CS0138 Assignment 3

Peter Nadel

November 2025

1 Introduction

For this assignment, we were asked to implement two temporal difference algorithms and answer two separate questions, one taken from Sutton and Barto and the other formulated by ourselves. For this problem, we took the same environment as in assignment 2: the racetrack problem as described in chapter 5, problem 12 of Sutton and Barto. Thus, the two questions are as follows:

1. To what extent are we able to solve this racetrack problem with temporal difference methods?
2. How does n-step backtracking affect agent learning in this context?

As described in assignment 2, the agent in on the racetrack must learn how to control its velocity in order to get to the finish line. In both the X and Y directions, it is able to increment its current velocity by -1, 0, or 1, with a total set of 9 actions. Additionally, the agent may *not* exceed a speed of 5 in either direction or fall below a speed of zero in either direction. In general, this environment requires the agent speed past the easier sections of the track, while moderating its speed through the left turns. This task is potentially quite difficult, especially if the track is large. Attempting to solve this problem with non-reinforcement learning methods can become unwieldy, but Monte Carlo and temporal difference methods do well in learning the shape of the track.

In particular, for this assignment we implement Sarsa as well as n-step backtracking Sarsa to solve this problem. As opposed to Monte Carlo, these temporal difference methods are able to learn and update state-action values during an episode. Not needing to finish the episode before an update occurs allows these algorithms to learn faster than Monte Carlo. Similarly, while Sarsa learns from and updates its current position, n-step backtracking is able to update n-steps in the past, allowing the agent to develop a memory of what actions get closer to the goal and which do not.

2 Methods

2.1 Sarsa racetrack problem

Sutton and Barto provide two 36x36 tracks to learn. For the sake of simplicity, we chose to only report results for the first track. Its array representation is shown below, with the starting positions marked in green and the finish line marked in yellow. The blue track is surrounded with purple off-track zones. Per the assignment, if the car touches these areas, the episode continues but the car is randomly replaced at a starting position.

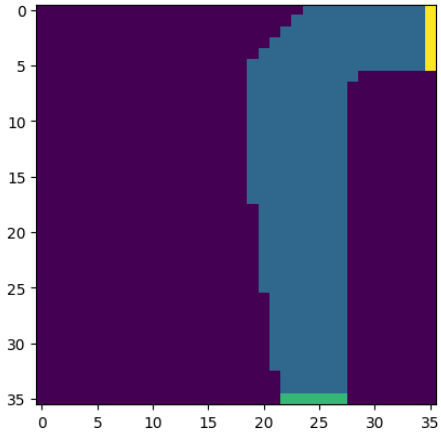

Figure 1: Track 1 from Sutton and Barto

So that the agent is able to learn how to navigate any track, we applied the On-policy Sarsa (State action reward state action) control algorithm as described in Sutton and Barto:

As in assignment 2, to implement the environment in Python, we added a few features. First, we added a number of maximum steps which an episode can last. Unlike in the Monte Carlo setting where this parameter is nearly necessary, as an update will not occur until an episode concludes, for Sarsa, the maximum steps are just used to speed up training, as the agent is still able to learn despite the epsiode not finishing.

Too, when using our trained Q table to develop completed trajectories, we needed to turn off all stochasticity. This includes both the noise factor mentioned in the problem description and the epsilon-soft policy.

2.2 Extra experiment

For our extra experiment, we chose to implement n-step backtracking for Sarsa. We applied the n-step Sarsa control algorithm as described in Sutton and Barto:

This algorithm lies between Monte Carlo and Sarsa. While Monte Carlo methods update their Q tables for each state based on the entire sequence of

Algorithm 1 Sarsa (on-policy TD control)

Initialize the following:
$\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+$, $a \in A^+$, arbitrarily except that $Q(terminal, \cdot) = 0$
loop
  Initialize S
  Choose A from S using policy derived Q (in our case, ϵ-greedy)
  loop
    Take action A, observe R, S'
    Choose A' from S' using the policy derived from Q (in our case, ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  end loop
  until S is terminal
end loop

observed rewards from the start of an episode to its conclusion, Sarsa updates are based solely on the next reward. With n-step bootstrapping, this algorithm is able to perform updates based on an n number of intermediate rewards before the given state action pair. In our lectures, we have described this notion as a way that the agent has to "remember" past actions and update its policy accordingly.

Sutton and Barto show that this bootstrapping allows the agent to find better paths faster than standard TD and Monte Carlo methods. In fact, n-step Sarsa may find a more optimal path than Sarsa itself. Indeed, because it follows the same steps as Sarsa and only updates other state action pairs based on an observed reward, n-step Sarsa and Sarsa will eventually visit the same state action pairs during training, but n-step Sarsa will more efficiently propagate rewards through the trajectory.

3 Results

3.1 Standard racetrack problem

After training an agent for each of the above track, we report the following trajectory:

This trajectory shows that, through standard Sarsa alone,the agent was able to learn the right turn behavior needed to complete the track. Note too the space between actions that the agent takes.

Algorithm 2 n-step Sarsa for estimating Q

Initialize $Q(s, a)$ arbitrarily, for all $s \in S$, $a \in A$
Algorithm parameters: $\alpha \in (0, 1]$, small $\epsilon > 0$, a positive integer n
loop
    Initialize and store $S_0 \neq$ terminal
    Select and store an action A_0 using policy derived from Q (in our case, ϵ-greedy)
    $T \leftarrow \infty$
    for $t = 0, 1, 2, \ldots$ **do**
        if $t < T$ **then**
            Take action A_t
            Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
            if S_{t+1} is terminal **then**
                $T \leftarrow t + 1$
            else
                Select and store an action A_{t+1} using policy derived from Q (in our case, ϵ-greedy)
            end if
        end if
        $\tau \leftarrow t - n + 1$
        if $\tau \geq 0$ **then**
            $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
            if $\tau + n < T$ **then**
                $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$
            end if
            $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$
        end if
        if $\tau = T - 1$ **then**
            break
        end if
    end for
end loop

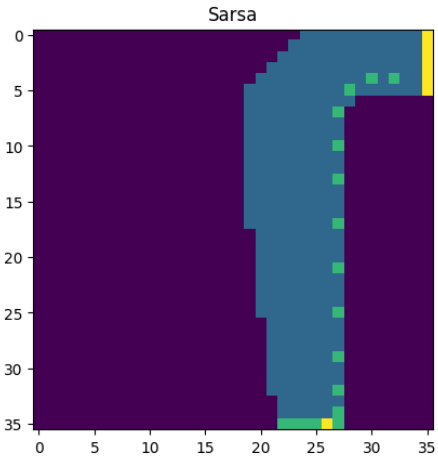

Figure 2: Learned path of track 1 from Sutton and Barto using Sarsa

3.2 Extra experiment

As mentioned above, we also report the result of the n-step algorithm. See the following trajectories:

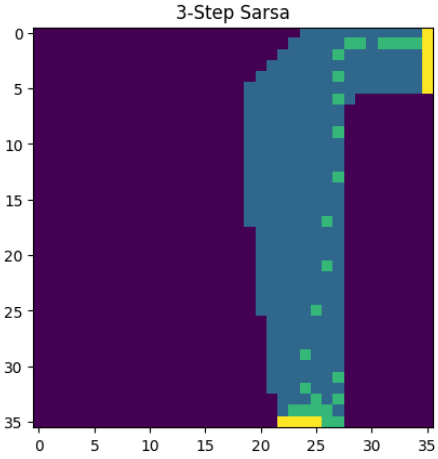

Figure 3: Learned path of track 1 from Sutton and Barto using 3-Step Sarsa

These trajectories from the n-step algorithms tell a similar story to the standard, or 1-step Sarsa, trajectory above, but we observe some clear differences. Most obviously, we see that the n-step algorithms use more actions to make the turn than the standard version of the algorithm. This tendency is likely due to how n-step Sarsa propagates rewards to state action pairs. While standard

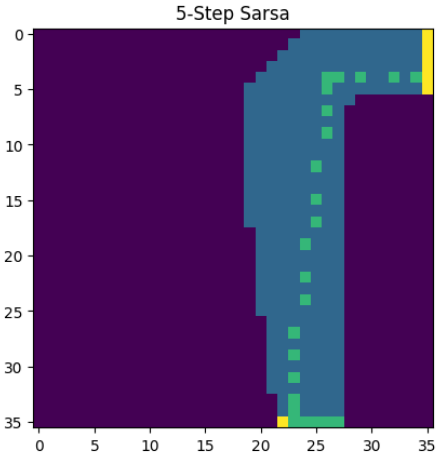

Figure 4: Learned path of track 1 from Sutton and Barto using 5-Step Sarsa

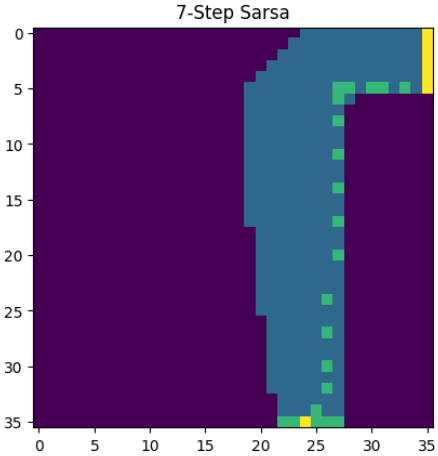

Figure 5: Learned path of track 1 from Sutton and Barto using 7-Step Sarsa

Sarsa has is highly biased towards the single update that it makes on its current step, n-step Sarsa is less biased and rewards variance. This leads to faster learning earlier but the policy that n-step Sarsa learns may not be optimal, as we see in these trajectories.

We see this relationship more clearly when we plot the length of a given episode (steps) against the amount of training episodes as below.

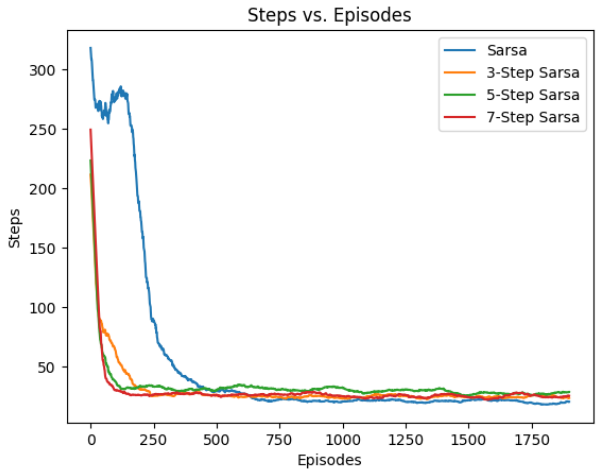

Figure 6: Steps vs. Episodes on track 1 from Sutton and Barto

As we see, the n-step variants do very well early on, learning much faster than standard Sarsa. Especially with higher and higher values of n, the learning seems to be more and more accelerated, with n=7 achieving an impressively fast time to converge to a solution. On the other hand though, standard Sarsa finishes its training with the overall lowest number of steps, while the n-step variant all learn non-optimal policies which take more steps than standard Sarsa. It may be that this environment is too simple for n-step Sarsa variants to find new ways to exploit their advantage. Especially early on in the track, the state space is so small that n-step bootstrapping does not help the agent learn anything more about its environment than standard Sarsa would and instead can lead the agent astray as we see in the case of the 3-step Sarsa trajectory.

To conclude, these experiments show that while n-step Sarsa can substantially speed up training, but this speed-up does not ensure that this learned policy will be good or better than standard Sarsa. Thus to answer the questions raised in the Introduction, temporal difference methods are sufficient to solve this problem. Too, while n-step backtracking can affect this learning, it varies in its utility.