CS 4180/5180: Reinforcement Learning and Sequential Decision Making (Fall 2021) Robert Platt
Northeastern University Due October 29, 2021

# Exercise 5: Temporal-Difference Learning

1. **1 point.** (RL2e 6.2) *Temporal difference vs. Monte-Carlo.*
**Written:** Read and understand Example 6.1, then answer the following:

> **Example 6.1: Driving Home** Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant. Say on this Friday you are leaving at exactly 6 o'clock, and you estimate that it will take 30 minutes to get home. As you reach your car it is 6:05, and you notice it is starting to rain. Traffic is often slower in the rain, so you reestimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn onto the side street where you live at 6:40. Three minutes later you are home. The sequence of states, times, and predictions is thus as follows:
>
> | *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 |
>
> The rewards in this example are the elapsed times on each leg of the journey.[1] We are not discounting ($\gamma = 1$), and thus the return for each state is the actual time to go from that state. The value of each state is the *expected* time to go. The second column of numbers gives the current estimated value for each state encountered.
>
> A simple way to view the operation of Monte Carlo methods is to plot the predicted total time (the last column) over the sequence, as in Figure 6.1 (left). The red arrows show the changes in predictions recommended by the constant-$\alpha$ MC method (6.1), for $\alpha = 1$. These are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go). For example, when you exited the highway you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes. Equation 6.1 applies at this point and determines an increment in the estimate of time to go after exiting the highway. The error, $G_t - V(S_t)$, at this time is eight minutes. Suppose the step-size parameter, $\alpha$, is 1/2. Then the predicted time to go after exiting the highway would be revised upward by four minutes as a result of this experience. This is probably too large a change in this case; the truck was probably just an unlucky break. In any event, the change can only be made offline, that is, after you have reached home. Only at this point do you know any of the actual returns.
>
> Is it necessary to wait until the final outcome is known before learning can begin? Suppose on another day you again estimate when leaving your office that it will take 30 minutes to drive home, but then you become stuck in a massive traffic jam. Twenty-five minutes after leaving the office you are still bumper-to-bumper on the highway. You now

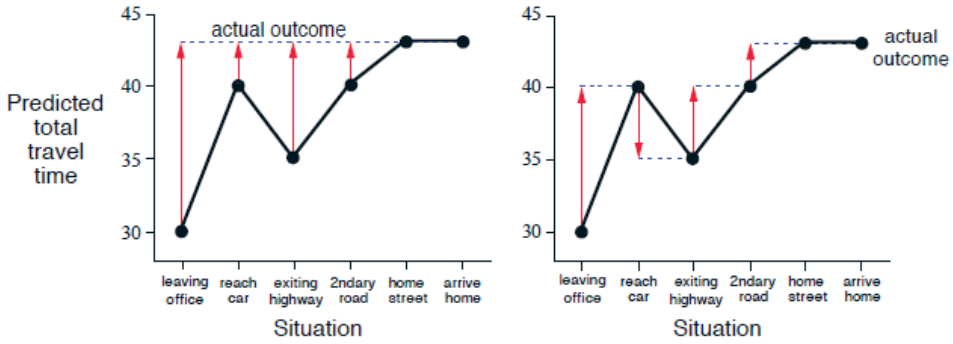



**Figure 6.1:** Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

estimate that it will take another 25 minutes to get home, for a total of 50 minutes. As you wait in traffic, you already know that your initial estimate of 30 minutes was too optimistic. Must you wait until you get home before increasing your estimate for the initial state? According to the Monte Carlo approach you must, because you don't yet know the true return.

According to a TD approach, on the other hand, you would learn immediately, shifting your initial estimate from 30 minutes toward 50. In fact, each estimate would be shifted toward the estimate that immediately follows it. Returning to our first day of driving, Figure 6.1 (right) shows the changes in the predictions recommended by the TD rule (6.2) (these are the changes made by the rule if $\alpha = 1$). Each error is proportional to the change over time of the prediction, that is, to the *temporal differences* in predictions.

Besides giving you something to do while waiting in traffic, there are several computational reasons why it is advantageous to learn based on your current predictions rather than waiting until termination when you know the actual return. We briefly discuss some of these in the next section. ■

(a) This is an exercise to help develop your intuition about why TD methods are often more efficient than Monte-Carlo methods. Consider the driving home example and how it is addressed by TD and Monte-Carlo methods. Can you imagine a scenario in which a TD update would be better on average than a Monte-Carlo update? Give an example scenario – a description of past experience and a current state – in which you would expect the TD update to be better.
*Hint*: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original scenario?

(b) Is there any situation (not necessarily related to this example) where the Monte-Carlo approach might be better than TD? Explain with an example, or explain why not.

2. **1 point.** (RL2e 6.11, 6.12) *Q-learning vs. SARSA.*
**Written:**

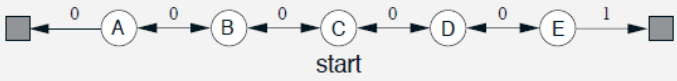
(a) Why is Q-learning considered an off-policy control method?

(b) Suppose action selection is greedy. Is Q-learning then exactly the same algorithm as SARSA? Will they make exactly the same action selections and weight updates?

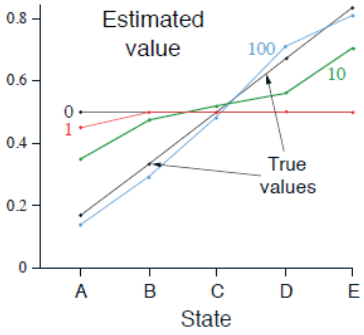
3. **3 points.** (RL2e 6.3, 6.4, 6.5, 7.3) *Random-walk task.*
**Written:** Read and understand Example 6.2 and Example 7.1, then answer the following:

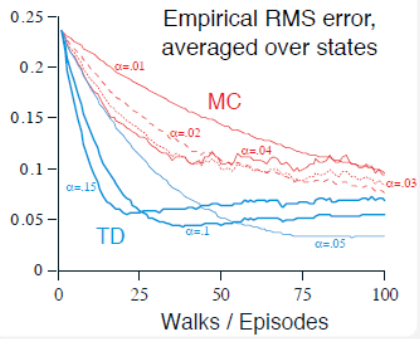
**Example 6.2 Random Walk**

In this example we empirically compare the prediction abilities of TD(0) and constant-$\alpha$ MC when applied to the following Markov reward process:




A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which there is no 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 left or the extreme right. When an 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, 0, 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 $v_\pi(\mathsf{C}) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$.




The left graph above shows the values learned after various numbers of episodes on a single run of TD(0). The estimates after 100 episodes are about as close as they ever come to the true values—with a constant step-size parameter ($\alpha = 0.1$ in this example), the values fluctuate indefinitely in response to the outcomes of the most recent episodes. The right graph shows learning curves for the two methods for various values of $\alpha$. The performance measure shown is the root mean-squared (RMS) error between the value function learned and the true value function, averaged over the five states, then averaged over 100 runs. In all cases the approximate value function was initialized to the intermediate value $V(s) = 0.5$, for all $s$. The TD method was consistently better than the MC method on this task.

**Example 7.1: $n$-step TD Methods on the Random Walk** Consider using $n$-step TD methods on the 5-state random walk task described in Example 6.2 (page 125). Suppose the first episode progressed directly from the center state, C, to the right, through D and E, and then terminated on the right with a return of 1. Recall that the estimated values of all the states started at an intermediate value, $V(s) = 0.5$. As a result of this experience, a one-step method would change only the estimate for the last state, $V(\mathsf{E})$, which would be incremented toward 1, the observed return. A two-step method, on the other hand, would increment the values of the two states preceding termination: $V(\mathsf{D})$ and $V(\mathsf{E})$ both would be incremented toward 1. A three-step method, or any $n$-step method for $n > 2$, would increment the values of all three of the visited states toward 1, all by the same amount.

Which value of $n$ is better? Figure 7.2 shows the results of a simple empirical test for a larger random walk process, with 19 states instead of 5 (and with a $-1$ outcome on the left, all values initialized to 0), which we use as a running example in this chapter. Results are shown for $n$-step TD methods with a range of values for $n$ and $\alpha$. The performance measure for each parameter setting, shown on the vertical axis, is the square-root of the average squared error between the predictions at the end of the episode for the 19 states and their true values, then averaged over the first 10 episodes and 100 repetitions of the whole experiment (the same sets of walks were used for all parameter settings). Note that methods with an intermediate value of $n$ worked best. This illustrates how the generalization of TD and Monte Carlo methods to $n$-step methods can potentially perform better than either of the two extreme methods.

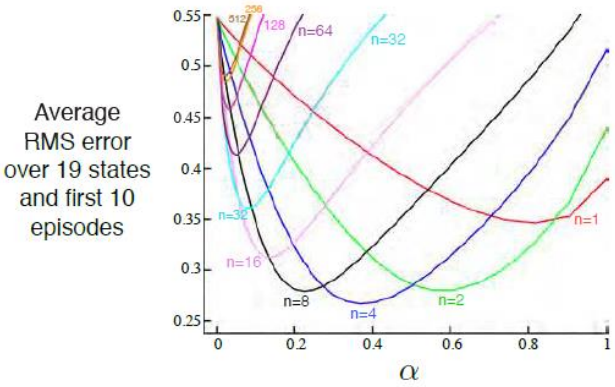



**Figure 7.2:** Performance of $n$-step TD methods as a function of $\alpha$, for various values of $n$, on a 19-state random walk task (Example 7.1). ■

(a) From the results shown in the left graph of the random-walk example it appears that the first episode results in a change in only $V(A)$. What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed?

(b) The specific results shown in the right graph of the random walk example are dependent on the value of the step-size parameter, $\alpha$. Do you think the conclusions about which algorithm is better would be affected if a wider range of $\alpha$ values were used? Is there a different, fixed value of $\alpha$ at which either algorithm would have performed significantly better than shown? Why or why not?

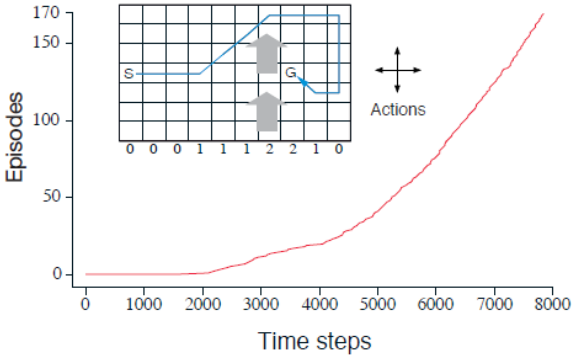
(c) In the right graph of the random walk example, the RMS error of the TD method seems to go down and then up again, particularly at high $\alpha$'s. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized?

(d) Why do you think a larger random walk task (19 states instead of 5) was used in Example 7.1?
Would a smaller walk (fewer states) have shifted the advantage to a different value of $n$?
How about the change in left-side outcome from 0 to $-1$ made in the larger walk?
Do you think that made any difference in the best value of $n$?

(e) **[Extra credit.]** Implement TD(0) on the random-walk task and computationally verify your answers to the above questions.

4. **4 points.** (RL2e 6.9, 6.10) *Windy gridworld.*
**Code/plot:** In this question, you will implement several TD-learning methods and apply them to the windy gridworld in Example 6.5.

(a) Implement the windy gridworld domain. Read the description in Example 6.5 carefully to find all details.

> **Example 6.5: Windy Gridworld** Shown inset below is a standard gridworld, with start and goal states, but with one difference: there is a crosswind running upward through the middle of the grid. The actions are the standard four—`up`, `down`, `right`, and `left`—but in the middle region the resultant next states are shifted upward by a "wind," the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted upward. For example, if you are one cell to the right of the goal, then the action `left` takes you to the cell just above the goal. This is an undiscounted episodic task, with constant rewards of $-1$ until the goal state is reached.
>
> 
> 
>
> The graph to the right shows the results of applying $\varepsilon$-greedy Sarsa to this task, with $\varepsilon = 0.1$, $\alpha = 0.5$, and the initial values $Q(s,a) = 0$ for all $s, a$. The increasing slope of the graph shows that the goal was reached more quickly over time. By 8000 time steps, the greedy policy was long since optimal (a trajectory from it is shown inset); continued $\varepsilon$-greedy exploration kept the average episode length at about 17 steps, two more than the minimum of 15. Note that Monte Carlo methods cannot easily be used here because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Online learning methods such as Sarsa do not have this problem because they quickly learn *during the episode* that such policies are poor, and switch to something else. ■

(b) Implement the following methods, to be applied to windy gridworld:

- On-policy Monte-Carlo control (for $\varepsilon$-soft policies) – consider using your code from Ex4
- SARSA (on-policy TD control)
- Expected SARSA
- $n$-step SARSA (use $n = 4$)
- Q-learning (off-policy TD control)
- *Optional*: Dynamic programming (to provide an upper bound)

To compare each method, generate line plots similar to that shown in Example 6.5 (do not generate the inset figure of the gridworld). Make sure you understand the axes in the plot, which is not the same as before (why is it different?).
As in previous exercises, perform at least 10 trials, and show the average performance with confidence bands ($1.96\times$ standard error).
If you implement the optional DP solution, use it to generate and plot an upper bound on performance.
*Note*: You may adjust hyperparameters for each method as necessary; for SARSA, use the values provided in the example ($\varepsilon = 0.1, \alpha = 0.5$) so that you can reproduce the plot in the textbook.

For the following parts, apply at least two of the above TD methods to solve them.

(c) *Windy gridworld with King's moves*: Re-solve the windy gridworld assuming eight possible actions, including the diagonal moves, rather than four. How much better can you do with the extra actions? Can you do even better by including a ninth action that causes no movement at all other than that caused by the wind?

(d) *Stochastic wind*: Re-solve the windy gridworld task with King's moves, assuming that the effect of the wind, if there is any, is stochastic, sometimes varying by 1 from the mean values given for each column. That is, a third of the time you move exactly according to these values, as you did above, but also a third of the time you move one cell above that, and another third of the time you move one cell below that. For example, if you are one cell to the right of the goal and you move left, then one-third of the time you move one cell above the goal, one-third of the time you move two cells above the goal, and one-third of the time you move to the goal.

5. **[CS 5180 only.] 2 points.** *Bias-variance trade-off.*
In lecture, we discussed that Monte-Carlo methods are unbiased but typically high-variance, whereas TD methods trade off bias to obtain lower-variane estimates. We will investigate this claim empirically in this question, from the perspective of prediction.

The overall experimental setup is as follows.

- We will continue with the deterministic (original) windy gridworld domain.
- A fixed policy $\pi$ will be specified.
- A certain number of "training" episodes $N \in \{1, 10, 50\}$ will be collected.
- Each method being investigated (TD(0), $n$-step TD, Monte-Carlo prediction) will estimate the state-value function based on the $N$ episodes.
- We then evaluate the distribution of learning targets each method experiences at a specified state $S$.
To do so, an additional 100 "evaluation" episodes will be generated. Instead of using these to perform further updates to the state-value function, we will instead evaluate the distribution of learning targets $V(S)$ will effectively experience based on the "evaluation" episodes. For example, TD(0) will experience a set of $\{R + V(S')\}$ targets, whereas Monte-Carlo will experience a set of $\{G\}$ targets.

Note that in practice you should pre-collect both the training and evaluation episodes for efficiency and to ensure consistency while comparing between different methods.

(a) **Code/plot:** Perform the above experiment for the specified methods and training episodes $N$.
Use a near-optimal stochastic policy $\pi$ (e.g., found by SARSA or other methods in Q4).
Perform the evaluation for the start state (indicated 'S' in Example 6.5).
For $n$-step TD, use $n = 4$, but you may consider trying more values of $n$ as well.
Plot the histogram of learning targets experienced in the evaluation episodes for each combination of $N$ and method (i.e., at least 9 histograms total).
*Optional*: Use dynamic programming or any other appropriate method to compute the true value of $v_\pi(s)$ for comparison purposes, and add this to your plots as well.

(b) **Written:** Describe what you observe from your histograms.
Comment on what they may show about the bias-variance trade-off between the different methods, and how it may depend on the amount of training that has already occurred.

(c) **[Extra credit.]** If we considered the scenario of control (i.e., we would use on-policy action-value methods, iteratively update the policy during training, and use it to generate the next training episode), would that change the results, and how? Implement this to computationally test your hypothesis.