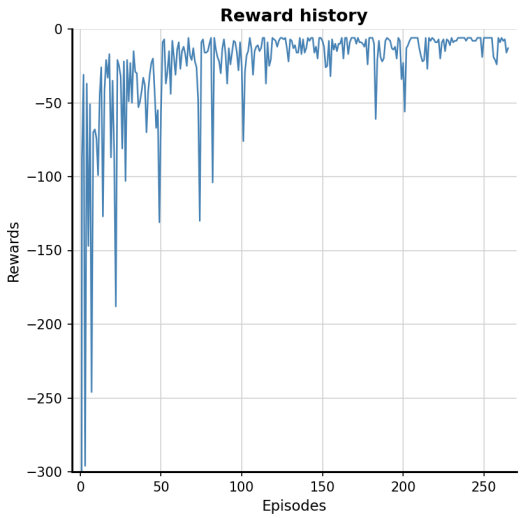


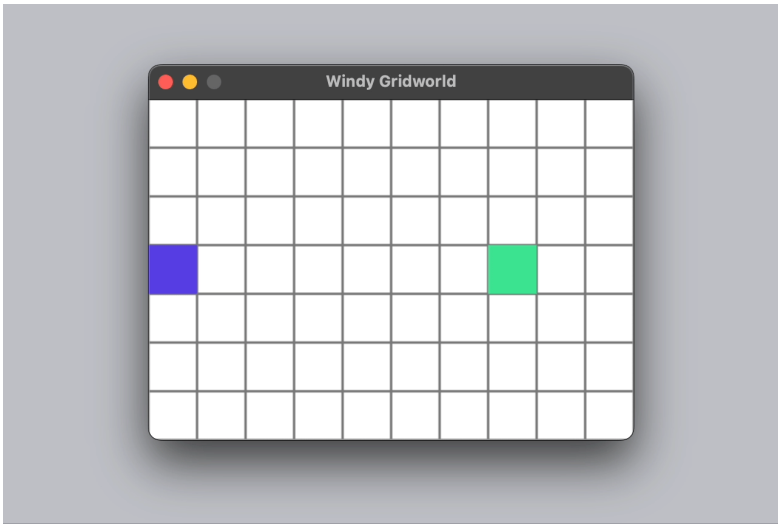
Exercise 6.9

Re-solve the windy gridworld assuming eight possible actions, including the diagonal moves, rather than four. Can also include the ninth action that causes no movement at all other than that caused by wind.

Solution:

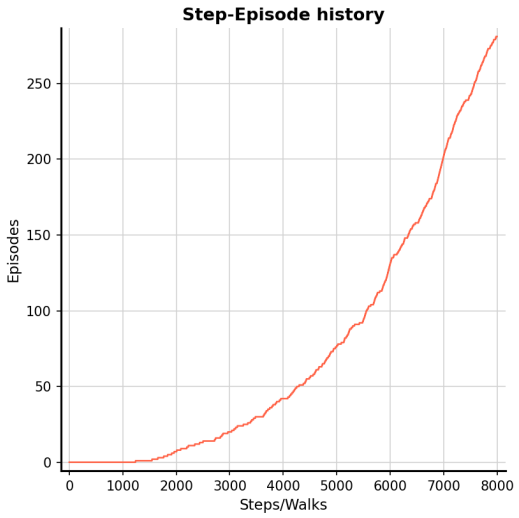


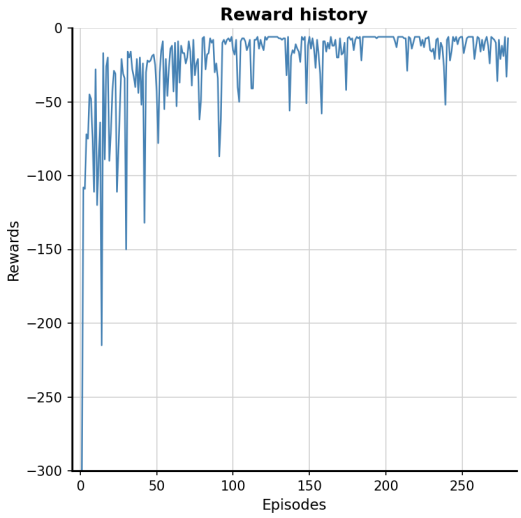


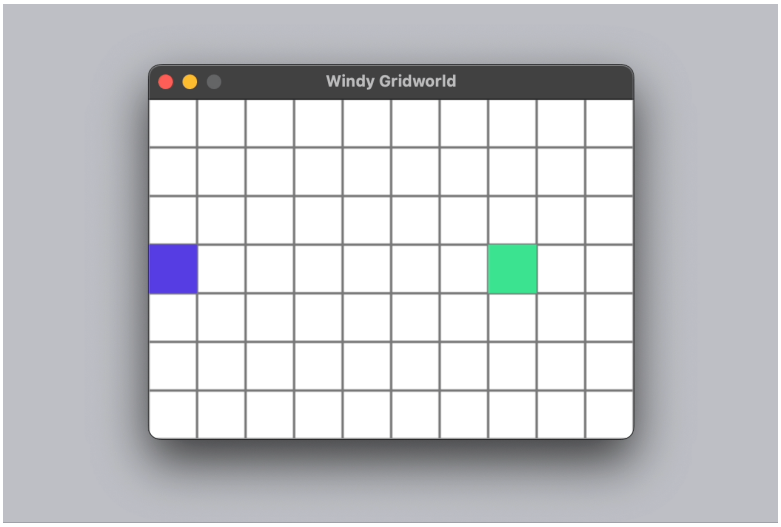


Exercise 6.10

Re-solve the windy gridworld with King's move, assuming that the effect of the wind, if there's any, is stochastic, sometimes varying by 1 from the mean values given for each column.







▼ 6.5 Q-Learning: Off-policy TD Control

- **Q-learning:**

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

  In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed.

- algorithm

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

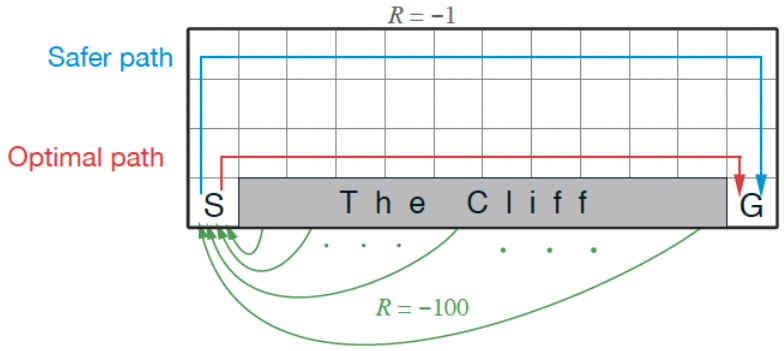
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

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'
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
    $S \leftarrow S'$
  until S is terminal

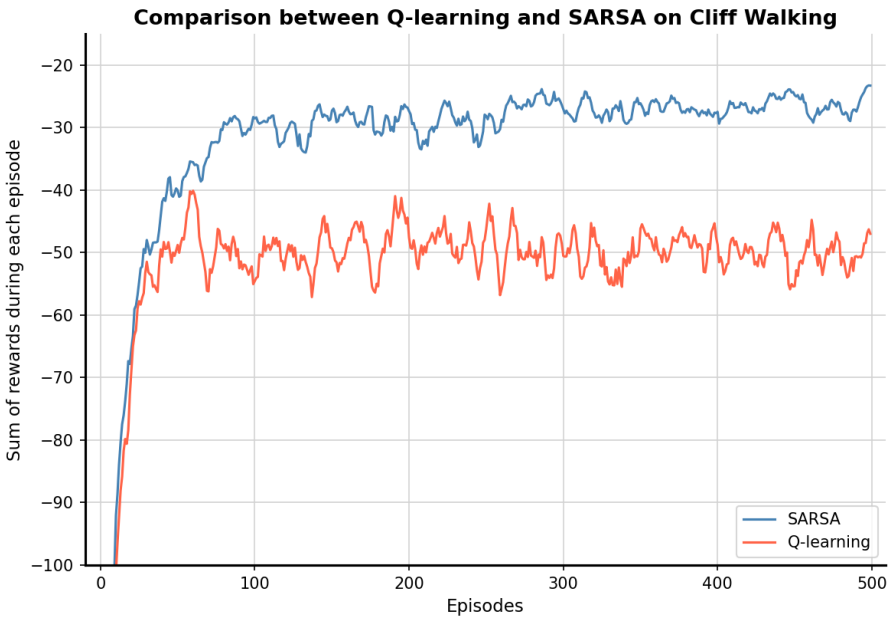
Example 6.6 Cliff Walking

This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. This is a standard undiscounted,

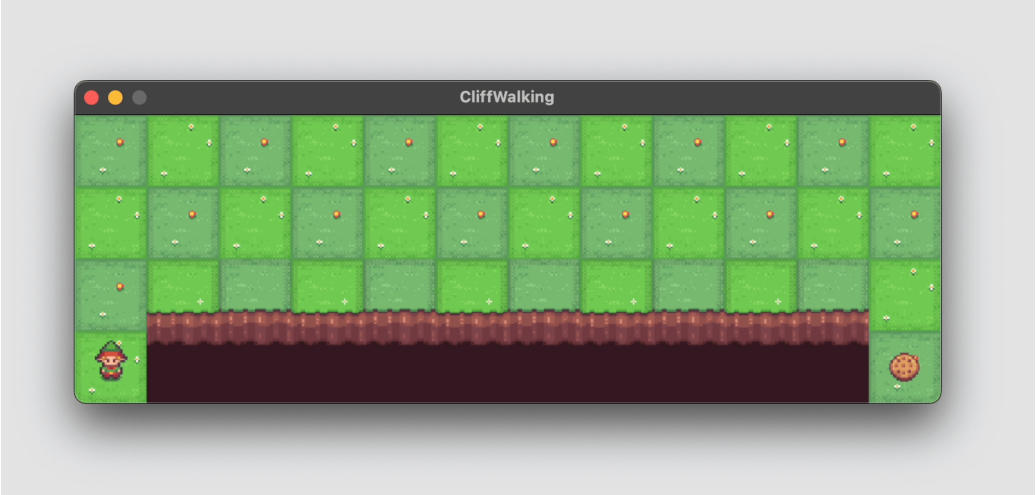
episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is −1 on all transitions except those into the region marked “The Cliff.” Stepping into this region incurs a reward of −100 and sends the agent instantly back to the start.



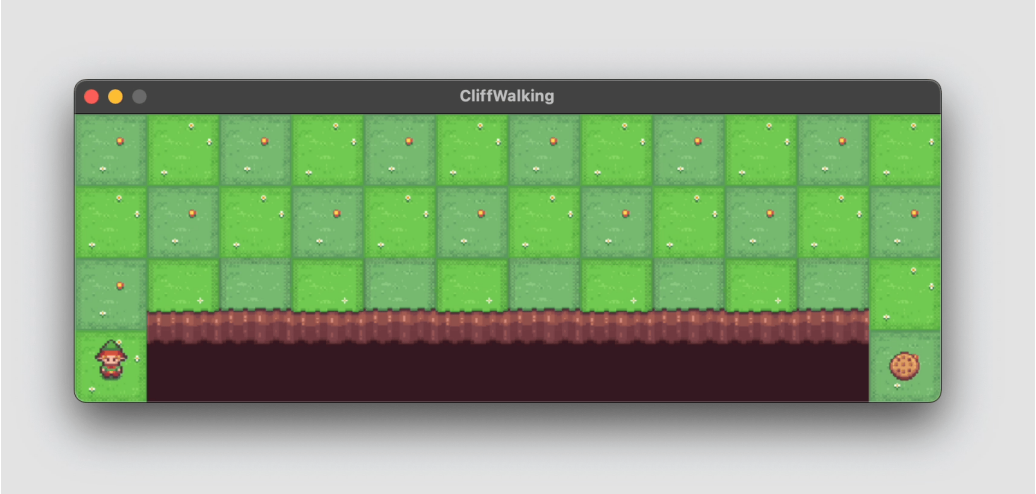
Implementation:



- Sarsa path:



- Q-learning path:



https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_06_temporal_difference_learning/example_6_6_clif_walking.py

▼ 6.6 Expected Sarsa

- **Expected Sarsa:**

  Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state-action pairs, it uses the expected value, taking into account how likely each action is under that current policy. That is, consider the algorithm with the update rule:

$$
\begin{aligned}
Q(S_t, A_t) &= Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E} \left[Q(S_{t+1}, A_{t+1} | S_{t+1}) \right] - Q(S_t, A_t) \right] \\
&= Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right]
\end{aligned}
$$

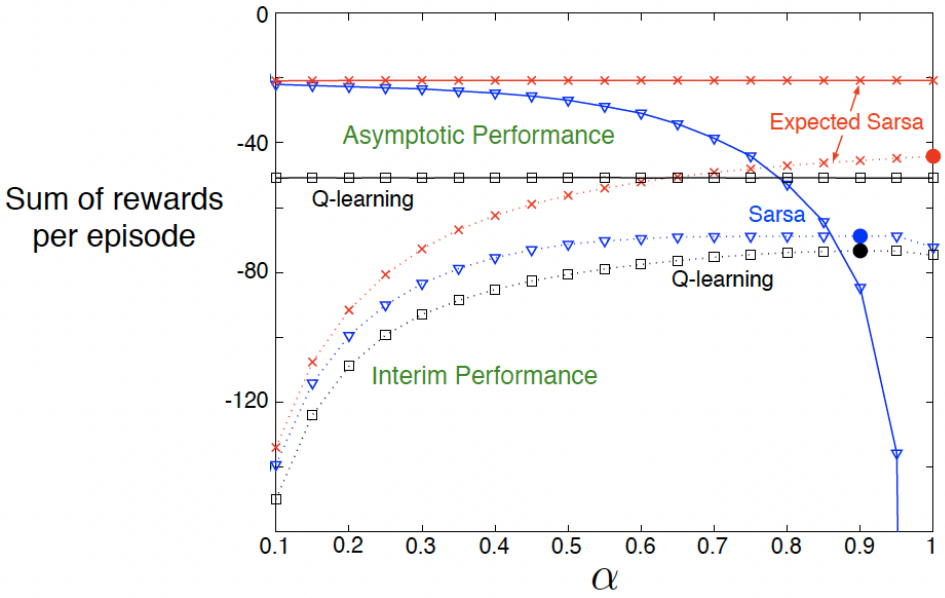
  Given the next state S_{t+1}, this algorithm move *deterministically* in the same direction as Sarsa moves *in expectation*, and accordingly it is called *Expected Sarsa*.

- Expected Sarsa eliminates the variance due to the random selection of A_{t+1}. It can safely set α = 1 without suffering any degradation of asymptotic performance, whereas Sarsa can only perform well in the long run at a small value of α.

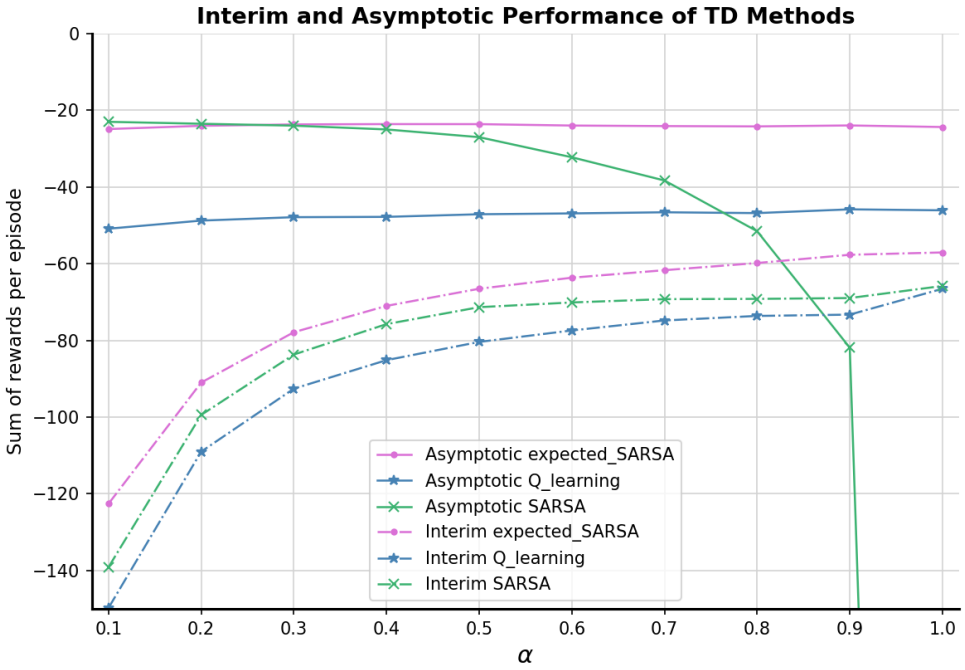
Figure 6.3 Performance of TD methods on Cliff Walking

Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α. All algorithms used an ε-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is

an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes.



Implementation:



https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_06_temporal_difference_learning/figure_6_3_TD_methods_performance.py

▼ 6.7 Maximization Bias and Double Learning

- **Maximization bias**: In Q-learning (greedy policy) and SARSA (ε-greedy policy), a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias.

Example 6.7 Maximization Bias Example

The small MDP shown inset in Figure 6.5 provides a simple example of how maximization bias can harm the performance of TD control algorithms.

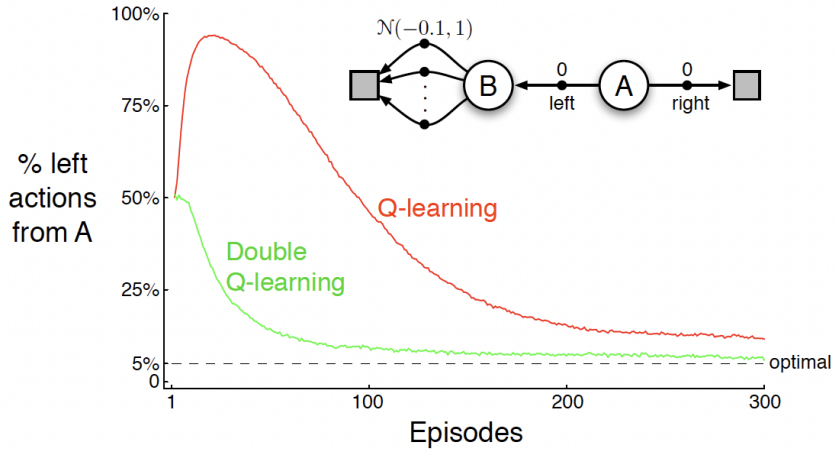
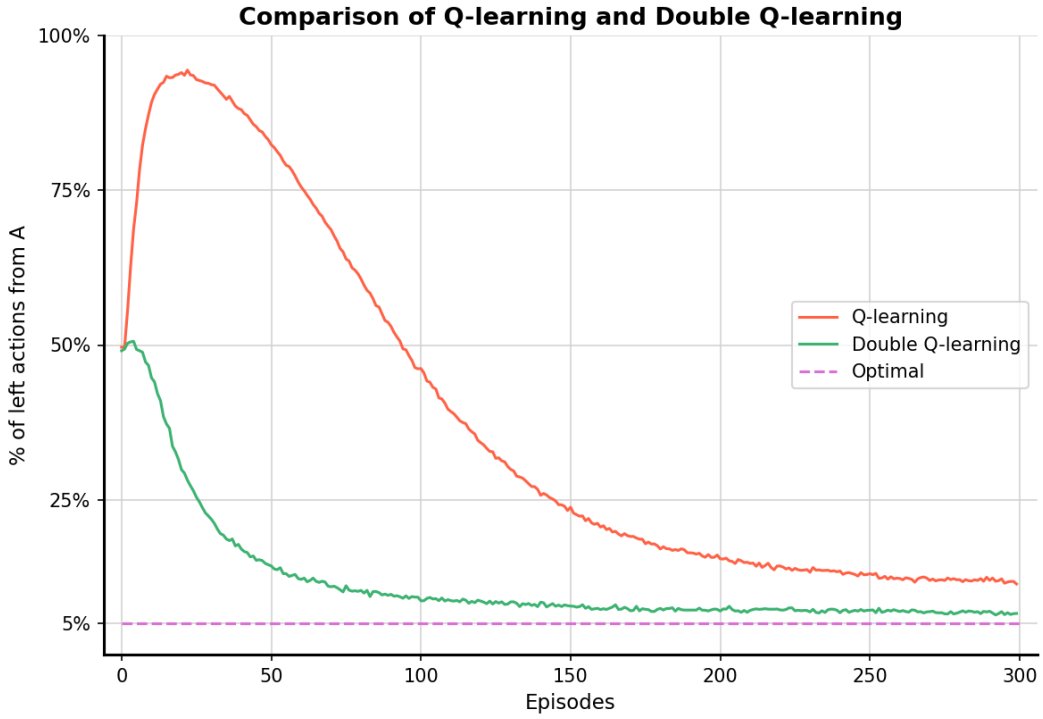


Figure 6.5: Comparison of Q-learning and Double Q-learning on a simple episodic MDP (shown inset). Q-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by ε-greedy action selection with $\varepsilon = 0.1$. In contrast, Double Q-learning is essentially unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. Any ties in ε-greedy action selection were broken randomly.

Implementation:



https://github.com/terrence-ou/Reinforcement-Learning-2nd-Edition-Notes-Codes/blob/main/chapter_06_temporal_difference_learning/example_6_7_max_bias.py

---

- **Double learning**

  The problem of maximization bias can be viewed that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value.

  Suppose we divided the plays in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$, for all $a \in \mathcal{A}$. We could then use one estimate, say Q_1, to determine the maximizing action $A^* = \arg\max_a Q_1(a)$, and the other, Q_2, to provide the estimate of its value, $Q_2(A^*) = Q_2(\arg\max_a Q_1(a))$. This estimate will them be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$.

- *Double Q-learning Algorithm:*

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\textit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

▼ 6.8 Games, Afterstates, and Other Special Cases

- *Afterstates* and *afterstate value functions*:

  A conventional state-value function evaluates states in which the agent has the option of selection an action, but the state-value function used in tic-tac-toe evaluates board positions *after* the agent has made its move. This is called *afterstates*, and value functions over these, *afterstate value function*.

> Afterstates arise in many tasks, not just games. For example, int queuing tasks there are actions such as assigning customers to servers, rejecting customers, or discarding information. In such cases the actions are in fact defined in terms of their immediate effects, which are completely known.