temporal-difference (TD) Learning is a combination of Monte Carlo (Monte-Carto Methods) and Dynamic Programming (DP) ideas. TD methods learn directly from raw experience without a model of the environment's dynamics. Additionally, TD updates estimates based in part on the other learned estimate, without waiting for the final outcome, *they bootstrap*. The relationship between DP, TD and Monte-Carlo methods is a recurring theme in the theory of Reinforcement Learning. Examples of combinations are, n-step algorithms, which provide a bridge from TD to Monte Carlo methods and TD(λ) which seamlessly unifies them.

TD Prediction

Using experience to solve the prediction problem, estimating the value function v_π for a given policy π. Both TD and MC methods update their estimate V of v_π for the non terminal states S_t occurring in that experience, given some experience following policy π.

Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A every-visit Monte Carlo method suitable for non-stationary environments updates as,

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

Where G_t is the actual return following time t, and α is a step-size parameter. The key difference here is this MC update rule needs to wait till the end of the episode only then will G_t be known. TD methods need to wait only until the next time step. As at time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$,

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

Comparing both rules, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called, $TD(0)$, or *one-step* TD, as this is a special case of $TD(\lambda)$ and n-step TD methods.

This $TD(0)$ prediction can be formalized as,

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

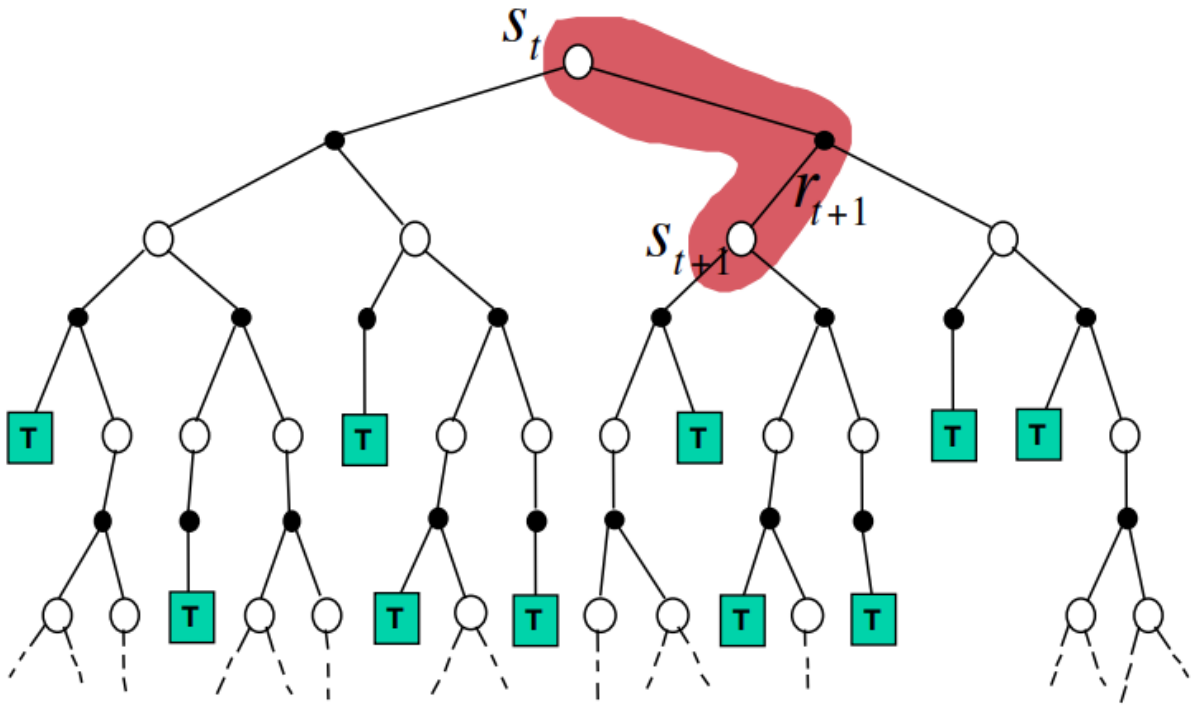
Because $TD(0)$ bases its update in part on an existing estimate, we say that it is a bootstrapping method, like DP.

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

This can be approximately translated as, Monte Carlo methods using the first line as an estimate of the target, whereas DP methods use the last line as a target.

- The Monte Carlo target is an *unbiased* estimate because the expected value in is not known.
- The DP target is an estimate not because of the expected values, which are not assumed to be completely provided by a model of environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$ is used instead. Hence, the TD target is an estimate for both reasons: it samples the expected values of the target and it uses the current estimate V instead of the true v_π. *Note, this TD target is an biased estimate of* $v_\pi(S_t)$, as Return G_t depends on *many* random actions, transitions and rewards. The TD target depends on *one* random action, transition and reward.

Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of DP. How this looks as a backup diagram is as follows,



We refer to TD and Monte Carlo updates as sample updates because they involve looking ahead to a sample successor state (or state–action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state–action pair) accordingly. **Sample updates** differ from the **expected updates** of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.

Finally, note that the zero 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, is called the *TD error*,

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

Notice that the TD error at each time is the error in the estimate *made at that time*. As, the TD error depends on the next state and next reward, it is not actually available until one time step later. If V does not change during the episode we can write the Monte Carlo error as the TD errors:

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

Where the first line follows from the definition of the discounted return $G_t = R_{t+1} + \gamma G_{t+1}$.

Advantages of TD Prediction Methods

Each error is proportional to the change over time of the prediction, that is, to the *temporal differences* in predictions (think of the example 6.1 in Barto & Sutton). TD methods update their estimates based in part on other estimates. They learn a guess from a guess—they *bootstrap*.

We will compare the advantages of TD methods against MC methods,

- TD can learn before knowing the final outcome
    - TD can learn online after every step
    - MC must wait until end of episode before return is known
- TD can learn without the final outcome
    - TD can learn from incomplete sequences
    - MC can only learn from complete sequences
    - TD works in continuing (non-terminating) environments
    - MC only works for episodic (terminating) environments

Furthermore,

- MC has high variance, zero bias
    - Good convergence properties (even with function approximation)
    - Not very sensitive to initial value
    - Very simple to understand and use
- TD has low variance, some bias
    - Usually more efficient than MC
    - $TD(0)$ converges to $v_\pi(s)$ (but, not always with function approximation)
    - More sensitive to initial value

Important to add is that convergence for TD methods has been proven. For any fixed policy π, $TD(0)$ has sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions.

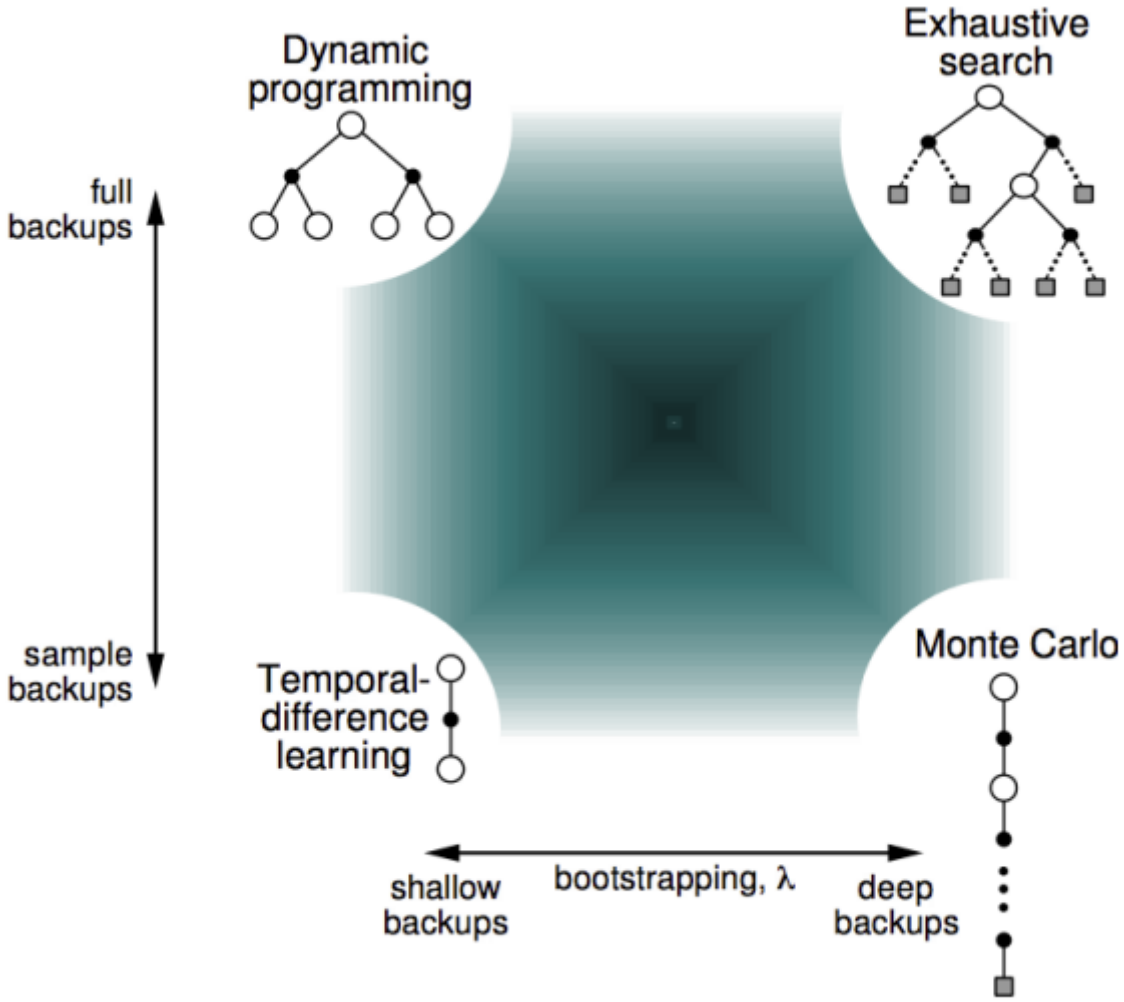
Optimality of $TD(0)$

If both TD and Monte Carlo methods converge asymptotically to the correct predictions, then a natural next question is “Which gets there first?” In other words, which method learns faster? Which makes the more efficient use of limited data? At the current time this is an **open question** in the sense that no one has been able to prove mathematically that one method converges faster than the other. In fact, it is not even clear what is the most appropriate formal way to phrase this question! In practice, however, TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks.

a general difference between the estimates found by batch $TD(0)$ and batch Monte Carlo methods. Batch Monte Carlo methods always find the estimates that minimize mean-squared error on the training set, whereas batch $TD(0)$ always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence* estimate because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch $TD(0)$ converges to the *certainty-equivalence* estimate.

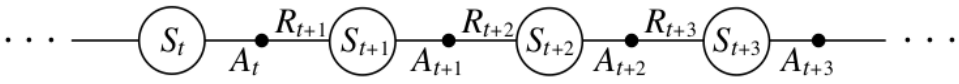
Finally, it is worth noting that although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly. If $n = |\mathcal{S}|$ is the number of states, then just forming the maximum-likelihood estimate of the process may require on the order of n^2 memory, and computing the corresponding value function requires on the order of n^3 computational steps if done conventionally. In these terms it is indeed striking that TD methods can approximate the same solution using memory no more than order n and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution. *These last two paragraphs are directly taken from Barto & Sutton*.

In short, bootstrapping updates involve an estimate. for example, MC does not bootstrap, but DP and TD do bootstrap. The other option is, update samples an expectation, DP does not sample, but MC and TD do use samples. How these methods can be depicted is as follows,



Sarsa: On-policy TD Control

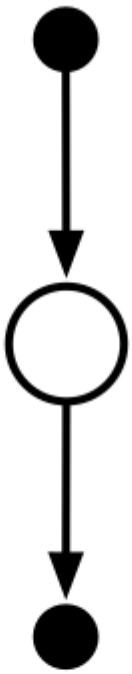
As we use an on-policy mehod we must estimate the action-value function, $q_\pi(s, a)$, for the current behavior policy π and for all states s and actions a. This can be done using essentially $TD(0)$ for learning v_π. An episode consists of an alternating sequence of states and state-action pairs:



Recall, that $TD(0)$ considers transitions from state to state and learned the values of states. Now we consider transitions from state-action pair to state-action pair, and learn the values of state-action pairs. The convergence of $TD(0)$ also applies to action values:

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

This update is done after every transition from a non-terminal state S_t. 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, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition form one state-action pair to the next. Hence, the *Sarsa* algorithm, the backup diagram looks as follows,



Sarsa

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q. For example, one could use ϵ-greedy or ϵ-soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all states-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (e.g. for ϵ-greedy policies we get $\epsilon = 1/t$).

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

Q-learning: Off-policy TD Control

The development of an off-policy TD control algorithm known as *Q-learning* is defined by,

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

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

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

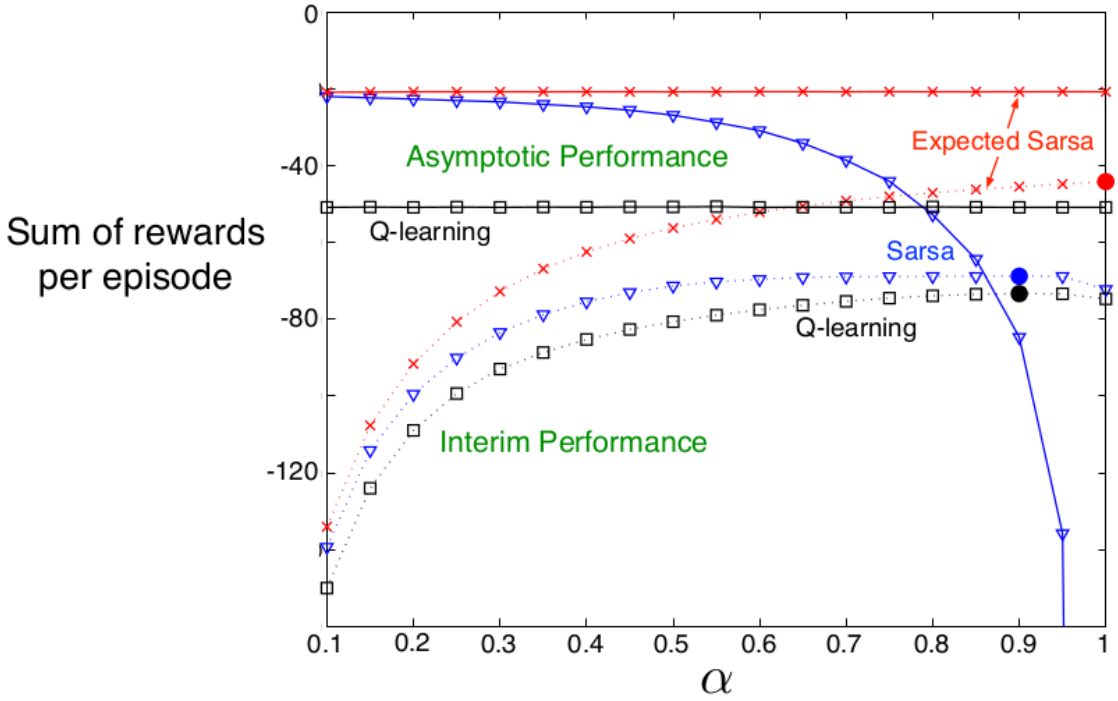
The reason that Q-learning is off-policy is that it updates its Q-values using the Q-value of the next state s and the greedy action a. In other words, it estimates the return for state-action pairs assuming a greedy policy were followed despite the fact that, actually an ϵ-greedy policy is followed. The reason that Sarsa is on-policy is that it updates it's Q-value of the next state s and the current policy's action a. It estimates the return for state-action pairs assuming the current policy continuous to be followed.

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 the current policy. The update rule that is considered,

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

Given the next state, S_{t+1}, this algorithm moves deterministically in the same direction as Sarsa moves in expectation, and accordingly it is called Expected Sarsa. Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_{t+1}. Given the same amount of experience we might expect it to perform slightly better than Sarsa, and generally does so. An comparison made on a simple environment (cliff walker) shows this,



Expected Sarsa is exactly Q-learning. In this sense Expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa. Except for the small additional computational cost, Expected Sarsa may completely dominate both of the other more-well-known TD control algorithms.

Maximization Bias and Double Learning

All the control algorithms that we have discussed so far involve maximization in the construction of their target policies. For example, in Q-learning the target policy is the greedy policy given the current action values, which is defined with a max, and Sarsa the policy is often ϵ-greedy, which also involves a maximization operation. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias.

To show this bias that is introduced, consider a single state s where there are many actions a whose true values, $q(s,a)$, are all zero but whose estimated values, $Q(s,a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive, a positive bias. We call this **maximization bias**.

How to avoid this positive maximization bias, through using the maximization of the estimates as an estimate of the maximum of the true values. Example, consider learning two independent estimates, $Q_1(a)$ 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 then be unbiased in the sense

that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(\arg\max_a Q_2(a))$. This is the idea of *double learning*. Note that although we learn two estimates, only one estimate is is updated on each play; double learning doubles the memory requirement, but does not increase the amount of computation per step.

We can extend this idea naturally to Q-learning, namely, *Double Q-learning* which divides the time steps in two, perhaps by flipping a coin on each step to switch up the estimators. Then, the update is,

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

Double learning seems to eliminate the harm caused by maximization bias. Additionally, this idea of double learning can be extended to Sarsa and Expected Sarsa.

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 $Q(terminal, \cdot) = 0$

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:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

References

- Barto & Sutton, Reinforcement Learning: An introduction 2nd edition
- David Silver's Course on Reinforcement Learning