COMPARISON OF EPISODIC SEMI GRADIENT N STEP SARSA AND TRUE ONLINE SARSA

Tengzhi Zhuo
UMass Amherst
tzhuo@umass.edu

Haochen Ren
UMass Amherst
haochenren@umass.edu

1. Description of Alogorithms

1.1. the Episodic Semi-Gradient n-step SARSA algorithm

This n-step algorithm is implemented based on the combination of the Episodic Semi-Gradient SARSA and the n-step SARSA algorithms. In contrast to the original SARSA algorithm, which optimizes the action value function $\hat{q}(s, a)$, the Episodic Semi-Gradient SARSA algorithm approximates the action-value function $\hat{q}(s, a, w)$, where $w \in \mathbb{R}^d$ is the weight vector. To determine the optimal policy for continuous problems, such as Mountain Car, we need to take a few steps backward to reconsider the values. Therefore, a semi-gradient update for w is given by:
$w_{t+1} = w_t + \alpha[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t)]$
$*\nabla\hat{q}(S_t, A_t, w_t)$
Just like the original SARSA algorithm, we can select the action greedily:

$$A_{t+1} = argmax_a \hat{q}(S_{t+1}, a, w_t)$$

Building upon this implementation, the Episodic Semi-Gradient n-step SARSA also incorporates the n-step SARSA. The idea is to commence updating the weights only after n steps. This approach prevents the low variance of one-step updates and encourages more explorations to learn about the environment.
With the combination of Semi-Gradient and n-step methods, the Episodic Semi-Gradient n-step SARSA algorithm can effectively approximate optimal policies for continuous environments such as Mountain Car and Cart Pole. It demonstrates faster convergence speed and improved performance.

1.2. the True Online SARSA(λ) algorithm

For the n-step return algorithms, the idea is to update the weights by the sum of the first n rewards with the state/action value after n steps[2]. The λ return algorithms offer a better computational efficiency with the eligibility trace $z_t \in \mathbb{R}^d$, the short term memory vector, that parallels the weight vector w_t, the long term memory vector. The eligibility trace serves as a mechanism to track the influence of past states on the current weight updates. When a part of w_t computes the estimated value, the related component of the z_t is temporarily increased and then gradually diminishes. The rate of decay is influenced by the trace-decay parameter $\lambda \in [0, 1]$. Therefore, the weights for SARSA(λ) are updated as:

$$\delta_t = R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t)$$

$$z_t = \gamma\lambda z_{t-1} + \nabla\hat{q}(S_t, A_t, w_t)$$

$$w_{t+1} = w_t + \alpha\delta_t z_t$$

where δ_t is the action-value of the TD error.
The True Online SARSA(λ) algorithm is implemented based on the SARSA(λ) algorithm. In contrast, SARSA(λ) has an input function $F(s, a)$ that returns the set of indices of active features for (s, a). The True Online version has a function $x_t = x(S_t, A_t) \rightarrow \mathbb{R}^d$ that directly outputs the state-action feature vector. This modification significantly reduces computational complexity, and the performance of accumulating and replacing traces also increases.[2]

1.3. Grid Tiling

We implemented these two algorithms to solve the Mountain Car and Cart Pole problems. Both problems have continuous states and discrete actions. To update the weights and calculate action values for continuous states, it is necessary to transform states and actions into unique discrete features. We implemented our Grid Tiling system based on the textbook[2] and the tile-coding software[1]. Each state, such as [position, velocity], contains float variables. Our goal is to convert these floats, associated with unique actions, into their feature representation with a shape corresponding to the number of tilings/features. Additionally, we aim to ensure that similar states and actions have close feature representations.

2. Pseudocode for methods

2.1. the Episodic Semi-Gradient n-step SARSA algorithm

Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer n
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or ε-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | 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 $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{q}(S_\tau, A_\tau, \mathbf{w})\right] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T - 1$

2.2. the True Online SARSA algorithm(λ)

True online Sarsa(λ) for estimating $\mathbf{w}^\top \mathbf{x} \approx q_\pi$ or q_*

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$, small $\varepsilon > 0$
Initialize: $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or ε-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
  $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
  $\mathbf{z} \leftarrow \mathbf{0}$
  $Q_{old} \leftarrow 0$
  Loop for each step of episode:
  | Take action A, observe R, S'
  | Choose $A' \sim \pi(\cdot|S')$ or ε-greedy according to $\hat{q}(S', \cdot, \mathbf{w})$
  | $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
  | $Q \leftarrow \mathbf{w}^\top \mathbf{x}$
  | $Q' \leftarrow \mathbf{w}^\top \mathbf{x}'$
  | $\delta \leftarrow R + \gamma Q' - Q$
  | $\mathbf{z} \leftarrow \gamma\lambda \mathbf{z} + \left(1 - \alpha\gamma\lambda \mathbf{z}^\top \mathbf{x}\right) \mathbf{x}$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
  | $Q_{old} \leftarrow Q'$
  | $\mathbf{x} \leftarrow \mathbf{x}'$
  | $A \leftarrow A'$
  until S' is terminal

3. Hyper-parameter Tuning

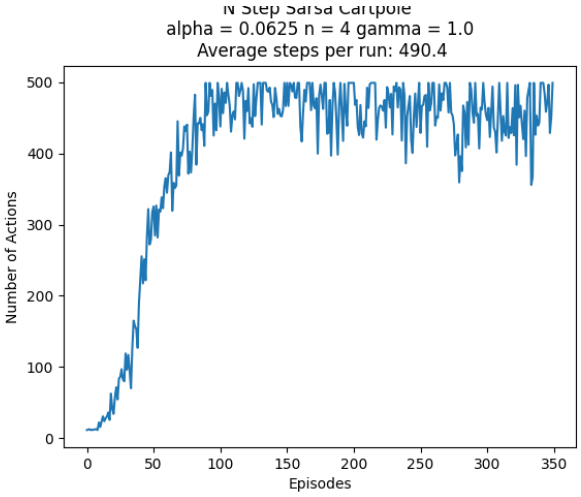
3.1. the Episodic Semi-Gradient n-step SARSA algorithm

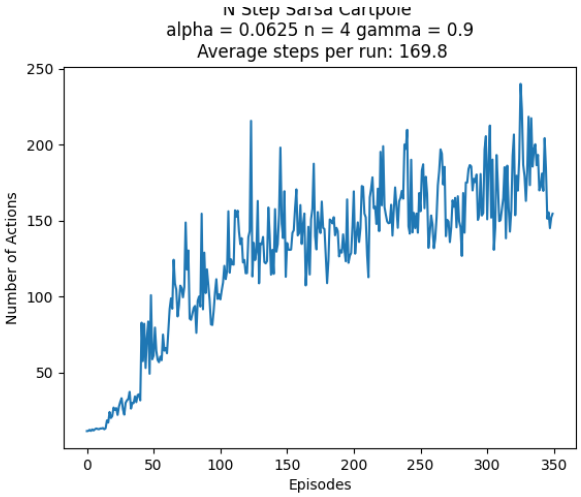
3.1.1 Cartpole

There are three hyper parameters that can be changed in this algorithm: alpha, n, and gamma. Alpha stands for the step size, which should be a number greater than 0. N stands for the "n" in "n-step", which should be a positive integer. And gamma is the discount rate, which should be a number between 0 and 1, and determine how much the agent cares about the reward in the future. In our process of hyper-parameter tuning, the following values were tested:

Alpha: $[0.3/8, 0.2/8, 0.5/8]$
N: $[4, 6, 8]$
Gamma: $[0.9, 0.95, 1.0]$

For each combination of these hyper-parameters, we ran the algorithms 10 times and take the average of the performances. The graph below are some graphs that we plot, showing the difference between good hyper-parameters and bad hyper-parameters.





We first implemented the algorithm in the Cartpole environment. These two graphs show the different curves that we get while changing the value of gamma and keep other parameters unchanged. Both graphs have number of episodes as their x-axis and number of actions(game time) as their y-axis. We can tell that the curve for gamma = 1.0 converges at a relatively high value of 500, while the curve for gamma = 0.9 converges at a lower value of around 150. Since this is a cartpole environment, we

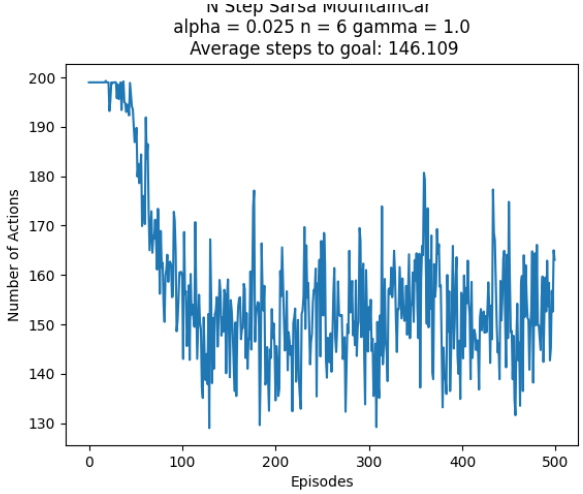
know that the more rounds the cart survive, the better our algorithm performs. Thus, we know that gamma = 1.0 is a better hyper-parameter than gamma = 0.9, while keeping others unchanged. By plotting many graphs like these and comparing their average steps per run, we finally figured out the best hyper-parameters for this algorithm and environment. The best hyper-parameters and the best performance plot will be shown in the result section.

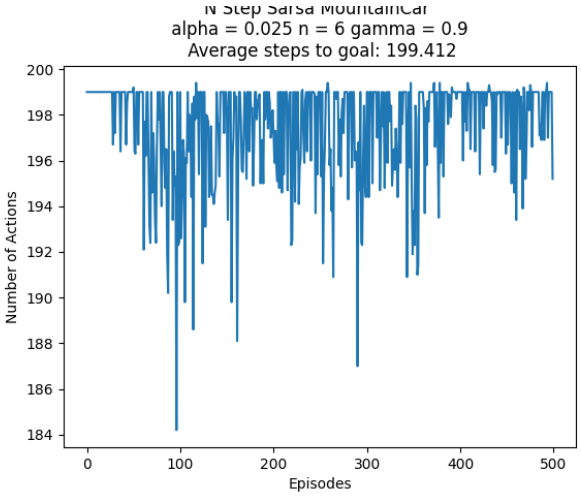
3.1.2 Mountain Car

Same as artpole environment, there are three hyper parameters that can be modified: alpha, n, and gamma. Alpha stands for the step size, which should be a number greater than 0. N stands for the "n" in "n-step", which should be a positive integer. And gamma is the discount rate, which should be a number between 0 and 1, and determine how much the agent cares about getting the reward in the future. In our project, the following values were tested:

Alpha: $[0.2/8, 0.3/8, 0.5/8]$
N: $[4, 6, 8]$
Gamma: $[0.9, 0.95, 1.0]$

For each of the combination of these hyper-parameters, we ran the algorithm 10 times and take the average performance of all cases. Graphs below are two graphs that we plotted.





In these two graphs, while we keep all other factors the same, the top one has a gamma of 1.0, and the bottom one has a gamma of 0.9. Both graphs were plotted by taking number of episodes as the x-axis and taking number of actions(game time) as the y-axis. In the mountain car environment, number of actions should be minimized to reach to the optimal solution. Thus, the less number of actions, the better the hyper-parameters. Based on this, we can say that the algorithm performs better when gamma = 1.0 than gamma = 0.9. For the top graph, it converges around 150, thus meaning that the algorithm has an average time of 150 to reach the mountain top. For the bottom graph, it is not learning anything and doesn't converge. Thus, we know that gamma 1.0 is better. For each combination of hyper-parameters, we compare their performance by plots and average steps to goal, and finally figured out the best set of hyper-parameters. The best hyper-parameters and the plot will be disccused in the result section later.

3.2. the True Online SARSA algorithm(λ)

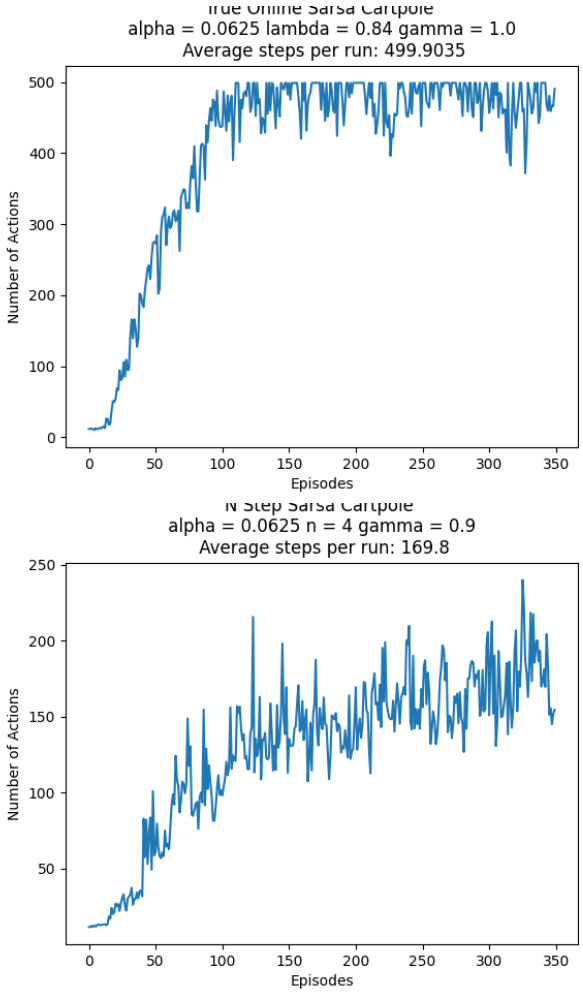
3.2.1 Cartpole

There are three hyper-parameters that can be modified in this algorithm: lambda, alpha, and gamma. Lambda stands for the trace decay rate, which should be a number between 0 and 1. Alpha is the step size, which should be a positive number. Gamma stands for the discount rate, which indicates how much the agent cares about getting rewards in the future. Gamma should be a number between 0 and 1. In our project, the following values of hyper-parameters were tested:

Lambda: $[0.84, 0.87, 0.92, 0.96]$
Alpha: $[0.2/8, 0.3/8, 0.5/8]$
Gamma: $[0.9, 0.95, 1.0]$

For each combination of parameters, we ran the algorithm 10 time and take the average of their performances.

Below are two graphs that we plot during the process.



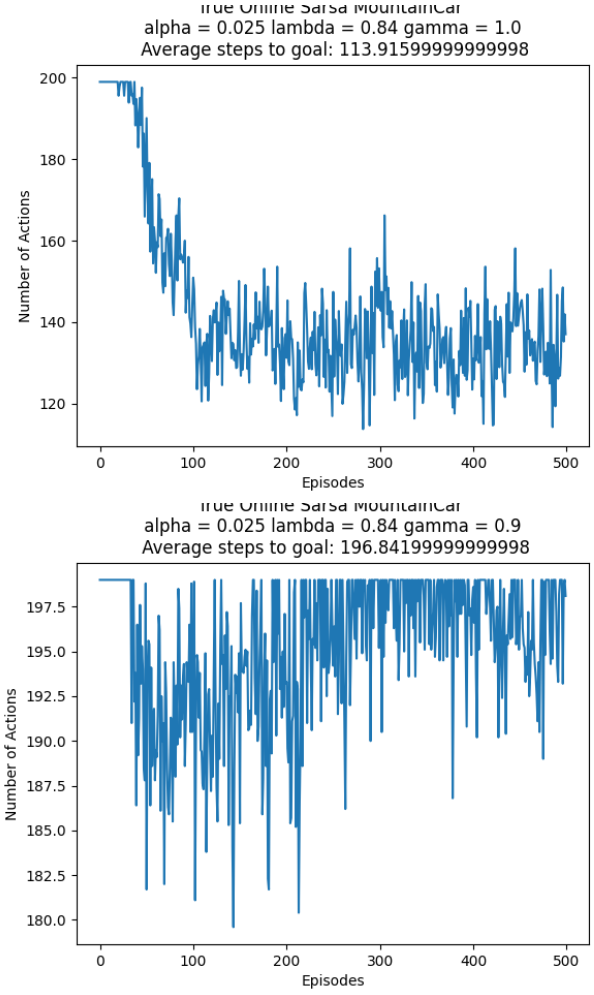
The top graph shows a convergence at number of actions equals 500, indicating that the algorithm is nearly optimal and is able to maintain the pole for over 500 steps. The bottom graph shows a convergence of around 170. Therefore, we know that gamma = 1.0 has a better performance than gamma = 0.9 while we keep other factors the same. By comparing all combinations of parameters using graphs and their value of average steps per run, we completed the hyper-parameter tuning and found out the optimal set of hyper-parameters. We will show our research result in the next section.

3.2.2 Mountain Car

Same as in the Cartpole environment, there are three hyper-parameters that we can modify: lambda, alpha, and gamma. Lambda is the trace decay rate between 0 and 1. Alpha is a positive number and it stands for the step size. Gamma stands for the discount rate and is always between 0 and 1. The following values were tested in our project:

Lambda = $[0.84, 0.92, 0.96]$
Alpha = $[0.2/8, 0.3/8, 0.5/8]$
Gamma = $[0.9, 0.95, 1.0]$

For each combination of hyper-parameters, we ran our algorithms 10 times and take the average performance. Graphs below are two examples with different hyper-parameters.



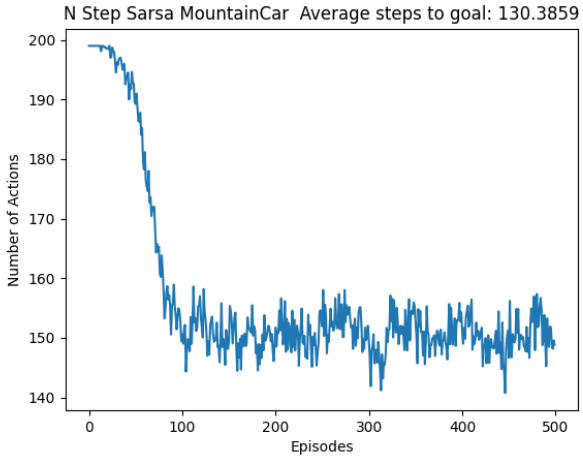
The top graph converges at around 130, which has a gamma = 1.0. The bottom graph doesn't converge and its value is never below 180, which means that it requires more than 180 actions to reach the mountain top. By having other factors fixed, we can tell that gamma = 1.0 has a better performance than gamma = 0.9. By comparing graphs like these and also comparing their average steps to goal, we finally found out the optimal set of hyper-parameters in our model. We will report our result in the next section.

4. Results

After determining the optimal hyperparameters, we applied them to train each algorithm on each environment 100 times. Additionally, we generated the average learning curve for each experiment for the purpose of comparison. In the learning curves, the x-axis represents the number of episodes, while the y-axis denotes the number of actions or steps taken in each episode. You can also see "Average Steps to goal" on the title for each plot. This is the evaluation result after training the weights. The evaluation was conducted 200 times, and we selected the average to showcase the performance.

4.1. the Episodic Semi-Gradient n-step SARSA algorithm

4.1.1 Mountain Car

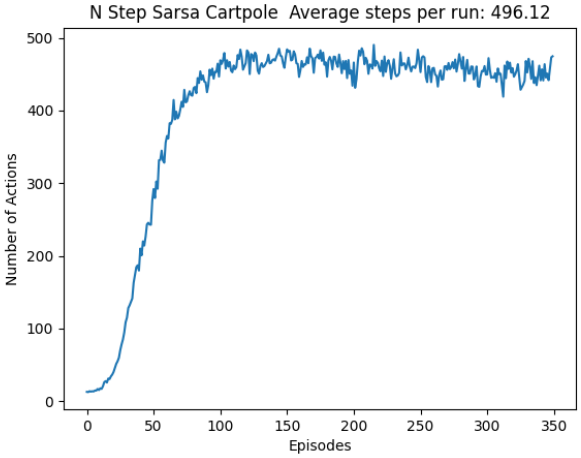


Best hyper-parameters:

Alpha = 0.025
N = 6
Gamma = 1.0

We can see that on average training, the agent starts to reach to the goal state more and more quickly for the first 100 episodes. After 100 episodes, the performance starts to converge around 150 steps. After training for 500 episodes, the agent averages a 130 steps to reach to the terminal state with this algorithm.

4.1.2 Cart Pole



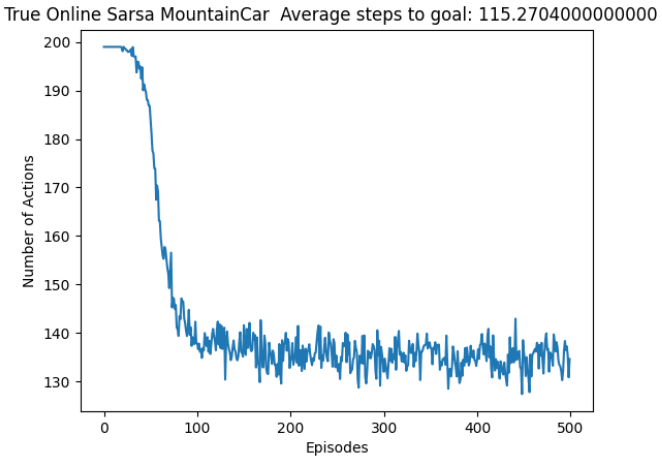
Best hyper-parameters:

Alpha = 0.0625
N = 4
Gamma = 1.0

For the first 125 episodes, the agent learns really quickly to stay longer without dropping. After 125 episodes, the performance starts to converge to 500, which is the optimal number of actions. After training, the agent averages a 496.12 steps without falling off with this algorithm.

4.2. the True Online SARSA algorithm(λ)

4.2.1 Mountain Car

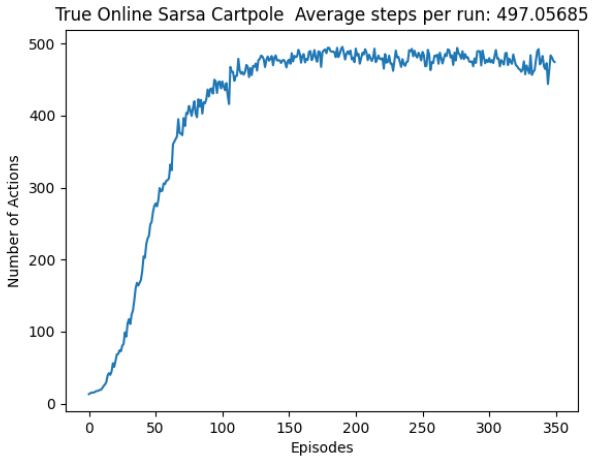


Best hyper-parameters:

Alpha = 0.025
Lambda = 0.84
Gamma = 1.0

On average training, the agent is able to improve itself rapidly to reach the terminal state with fewer states for the first 100 episodes. After 100 episodes, the performance starts to converge around 140 steps. For evaluations, the agent is able to score an average of 115 steps to reach to the goal state with this algorithm.

4.2.2 Cart Pole



Best hyper-parameters:

Alpha = 0.0625
Lambda = 0.84
Gamma = 1.0

The agent is able to learn really quickly for the first 125 episodes to stay longer without falling off. After 125 episodes, the performance converges to the optimal steps 500. For the evaluations, the agent is able to score an average of 497 steps without dropping.

4.3. Comparison

By looking at the plots and the average scores of each algorithm for each environment, we can conclude that the True Online SARASA algorithm performs better than the Episodic Semi-Gradient n-step SARSA algorithm. For both environments learning curves, we can clearly see that the True Online SARSA have a more stable performances between episodes. We can also see that the True Online SARSA outputs better scores, 115 steps to goal, than the Episodic Semi-Gradient n-step SARSA algorithm. Moreover, we found out that it takes less time to run episodes for the True Online SARSA than the other. We believe that this happens because the True Online SARSA is able to get to the optimal weight with a higher stability. Because of this, it requires fewer actions/steps for each episode, which reduces computational complexity.

5. Group work

5.1. Code

We worked together on the whole process of this project. In the first stage, we figured out which two algorithms to use and which environments to apply. Then for the coding part, Tengzhi wrote the main structure for the algorithm, while Haochen helped to debug and fix potential problems.

5.2. Report

Tengzhi worked on section 1 and 4 of this final report, while Haochen took the rest of the work.

References

[1] Richard S. Sutton. Tile coding software – reference manual, version 3.0.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction(2nd ed.)*. 2018.