Temporal-Difference Learning: Reinforcement Learning

Sanyam Singhal

March 27, 2022

1 Summary

1. TD learning is a combination of Monte carlo methods and dynamic programming. It uses sampled experience like Monte Carlo, and bootstraps like dynamic programming
2. TD Prediction: Monte Carlo methods wait until the return following the visit to state s_t is known and then use that return to update $v(s_t)$. Unlike MC method, TD learning waits only till the next time step to update $v(s_t)$. This method is called one-step TD or TD(0).
3. Since, it uses the existing estimate for $v(s_{t+1})$ to update $v(s_t)$, it is a bootstrapping method. The algorithm is as follows:

   Tabular TD(0) for estimating v_π

   (a) **Input:** The policy π to be evaluated.

   (b) **Algorithm parameter:** Step size $\alpha \in (0, 1]$.

   (c) **Initialize:** $v(s)$ for all states $s \in S^+$ arbitrarily except that $v(terminal) = 0$.

   (d) **Loop for each episode:**
   Initialize S (state)
   Loop for each step of episode:
   $A \leftarrow$action taken by π for state S.
   Take action A, observe R, S'
   $v(S) \leftarrow v(S) + \alpha[R + \gamma v(S') - v(S)]$
   $S \leftarrow S'$
   until S is terminal

---

4. The quantity $R + \gamma v(S_{t+1}) - v(S_t)$ is called TD error and is denoted as δ_t.
5. For any fixed policy π, TD(0) has been proved to converge to v_π with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions seen in bandit problem discussion. In practise, TD learning has been found to converge faster than MC methods.
6. If we have access only to a finite T episodes, the estimate of TD(0) will depend on the initial estimate V^0. To forget V^0, we can run the T collected episodes over and over again, and make TD(0) updates. This estimate will converge to $V_{MLE\ \pi}$ where $MLE\ \pi$ is the MDP with policy π that has the highest likelihood of generating our T episodes. So, our result need not be the same as actual v_π. This method is called Batch TD(0).
7. Control with TD learning: Maintain and update action value function by following a policy π_t that explores all possible state-action pair (ϵ_t-greedy for example. So, the algorithm in general can be stated as:

   TD control for estimating q_*

(a) **Algorithm parameters:** Step size $\alpha \in (0, 1]$, small $\epsilon > 0$.
Discount factor $\gamma \in [0, 1)$.
Target scheme $T(\pi, q, r)$

(b) **Initialize:** $q(s, a)$ for states s and actions a arbitrarily except $q(terminal, \cdot) = 0$.

(c) Loop for each episode:
Initialize S
Choose A from s using policy derived from q.
Loop for each step of episode:
Take action A, observe R, S'
Choose A' from S' using policy derived from q
$q(s, a) \leftarrow q(s, a) + \alpha[T(\pi, q', a') - q(s, a)]$
$S \leftarrow S'; A \leftarrow A'$
until S is terminal.

---

8. Different target schemes in TD control are:
   (a) Q-learning: $Target = r_t + \gamma \max_a q_t(s_{t+1}, a)$.
   (b) Sarsa: $Target = r_t + \gamma q_t(s_{t+1}, a_{t+1})$
   (c) Expected Sarsa: $Target = r_t + \gamma \sum_a \pi_t(s_{t+1}, a) q_t(s_{t+1}, a_{t+1})$
9. Sarsa and expected sarsa are on-policy algorithms whereas the Q-learning algorithm is an off-policy control algorithm because we take greedy actions at every time-step wrt the approximation made and the method converges to q_* irrespective of the optimal policy governing the MDP. Although, Q-learning actually learns the value of the optimal policy, its online performance is still worse than that of Sarsa.
10. Expected Sarsa moves in the same direction as Sarsa in expectation. It removes the variance due to the random selection of A_{t+1}.

    In general, it can use a policy different from the target policy to generate the behavior. In that case, it becomes an off-policy algorithm.
11. Since most of our methods involve some sort of maximization of value estimations, we often have a positive bias called the maximization bias, that creeps into the estimates for finite iterations. Read example 6.7 from Sutton and Barto, 2e for illustration of the detrimental effects of the maximization bias.
12. One way to tackle maximization bias is to use one estimate q_1 for taking the maximizing action and the other estimate q_2 to store the estimate for q_*. So, $q_2(A^*) = q_2(arg\max_a q_1(a))$. This is called double learning. Only one estimate is updated in a particular sweep. An example of the double Q-learning algorithm is given as follows:

    Double Q-learning for estimating $q_1 = q_2 = q_*$

    (a) **Algorithm parameters:** Step size $\alpha \in (0, 1]$, small $\epsilon > 0$.
    Discount factor $\gamma \in [0, 1)$.
    Target scheme $T(\pi, q, r)$

    (b) **Initialize:** $q_1(s, a)$ and $q_2(s, a)$ for states s and actions a arbitrarily except $q_i(terminal, \cdot) = 0$ for $i = 1, 2$.

    (c) Loop for each episode:
    Initialize S. Loop for each step of episode:
    Choose A from s using policy ϵ-greedy in $q_1 + q_2$.
    Take action A, observe R, S'
    With 0.5 probability:

$q_1(s,a) \leftarrow q_1(s,a) + \alpha[R + \gamma q_2(S', arg\max_a q_1(S',a)) - q_1(s,a)]$

else:

$q_2(s,a) \leftarrow q_2(s,a) + \alpha[R + \gamma q_1(S', arg\max_a q_2(S',a)) - q_2(s,a)]$

$S \leftarrow S'; A \leftarrow A'$

until S is terminal.