6. Temporal-Difference Learning

Notes

- Temporal-difference learning is central and novel to RL.
- Temporal-difference (TD) methods is a combination of MC and DP ideas, where it can learn from raw experience without a model of the environment's dynamics, as well as update estimates based on other learned estimates without waiting for a final outcome (bootstrapping).
- TD Prediction:
  - Recall that MC methods wait until an episode is completed and returns are known to update the value $V(S_t)$. The *constant-α MC method* can be denoted by $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]]$.
  - TD methods do not need to wait till the end of an episode, but only needs to wait until the next time step to make an update: $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$. Here the target for the update is $R_{t+1} + \gamma V(S_{t+1})$ rather than G_t. This method is called the *TD(0)*, or one-step TD, where it is a special case of the TD(λ) and n-step TD methods. Implementation can be found in page 120.
  - MC methods is an estimate because the expected value is not known, only a sample, rather than real expected, return is used. DP is an estimate because $v_\pi(S_{t+1})$ is not known and the current estimate $V(S_{t+1})$ is used. TD uses both, where it samples the expected value through $R_{t+1} + \gamma V(S_{t+1})$ and it uses the current estimate of V instead. It uses sampling and bootstrapping.
  - TD and MC updates are called *sample updates*, as it uses the value of sample successor state and reward to compute and update its value. It is different from the *expected update* of DP methods because they are based off a single sample successor rather than on a complete known distribution of all possible successors.
  - The quantity $\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is known as the TD error, and it is not actually available until one time step later, where δ_t is only available at time step $t + 1$.
  - If V does not change in an episode, the Monte Carlo error can be written as a sum of TD errors. This identity plays an important role in TD learning.

$$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k$$

- Advantages of TD Prediction Methods:
  - Advantage over DP methods is that they do not need a model of the environment and next-state probability distribution.
  - Compared to MC methods, TD methods are implemented online and fully incremental, where it needs to only wait one step, not one episode, to update its value estimates. For MC methods, if the episodes are very long, or the episodes contain many experimental steps, the learning can be very slow. These issues do not plague TD methods.
  - TD methods are guaranteed to converge to the value for a certain fixed policy. TD methods have also been found to converge faster than constant-α MC methods on stochastic tasks.
- Optimality of TD(0)
  - Under *batch updating*, which is updates made only after processing a batch of training data, TD(0) and MC methods converge deterministically to two different optimal values, which are both independent of α, assuming that α is sufficiently small.
  - MC methods finds the value estimates which minimise the mean-squared error on the training set, whereas TD(0) finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process.
  - The *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest. The TD(0) converges to the *certainty-equivalence estimate*, where it assumes that the estimate of the underlying process was known with certainty rather than being approximated.
  - This can explain why TD methods converge more quickly than MC methods in batch form, where the TD(0) computes the exact certainty-equivalence estimate.
  - for $n = |\mathcal{S}|$ states, it takes n^2 memory to compute the maximum-likelihood estimate of the process and n^3 to compute the corresponding value function via conventionally (non-TD) methods. For TD methods, it can approximate the same solution in memory no more than n. Thus, TD methods perform very well in tasks with large state space.
- Sarsa: On-policy TD control
  - We learn the action-value, rather than state-value, where we estimate $q_\pi(s, a)$ for each state-action pair. The update is written as:

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

This update uses the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ which represents a transition from one state-action pair to the next, and is what gives *Sarsa* its name.

  - As with MC methods, Sarsa converges to the 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. Implementation found on page 130.

- Q-learning: Off-policy TD Control:
  - The algorithm is given by

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

    Here, the learned action-value function Q directly approximates the optimal action-value function q_*, independent of the policy being followed. Like before, to ensure convergence, we require that all state-action pairs continue to be visited/updated. Implementation can be found in page 131.

  - The backup diagram includes a filled action node at the top, follow by a hollow state node, and maximising (arc) over the state-action nodes branching out from the hollow state node.

- Expected Sarsa:
  - Instead of using the maximum over the following state-action pairs like Q-learning, Expected Sarsa uses the expected value, where:

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

  - Compared to Sarsa, Expected Sarsa eliminates the variance due to the random selection of the next action. It generally performs better than Sarsa, where it has consistent performance across all values of α, and doesn't suffer from the degradation of asymptotic performance.
  - Expected Sarsa can also be an off-policy algorithm, where we have a greedy target policy and a exploratory behaviour policy. Expected Sarsa can be said to be a generalisation of Q-learning.

- Maximisation Bias and Double Learning:
  - The TD control algorithms so far all involve maximization in their target policies, and this can lead to a positive bias, called the *maximization bias*. This occurs due to the

estimates often being noisy, and maximising the value estimates will always result in higher than expected value estimates in the short run.

- *Double learning* is the idea of learning two independent estimates $Q_1(a)$ and $Q_2(a)$, which are estimates of the true value $q_*(a)$, off two sets of the training data. Here, we would use one estimate, let's say $Q_1(a)$ to determine the maximising action $A^* = \arg\max_a Q_1(a)$, and the other to estimate $Q_2(A*) = Q_2(\arg\max_a Q_1(a))$, which is unbiased. The same will be done for the reversed case as well.
- For double Q-learning, the time steps are divided equally into two, and on one side, the update will be

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

  and Q_1 and Q_2 will be switched round for the other side. The two value functions are treated symmetrically, such that the behaviour policy can use both action-value estimates, where the policy can be based on the average of the two estimates. Note that the amount of computation is the same, but double the memory space is required. Implementation can be found in page 136.

- Games, Afterstates, and Other Special Cases
  - In games that involve an opponent, it is more efficient for us to consider the state after the agent has made its moves, which are known as *afterstates*. This is useful when we have knowledge of an initial part of the environment's dynamics but not necessarily the full dynamics. Take a game of chess for example, we know what actions each chess piece makes, but we do not know how our opponent is going to move.
  - Afterstates are more efficient because multiple state-action pairs can map to the same afterstate, which means that these different state-action pairs have the same value. Thus, estimate a single afterstate, rather than multiple state-action pairs, is more efficient.
- The methods here form the foundation of RL methods used nowadays. Modern methods are often just built on top of these methods, and are more complicated variants of them. The special case of TD methods in this Chapter are called one-step, tabular, model-free TD methods.
- TD methods are more general than solving reinforcement learning problems, and can be applied to many other fields, such as predicting financial data, animal behaviour, demands on a power station etc.

Exercises

6.1 Let us use V_t to denote the value estimate at time t, in order to keep track of the changes of V in an episode. Recall that TD(0) gives an update of $V_{t+1}(S_t) = V_t(S_t) + \alpha[R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)]$

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

6.2 In the scenario proposed in the book, a TD method here would be better because the highway part of the drive remains the same as the previous drive from the old office. A TD method is more efficient, since the method only creates estimates for the new first part of the drive, but keep the highway part the same. In addition, the TD method will update its value estimates based off the value estimates that follow it, which creates an acceleration effect of the value estimation. On the other hand, an MC method would have the complete a whole episode before updating its value estimates, which can be long and slow.

6.3 The first episode terminated on the left of the MRP with reward of 0. Given the undiscounted update equation $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + V(S_{t+1}) - V(S_t)]$, and all the value estimates started at 0.5, only the state before termination, which is A, will be updated, since $V(S_T) = 0$, so $V(A) \leftarrow 0.5 + \alpha[0 + 0 - 0.5] = 0.5(1 - \alpha)$.

6.4 A small α is required for convergence of both TD and MC methods. We should arrive at the same conclusion of the relative performance of both algorithms regardless of the α used, thus there is not such fixed value of α which can make either algorithm perform better.

6.5 The value of α determines the magnitude in which the value V is updated, and a higher α would make the estimate rely more on the returns of the next step. The increase in the error later on is a reflection of the random walk process of the problem, where the random walk causes the error in the estimate increase as the estimate is much more sensitive to such randomness at high α.

6.6 We can use probability to compute the true value of the random walk example. Let $P(L|C)$ and $P(R|C)$ denote the probability of terminating at the left and the right respectively, given the current state is C.

$$
\begin{aligned}
P(R|C) &= P(L|C) = \frac{1}{2} \quad \text{by symmetry} \\
P(R|D) &= P(C|D)P(R|C) + P(E|D)P(R|E) \\
&= \frac{1}{2}\left[\frac{1}{2} + P(R|E)\right] \\
P(R|E) &= \frac{1}{2} + P(D|E)P(R|D) \\
&= \frac{1}{2}\left[1 + P(R|D)\right] \\
P(R|D) &= \frac{1}{2}\left[1 + \frac{1}{2}P(R|D)\right] \\
P(R|D) &= \frac{1/2}{3/4} = \frac{2}{3}, \quad P(R|E) = \frac{1}{2} \cdot \frac{5}{3} = \frac{5}{6} \\
P(R|B) &= \frac{1}{3}, \quad P(R|A) = \frac{1}{3} \quad \text{by symmetry}
\end{aligned}
$$

6.7

Input: an arbitraty 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

$W \leftarrow 1$

Loop for each step of the episode (generated by b), ignoring first time step:

$$W = \frac{\pi(A_{t-1}|S_{t-1})}{b(A_{t-1}|S_{t-1})}$$

$$C(S_{t-1}, A_{t-1}) \leftarrow C(S_{t-1}, A_{t-1}) + W$$

$$Q(S_{t-1}, A_{t-1}) \leftarrow Q(S_{t-1}, A_{t-1}) + \frac{W}{C(S_{t-1}, A_{t-1})}[R_t + \gamma Q(S_t, A_t) - Q(S_{t-1}, A_{t-1})]$$

6.8

$$
\begin{aligned}
G_t - Q(S_t, A_t) &= R_{t+1} + \gamma G_{t+1} - Q(S_t, A_t) + \gamma Q(S_{t+1}, A_{t+1}) - \gamma Q(S_{t+1}, A_{t+1}) \\
&= \delta_t + \gamma(G_{t+1} - Q(S_{t+1}, A_{t+1})) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - Q(S_{t+2}, A_{t+2})) \\
&= \sum_{k=t}^{T-1} \gamma^{k=t}\delta_k, \quad \delta_k = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)
\end{aligned}
$$

6.9 With the availability of the extra 4 diagonal actions, the agent is able to improve on the number of steps it takes, where it can get from start to end in 13 steps, rather than the 16 steps required when only 4 steps are available. The inclusion of not moving does not improve the policy.

6.10 With the stochastic wind, the number of steps is able to decrease to a minimum of 9 steps. However, this relies on the stochastic nature of the wind.

6.11 From the implementation shown, the action selection follows a policy that is ϵ-greedy, whilst the action-value function is updated with a greedy method (selecting the maximum value). In this case, the ϵ-greedy method can be said to be the behaviour policy, whilst the greedy update is the target policy.

6.12 If action selection is greedy. it is likely that not all state-action pairs will be visited and this will not lead to convergence. Thus, this is not the same as the algorithm for Sarsa, since Sarsa uses an ϵ-greedy policy.

6.13 For Double Expected Sarsa:

$$
Q_1(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q_2(S_{t+1}, a) - Q(S_t, A_t)\Big]
$$

where $\pi(a|S_{t+1})$ is the ϵ-greedy target policy:

$$
\pi(a|S_{t+1}) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(S_{t+1})|} & \text{if } a = \arg\max_{a'}[Q_1(S_{t+1}, a') + Q_2(S_{t+1}, a')], \\ \epsilon & \text{otherwise} \end{cases}
$$

6.14 In Jack's Care Rental, we needed to estimate the state-action value functions for each action of each state, which can increase exponentially with the number of states. This is rather inefficient since some state-action pairs lead to the same afterstate, for example moving 1 car from a state of 10 cars at A and 10 cars at B, and moving 2 cars from a state of 11 cars and 9 cars at B, lead to the same afterstate of 9 cars at A and 11 cars at B. Using the idea of afterstates, we can treat the Poisson distribution of cars being rented out and returned as the 'opponent' in this scenario, and the afterstates can be the states of the car rental before the day starts, but after the cars are moved. This would significantly reduce the amount of computation required.