1 Temporal-Difference Learning

We first focus on the prediction problem, that is, finding v_pi given a π. The control problem, finding π_*, is approached using the GPI framework.

1.1 TD Prediction

Connection between TD, MC & DP

Monte-Carlo methods wait until the end of an episode to update the values. A simple MC update suitable for non-stationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{1}$$

we will call this *constant-α MC*. Temporal difference learning (TD) increments the values at each timestep. The following is the TD(0) (or one-step TD) update which is made at $t+1$ (we will see TD(λ) in Chapter 12)

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{2}$$

The key difference is that MC uses G_t as the target whereas TD(0) uses $R_{t+1} + \gamma V(S_{t+1})$. TD uses an estimate in forming the target, hence is known as a *bootstrapping method*. Below is TD(0) in procedural form.

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

The core of the similarity between MC and TD is down to the following relationship

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t | S_t = s] && (3)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] && (4)\\ &= \mathbb{E}_\pi[R_{t+1} + \gamma V(S_{t+1}) | S_t = s] && (5) \end{aligned}$$

- MC uses an estimate of the first line, since it uses sample returns to approximate the expectation
- DP uses an estimate of the final line, because it approximates v_π by V
- TD does both, it samples the returns like MC and also uses the current value estimates in the target

TD Error

We can think of the TD(0) update as an error, measuring the difference between the estimated value for S_t and the better estimate of $R_{t+1} + \gamma V(S_{t+1})$. We define the *TD error*

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t), \tag{6}$$

now if the array V does not change within the episode we can show (by simple recursion) that the MC error can be written

$$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma_{k-t} \delta_k. \tag{7}$$

1.2 Advantages of TD Prediction Methods

- TD methods do not require a model of the environment
- TD methods are implements online, which can speed convergence vs. MC methods which must wait until the end of (potentially very long) episodes before learning. TD methods can be applied to continuing tasks for the same reason
- TD methods learn from all actions, whereas MC methods required that the tails of the episodes be greedy
- For any fixed policy π, TD(0) has been proved to converge to v_π, in the mean with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions
- It is an open question as to whether TD methods converge faster than constant-α MC methods in general, though this seems to be the case in practice

1.3 Optimality of TD(0)

Given a finite number of training steps or episodes, a common method for estimating V is to present the experience repeatedly until V converges. We call the following *batch updating*: given finite experience following a policy and an approximate value function V, calculate the increments for each t that is non-terminal and change V once by the sum of all the increments. Repeat until V converges.

Under batch updating, we can make some comments on the strengths of TD(0) relative to MC. In an online setting we can do no better than to guess that online TD is faster than constant-α MC because it is similar towards the batch updating solution.

- Under batch updating, MC methods always find estimates that minimize the mean-squared error on the training set.
- Under batch updating, TD methods always finds the estimate that would be exactly correct for the maximum-likelihood model of the Markov process. The MLE model is the one in which the estimates for the transition probabilities are the fraction of observed occurrences of each transition.
- We call the value function calculated from the MLE model the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process is exact. In general, batch TD(0) converges to the certainty equivalence estimate.

1.4 Sarsa: On-policy TD Control

We now use TD methods to attack the control problem. The *Sara* update is as follows

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{8}$$

This update is done after every transition from a non-terminal state S_t. If S_{t+1} is terminal then we set $Q(S_{t+1}, A_{t+1}) = 0$. Note that this rule uses the following elements $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ which gives rise to the name Sarsa. The theoroms regarding convergene of the state-value versions of this update apply here too.

We write an on-policy control algorithm using Sarsa in the box below, at each time step we move the policy towards the greedy policy with respect to the current action-value function. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state action pairs are visited infinitely often and the policy also converges to the greedy policy in the limit (e.g. maybe π is ε-greedy with $\varepsilon = \frac{1}{t}$).

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

1.5 Q-learning: Off-policy TD Control

The *Q-learning* update is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{9}$$

The learning function Q directly approximates q_*. All that is required for convergence is that all pairs continue to be updated. An algorithm for Q-learning is given in the box below.

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

1.6 Expected Sarsa

The update rule for *Expected Sarsa* is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)] \tag{10}$$

$$\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{11}$$

This algorithm moves deterministically in the same direction as Sarsa moves in *expectation*, hence the name. It is more computationally complex than Sarsa, but eliminates the variance due to random selection of A_{t+1}. Given the same amount of experiences, it generally performs slightly better than Sarsa.

1.7 Maximisation Bias and Double Learning

All the control algorithms we have discussed so far involve some sort of maximisation in the construction of their target policies. This introduces a positive bias to the value estimates because they form uncertain estimates of the true values. This is known as the *maximisation bias*. It is essentially down to the fact the the expectation of the max of a sample is $\geq$ the max of the expected values of the samples.

To solve this we introduce the idea of *double learning*, in which we learn two independent sets of value estimates Q_1 and Q_2, then at each time step we choose one of them at random and update it using the other as a target. This produces two unbiased estimates of the action-values (which could be averaged). Below we show an algorithm for *double Q-learning*.

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

1.8 Games, Afterstates, and other Special Cases

In this book we try to present a uniform approach to solving tasks, but sometimes more specific methods can do much better.

We introduce the idea of *afterstates*. Afterstates are relevant when the agent can deterministically change some aspect of the environment. In these cases, we are better to value the resulting state of the environment, after the agent has taken action and before any stochasticity, as this can reduce computation and speed convergence.

Take chess as an example. One should choose as states the board positions *after* the agent has taken a move, rather than before. This is because there are multiple states at t than can lead to the board position that the opponent sees at $t+1$ (assuming we move second) via deterministic actions of the agent.