Temporal-Difference Learning

Diego Ascarza-Mendoza

Escuela de Gobierno y Transformación Pública

Outline

Introduction

- A central a novel idea in RL is *temporal-difference (TD)* learning.
- It mixes MC and DP ideas.
- Like in MC, TD can learn directly from raw experience without a model of the environment's dynamics.
- Like DP, TD methods update estimates based in part on other learned estimates without waiting for final outcome (bootstraps).
- We will see that ideas from TDP, DP and MC blend into each other and can be combined in many ways.
- We start by studying policy evaluation and for the control problem, as in the previous methods, we will use some sort of GPI.

Outline

TD Prediction

- Both TD and MC methods use experience to solve the prediction problem.
- Given some experience following a policy π, both methods update their estimate V of v_π for the nonterminal states S_t occurring in that experience.
- MC methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$.
- An every-visit MC method suitable for nonstationary environments is:

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

Where G_t is the actual return following time t, and α is a constant step-size parameter (constant-α) MC.

TD Prediction

- Whereas MC methods must wait until the end of the episode to determine the increment to $V(S_t)$, TD methods need to wait only until the next time step.
- At $t+1$ they immediately form a target and make a useful update using R_{t+1} and the estimate $V(S_{t+1})$:

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

- Immediately on transition to S_{t+1} and receiving R_{t+1}.
- While in MC the target is G_t, in TD the target is $R_{t+1} + \gamma V(S_{t+1})$.
- This method is called $TD(0)$, or one-step T, because it is a special case of $TD(\lambda)$ that you guys will present later.

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 $V(\mathit{terminal}) = 0$.
- Loop for each episode:
  1. Initialize S.
  2. Loop for each step of episode:

$$
\begin{aligned}
A &\leftarrow \text{action given by } \pi \text{ for } S \\
&\text{Take action A, observe } R, S' \\
V(S) &\leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)] \\
S &\leftarrow S' \\
&\text{until S is terminal}
\end{aligned}
$$

Tabular TD(0) for estimating v_π

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

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

- MC methods basically use an estimate for (3) as a target, whereas DP uses (5) as a target.
- Why are the targets for both MC and DP estimates?
- The TD target is an estimate for both reasons: it samples the expected values in (5) and it uses the current estimate V instead of the true v_π.
- In general, we can see that TD combines the sampling of MC with the bootstrapping of DP.

Sample updates and expected updates

- TD and Monte Carlo updates are called *sample updates* because they involve looking ahead to a sample successor state (or state-action pair).
- They do this by 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.
- DP does *expected updates* which differ from sample updates since the latter uses a single sample successor rather than a complete distribution of all possible successors.

- The quantity in brackets in $TD(0)$ update is a sort of error. This errors arises in varios forms throughout RL:

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

- Notice that the TD error at each time is the error in the estmate *made at that time*.
- Because the TD error depends on the next state and next reward, it is not actually available until one time step later (δ_t available in $t+1$).
- Note that if the array V does not change during the episode (as in MC), then the MC error can be written as a sum of TD errors:

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

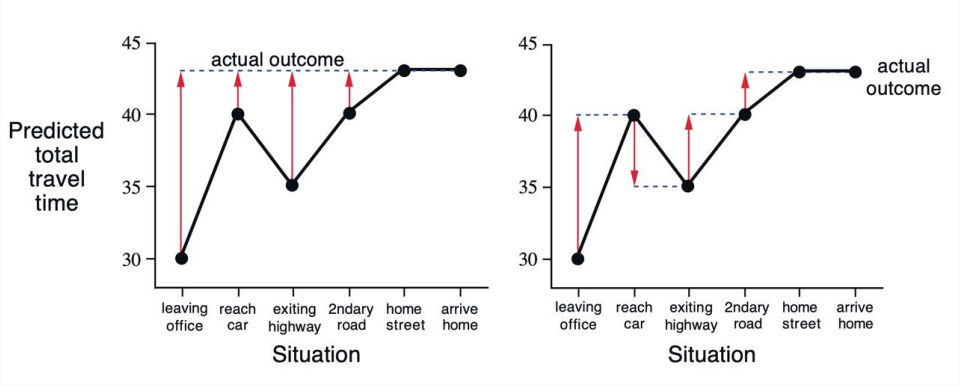
TD ERROR

- The identity is not exact if V is updated during the episode (as it is in TD(0)).
- However, if the step size is small, then it may still hold approximately. This is going to matter in general in the theory and algorithms of TD learning.

Example 6.1

State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

Example 6.1


Predicted total travel time
45
40
35
30
actual outcome
leaving office
reach car
exiting highway
2ndary road
home street
arrive home
Situation
45
40
35
30
actual outcome
leaving office
reach car
exiting highway
2ndary road
home street
arrive home
Situation

Outline

Advantages of TD Prediction Methods

- TD methods update their estimates based in part on other estimates (they bootstrap). Is that good?
- Here we are going to give some spoilers of why TD methods has advantages over MC and DP.
- First of all, TD has an advantage over DP in that they do not require a model of the environment, of its reward, and next-state probability distributions.
- The next most obvious advantage of TD methods over MC is that they are naturally implemented in an online, fully incremental fashion (you have to wait until the end of the episode with MC).
- TD methods wait for only one type of step. This is useful, for instance, if there are applications with very long episodes or continuing tasks.
- Finally, MC methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods learn from each transition regardless of what subsequent actions are taken.

Are TD methods sound?

- Can we guarantee convergence by learning one guess from the next, without waiting for an actual outcome? **YES!**
- For any fixed policy π, $TD(0)$ has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions.
- If both TD and MC methods converge asymptotically to the correct predictions, then a natural question is "Which gets there first?"
- This is an open question. In practice, however, TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks. Let's see an example to illustrate this.

Random Walk

- Let's compare the prediction abilities of TD(0) and constant-α MC when applied to the following Markov reward process.
- A *Markov reward process (MRP)* is a Markov decision process without actions.
- It is useful because we do not 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 right or the extreme left.
- When the 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: $C, 0, B, C, 0, D, 0, E, 1$.
- 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 $v_\pi(C) = 0.5$. The true values of all the states, A through E, are $1/6, 2/6, 3/6, ..., 5/6$.
- If we measure performance as the root of the mean-squared error (RMS between the value function learned and the true value function, averaged over the five stages, then averaged over 100 runs, TD is consistently better than the MC on this task.
- What is happening?

Outline

Optimality of the TD(0)

- Suppose there is available only a finite amount of experience, like 10 episodes or 100 time steps.
- A common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer.
- Give an approximate value function, V, the increments specified by equations 1 or 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.
- We call this *batch updating* because updates are made only after processin each complete *batch* of training data.

Optimality of the TD(0)

- Under batch updating, $TD(0)$ converges deterministically to a single answer independent of the step-size parameter α, as long as α is chosen to be sufficiently small.
- The constant-α MC method also converges deterministically under the same conditions, but to a different answer.
- Understanding why this is the case will help us understand the differences between the two methods.
- Under normal updating, the methods do not move all the way to their respective batch answers, but in some sense they take steps in these directions.

Random Walk Under Batch Updating

- We did batch-updating versions of TD(0) and constant-α MC to the random walk prediction example as follows.
- 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.
- The resulting value function was then compared with v_π, and the average root mean-squared error across the five states was plotted to obtain the learning curves.
- Note that the batch TD method was consistently better than the batch MC method.

Random Walk Under Batch Updating

- Under batch training, constant-α MC converges to values, $V(s)$, that are sample averages of the actual returns experienced after visiting each state s.
- These are optimal estimates in the sense that they minimize the MSE from the actual returns in the training set.
- In this sense, it is surprising that the batch TD method was able to perform better according to our measures.
- The reason is that MC methods are optimal only in a limited way, and that TD is optimal in a way that is more relevant to predicting returns.

Example: You are the Predictor

- Suppose you are now in the role of the predictor of returns for an unknown Markov reward process, and you observe the following eight episodes:

$$
\begin{array}{ll}
A, 0, B, 0 & B, 1 \\
B, 1 & B, 1 \\
B, 1 & B, 1 \\
B, 1 & B, 0
\end{array}
$$

Example: You are the Predictor

- Given this batch of data, what would you say are the optimal predictions, the best values for $V(A)$ and V(B)?
- Probably everyone would agree that the optimal value for $V(B)$ is 0.75 (why?).
- But what is the optimal value for the estimate $V(A)$ given this data? There are two reasonable answers.
- One is to observe that 100% of the time, the process was in A, it led to B. Since the value of B is 0.75, then A must have a value of 0.75 as well. This can follow based on first modeling the Markov process.
- This is also the answer that batch TD(0) gives.
- The other reasonable answer is simply to observe that we have seen A once and the return that followed it was 0; we therefore estimate $V(A)$ as 0. This is the answer that batch MC methods give.

Example: You are the Predictor

- Notice that it is also the answer that gives the minimum squared error on the training data. In fact, it gives zero error on the data.
- But still, we expect the first answer to be better.
- If the process is Markov, we expect that the first answer will produce lower error on *future data*, even though the MC answer is better on the **existing** data.
- This example illustrates a general difference between the estimates found by batch TD(0) and batch MC methods.

The Key Difference

- MC methods find the estimates that minimize MSE on the training set. Batch TD(0) always finds the estimates that would be exactly correct for the ML model of the Markov process.
- In general, the ML estimate of a parameter is the parameter value whose probability of generating the data is greatest.
- In this case, the ML estimate is the model of the Markov process formed in the obvious way from the observed episodes:
  1. The estimated transition probability from i to j is the fraction of observed transitions from i that went to j.
  2. The associated expected reward is the average of the rewards observed on those transitions.
- Given this model, we can compute the estimate of the VF that would be exactly correct if the model were exactly correct (*certainty-equivalence estimate*). 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.

The Key Difference

- This feature helps to explain why TD methods converge more quickly than MC methods.
- This is because TD(0) computes the true certainty-equivalence estimate. This relationship to the certainty equivalence estimate may also explain in part the speed advantage of non-batch TD(0).
- Although the nonbatch methods do not achieve either the certainty-equivalence or the minimum squared-error estimates, they can be understood as moving roughly in these directions.
- Nonbatch TD(0) may be faster than constant-α MC because it is moving toward a better estimate, even though it is not getting all the way there.
- It is still an open question what is the relative efficiency of online TD and MC methods.

The Key Difference

- 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 ML estimate of the process may require on the order of n^2 memory, and computing the corresponding VF 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 solutions.

Outline

Sarsa: On-policy TD Control

- Let's turn now to the use of TD prediction methods for the control problem.
- As usual, we follow the pattern of GPI, only this time using TD methods for the evaluation part.
- As with MC methods, we face the need to trade off exploration and exploitation, and again, approaches fall into two main classes: on-policy and off-policy. Let's start with on-policy.
- The first step is to learn an action-value function rather than a state-value function.
- For an on-policy method we must estimate $q_\pi(s,a)$ for the current π and for all states s and actions a.
- This can be done using essentially the same TD method described above for learning v_π.

Sarsa: On-policy TD Control

- In the previous section we considered transitions from state to state and learned the value of states.
- Now we consider transitions from state-action pair to state-action pair, and learn the values of state-action pairs. Formally, these cases are identical: Markov chains with a reward process.
- The theorems assuring the converge of state values under $TD(0)$ also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right]. \tag{7}$$

- This update is done after every transition from a nonterminal 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 from one state-action pair to the next.
- This quintuple gives rise to the name *Sarsa* for the algorithm.

Sarsa: On-policy TD Control

- It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method.
- As in all on-policy methods, we continually estimate q_π for π, and at the same time change π toward greediness with respect to q_π.
- The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q.
- For example, one could use $\epsilon-$greedy for ϵ-soft policies.
- 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 (for instance with ϵ-greedy policies by setting $\epsilon = 1/t$).

Sarsa (On-policy TD Control) algorithm for estimating $Q \approx q_*$

- Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$.
- Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, .) = 0$.
- Loop for each episode:
  1. Initialize S.
  2. Choose A from S using policy derived from Q (e.g., $\epsilon-$greedy).
  3. Loop for each step of episode:
     - Take action A, observe R, S'.
     - Choose A' from S' using policy derived from Q (e.g., $\epsilon-$greedy).
     - $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
     - $S \leftarrow S'$; $A \leftarrow A'$;

     until S is terminal.

Example: Windy Gridworld

Outline

Q-learning: Off-policy TD Control

- One of the early breakthroughs in RL was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989), defined by:

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

- In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed.
- This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs.
- The policy still has an effect in that it determines which state-action pairs are visited and updated.
- However, all that is required for correct convergence is that all pairs continue to be updated. As we saw in MC chapter, this is a minimal requirement to guarantee finding optimal behavior.

- Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size paremeters, Q has been shown to converge with probability 1 to q_*.
- Let's take a look at the Q-learning algorithm.

Q-LEARNING (OFF-POLICY TD CONTROL) FOR ESTIMATING $\pi \approx \pi_*$

- Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$.
- Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, .) = 0$.
- Loop for each episode:
  1. Initialize S.
  2. Loop for each step of episode:
     - Choose A from S using policy derived from Q (e.g., $\epsilon-$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.

Outline

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.
- That is, it takes into account how likely each action is under the current policy.
- That is, consider the algorithm with the update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi [Q(S_{t+1}, A_{t+1}) | S_{t+1}] - Q(S_t, A_t) \right] \tag{9}$$

$$\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right], \tag{10}$$

- But otherwise follows the schema of Q-learning. 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

- 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 indeed it generally does.
- If you check the examples in the book, you will see that Expected Sarsa retains the significant advantage of Sarsa over Q-learning.
- Furthermore, Expected Sarsa might use a policy different from the target policy π to generate behavior, in which case it becomes an off-policy algorithm.
- For example, suppose π is the greedy policy while behavior is more exploratory; then Expected Sarsa is exactly Q-learning.
- In this sense, Expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa (at a small computational cost). Expected Sarsa may completely dominate both of the other more-well-known TD control algorithms.

Outline

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 instance, in Q-learning, the target policy is the greedy policy given the current action values, which is defined with a max.
- In the 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 see why, 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 below zero.
- The maximum of the true values is zero, but the maximum of the estimates is positive. We call this *maximization bias*.

- Are there algorithms that avoid maximization bias? Consider a bandit case in which we have noisy estimates of the value of each of many actions.
- Suppose these estimates are obtained as sample averages of the rewards received on all the plays with each action.
- As we said before, there will be a positive maximization bias if we use the maximum of the estimates to estimate the maximum of the true values.
- One way to see the problem is that it is due to using the same samples both to determine the maximizing action and to estimate its value.
- Suppose we divided the plays into two sets and used them to learn two independent estimates ($Q_1(a), Q_2(a)$) to estimate $q(a) \forall a \in \mathcal{A}$.

- We could then use one estimate, say Q_1, to determine the maximizing action $A^* = \arg\max_a Q_1(a)$.
- An the other, Q_2, to provide the estimate of tis 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 updated on each play. Double learning doubles the memory requirements, but does not increase the amount of computation per step.

- The idea of double learning extends naturally to algorithms for full MDPs.
- For example, the double learning algorithm analogous to Q-learning, called Double Q-learning, divides the time steps in two, perhaps by flipping a coin on each step. If the coin comes up heads, the update is:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right] \tag{11}$$

- If the coin comes up tails, then the same update is done with Q_1 and Q_2 switched, so that Q_2 is updated. The two approximate value functions are treated completely symmetrically.
- The behavior policy can use both action-value estimates. For instance, an $\epsilon-$greedy policy for Double Q-learning could be based on the average (or sum) of the two action-value estimates.
- Of course, there are also double versions of Sarsa and Expected Sarsa. Let's take a look now to the algorithm.

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

- Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$.
- Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, such that $Q(\textit{terminal}, .) = 0$.
- Loop for each episode:
  1. Initialize S.
  2. Loop for each step of episode:
     - Choose A from S using the policy $\epsilon-$greedy in $Q_1 + Q_2$.
     - Take action A, observe R, S'.
     - With 0.5 probability:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S', \arg\max_a Q_1(S', a)) - Q_1(S, A) \right]$$

     else:

$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_1(S', \arg\max_a Q_2(S', a)) - Q_2(S, A) \right]$$

     - $S \leftarrow S'$

     until S is terminal.

Outline

Games, Afterstates, and Other Special Cases

- There are always exceptional tasks that are better treated in a specialized way.
- For example, our general approach involves learning an *action-value* function, but in the first class we presented a TD method for learning to play *tic-tac-toe* that learned something much more like a *state-value* function.
- If we look carefully, the function learned there is neither an action-value function nor a state-value function in the usual sense.
- A conventional state-value function evaluates states in which the agent has the option of selecting an action, but the state-value function in tic-tac-toe evaluates board positions *after* the agent has made its move.
- Let us call these *afterstates*, and value functions over these, *afterstate value functions*.
- Afterstates are helpful when we know an initial part of the environment's dynamics but not necessarily of the whole dynamics.

- For example, in games we typically know the immediate effects of our moves. We know for each possible chess move what the resulting position will be, but not how our opponent will reply.
- Afterstate value functions are a natural way to take advantage of this kind of knowledge and thereby produce a more efficient learning method.
- The reason it is more efficient to design algorithms in terms of afterstates is apparent from the tic-tac-toe example.
- A conventional action-value function would map from positions and moves to an estimate of the value.
- But many position-move pairs produce the same resulting position. Let's give some examples.

Games, Afterstates, and Other Special Cases

- Position-move pairs are different but produce the same ”afterposition” and thus must have the same value.
- A conventional action-value function would have to separately assess both pairs, where as an afterstate value function would immediately assess both equally.
- Any learning about the position-move pair would immediately transfer to the pair that has the same afterstate.
- These cases show up in many cases, not only in games. For instance, assigning customers to servers, rejecting customers, or discarding information.
- It is impossible to describe all the possible kinds of specialized problems and corresponding specialized learning algorithms. What we learned in this course should apply widely.
- For instance, afterstate methods are still aptly described in terms of GPI, with a policy and (afterstate) value function interacting in essentially the same way.