687 Final Project

Vayunandhan Reddy Yekkaluri, Venkata Sai Phanindra Pamidimukkala

December 2022

Introduction

As part of this project, we have implemented True Online SARSA(λ), One-Step Actor-Critic, and Episodic Semi-Gradient n-step SARSA algorithms and evaluated on Mountain Car, Acrobot, and Cart Pole MDPs.

Algorithms

1. True Online SARSA(λ) algorithm

It is a value function approximation algorithm that uses the concept of eligibility traces to produce an efficient backward-view algorithm.

This algorithm determines the new weight vector w_t at each step t during an episode, using only information available at time t. The mechanism uses a short-term memory vector, the eligibility trace z_t, that parallels the long-term weight vector w_t. The rough idea is that when a component of w_t participates in producing an estimated value, then the corresponding component of z_t is bumped up and then begins to fade away. Learning will then occur in that component of w_t if a nonzero TD error occurs before the trace falls back to zero. The trace-decay parameter λ determines the rate at which the trace falls.

Eligibility trace is a mechanism by which we can keep track of states/actions previously visited by the agent.

State feature function are represented with Fourier Basis of order-n which results in $d = (n+1)^k$ features (k - number of variables used to represent a state).

Pseudo Code:

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

2. One-Step Actor-Critic algorithm

One step actor critic is the policy gradient method which learns a parameterized policy that can select actions without consulting a value function. The value function is used to learn the policy parameter, but not used for action selection. Actor is a reference to the learned policy, and critic refers to the learned value function. In the policy gradient methods, we use the soft-max in action preference rather than ϵ greedy. Because, the approximate policy using soft max approached a deterministic policy where as the action selection over action values using ϵ greedy always has ϵ probability of selecting a random action. So, we are using the soft-max while picking the actions.

In the algorithm,

1. Initialise both value function and policy with features of order-n which results in $d = (n + 1)^k$ features (k - number of variables used to represent a

state).

2. Initialise the weights of size ($d = (n+1)^k$)

Equations for parameterized value function:

$$\hat{v}_w(s) = w^T \phi(s) = \sum_{i=1}^{d} w_i \phi_i(s)$$

$\phi(s)$ - value feature representation and w is weights attached

Equations for parameterized policy function:

$$\pi(a|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}$$
$$h(s,a,\theta) = \theta^T x(s,a)$$

h(s, a, θ) is the parameterized numerical preference for a state-action pair. For discrete action space, this is the common way of parameterization. $x(s,a)$ is the feature vectors. And θ are the weights

3. α^θ and α^w are the step sizes

4. Using these weights and feature vectors we running the episodes initializing the starting state randomly and updating the weights at every timestep. At a timestep, we are picking an action using the current state(S). The action is picked by calculating the parameterized numerical preference $h(s,a,\theta)$ for each action and using the soft-max to calculate the probabilities and based on those probabilities we are picking an action. Based on this action, we are getting into new state and a reward is obtained. Using the reward and current state(S) and next state(S'), we calculate the temporal difference(δ). We gonna update the weights of value approximation with the step size(α),(δ), and gradient of valuation approximation. Similarly to the weights for policy approximation.

Based on step size(α^θ and α^w) and features order (n), the policy will be learnt. With the right values for those, the policy will be learnt quicker and with very lesser deviation.

Pseudo Code:

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

3. Episodic Semi-Gradient n-step SARSA algorithm

Semi-Gradient n-step SARSA algorithm is the value approximation gradient method which is used to learn a parametermized value function $\hat{q}(s, a, w)$. In this algorithm, at a given timestep we update the weights by using the temporal difference which is difference of the n-step return of the state, action(S_τ, A_τ) and value obtained from parametermized value function $\hat{q}(S_\tau, A_\tau, w)$ where $\tau = t - n + 1$.

In the alogrithm,

1. Initialise both value function with features of order-n which results in $d = (features_n + 1)^k$ features (k - number of variables used to represent a state).

2. Initialise the weights of size $(d = (features_n + 1)^k)$

Equations for parametermized value function:

$$\hat{q}_w(s, a) = w^T \phi(s, a) = \sum_{i=1}^{d} w_i \phi_i(s, a)$$

3. α^w are the step sizes

4. storing state, action, reward for the corresponding timestep which will be used to update the weights

5. Using these weights and feature vectors we running the episodes initializing the starting state randomly and updating the weights when $\tau \geq 0$. At each timestep, an action is sampled from the ϵ - greedy with respect to $\hat{q}_w(s, a)$. Based on this action, we transit to next state and reward is obtained. We are storing the these values in their corresponding store. When timesteps(t) is greater than equal to n-1 steps, we start updating the weights. From this point, we calculate the return for the state S_{t-n+1} and action A_{t-n+1} corresponding all the rewards till the next n steps. Using this state we gonna update the weight.

Since we calculating the return based on next rewards, the variance will be less and method tends to learn quicker. Based on step size α^w), steps(n) and features order ($features_n$), the policy will be learnt. With the right values for those, the policy will be learnt quicker and with very lesser deviation

Pseudo Code:

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 $\tau = T - 1$

MDPs

1. Mountain Car

The Mountain Car MDP is a deterministic MDP that consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The goal of the MDP is to strategically accelerate the car to reach the goal state on top of the right hill.

State: $s = (x, v)$, where $x \in \mathbb{R}$ is the position of the car and $v \in \mathbb{R}$ is the velocity.

Actions: $a \in \{\texttt{reverse}, \texttt{neutral}, \texttt{forward}\}$. These actions are mapped to numerical values as follows: $a \in \{-1, 0, 1\}$.

Dynamics:

$$\begin{aligned} v_{t+1} &= v_t + 0.001a_t - 0.0025\cos(3x_t) \\ x_{t+1} &= x_t + v_{t+1}. \end{aligned}$$

After the next state, $s' = [x_{t+1}, v_{t+1}]$ has been computed, the value of x_{t+1} is clipped so that it stays in the closed interval $[-1.2, 0.5]$. Similarly, the value v_{t+1} is clipped so that it stays in the closed interval $[-0.07, 0.07]$. If x_{t+1} reaches the left bound (i.e., the car is at $x_{t+1} = -1.2$), or if x_{t+1} reaches the right bound (i.e., the car is at $x_{t+1} = 0.5$), then the car's velocity is reset to zero: $v_{t+1} = 0$. This simulates inelastic collisions with walls at -1.2 and 0.5.

Terminal States: If $x_t = 0.5$, then the state is terminal (it always transitions to s_∞). The episode may also terminate due to a timeout; in particular, it terminates if the agent runs for more than 1000 time steps.

Rewards: $R_t = -1$ always, except when transitioning to s_∞ (from s_∞ or from a terminal state), in which case $R_t = 0$.

Discount: $\gamma = 1.0$.

Initial State: $S_0 = (X_0, 0)$, where X_0 is an initial position drawn uniformly at random from the interval $[-0.6, -0.4]$.

2. Acrobot

The Acrobot MDP consists of two links connected linearly to form a chain, with one end of the chain fixed. The joint between the two links is actuated. The goal is to apply torques on the actuated joint to swing the free end of the linear chain above a given height while starting from the initial state of hanging downwards.

State: $s = (\theta 1, \theta 2, \mathit{Angular\_velocity\_of\_}\theta 1, \mathit{Angular\_velocity\_of\_}\theta 2)$, where $\theta 1$ is the angle of the first joint, where an angle of 0 indicates the first link is pointing directly downwards and $\theta 2$ is relative to the angle of the first link. An angle of 0 corresponds to having the same angle between the two links.

Actions: $a \in \{0, 1, 2\}$. The action represents the torque applied on the actuated joint between the two links.

Dynamics: Next state is determined using the gym library.

Terminal States:
The free end reaches the target height, which is constructed as: $-\cos(\theta 1) - \cos(\theta 2 + \theta 1) > 1.0$
The episode length is greater than 500.

Rewards: $R_t = -1$ always, except when transitioning to s_∞ (from s_∞ or from a terminal state), in which case $R_t = 0$.

Discount: $\gamma = 1.0$.

Initial State: $S_0 = (\theta 1, \theta 2, Angular\_velocity\_of\_\theta 1, Angular\_velocity\_of\_\theta 2)$, where all values are drawn uniformly at random from the interval $[-0.1, 0.1]$.

3. Cart Pole

In Cart Pole MDP a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

State: $s = (x, v, \theta, \theta^{\cdot})$, where x denotes cart position, v denotes cart velocity, θ denotes pole angle, $\theta^{\cdot}$ denotes pole angular velocity
$x \in [-4.8, 4.8]$
$v \in [-5, 5]$
$\theta \in [-0.418, 0.418]$
$\theta^{\cdot} \in [-4, 4]$
Note: The true range of v, $\theta^{\cdot}$ is [-∞, ∞] as mentioned in the gym library. While running experiments the values for v, $\theta^{\cdot}$ never exceeded the above-mentioned ranges, so we are using these ranges.

Actions: $a \in \{0, 1\}$. The action indicates the direction of the fixed force the cart is pushed with.

Dynamics: Next state is determined using the gym library.

Terminal States:
$|\theta| \geq 0.2095$
$|x| \geq 2.4$
The episode length is greater than 500.

Rewards: $R_t = -1$ always, except when transitioning to s_∞ (from s_∞ or from a terminal state), in which case $R_t = 0$.

Discount: $\gamma = 1.0$.

Initial State: $S_0 = (x, v, \theta, \theta^{\cdot})$, where all values are drawn uniformly at random from the interval $[-0.05, 0.05]$.

Experiments & Results

1. True Online SARSA(λ) algorithm

(a) Mountain Car

Design decisions made:
(i) Used $\lambda = 0.95$. The trace-decay parameter, λ determines the rate at which the TD error fades away. Whenever a TD error occurs it facilitates learning and a higher λ value in some sense stimulates looking much far into the future to determine the update and this facilitates learning. But for higher λ values ($\in$ (0.95, 1]) the algorithm's performance degraded sharply.

Hence we have used 0.95 for λ.

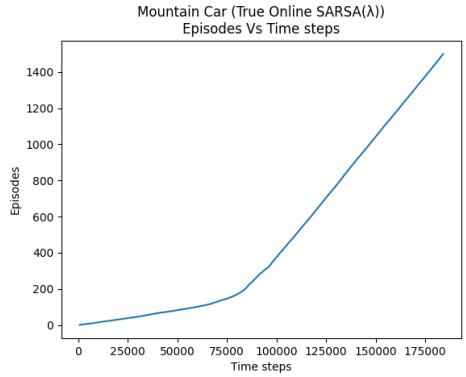
(ii) Used n=3 => $d = 16$ ($k = 2$) for the order of the Fourier Series used to represent the state. Experimented with multiple values and observed that for smaller values ($\in \{1, 2\}$) the algorithm is taking a larger number of episodes to converge also the learning is not smooth and as the order value increases the algorithm converges faster and the learning is a lot smoother. The convergence and learning are almost the same for higher order values (≥ 3). But as the order size increases the algorithm run time is spiking really fast and also the weights are exploding.
Hence we have used a reasonably higher value of 3 for n.

(iii) Used α=0.00001 (learning rate). It is a factor by which the newly observed value changes the current estimated value. As the λ value is higher the samples become unreliable and with a higher α value there is a lot of fluctuation in the learning curve. This can be prevented using smaller α values. Hence we have used a minute value of 0.00001 for α.

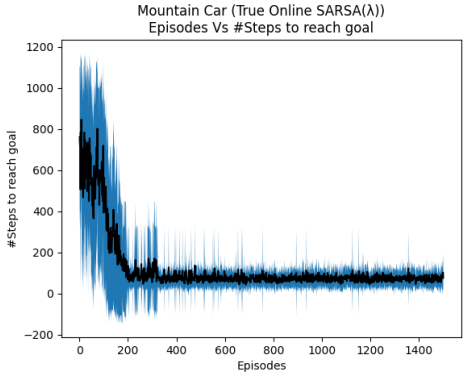
(iv) Used $\epsilon = 0.001$ in ϵ-greedy exploration and ϵ is fixed over time. According to the Mountain Car domain for a given $< x, v >$ and action the number of steps to reach the goal state is always the same as there is no stochasticity in the MDP. If we find out the true estimates of all actions there will not be much gain if we force the agent to explore, it is better to exploit. For higher values of ϵ (≥ 0.1), the learning curve is fluctuating a lot in the initial episodes and is also not stabilizing over time.
Hence we have used a minute value of 0.001 for ϵ.

(v) Used Number of Episodes = 1500. Experimented with various episode counts ranging from 500 to 3000, while using a smaller number of episodes the learning curve had high variance and the variance gradually decreased with an increase in the Number of Episodes and mostly stabilized after 1500 episodes.

Learning Curve:(computed over 20 runs)



Learning Curve:(computed over 20 runs)



(b) # Acrobot

Design decisions made:

(i) Used $\lambda = 0.95$. The trace-decay parameter, λ determines the rate at which the TD error fades away. Whenever a TD error occurs it facilitates learning and a higher λ value in some sense stimulates looking much far into the future to determine the update and this facilitates learning. But for higher λ values ($\in$ (0.95, 1]) the algorithm's performance degraded sharply.
Hence we have used 0.95 for λ.

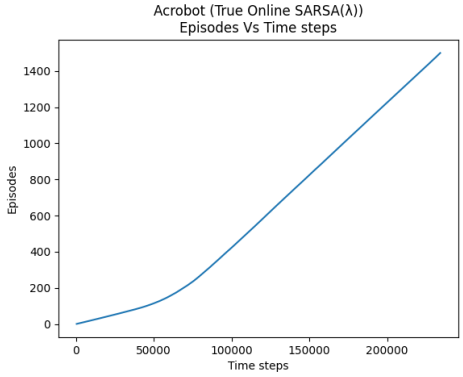
(ii) Used n=1 => $d = 64$ ($k = 6$) for the order of the Fourier Series used to represent the state. Experimented with multiple values and observed that for all values (≥ 1) the algorithm's convergence and learning are similar. Even though the convergence and learning are a little better at higher order values the algorithm run time is spiking really fast as order increases and also the weights are exploding.
Hence we have used a reasonable value of 1 for n.

(iii) Used α=0.00001 (learning rate). It is a factor by which the newly observed value changes the current estimated value. As the λ value is higher the samples become unreliable and with a higher α value there is a lot of fluctuation in the learning curve. This can be prevented using smaller α values. Hence we have used a minute value of 0.00001 for α.

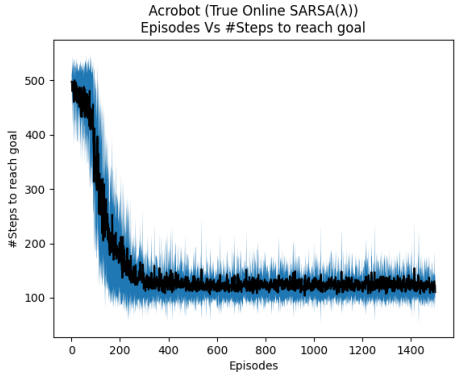
(iv) Used $\epsilon = 0.001$ in ϵ-greedy exploration and ϵ is fixed over time. According to the Acrobot domain for a given $< \cos(\theta 1), \sin(\theta 1), \cos(\theta 2), \sin(\theta 2), \theta 1, \theta 2 >$ and action the number of steps to swing the free end of the linear chain above a given height is always the same as there is no stochasticity in the MDP. If we find out the true estimates of all actions there will not be much gain if we force the agent to explore, it is better to exploit. For higher values of ϵ (≥ 0.1), the learning curve is fluctuating a lot in the initial episodes and is also not stabilizing over time.
Hence we have used a minute value of 0.001 for ϵ.

(v) Used Number of Episodes = 1500. Experimented with various episode counts ranging from 500 to 3000, while using a smaller number of episodes the learning curve had high variance and the variance gradually decreased with an increase in the Number of Episodes and mostly stabilized after 1500 episodes.

Learning Curve:(computed over 20 runs)



Learning Curve:(computed over 20 runs)



(c) **Cart Pole Design decisions made:**

(i) Used $\lambda = 0.9$. The trace-decay parameter, λ determines the rate at which the TD error fades away. Whenever a TD error occurs it facilitates learning and a higher λ value in some sense stimulates looking much far into the future to determine the update and this facilitates learning. But for higher λ values ($\in$ (0.9, 1]) the algorithm's performance degraded sharply.
Hence we have used 0.9 for λ.

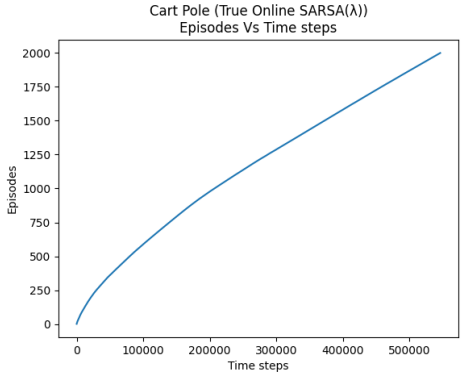
(ii) Used n=5 => $d = 1296$ ($k = 4$) for the order of the Fourier Series used to represent the state. Experimented with multiple values and observed that for smaller values (≤ 4) the algorithm's convergence and learning are poor an the learning is stabilizing in a local minima of around 200. Even though the convergence and learning are getting better at higher order values the algorithm run time is spiking really fast as order increases.
Hence we have used a reasonable value of 5 for n.

(iii) Used α=0.00001 (learning rate). It is a factor by which the newly observed value changes the current estimated value. As the λ value is higher the samples become unreliable and with a higher α value there is a lot of fluctuation in the learning curve. This can be prevented using smaller α values. Hence we have used a minute value of 0.00001 for α.

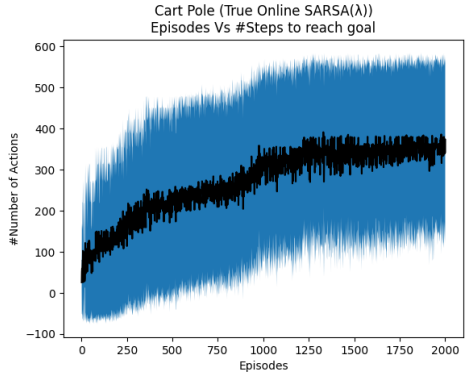
(iv) Used $\epsilon = 0.0001$ at the start in ϵ-greedy exploration and ϵ is decayed by a factor of 10 after every 100 episodes. We have observed that this particular MDP is more sensitive to the exploration rate(ϵ) and the algorithms convergence is significantly effected even if there is a small increase in ϵ.
Hence we have used a minute value of 0.0001 at the start for ϵ and it is decayed by a factor of 10 after every 100 episodes.

(v) Used Number of Episodes = 2000. Experimented with various episode counts ranging from 500 to 3000, while using a smaller number of episodes (≤ 1500) the algorithm is stuck in a local optima and also the learning curve had high variance. As the Number of Episodes increased the algorithm reached the global optima after 1750 episodes.

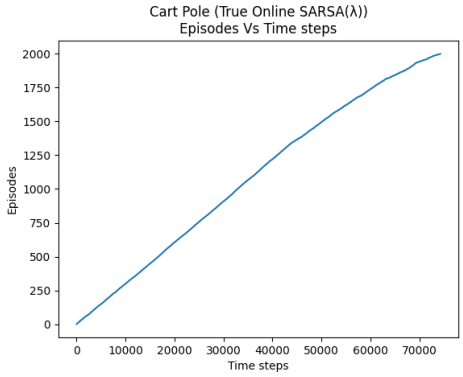
Learning Curve:(computed over 20 runs)

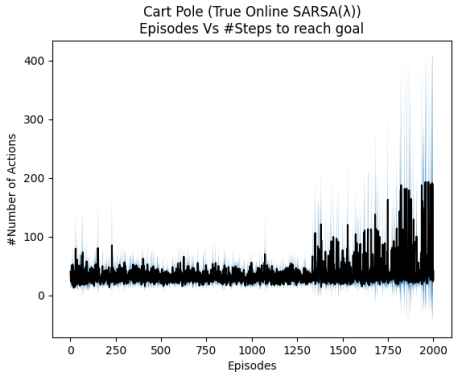


Learning Curve:(computed over 20 runs)

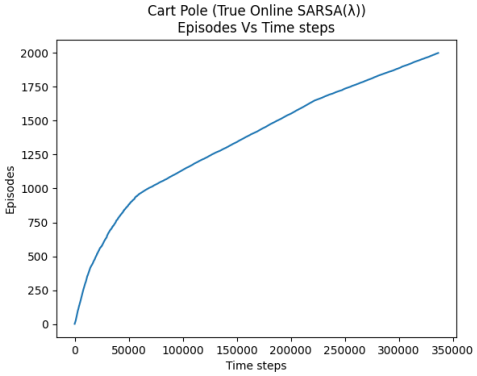


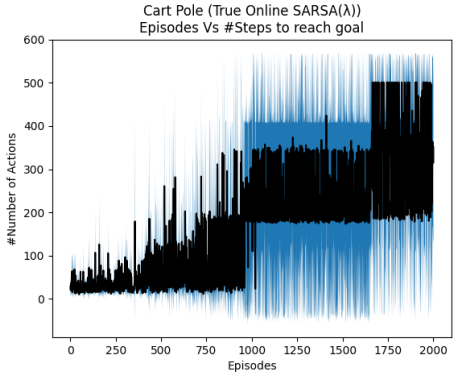
Observations:
Number of Episodes: 2000, α: 0.00001, ϵ: 0.01, λ: 0.9, d: 1296(n: 5)
Learning Curve:(computed over 20 runs) **Learning Curve:**(computed over 20 runs)





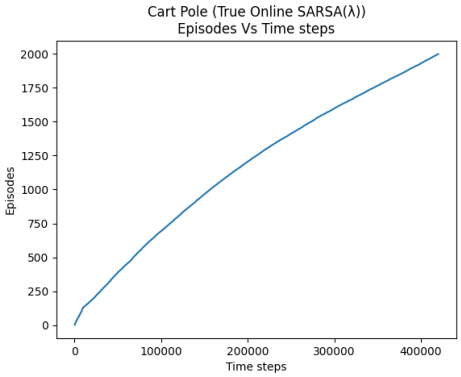
Number of Episodes: 2000, α: 0.00001, ϵ: 0.001, λ: 0.95, d: 1296(n: 5)
Learning Curve:(computed over 20 runs) **Learning Curve:**(computed over 20 runs)



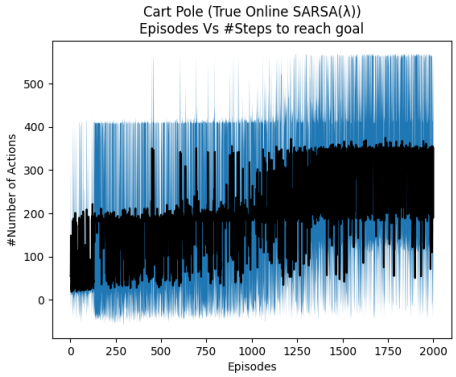


Number of Episodes: 2000, α: 0.00001, ϵ: 0.001, λ: 0.9, d: 256(n: 3)

Learning Curve:(computed over 20 runs)

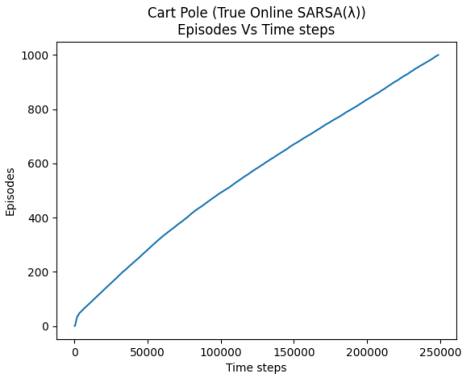


Learning Curve:(computed over 20 runs)

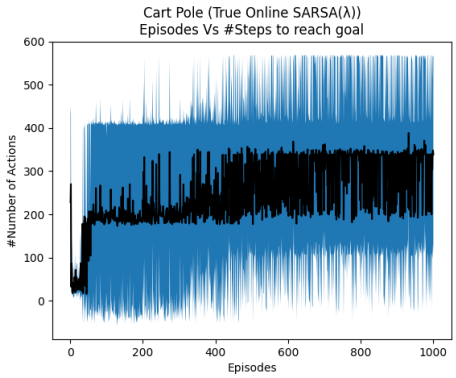


Number of Episodes: 1000, α: 0.00001, ϵ: 0.001, λ: 0.9, d: 256(n: 3)

Learning Curve:(computed over 20 runs)



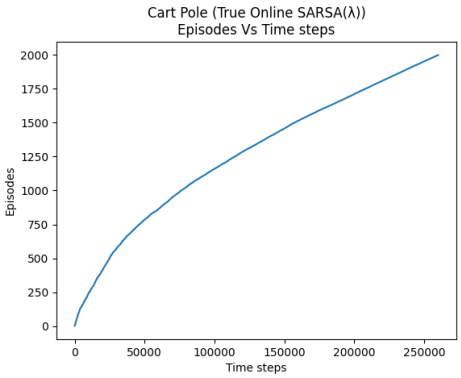
Learning Curve:(computed over 20 runs)

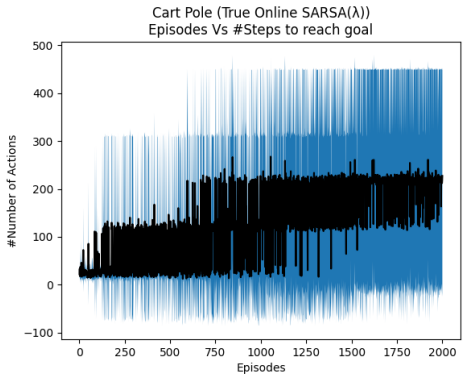


Number of Episodes: 2000, α: 0.000001, ϵ: 0.001, λ: 0.9, d: 256(n: 3)

Learning Curve:(computed over 20 runs)

Learning Curve:(computed over 20 runs)





2. One-Step Actor-Critic algorithm

We used soft max policy for picking an action for all the following MDP and with no exploration.

(a) Mountain Car

Design decisions made:

(i) Exploration Parameter (σ): Used $\sigma = 1$ for the soft max to calculate the action probabilities. The parameterized policy function contains σ and the finding the gradient of it would be tough. As specified in the class, to make the implementation easy we picked $\sigma = 1$

(ii) Discount Parameter (λ): Used discount parameter $\lambda = 1$ as specified in the MDP.

(iii) Feature Representation: Used Fourier series.

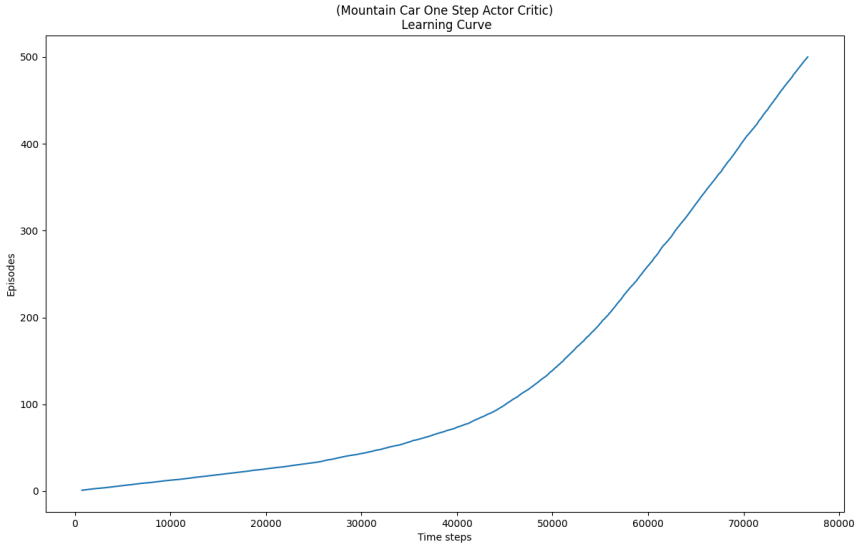
(iv) Features order (n) - As n increases, the feature representation increases which implies the weights to be updated also increases. Adding more features gives better representation of a state. But the time complexities will be increased. With very exploration rate and very high feature representation, it takes very long time to optimal one with lesser deviation. With the episodes = 500 and making other hyper parameters constant, I tried with different values of $n \in [1, 10]$. Since the MDP is simple, it reaches the optimal value after 100 episodes, but with higher n there is a lot of deviations. So I preferred taking order (n=1), to make computing faster and also it provided the best result.

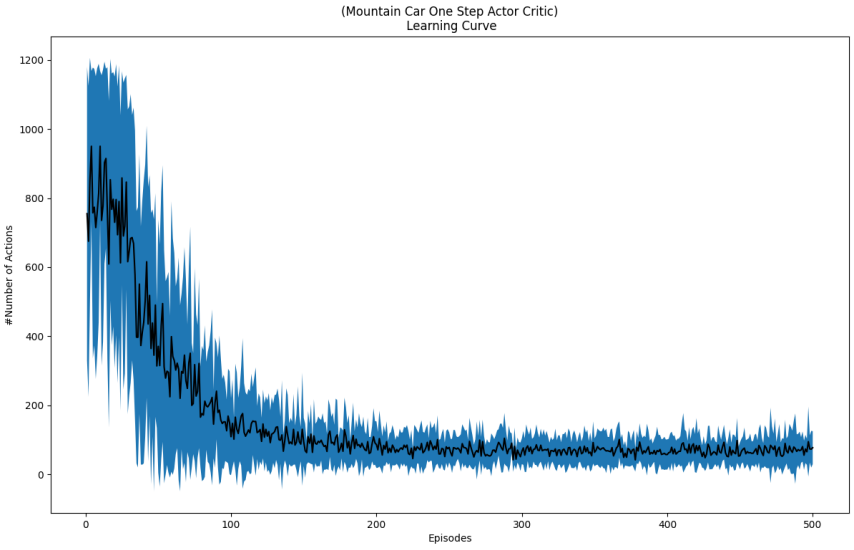
(v) Step Size (α^{θ}): Step size helps in the learning the policy approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^{\theta} \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution and higher value is oscillating and resulting me more deviation. With various values,e value $\alpha^{\theta} = 0.001$ has given the best result.

(vi) Step Size (α^{w}): Step size helps in the learning the value approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^{w} \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution and higher value is oscillating and resulting me more deviation. With various values, the value $\alpha^{w} = 0.001$ has given the best result.

The curves are based on $\alpha^{w} = 0.001, \alpha^{\theta} = 0.001, n = 1, \lambda = 1, \sigma = 1, epsiodes = 500$

Learning Curve:(computed over 20 runs)





(b) Acrobot

Design decisions made:

(i) Exploration Parameter (σ): Used $\sigma = 1$ for the soft max to calculate the action probabilities. The parameterized policy function contains σ and the finding the gradient of it would be tough. As specified in the class, to make the implementation easy we picked $\sigma = 1$

(ii) Discount Parameter (λ): Used discount parameter $\lambda = 1$ as specified in the MDP.

(iii) Feature Representation: Used Fourier series.

(iv) Features order (n) - As n increases, the feature representation increases which implies the weights

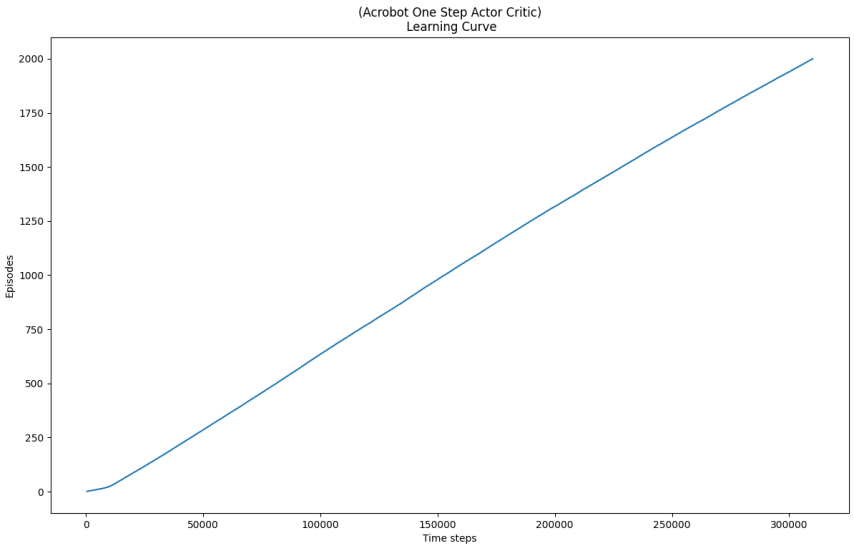
to be updated also increases. Adding more features gives better representation of a state. But the time complexities will be increased. With very exploration rate and very high feature representation, it takes very long time to optimal one with lesser deviation. With the episodes = 2000 and making other hyper parameters constant, I tried with different values of $n \in [1, 5]$ and everything showed similar performance where it converges to the optimal value but with lot of deviation. And state variables are (k=6) for current MDP, even with taking n = 2 it leads to 3^6 which is very huge and takes very long time to compute. So, I preferred taking $n = 1$.

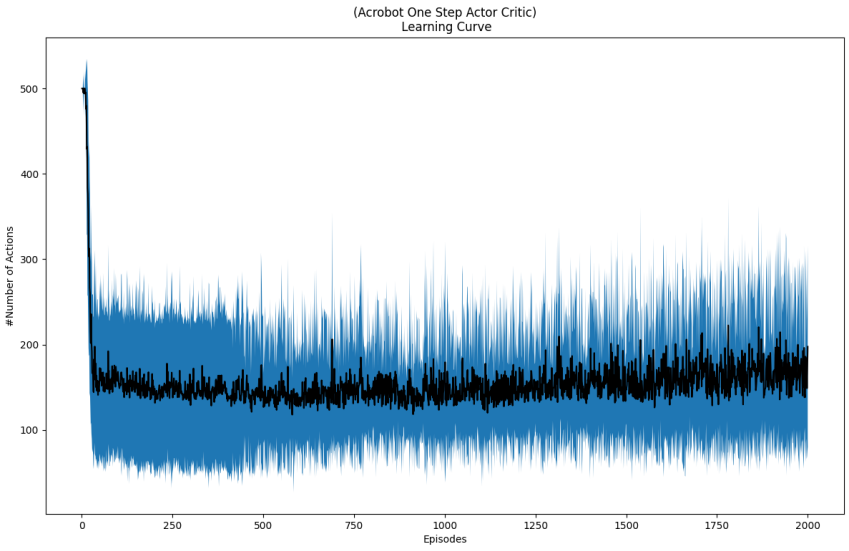
(v) Step Size (α^{θ}): Step size helps in the learning the policy approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^{\theta} \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution and higher value is oscillating and resulting me more deviation. With various values, the value $\alpha^{\theta} = 0.001$ has given the best result.

(vi) Step Size (α^{w}): Step size helps in the learning the value approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^{w} \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution and higher value is oscillating and resulting me more deviation. With various values, $\alpha^{w} = 0.001$ has given the best result.

The curves are based on $\alpha^{w} = 0.001, \alpha^{\theta} = 0.001, n = 1, \lambda = 1, \sigma = 1, epsiodes = 2000$

Learning Curve:(computed over 20 runs)





(c) **Cart Pole**

Design decisions made:

(i) Exploration Parameter (σ): Used $\sigma = 1$ for the soft max to calculate the action probabilities. The parameterized policy function contains σ and the finding the gradient of it would be tough. As specified in the class, to make the implementation easy we picked $\sigma = 1$

(ii) Discount Parameter (λ): Used discount parameter $\lambda = 1$ as specified in the MDP.

(iii) Feature Representation: Used Fourier series.

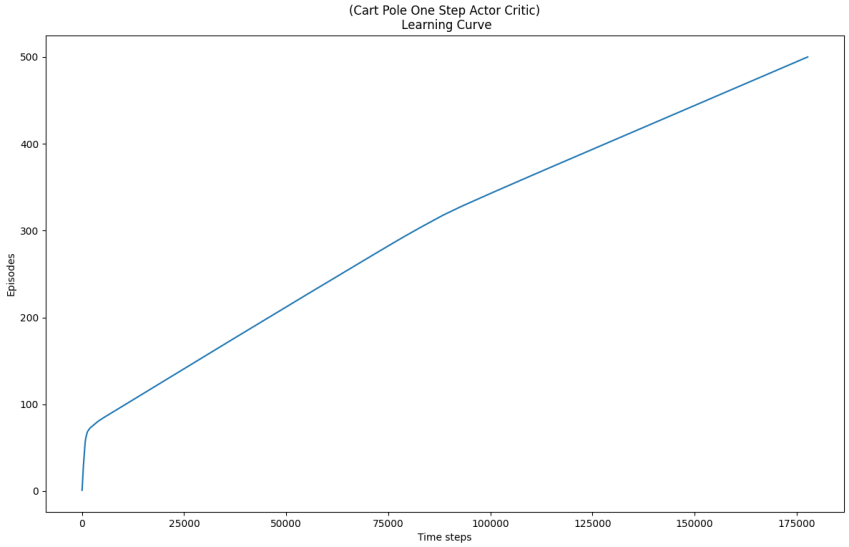
(iv) Features order (n) - As n increases, the feature representation increases which implies the weights to be updated also increases. Adding more features gives better representation of a state. But the time complexities will be increased. With very exploration rate and very high feature representation, it takes very long time to optimal one with lesser deviation. With the episodes = 500 and making other hyper parameters constant, I tried with different values of $n \in [1, 10]$. When $n \in [3, 5]$, the agent is reaching the optimal value(500) less than 100 episodes. With $n \in [1, 2]$ it is reaching at 300 episodes. With $n \geq 6$, it is not reaching optimal solution. I preferred taking $n = 1$ since it takes lesser time eventhough it takes 300 episodes to reach the optimal one.

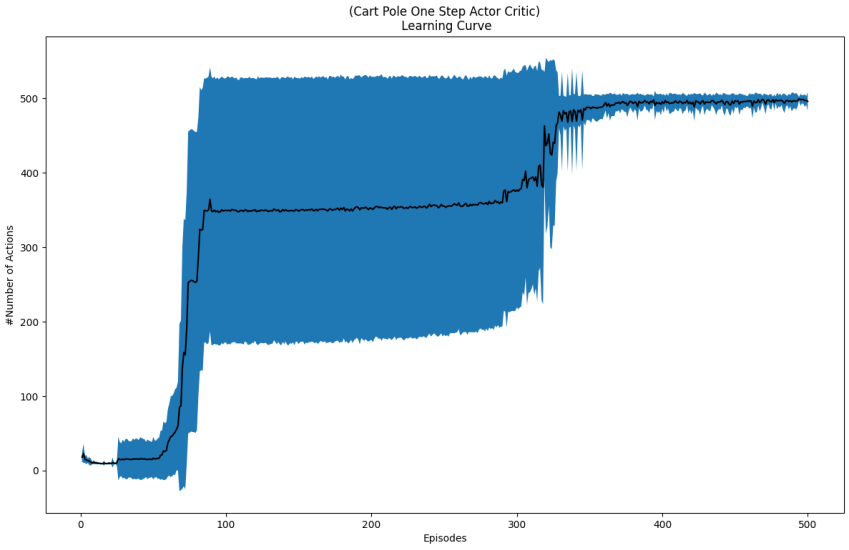
(v) Step Size (α^{θ}): Step size helps in the learning the policy approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^{\theta} \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution and higher value is oscillating and resulting me more deviation. With various values, the value $\alpha^{\theta} = 0.01$ has given the best result.

(vi) Step Size (α^{w}): Step size helps in the learning the value approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^{w} \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution and higher value is oscillating and resulting me more deviation. With various values, $\alpha^{w} = 0.01$ has given the best result.

The curves are based on $\alpha^{w} = 0.01, \alpha^{\theta} = 0.01, n = 1, \lambda = 1, \sigma = 1, epsiodes = 500$

Learning Curve:(computed over 20 runs)





3. Episodic Semi-Gradient n-step SARSA algorithm

We used ϵ - greedy policy for picking an action for all the following MDP.

(a) Mountain Car Design decisions made:

(i) (ϵ): Initialised with $\epsilon = 0.1$ and used ϵ greedy policy to pick an action.

(ii) Discount Parameter (λ): Used discount parameter $\lambda = 1$ as specified in the MDP.

(iii) Feature Representation: Used Fourier series.

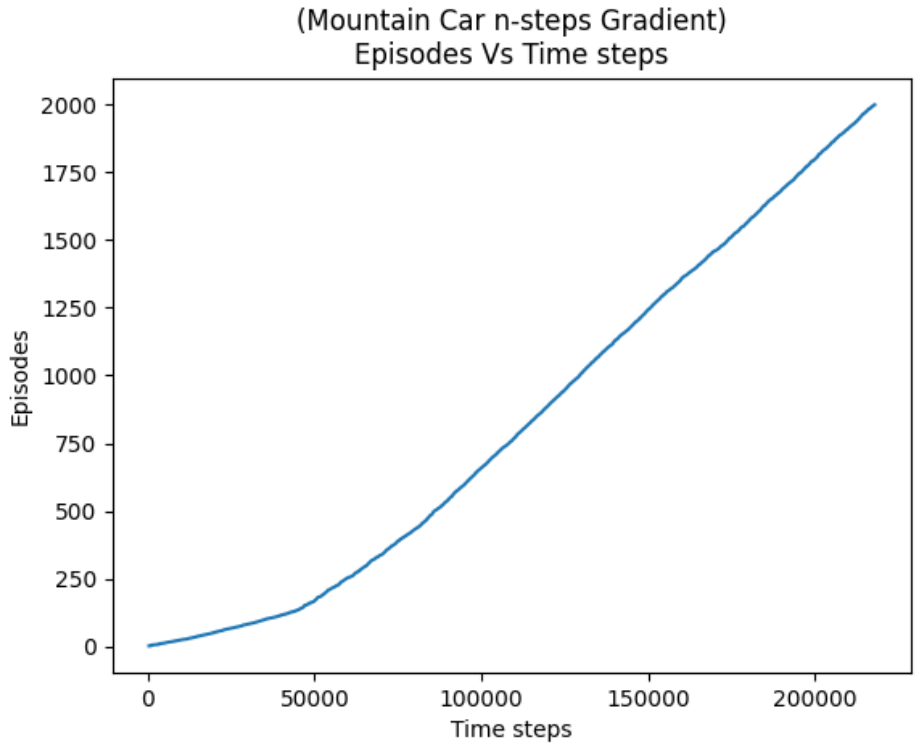
(iv) Features order ($features_n$) - As n increases, the feature representation increases which implies the weights to be updated also increases. Adding more features gives better representation of a state. But the time complexities will be increased. With very exploration rate and very high feature representation, it takes very long time to optimal one with lesser deviation. With the episodes = 2000 and making other hyper parameters constant, I tried with different values of $features_n \in [1, 10]$. And for value $features_n = 1$ it is performing better

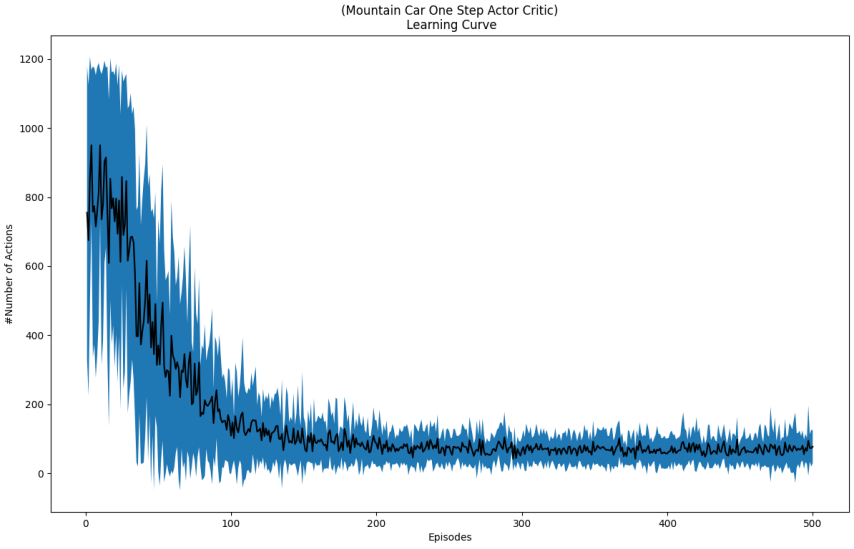
(v) sarsa n step (n) - As the n step increases, we are considering the rewards for the next n steps and based on that we are updating the equation. I tried different values of n and for higher values of n, its not reaching the optimal solution and even for the lower one it does the same. And I observed, $n = 8$ performed really well for this MDP

(vi) Step Size (α^w): Step size helps in the learning the value approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^w \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution same for the higher value. With various values, $\alpha^w = 0.04$ has given the best result.

The curves are based on $\alpha^w = 0.04, features_n = 1, \lambda = 1, \epsilon = 0.9, epsiodes = 2000, n_{steps} = 8$

Learning Curve:(computed over 20 runs)





(b) **Acrobot**

Design decisions made:

(i) (ϵ): Initialised with $\epsilon = 0.1$ and used ϵ greedy policy to pick an action.

(ii) Discount Parameter (λ): Used discount parameter $\lambda = 1$ as specified in the MDP.

(iii) Feature Representation: Used Fourier series.

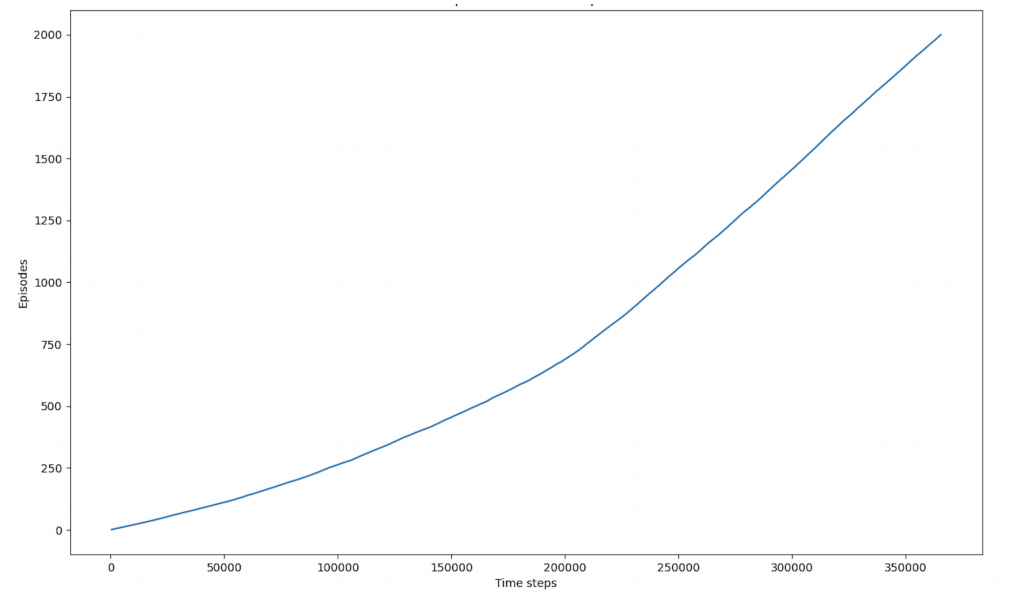
(iv) Features order ($features_n$) - As n increases, the feature representation increases which implies the weights to be updated also increases. Adding more features gives better representation of a state. But the time complexities will be increased. With very exploration rate and very high feature representation, it takes very long time to optimal one with lesser deviation. With the episodes = 500 and making other hyper parameters constant, I tried with different values of $features_n \in [1, 10]$. And for value $features_n = 1$ it is performing better

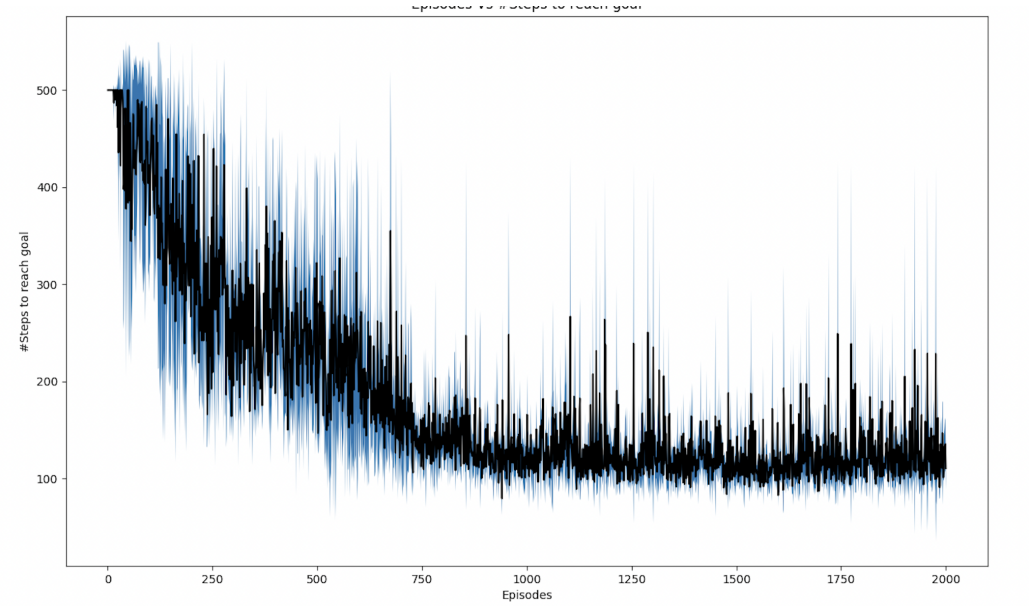
(v) sarsa n step (n) - As the n step increases, we are considering the rewards for the next n steps and based on that we are updating the equation. I tried different values of n and for higher values of n, its not reaching the optimal solution and even for the lower one it does the same. And I observed, $n = 8$ performed really well for this MDP

(vi) Step Size (α^w): Step size helps in the learning the value approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^w \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution same for the higher value. With various values, $\alpha^w = 0.001$ has given the best result.

The curves are based on $\alpha^w = 0.001, features_n = 1, \lambda = 1, \epsilon = 0.1, epsiodes = 2000, n_{steps} = 8$

Learning Curve:(computed over 20 runs)





(c) **Cart Pole**

Design decisions made:

(i) (ϵ): Initialised with $\epsilon = 0.9$ and decaying at every 50 episodes by 0.1. Used decaying as the states are continuous and wanna try to out explore as much as it can. I tried it because for lower value, it is not exploring because of that it is not reaching the optimal value. And for higher value, it is reaching the optimal value after some episodes but due higher exploration, there is a lot of deviations.

(ii) Discount Parameter (λ): Used discount parameter $\lambda = 1$ as specified in the MDP.

(iii) Feature Representation: Used Fourier series.

(iv) Features order ($features_n$) - As n increases, the feature representation increases which implies the weights to be updated also increases. Adding more features gives better representation of a state. But the time complexities will be increased. With very exploration rate and very high feature

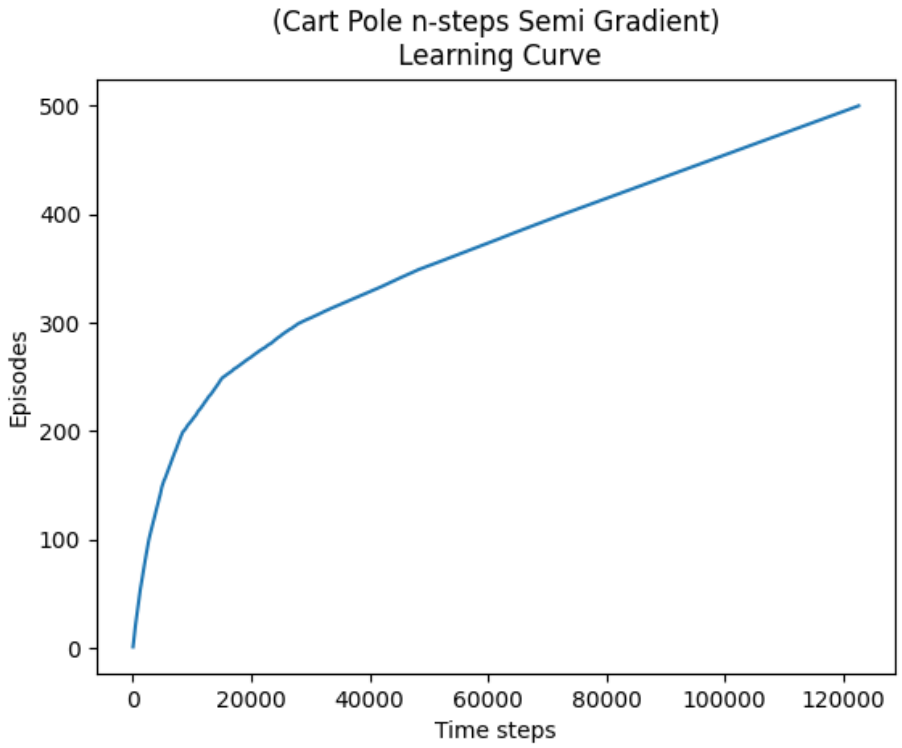
representation, it takes very long time to optimal one with lesser deviation. With the episodes = 500 and making other hyper parameters constant, I tried with different values of $features_n \in [1, 10]$. And for value $features_n = 2$ it is performing better

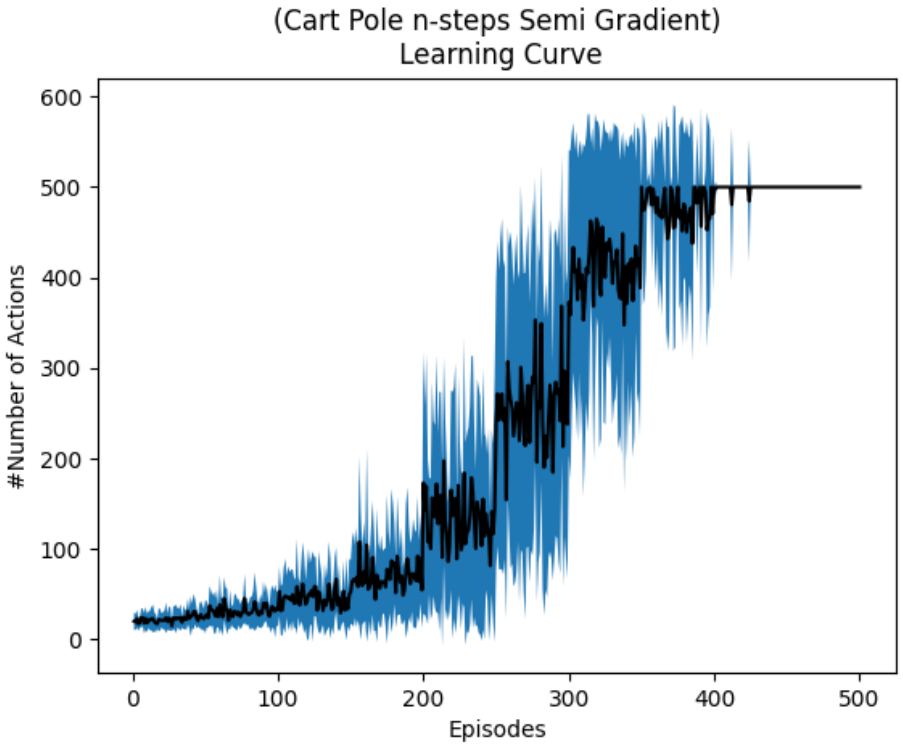
(v) sarsa n step (n) - As the n step increases, we are considering the rewards for the next n steps and based on that we are updating the equation. I tried different values of n and for higher values of n, its not reaching the optimal solution and even for the lower one it does the same. And I observed, $n = 8$ performed really well for this MDP

(vi) Step Size (α^w): Step size helps in the learning the value approximation function. Higher step size helps to converge faster, but sometimes it reaches the optimal solution and oscillates. And lower step size avoid the oscillation but takes every long time to converge. I tried different values of $\alpha^w \in [10^{-5}, 10]$ and observed lower value is not reaching the optimal solution same for the higher value. With various values, $\alpha^w = 0.001$ has given the best result.

The curves are based on $\alpha^w = 0.001, features_n = 2, \lambda = 1, \epsilon = 0.9, epsiodes = 500, n_{steps} = 8$ and decaying ϵ by 0.1 for every 50 episodes

Learning Curve:(computed over 20 runs)




(Cart Pole n-steps Semi Gradient)
Learning Curve
#Number of Actions
Episodes
0
100
200
300
400
500
600