CMPSCI 687 Final Project - Fall 2022

Piusha Gullapalli (pgullapallli@umass.edu), Debabrata Halder (dhalder@umass.edu)

December 2022

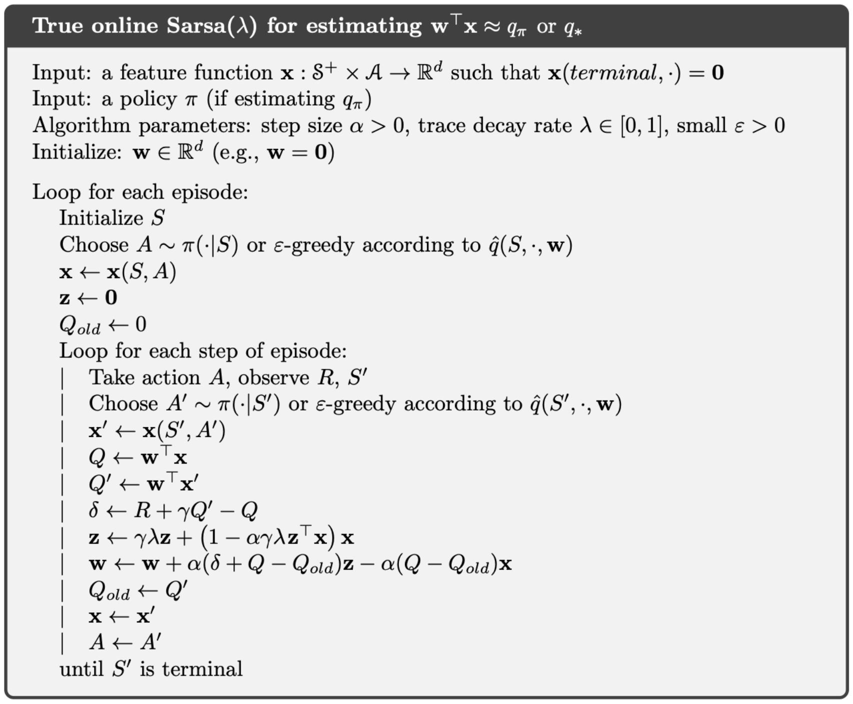
1 Introduction

In this project, for the main portion of it, we have implemented two new algorithms - True Online SARSA(λ) and Episodic Semi-Gradient n-step SARSA.
True Online SARSA(λ) is implemented on the domains Mountain Car and Gridworld.
Episodic Semi-Gradient n-step SARSA is implemented on Windy Gridworld, CliffWalk, and Cart-pole.
In addition to the two new algorithms on at least 2 existing MDPs each, we have implemented:
- The priority sweeping algorithm on Gridworld.
- A new MDP - Room Temperature control agent.
- Evaluating how well Value Iteration would perform if given an estimated/learned model.

2 Implementing two new algorithms

2.1 True Online SARSA(λ)

2.1.1 Mountain Car



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

```
Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \rightarrow \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: a policy $\pi$ (if estimating $q_\pi$)
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 $\varepsilon$-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 $\varepsilon$-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
```

i Description of the methods:
normaliseS : Used to normalise the position and velocity of the state for the Fourier basis.
getFourierBasis: Performs the Fourier basis using the cosine variant to represent the state feature vector
getAction: Using epsilon greedy to get an action with the given x, epsilon, and weights
generateEpisode: Implements the same functionality as the pseudo-code above for each episode
generateLearningCurve: Runs 1000 episodes to collect the number of steps in each episode to plot the learning curves.
generateLearningCurve is run 20 times to plot the average for the learning curves.

ii Pseudocode for the methods:
normaliseS:
input : state (position and velocity)
normalises position and velocity to values in [0,1]
return normalised values

getFourierBasis:
input: position x
input: velocity v
input: Fourier order - order
compute the feature vector

1. $\phi(s) = [1, \cos(1\pi x), \cos(2\pi x), \ldots, \cos(M\pi x), \cos(1\pi v), \cos(2\pi v), \ldots, \cos(M\pi v)]^\top$.

returns the feature vector

getAction:
input: state feature vector x
input: epsilon
input: weights
Picks the action with the highest$\hat{q}$ or a random action based on epsilon
returns action

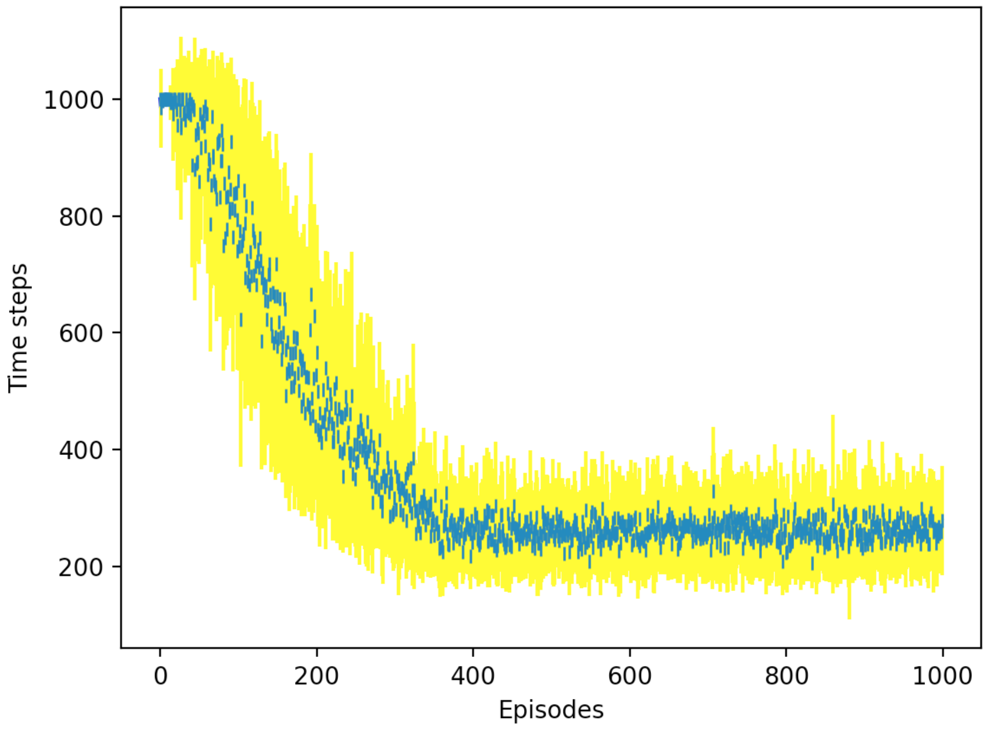
generateEpisode:
Same as the picture above with the pseudo-code for True online Sarsa(λ)

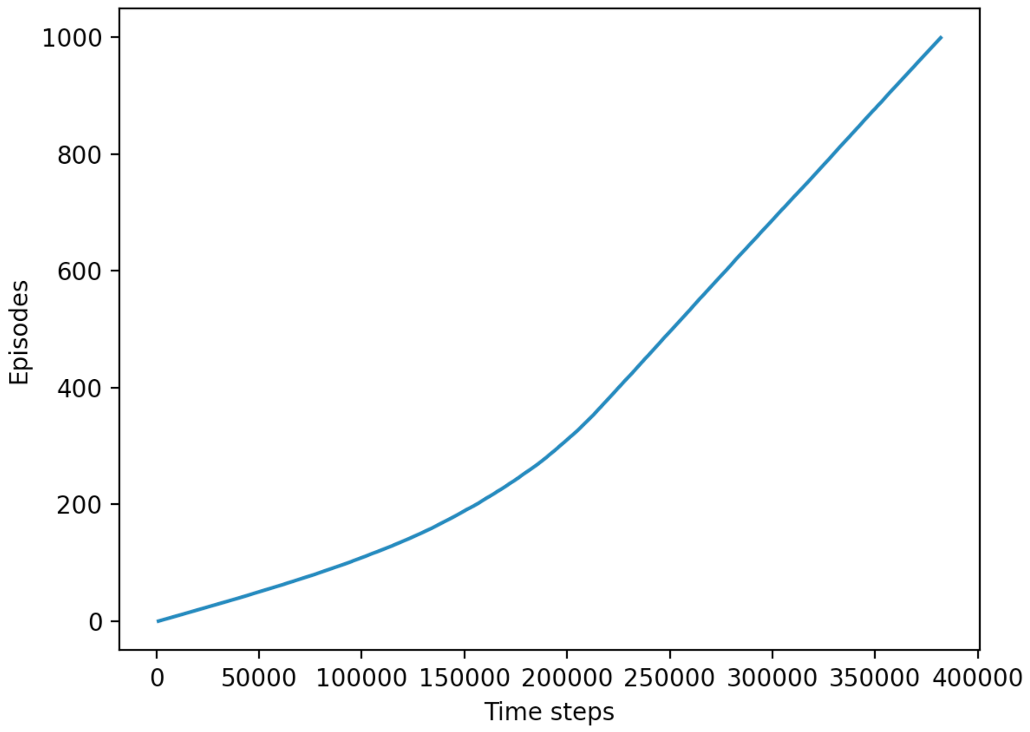
generateLearningCurve:
Set all the hyper-parameters and initialize variables for the episodes.
$\alpha; \lambda; \gamma; \epsilon$ and order
Calls generateEpisode for 1000 episodes and stores the number of steps taken in each to plot the learning curves.
Decays epsilon every 20 episodes by 0.05.
output: array of number of steps for every episode

iii Tuning the hyper-parameters:
We initially started out with $\alpha = 0.5; \lambda = 0.9; \epsilon = 0.5$, order = 8
First, we reduced the order to 3, and got a better learning curve, so we used the same further.
Once we generated the learning curve, the results were not satisfying, the algorithm was not learning well. So we changed the params to $\alpha = 0.3; \lambda = 0.5; \epsilon = 1$
This learning curve turned out better than earlier, since we increased exploration in the start and reduced α.
To check if we can improve this further, we tried out the params $\alpha = 0.1; \lambda = 0.1; \epsilon = 1$
This gave us lower steps than earlier and we can see the improved performance in the learning curves.

iv Learning curves:
The first learning curve plots the number of steps taken per episode, averaged over the 20 runs.

The second learning curve plots the total number of steps taken for 1000 episodes, averaged over the 20 runs.





2.1.2 Gridworld

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 $\pi$ (if estimating $q_\pi$)
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 $\varepsilon$-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 $\varepsilon$-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
```

i Description of the methods:
update_policy: After every episode, we update the policy by reevaluating the actions based on the q values.
get_action_dist: Creates the action distribution for the given state.
create_p_values: Creates the state distribution given a state and an action
generate_episode: Main implementation of the algorithm, same as the picture of the pseudo code.
generate_learning_curve: Runs generate_episode 200 times and stores the step counts, and cost function for the learning curves.

ii Pseudocode for the methods:

update_policy:
input: epsilon, $\pi(s, a)$, deterministic policy
Look at the deterministic policy and update $\pi(s, a)$ with corresponding epsilon greedy values
output: updated $\pi(s, a)$

get_action_dist:
input: state
Collect the action distributions for that state
output: actions and their distributions

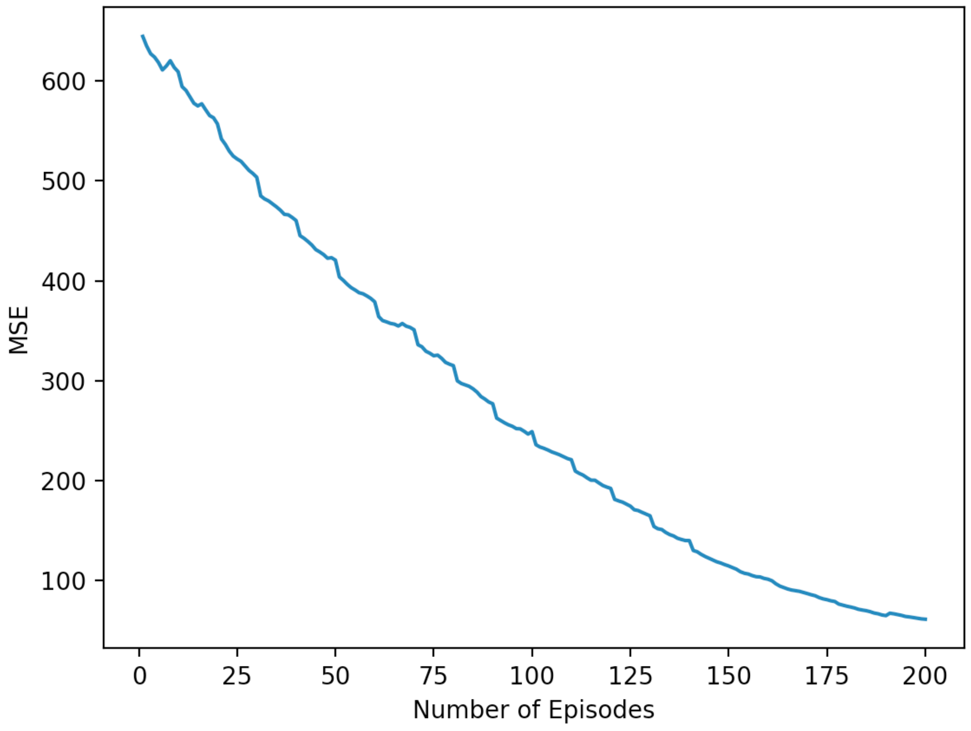
create_p_values:
input: state, action
based on the gridworld domain, get the next state distribution
output: distribution of the next state

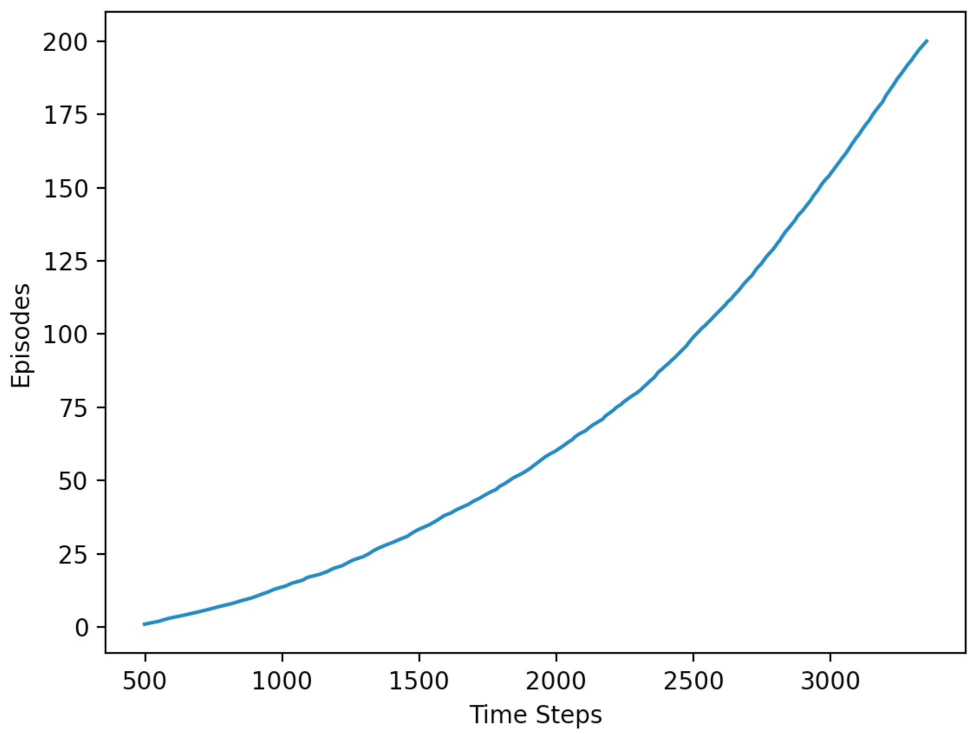
generate_episode:
Same code as the pseudo-code in the textbook, and in the picture above

generate_learning_curve:
Set all the hyper-parameters and initialize variables for the episodes.
$\alpha; \lambda; \gamma; \epsilon$
Call generate_episode: for 200 episodes and stores the number of steps taken in each to plot the learning curves.
Update the policy for the states in every episode based on the $q_p i$ value. Compute value function for the cost function
Decay epsilon every 20 episodes by 0.05.
output: array of number of steps for every episode

iii Tuning the hyper-parameters:
We initially started out with $\alpha = 0.5; \lambda = 0.9; \epsilon = 0.5$
Once we generated the learning curves, and compared the mean squared error, the results were not satisfying, the algorithm was not learning well, the number of steps did not reduce with the increase in episodes.
We changed the params to $\alpha = 0.3; \lambda = 0.5; \epsilon = 1$
This learning curve turned out better than earlier, since we increased exploration in the start and reduced the learning rate α.
To check if we can improve this further, we tried out the params $\alpha = 0.3; \lambda = 0.1; \epsilon = 0.5$
This gave us lower steps on an average than earlier and we can see the improved performance in the learning curves.
The cost function also reduced with the number of episodes.

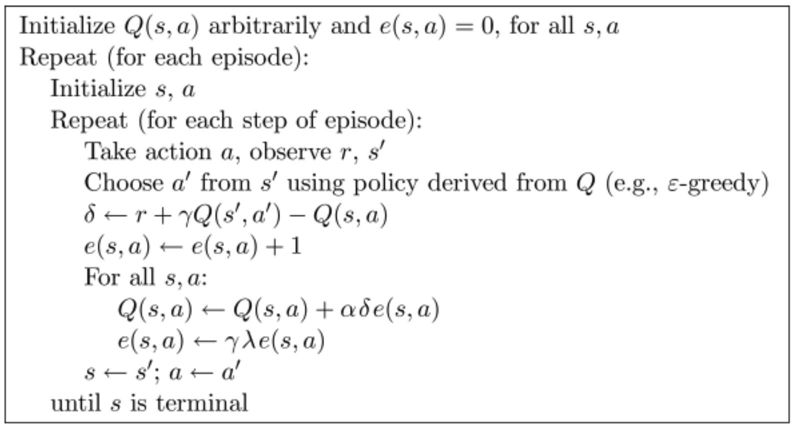
iv Learning curves:
The first learning curve plots the cost function compared to the optimal value function averaged over the 20 runs.
The second learning curve plots the total number of steps taken for 200 episodes, averaged over the 20 runs.





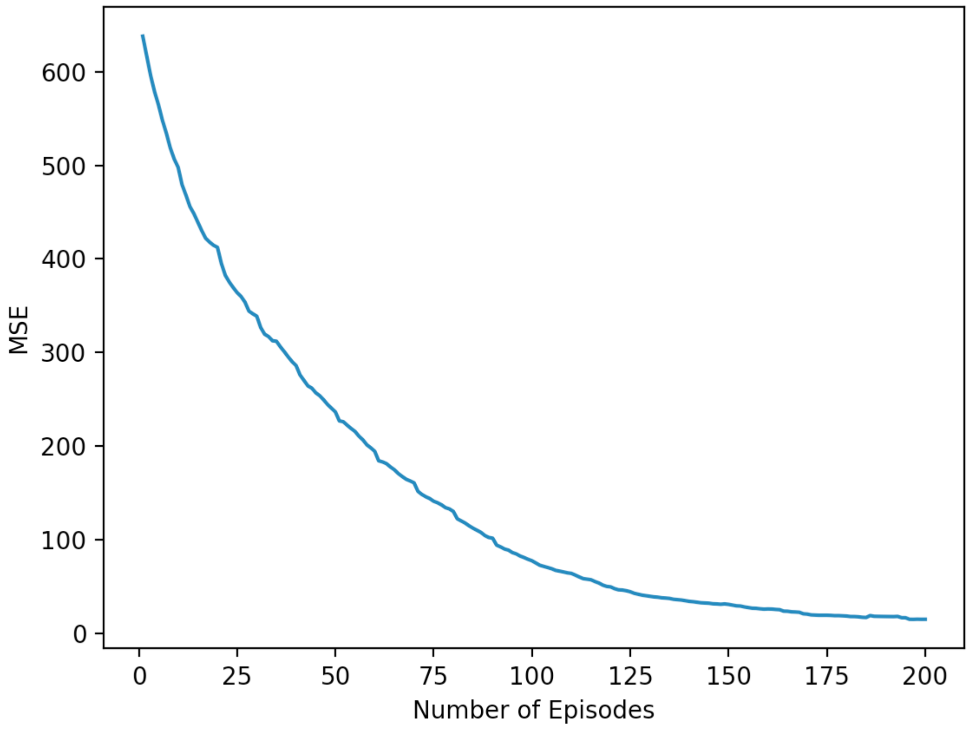
v After implementing true online SARSA on gridworld, we were not completely satisfied with the results even after tuning the hyperparameters.
We decided to implement the tabular SARSA (λ) using traces on gridworld since we are working with discrete states.

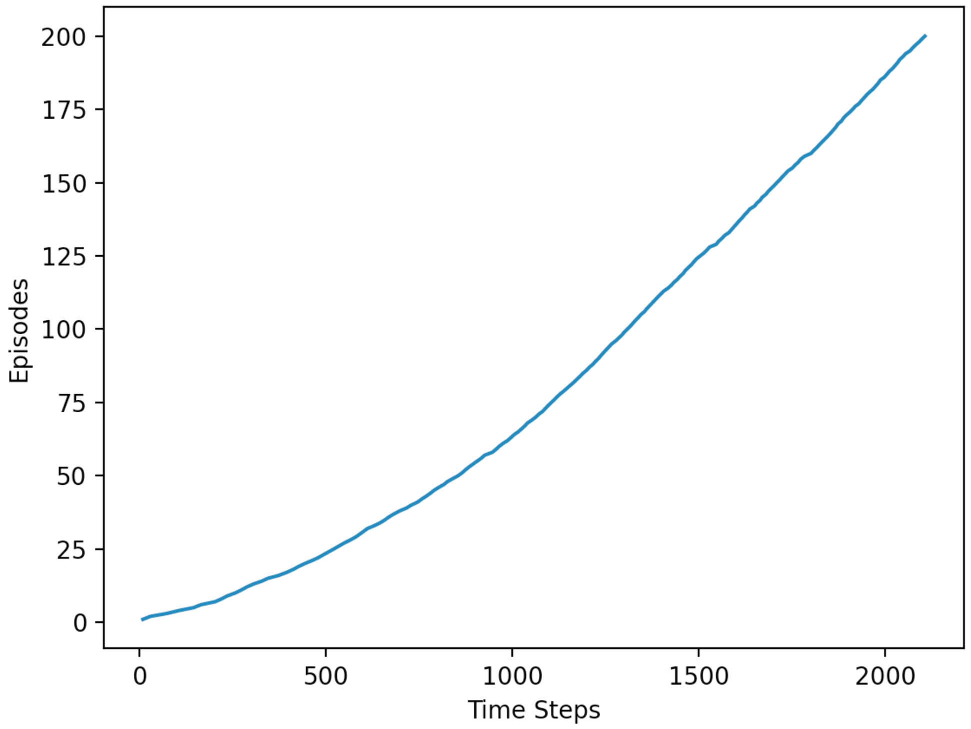
All the methods remained the same, except for generate_episode.
The following is the psuedo-code for generate_episode:



```
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., $\varepsilon$-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
```

Learning curves:
The first learning curve plots the cost function compared to the optimal value function averaged over the 20 runs.
The second learning curve plots the total number of steps taken for 200 episodes, averaged over the 20 runs.



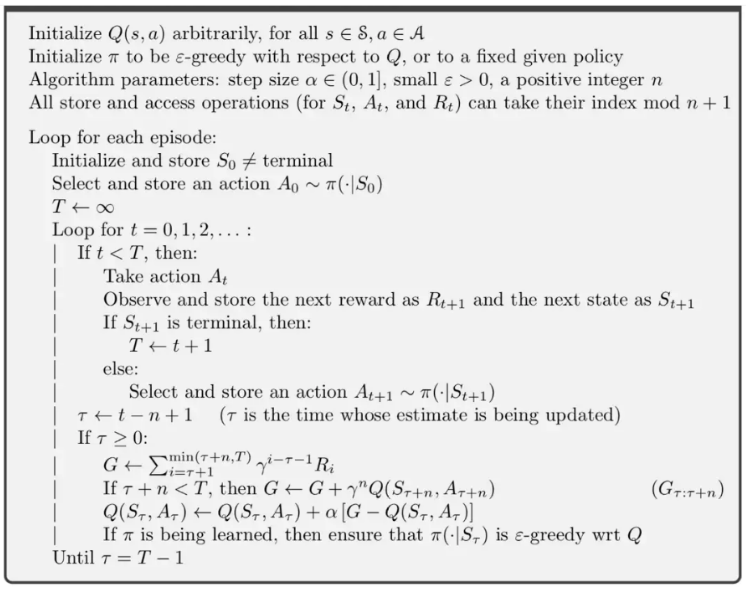


Comparing the learning curves, we can see that the MSE is closer to 0 with tabular SARSA and the total

number of steps reduced by around 1000 as well.
Looking the two results, we found the tabular version performing better on the gridworld domain in our implementations.

2.2 Episodic Semi-Gradient n-step SARSA

2.2.1 Windy Grid World



```
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be $\varepsilon$-greedy with respect to $Q$, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, 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 $A_0 \sim \pi(\cdot|S_0)$
   $T \leftarrow \infty$
   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 an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
   |   $\tau \leftarrow t - n + 1$   ($\tau$ 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 $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$          $(G_{\tau:\tau+n})$
   |       $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
   |       If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is $\varepsilon$-greedy wrt $Q$
   Until $\tau = T - 1$
```

i Description of the methods:
nStepSARSA:
This method mainly update the Q value based on n value which is temporal difference between current time and update time. Here updating the policy is implicit as we are updating the Q value. And updating the Q value based on n value. In the algorithm, the t value is current time stamp and τ is the time stamp of Q value is being updated. Like mentioned in the algorithm, we are also updating the Q value in the main n step sarsa method.

calculate_G:
This method calculate the total discounted return from tau timestamp to current timestamp.
epsilonGreedyPolicy: This method gives the next action according to the epsilon greedy policy.
generate_learning_curve: This method is responsible for generating all learning curves like episode length vs episode, episode vs steps etc.

ii Pseudo-code for the methods:
The screenshot of the pseudo code is mentioned above. The implementation was based on this pseudo code.

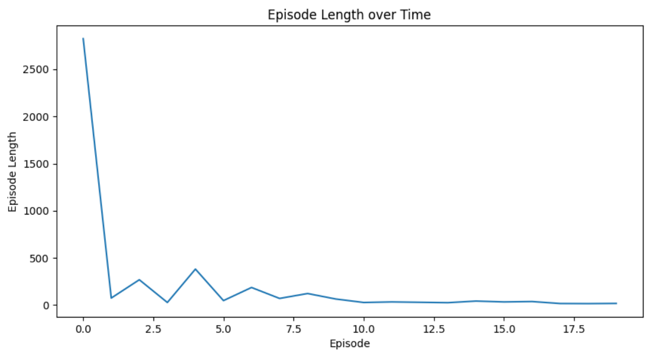
iii Tuning the hyper-parameters:
The hyper-parameters associated with the algorithms are -

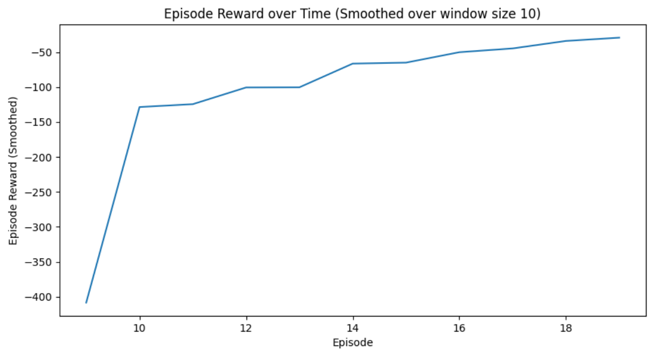
- n:
  The n value tells about the current update of q value how many step away. So, changing the n value we are changing the temporal difference between current time and updating time. For this algorithm we kept changing n value from [2, 20] range.
- Number of Episode:
  The number of episode for windy grid world was around 40 is sufficient to reach optimal policy. For that

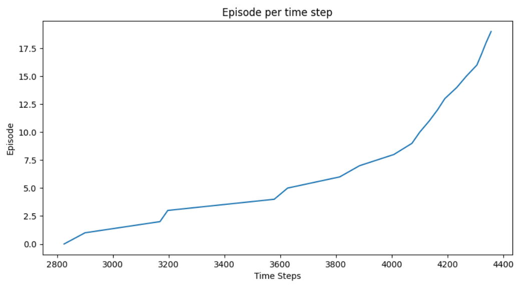
we kept changing the the number of episodes from [20-100] range. But after 50, the performance was quite stable.

- γ: For this problem, we tried with three variation of $\gamma = 1, 0.9, 0.75$ etc. But with $\gamma = 0.75$, we was getting better result.
- α: We also tried with five variation of $\alpha = 0.4, 0.3, 0.2, 0.1$ etc. with $\alpha = 0.4$ we am getting good and fastest result as increasing α can lead to non-convergence and reducing it may lead to slow learning.
- ϵ: It hyper parameter has been used for epsilon-greedy policy. Throughout the experiment we kept it constant value 0.1.

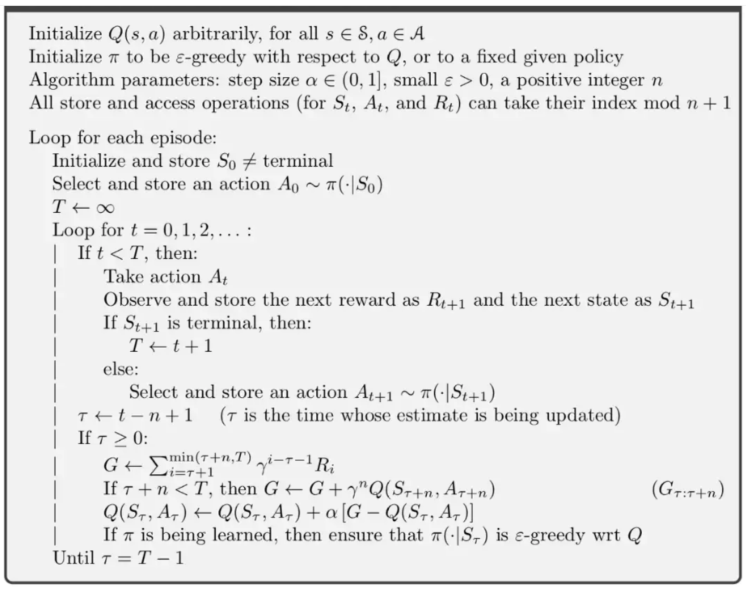
iv Learning curves:







2.2.2 CliffWalk



```
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be $\varepsilon$-greedy with respect to $Q$, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, 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 $A_0 \sim \pi(\cdot|S_0)$
    $T \leftarrow \infty$
    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 an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
    |   $\tau \leftarrow t - n + 1$    ($\tau$ 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 $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$        $(G_{\tau:\tau+n})$
    |       $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$
    |       If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is $\varepsilon$-greedy wrt $Q$
    Until $\tau = T - 1$
```

i Description of the methods:
nStepSARSA:
This method mainly update the Q value based on n value which is temporal difference between current time and update time. Here updating the policy is implicit as we are updating the Q value. And updating the Q value based on n value. In the algorithm, the t value is current time stamp and τ is the time stamp of Q value is being updated. Like mentioned in the algorithm, we are also updating the Q value in the main n step sarsa method.

calculate_G:
This method calculate the total discounted return from tau timestamp to current timestamp.
epsilonGreedyPolicy: This method gives the next action according to the epsilon greedy policy.
generate_learning_curve: This method is responsible for generating all learning curves like episode length vs episode, episode vs steps etc.

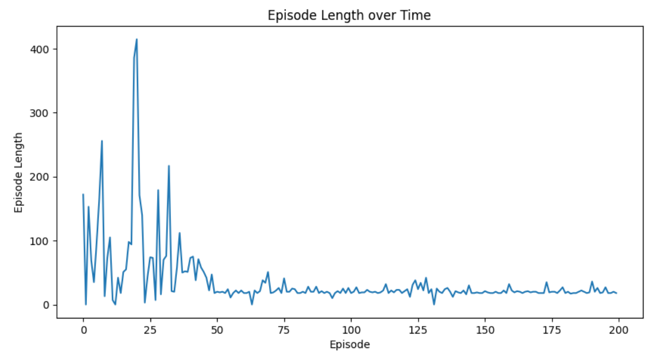
ii Pseudo-code for the methods:
The screenshot of the pseudo code is mentioned above. The implementation was based on this pseudo code.

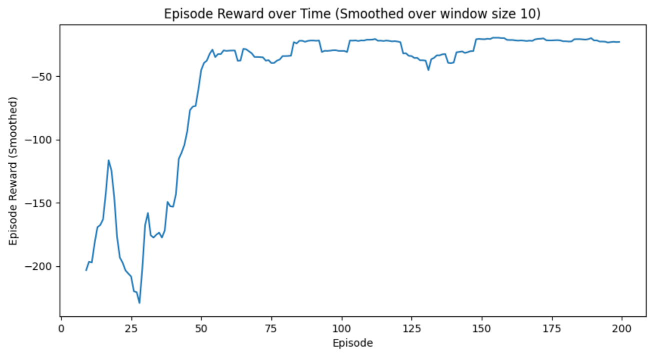
iii Tuning the hyper-parameters:
The hyper-parameters associated with the algorithms are -

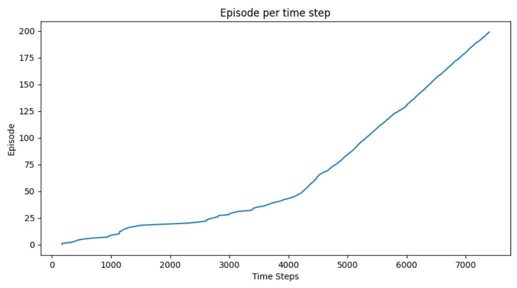
- n:
  The n value tells about the current update of q value how many step away. So, changing the n value we are changing the temporal difference between current time and updating time. For this algorithm we kept changing n value from [2, 20] range.
- Number of Episode:
  The number of episode for windy grid world was around 200 is sufficient to reach optimal policy. For that we kept changing the the number of episodes from [50-500] range. But after 100, the performance was quite stable.
- γ: For this problem, we tried with three variation of $\gamma = 1, 0.9, 0.75$ etc. But with $\gamma = 0.75$, we was getting better result.
- α: we also tried with five variation of $\alpha = 0.4, 0.3, 0.2, 0.1$ etc. with $\alpha = 0.4$ we are getting good and fastest result as increasing α can lead to non-convergence and reducing it may lead to slow learning.

- ϵ: It hyper parameter has been used for epsilon-greedy policy. Throughout the experiment we kept it constant value 0.1.

iv Learning curves:







2.2.3 Cart-pole

i Description of the methods:
Same as above.

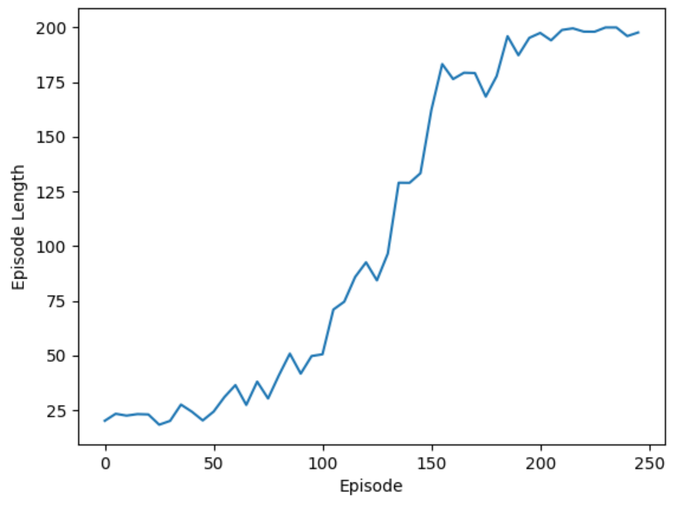
ii Pseudo-code for the methods:
The screenshot of the pseudo code is mentioned above. The implementation was based on this pseudo code.

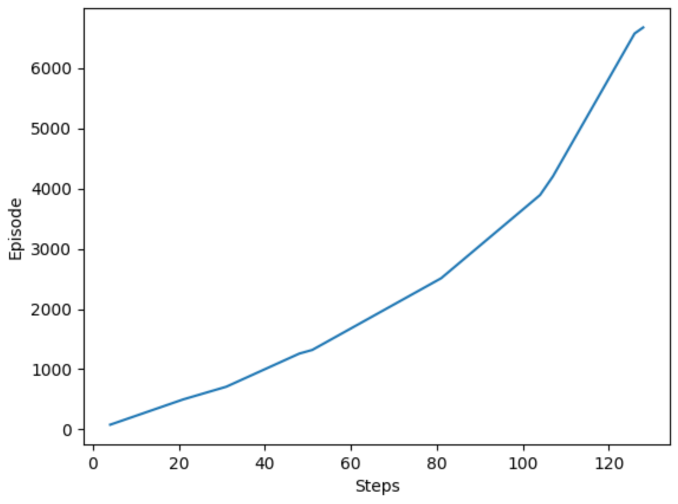
iii Tuning the hyper-parameters:
The hyper-parameters associated with the algorithms are -

- n:
  The n value tells about the current update of q value how many step away. So, chaining the n value we are changing the temporal difference between current time and updating time. For this algorithm we kept changing n value from [2, 20] range.

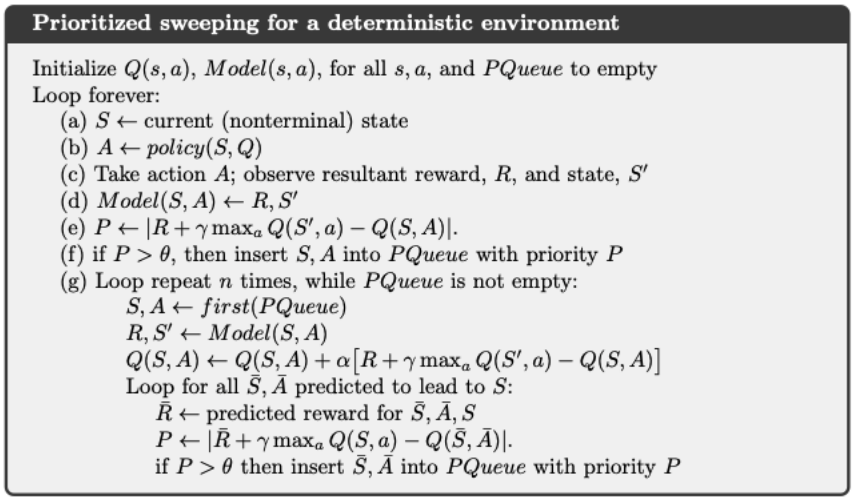
- Number of Episode:
  The number of episode for windy grid world was around 200 is sufficient to reach optimal policy. For that we kept changing the the number of episodes from [50-500] range. But after 100, the performance was quite stable.
- γ: For this problem, we tried with three variation of $\gamma = 1, 0.9, 0.75$ etc. But with $\gamma = 0.75$, we were getting better result.
- α: we also tried with five variation of $\alpha = 0.4, 0.3, 0.2, 0.1$ etc. with $\alpha = 0.4$ we am getting good and fastest result as increasing α can lead to non-convergence and reducing it may lead to slow learning.
- ϵ: It hyper parameter has been used for epsilon-greedy policy. Throughout the experiment we kept it constant value 0.1.

iv Learning curves:





2.3 Prioritized sweeping for a deterministic environment - Gridworld



Prioritized sweeping for a deterministic environment

```
Initialize $Q(s,a)$, $Model(s,a)$, for all $s, a$, and $PQueue$ to empty
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow policy(S, Q)$
  (c) Take action $A$; observe resultant reward, $R$, and state, $S'$
  (d) $Model(S, A) \leftarrow R, S'$
  (e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|$.
  (f) if $P > \theta$, then insert $S, A$ into $PQueue$ with priority $P$
  (g) Loop repeat $n$ times, while $PQueue$ is not empty:
        $S, A \leftarrow first(PQueue)$
        $R, S' \leftarrow Model(S, A)$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        Loop for all $\bar{S}, \bar{A}$ predicted to lead to $S$:
          $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
          $P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A})|$.
          if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority $P$
```

i Description of the methods:
pritoritzed_sweeping:
The pseudo-code in the picture above is implemented in this method.

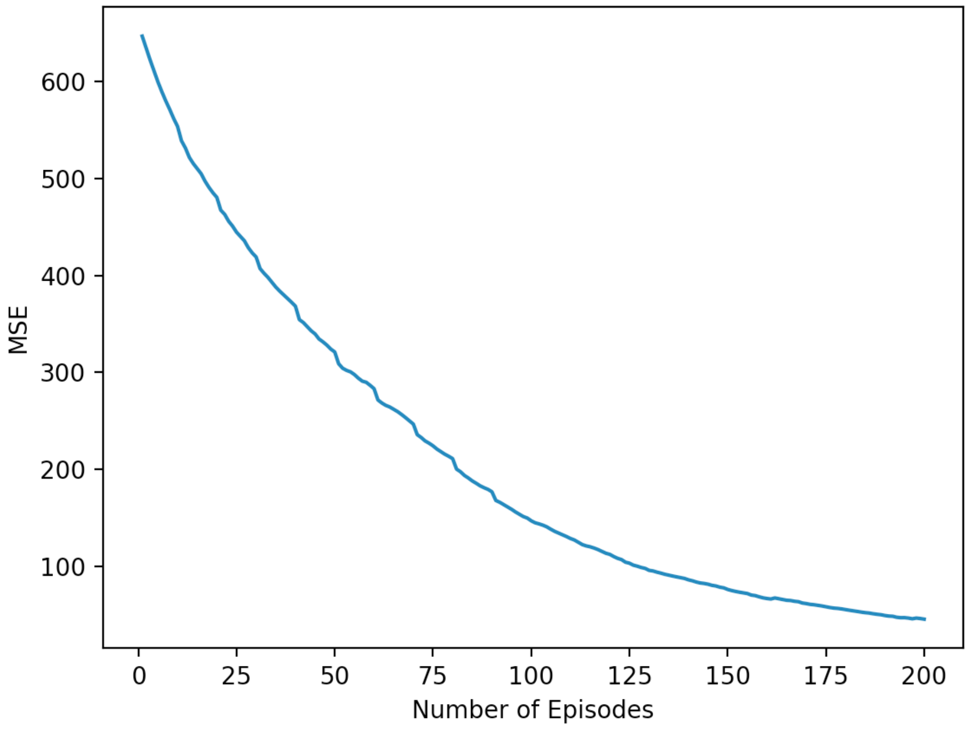
generate_value_function:
This method is used to get the value function from the Q values obtained and the policy.

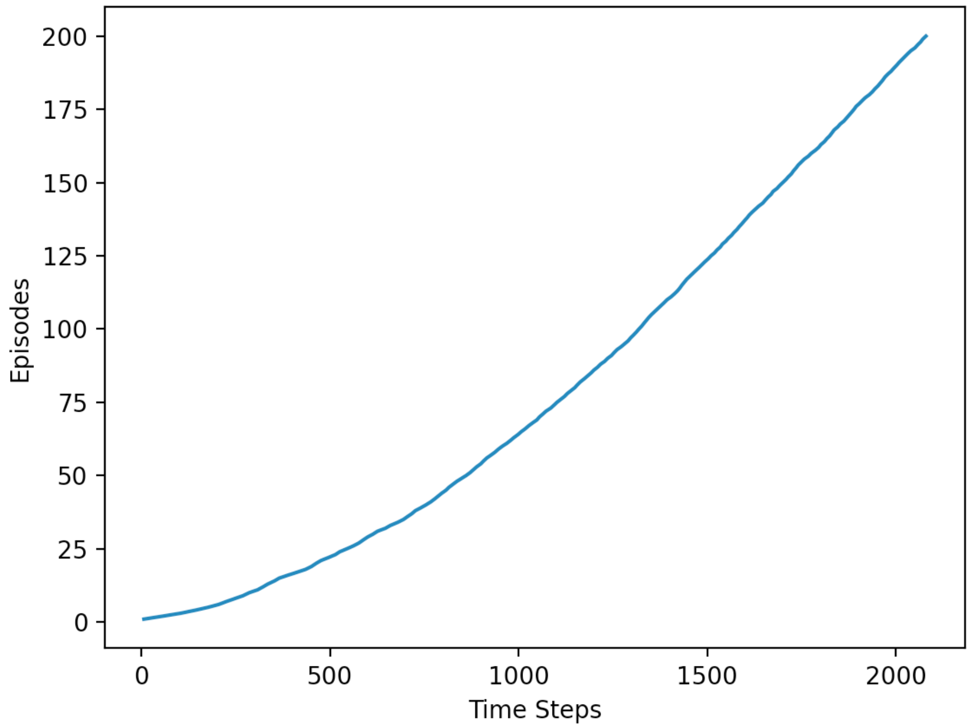
displayValueFunction:
Used to format and display the value function.

ii Pseudocode of the methods:
The screenshot of the pseudo code is added above. The implementation was based on the same.

iii Tuning the hyperparameters:
We initially started out with an α of 0.05 and went till 0.6.
The best results were obtained at $\alpha = 0.2$

iv Learning curves: The first learning curve plots the cost function compared to the optimal value function averaged over the 20 runs.
The second learning curve plots the total number of steps taken for 10000 episodes, averaged over the 20 runs.


MSE
600
500
400
300
200
100
0
25
50
75
100
125
150
175
200
Number of Episodes


Episodes
200
175
150
125
100
75
50
25
0
0
500
1000
1500
2000
Time Steps

3 New MDP

3.1 Room Temperature Controller Agent

As a part of this project, we tried to come up a new MDP which could be useful in real life. Here, we tried to create an reinforcement agent who tries to keep room temperature comfortable by keeping it within a predefined range. This could be very useful in multiple real example like air-conditioner control, food or medicine preservation center etc.

- **State**: The state is combination of three parameters current room temperature, outside temperature, and the temperature changing rate outside. So, S:(T_r, T_o, $\frac{\partial T_o}{\partial t}$). To make it simpler, let outside temperature changing rate is constant for a episode. Let call it α. So, state becomes S:(T_r, T_o, α)

- **Actions**: Agent can take only three discrete actions temperature down, neutral, temperature up which is mapped with $[-1, 0, 1]$ respectively.
- **Dynamics**: The room temperature dynamics depend on three factors - current room temperature, outside temperature, and a factor named heat transfer constant. let called it β,

$$T_o = T_o + \alpha$$
$$T_r = action + T_r + \beta(T_o - T_r)$$

- **Rewards** : 0 if $65 \le T_r \le 85$, else -1
- **Initial State**: initial room temperature is between $[67, 77]$ and initial outside temperature is $[57, 87]$ range and outside temperature change rate in [-0.05, 0.05] range.
- **Discount**: $\gamma = 1$

Custom Training on Stable Baseline Model:

- Double Q learning agent from (DQNAgent) from rl.agents
- Proximal Policy Optimization with MLP Policy
- DQN with MLP Policy from stable baseline

But due to time constrains we could not make it work. The simpler version when room temperature independent of outside temperature, we could make it work. and models are learning successfully.

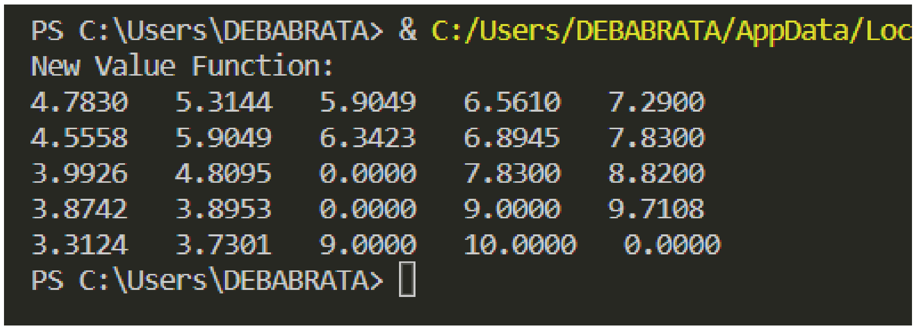
4 Value Iteration performance if given an estimated/learned model

Input: A optimal policy for grid world 687

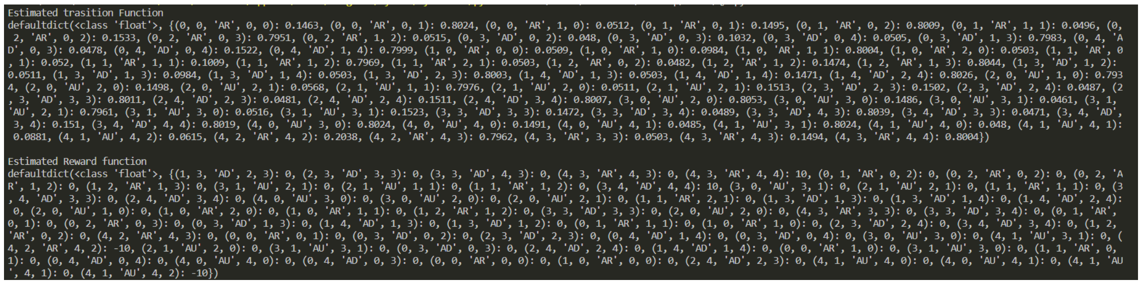
Now to get the environment dynamic and reward function we generated a lot of episodes. For every episode we stored the the reward function in a dictionary. Similarly we stored the count of (s,a,s') in dictionary. Now, $p(s, a, s') = count(s, a, s')/count(s, a, s')$. Thus, we have calculated the p(s,a,s').

Now, we have p(s,a,s'), and R, we can use value iteration. For that we used our old implementation of value iteration in the assignment 3.

The final value state we got,



```
PS C:\Users\DEBABRATA> & C:/Users/DEBABRATA/AppData/Loc
New Value Function:
4.7830   5.3144   5.9049   6.5610   7.2900
4.5558   5.9049   6.3423   6.8945   7.8300
3.9926   4.8095   0.0000   7.8300   8.8200
3.8742   3.8953   0.0000   9.0000   9.7108
3.3124   3.7301   9.0000   10.0000   0.0000
PS C:\Users\DEBABRATA> 
```



```
Estimated trasition Function
defaultdict(<class 'float'>, {(0, 0, 'AR', 0, 0): 0.1463, (0, 0, 'AR', 0, 1): 0.8024, (0, 0, 'AR', 1, 0): 0.0512, (0, 1, 'AR', 0, 1): 0.1495, (0, 1, 'AR', 0, 2): 0.8009, (0, 1, 'AR', 1, 1): 0.0496, (0,
 2, 'AR', 0, 2): 0.1533, (0, 2, 'AR', 0, 3): 0.7951, (0, 2, 'AR', 1, 2): 0.0515, (0, 3, 'AD', 0, 2): 0.048, (0, 3, 'AD', 0, 3): 0.1032, (0, 3, 'AD', 0, 4): 0.0505, (0, 3, 'AD', 1, 3): 0.7983, (0, 4, 'A
D', 0, 3): 0.0478, (0, 4, 'AD', 0, 4): 0.1522, (0, 4, 'AD', 1, 4): 0.7999, (1, 0, 'AR', 0, 0): 0.0509, (1, 0, 'AR', 1, 0): 0.0984, (1, 0, 'AR', 1, 1): 0.8004, (1, 0, 'AR', 2, 0): 0.0503, (1, 1, 'AR', 0
, 1): 0.052, (1, 1, 'AR', 1, 1): 0.1009, (1, 1, 'AR', 1, 2): 0.7969, (1, 1, 'AR', 2, 1): 0.0503, (1, 2, 'AR', 0, 2): 0.0482, (1, 2, 'AR', 1, 2): 0.1474, (1, 2, 'AR', 1, 3): 0.8044, (1, 3, 'AD', 1, 2):
0.0511, (1, 3, 'AD', 1, 3): 0.0984, (1, 3, 'AD', 1, 4): 0.0503, (1, 3, 'AD', 2, 3): 0.8003, (1, 4, 'AD', 1, 3): 0.0503, (1, 4, 'AD', 1, 4): 0.1471, (1, 4, 'AD', 2, 4): 0.8026, (2, 0, 'AU', 1, 0): 0.793
4, (2, 0, 'AU', 2, 0): 0.1498, (2, 0, 'AU', 2, 1): 0.0568, (2, 1, 'AU', 1, 1): 0.7976, (2, 1, 'AU', 2, 0): 0.0511, (2, 1, 'AU', 2, 1): 0.1513, (2, 3, 'AD', 2, 3): 0.1502, (2, 3, 'AD', 2, 4): 0.0487, (2
, 3, 'AD', 3, 3): 0.8011, (2, 4, 'AD', 2, 3): 0.0481, (2, 4, 'AD', 2, 4): 0.1511, (2, 4, 'AD', 3, 4): 0.8007, (3, 0, 'AU', 2, 0): 0.8053, (3, 0, 'AU', 3, 0): 0.1486, (3, 0, 'AU', 3, 1): 0.0461, (3, 1,
'AU', 2, 1): 0.7961, (3, 1, 'AU', 3, 0): 0.0516, (3, 1, 'AU', 3, 1): 0.1523, (3, 3, 'AD', 3, 3): 0.1472, (3, 3, 'AD', 3, 4): 0.0489, (3, 3, 'AD', 4, 3): 0.8039, (3, 4, 'AD', 3, 3): 0.0471, (3, 4, 'AD',
 3, 4): 0.151, (3, 4, 'AD', 4, 4): 0.8019, (4, 0, 'AU', 3, 0): 0.8024, (4, 0, 'AU', 4, 0): 0.1491, (4, 0, 'AU', 4, 1): 0.0485, (4, 1, 'AU', 3, 1): 0.8024, (4, 1, 'AU', 4, 0): 0.048, (4, 1, 'AU', 4, 1):
 0.0881, (4, 1, 'AU', 4, 2): 0.0615, (4, 2, 'AR', 4, 2): 0.2038, (4, 2, 'AR', 4, 3): 0.7962, (4, 3, 'AR', 3, 3): 0.0503, (4, 3, 'AR', 4, 3): 0.1494, (4, 3, 'AR', 4, 4): 0.8004})

Estimated Reward function
defaultdict(<class 'float'>, {(1, 3, 'AD', 2, 3): 0, (2, 3, 'AD', 3, 3): 0, (3, 3, 'AD', 4, 3): 0, (4, 3, 'AR', 4, 3): 0, (4, 3, 'AR', 4, 4): 10, (0, 1, 'AR', 0, 2): 0, (0, 2, 'AR', 0, 2): 0, (0, 2, 'A
R', 1, 2): 0, (1, 2, 'AR', 1, 3): 0, (3, 1, 'AU', 2, 1): 0, (2, 1, 'AU', 1, 1): 0, (1, 1, 'AR', 1, 2): 0, (3, 4, 'AD', 4, 4): 10, (3, 0, 'AU', 3, 1): 0, (2, 1, 'AU', 2, 1): 0, (1, 1, 'AR', 1, 1): 0, (3
, 4, 'AD', 3, 3): 0, (2, 4, 'AD', 3, 4): 0, (4, 0, 'AU', 3, 0): 0, (3, 0, 'AU', 2, 0): 0, (2, 0, 'AU', 2, 1): 0, (1, 1, 'AR', 2, 1): 0, (1, 3, 'AD', 1, 3): 0, (1, 3, 'AD', 1, 4): 0, (1, 4, 'AD', 2, 4):
 0, (2, 0, 'AU', 1, 0): 0, (1, 0, 'AR', 2, 0): 0, (1, 0, 'AR', 1, 1): 0, (1, 2, 'AR', 1, 2): 0, (3, 3, 'AD', 3, 3): 0, (2, 0, 'AU', 2, 0): 0, (4, 3, 'AR', 3, 3): 0, (3, 3, 'AD', 3, 4): 0, (0, 1, 'AR',
0, 1): 0, (0, 2, 'AR', 0, 3): 0, (0, 3, 'AD', 1, 3): 0, (1, 4, 'AD', 1, 3): 0, (1, 3, 'AD', 1, 2): 0, (0, 1, 'AR', 1, 1): 0, (1, 0, 'AR', 1, 0): 0, (2, 3, 'AD', 2, 4): 0, (3, 4, 'AD', 3, 4): 0, (1, 2,
'AR', 0, 2): 0, (4, 2, 'AR', 4, 3): 0, (0, 0, 'AR', 0, 1): 0, (0, 3, 'AD', 0, 2): 0, (2, 3, 'AD', 2, 3): 0, (0, 4, 'AD', 1, 4): 0, (0, 3, 'AD', 0, 4): 0, (3, 0, 'AU', 3, 0): 0, (4, 1, 'AU', 3, 1): 0, (
4, 2, 'AR', 4, 2): -10, (2, 1, 'AU', 2, 0): 0, (3, 1, 'AU', 3, 1): 0, (0, 3, 'AD', 0, 3): 0, (2, 4, 'AD', 2, 4): 0, (1, 4, 'AD', 1, 4): 0, (0, 0, 'AR', 1, 0): 0, (3, 1, 'AU', 3, 0): 0, (1, 1, 'AR', 0,
1): 0, (0, 4, 'AD', 0, 4): 0, (4, 0, 'AU', 4, 0): 0, (0, 4, 'AD', 0, 3): 0, (0, 0, 'AR', 0, 0): 0, (1, 0, 'AR', 0, 0): 0, (2, 4, 'AD', 2, 3): 0, (4, 1, 'AU', 4, 0): 0, (4, 0, 'AU', 4, 1): 0, (4, 1, 'AU
', 4, 1): 0, (4, 1, 'AU', 4, 2): -10})
```

5 References and Sources:

- **True Online SARSA(λ):**
  References: Reinforcement Learning, 2nd Ed. (Sutton & Barto)
  Screenshot of the algorithm is from the reference.
  Algorithm implemented from scratch for the project by Piusha Gullapalli.
  Mountain Car:
  Some of the code for mountain car domain is reused from homework implementation of Mountain car by Piusha Gullapalli, which was implemented from scratch.
  Gridworld:
  Some of the code for Gridworld domain is reused from homework implementation of Gridworld by Piusha Gullapalli, which was implemented from scratch.

- **Tabular SARSA(λ):**
  Algorithm implemented from scratch for the project. Some of the code for Gridworld domain is reused from homework implementation of Gridworld by Piusha Gullapalli.

- **Episodic Semi-Gradient n-step SARSA:**
  References: Reinforcement Learning, 2nd Ed. (Sutton & Barto)
  Screenshot of the algorithm is from the reference.
  Algorithm implemented from scratch for the project by Debabrata Halder.
  Reference for the library: https://github.com/makaveli10/reinforcementLearning **Windy Gridworld:**
  Used windy_gridworld library.
  CliffWalk:
  Used lib.envs.cliffwalking library.
  Cart-pole:
  Used OpenAI gym Cart Pole.

- **Prioritized Sweeping:**
  Algorithm implemented from scratch for the project.
  Gridworld:
  Some of the code for Gridworld domain is reused from homework implementation of Gridworld by Debabrata Halder, which was implemented from scratch.

- **Room Temperature Control Agent:**
  Libraries used for the algorithms:
  - gym
  - tensorflow.keras
  - stable_baselines3 - rl.agents
  - rl.policy
  - rl.memory

- **Value Iteration Performance:**
  Code reused from homework for Value Iteration and Monte Carlo by Debabrata Halder.

6 Contributions:

We discussed the ideas together for all parts of the project, below shows the major contributor to each section:
True Online SARSA - Piusha Gullapalli
Tabular SARSA - Piusha Gullapalli
Episodic Semi-Gradient n-step SARSA - Debabrata Halder
Room Tempterature Control agent implementation - Debabrata Halder
Value Iteration Performance implementation - Debabrata Halder
Prioritized Sweeping - Worked together
Report : Worked together