National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Faculty of Computing

CS 368: Reinforcement Learning

Lab 08: Model Free Reinforcement Learning - Control

(n-Step SARSA and SARSA (λ))

Date: 15 November 2024

Time: 02:00 PM – 5:00 PM

Instructor: Dr. Zuhair Zafar

Lab 08: Model Free Control (n-step SARSA & SARSA (λ) Algorithm)

Objectives

Given a simulated environment, find the optimal policy using n-step SARSA & SARSA (λ) Algorithm.

Tools/Software Requirement:

Google Colab, Python, Gymnasium Library

Introduction

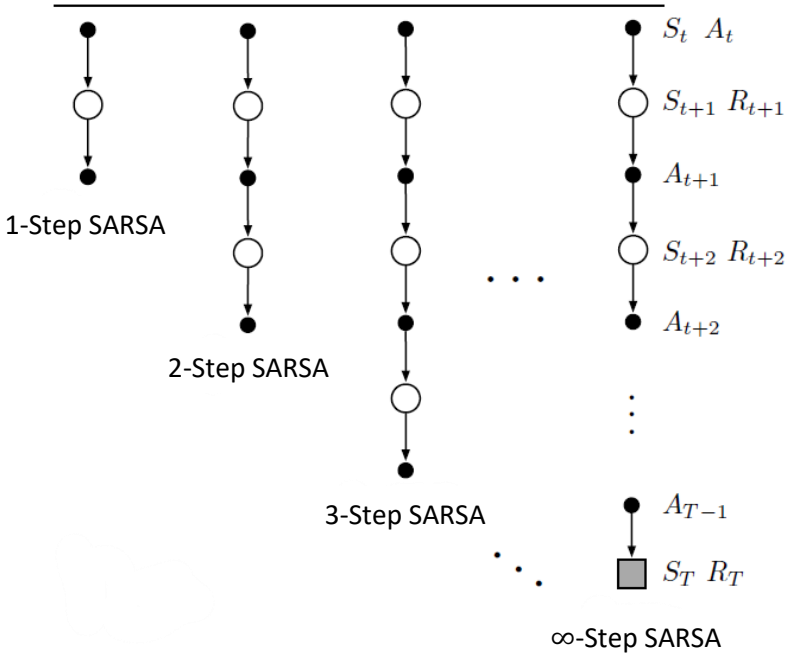
In control task, the objective is to find an optimal policy rather than evaluating a policy. SARSA implements a $Q(s,a)$ value-based generalized policy iteration (GPI) and naturally follows as an enhancement from the $\epsilon - greedy$ policy improvement step of MC control.

We meet also here the familiar two steps:

1. The first is a technique for learning the Q-function via TD-learning that we have seen in the prediction section. This is similar to evaluating a policy.
2. The second is a method for evolving the policy using the learned Q-function.

n-Step SARSA Algorithm

To solve the disadvantage of biasedness, we can use more than one step to perform an update. In this way, we can balance the biasness and variance trade-off, which affects Monte Carlo with generalized policy iteration and SARSA (0). Moreover, by taking multiple steps it can lead to comparatively faster convergence than SARSA (0).



While conceptually this is not so difficult, an algorithm for doing n-step learning needs to store the rewards and observed states for n steps, as well as keep track of which step to update. An algorithm for n-step SARSA is shown below.

Input : MDP $M = \langle S, s_0, A, P_a(s' \mid s), r(s,a,s') \rangle$ number of steps n
Output : Q-function Q

Initialise Q arbitrarily; e.g., $Q(s,a) = 0$ for all s and a

repeat
  Select action a to apply in s using Q-values in Q and
    a multi-armed bandit algorithm such as ϵ-greedy
  $\vec{s} = \langle s \rangle$
  $\vec{a} = \langle a \rangle$
  $\vec{r} = \langle \rangle$
  while $\vec{s}$ is not empty **do**
    if s is not a terminal state **then**
      Execute action a in state s
      Observe reward r and new state s'
      $\vec{r} \leftarrow \vec{r} + \langle r \rangle$
      if s' is not a terminal state **then**
        Select action a' to apply in s' using Q and a multi-armed bandit algorithm
        $\vec{s} \leftarrow \vec{s} + \langle s' \rangle$
        $\vec{a} \leftarrow \vec{s} + \langle a' \rangle$
    if $|\vec{r}| = n$ **or** s is a terminal state **if**
      $G \leftarrow \sum_{i=0}^{|\vec{r}|-1} \gamma^i \vec{r}_i$
      if s is not a terminal state **then**
        $G \leftarrow G + \gamma^n Q(s', a')$
      $Q(\vec{s}_0, \vec{a}_0) \leftarrow Q(\vec{s}_0, \vec{a}_0) + \alpha[G - Q(\vec{s}_0, \vec{a}_0)]$
      $\vec{r} \leftarrow \vec{r}_{[1:n+1]}$
      $\vec{s} \leftarrow \vec{s}_{[1:n+1]}$
      $\vec{a} \leftarrow \vec{a}_{[1:n+1]}$
    $s \leftarrow s'$
    $a \leftarrow a'$
until Q converges

This is similar to standard SARSA, except that we are storing the last n states, actions, and rewards; and also calculating the rewards on the last five rewards rather than just one. The variables s→, a→, and r→ as the list of the last n states, actions, and rewards respectively. We use the syntax s→i to get the ith element of the list, and the Python-like syntax s→[1:n+1] to get the

elements between indices 1 and n+1 (remove the first element).

As with SARSA and Q-learning, we iterate over each step in the episode. The first branch simply executes the selected action, selects a new action to apply, and stores the state, action, and reward.

It is the second branch where the actual learning happens. Instead of just updating with the 1-step reward r, we use the n-step reward G. This requires a bit of "book-keeping". The first thing we do is calculate G. This simply sums up the elements in the reward sequence $r\rightarrow$, but remembering that they must be discounted based on their position in $r\rightarrow$. The next line adds the TD-estimate $\gamma^n Q(s',a')$ to G, but only is the most recent state is not a terminal state. If we have already reached the end of the episode, then we must exclude the TD-estimate of the future reward, because there will be no such future reward. Of importance, also note that we multiple this by γn instead of γ. Why? This because the future estimated reward is n steps from state $s\rightarrow 0$. The n−step reward in G comes first. Then we do the actual update, which updates the state-action pair $(s\rightarrow 0, a\rightarrow 0)$ that is n-steps back.

The final part of this branch removes the first element from the list of states, actions, and rewards, and moves on to the next state.

What is the effect of all of the computation? This algorithm differs from standard SARSA as follows: it only updates a state-action pair after it has seen the next n rewards that are returned, rather than just the single next reward. This means that there are no updates until the nth steps of the episode; and no TD-estimate of the future reward in the last n steps of the episode.

Computationally, this is not much worse than 1-step learning. We need to store the last n states, but the per-step computation is small and uniform for n-step, just as for 1-step.

Backward View of SARSA (λ)

How can eligibility traces be used not just for prediction, as in TD(λ), but for control? As usual, the main idea of one popular approach is simply to learn action values, $Q_t(s,a)$, rather than state values, $V_t(s)$. In this lab, we analyze how eligibility traces can be combined with Sarsa in a straightforward way to produce an on-policy TD control method. The eligibility trace version of Sarsa we call *Sarsa*(λ), and the original version presented in the previous lab we henceforth call *one-step Sarsa* or SARSA (0).

The idea in Sarsa(λ) is to apply the TD(λ) prediction method to state-action pairs rather than to states. Obviously, then, we need a trace not just for each state, but for each state-action pair. Let $e_t(s,a)$ denote the trace for state-action pair s,a. Otherwise the method is just like TD(λ), substituting state-action variables for state variables--$Q(s,a)$ for $V(s)$ and $e_t(s,a)$ for $e_t(s)$:

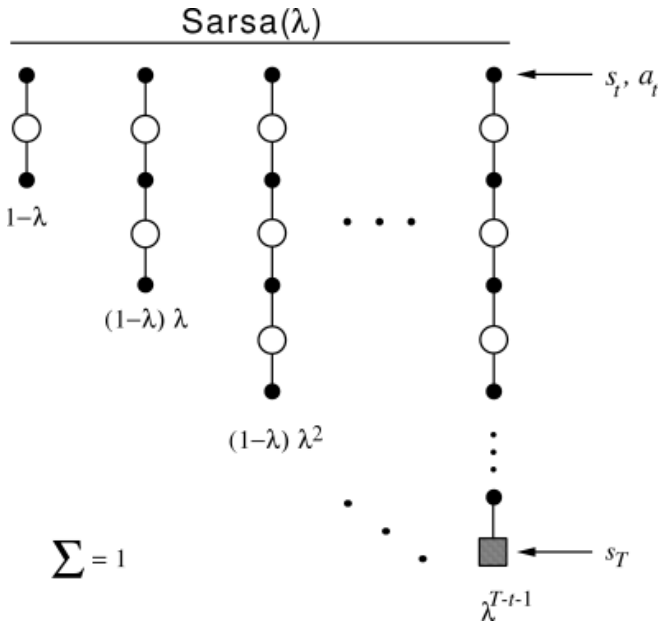
$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a), \qquad \text{for all } s,a$$

where

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$

and

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise.} \end{cases} \qquad \text{for all } s, a$$



The above figure shows the backup diagram for SARSA (λ). Notice the similarity to the diagram of the TD(λ) algorithm from the previous lab. The first backup looks ahead one full step, to the next state-action pair, the second looks ahead two steps, and so on. A final backup is based on the complete return. The weighting of each backup is just as in TD(λ) and the λ-return algorithm.

One-step Sarsa and Sarsa(λ) are on-policy algorithms, meaning that they approximate $Q^{\pi}(s,a)$, the action values for the current policy, π, then improve the policy gradually based on the approximate values for the current policy. The policy improvement can be done in many different ways, as we have seen throughout this book. For example, the simplest approach is to use the ε-greedy policy with respect to the current action-value estimates. The following figure shows the complete SARSA (λ) algorithm for this case.

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, a
Repeat (for each episode):
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all s, a:
      $Q(s,a) \leftarrow Q(s,a) + \alpha \delta e(s,a)$
      $e(s,a) \leftarrow \gamma \lambda e(s,a)$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal

Lab Tasks

- Learning rate, alpha, α = 0.1
- Discount factor, gamma, γ = 0.9
- Exploration factor, epsilon, ε = 0.2
- Lambda, λ = 0.85
- Number of Episodes = 20000

1. **Solve the Cart Pole problem given in the Gymnasium library using the n-step SARSA algorithm. You have to simulate the episodes using n-step SARSA algorithm and generalized policy iteration and see its convergence. Remember, you have to implement epsilon greedy policy improvement to pick actions for each state. Plot the returns for episodes for n=0, 2 and 4.**

2. **Solve the Pendulum problem given in the Gymnasium library using the SARSA (λ) algorithm. You have to simulate the episodes using SARSA (λ) algorithm and generalized policy iteration and see its convergence. Remember, you have to implement epsilon greedy policy improvement to pick actions for each state. Plot the returns for episodes.**

Deliverable:

Please submit your notebook on LMS before the deadline **(19th November 2024, 11:59pm).**

Lab Rubrics

Assessment	Does not meet expectation (1/2 marks)	Meets expectation (3/4 marks)	Exceeds expectation (5 marks)
Software Problem Realization **(CLO1, PLO1)**	The student struggles to formulate the problem as RL and does not apply SARSA algorithm to solve it. There is a lack of understanding of the problem's requirements and no attempt to find the optimal policy effectively.	The student formulates the problem as RL with some guidance, applies SARSA algorithm with hints, and shows it's working. However, the approach might not be fully optimized or lacks a thorough justification.	The student independently formulates the given problem as RL, applies SARSA algorithm without guidance, and effectively find an optimal policy. The approach is fully optimized and can be applied to different similar problems.
Software Tool Usage **(CLO4, PLO5)**	Code has syntax errors, and the implementation of the SARSA (0) algorithm is incorrect or incomplete. The code is not modular and lacks comments for readability and reuse. The student shows limited ability to use gymnasium library functions where required.	The student has implemented the SARSA algorithm correctly for the given problem with minor mistakes. The code is mostly correct in terms of syntax and functionality but might not be optimized or well-structured for reuse. Some documentation is provided. The student also shows working knowledge of gymnasium library where required.	The student has implemented the SARSA algorithm efficiently and correctly. The code is clean, modular, well-documented, and follows best practices for reproducibility and reuse. The student demonstrates full command of the gymnasium library and its functions.