COMPSCI 687 Final Project - Fall 2021

Akshay Sharma (32219529)
Pragya Prakash (33167466)

1 Problem Statement

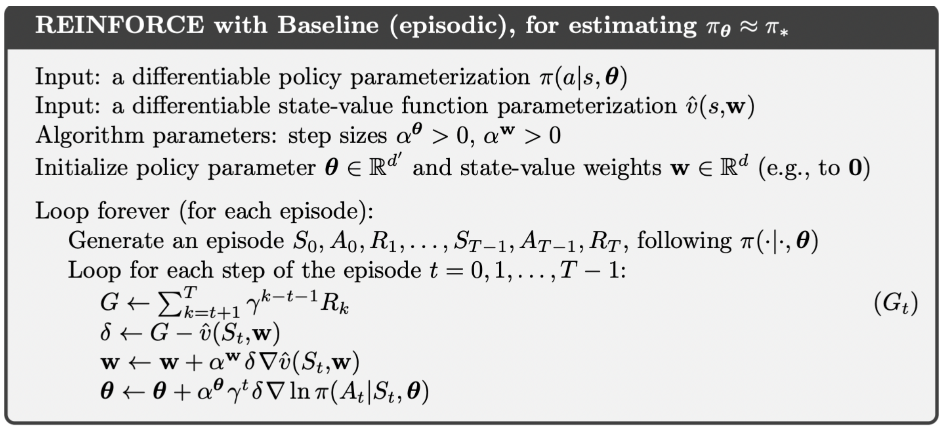
For the scope of this project, we aim to implement 3 Reinforcement Learning algorithms that we haven't studied in class, from scratch and demonstrate their working on classic environments available within OpenAI Gym. We have chosen the following algorithms:

- REINFORCE With Baseline: An extension of REINFORCE: Monte-Carlo POlicy Gradient theorem, where the policy parameters are estimated directly using gradient methods, instead of learning Q-functions/Value functions.
- Episodic Semi-Gradient n-Step SARSA: An extension of Semi-Gradient SARSA, where only the n-step returns are considered (for Q value updates), from the simulations of an episode.
- Prioritized Sweeping: A model-based algorithm that tries to optimize the agent planning phase so that fewer and only relevant updates need to be made during the learning process.

2 REINFORCE With Baseline

REINFORCE With Baseline is a more efficient version of the simple REINFORCE algorithm. It is a Policy-Gradient algorithm, meaning that it models the policy as some continuous function (say, linear function or softmax) and aims to learn the parameters that can clearly define the policy function. Policy-Gradient methods don't select actions from maximizing the Q-function, rather they simply select an action from the probability distribution governed by the policy approximation function.

The pseudocode shown in figure 1 (refer Section 13.4 of Barto and Sutton's RL Book) is implemented in Python. The code for the same can be found in the Algorithms/REINFORCE.ipynb file.



```
REINFORCE with Baseline (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 $\hat{v}(s,\mathbf{w})$
Algorithm 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):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol\theta)$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
        $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol\theta)$
```

Figure 1: Pseudocode for REINFORCE with Baseline Algorithm

2.1 Setup

We have modelled the policy as a Softmax function. We create a 1 layer Neural Network to learn policy parameters (128 nodes with ReLU activation for the hidden layer and Softmax activation for the output node). At each iteration, the next action given a current state is chosen from the probability distribution obtained by a forward pass through the Neural network representing the current policy being followed.

Another 1 layer Neural Network is trained simultaneously for learning the State Value function (also approximated by a Softmax function). Monte Carlo simulations are done using Gym OpenAI environment functions: step(): given a state and action taken, returns the next state reached and reward collected; reset() to reset the environment once an episode is completed. Once an episode has been generated, the policy and state value approximation parameters are updated by performing a backward pass on each Neural Network. The parameters that need to be finetuned for this method are the step sizes as follows:

1. Policy Learning Rate (α^{θ}): The step size for gradient based learning of the policy parameters.
2. State Value Learning Rate (α^{w}): The step size for gradient based learning of the State value parameters.

These values are carefully chosen: too large and the algorithm wont converge to the optimal parameter values as it will jump over the local solution that minimized the loss, and too small a value will take too long to converge.

2.2 Experiments and Results

To evaluate the correctness of our model, we use a ground truth training score value which is the total reward taken from the leaderboards of each of the environment on OpenAI Gym which translates to a sufficiently good performance in terms of training. We use this as a stopping condition for our training process, i.e. once the score is sufficiently stable and above this threshold, we can say the model's learning can be stopped.

2.2.1 Results on CartPole:

The Score for the CartPole environment is defined as the total reward it gets before the pole imbalances and falls. A reward of +1 is provided to the agent for every timestamp that the pole remains upright on the cart. So the total reward is the total time that the pole is balanced successfully by the agent.

Learning rate for policy parameters is 1e-3 and learning rate for state value parameters is set as 1e-2. The graph in figure 2 shows the Training Score w.r.t number of episodes generated by Monte Carlo simulations. This is seen to be increasing as more and more episodes are generated to learn to adjust the policy and state value in the direction of the optimal solutions.

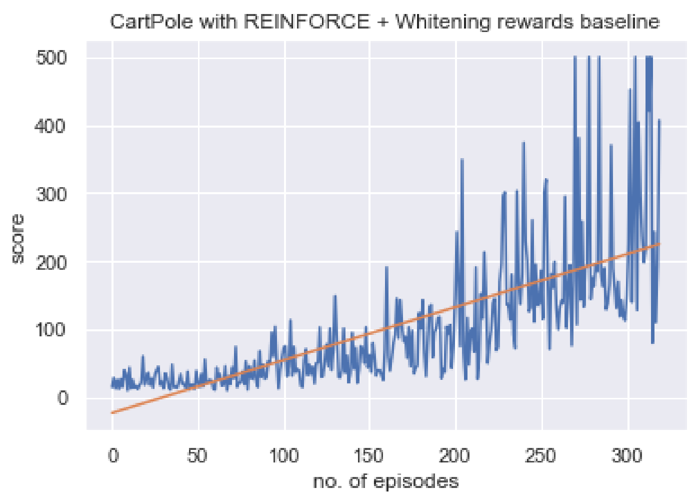


Figure 2: Training Score of REINFORCE with Whitening Rewards as Baseline for CartPole environment.

Testing the policy: we get an average score of 271.78 for CartPole, i.e. the agent is able to balance the pole for approximately 270 timestamps. The leaderboard average score value for this is 200.

2.2.2 Results on Acrobat:

The Acrobat system consists of 2 joins and 2 rods, such that the rods can swing about hinged at these joints. The aim is for the agent to learn how much thrust to apply on the first joint so that the second rod swings and reaches above a horizontal level. The score/rewards for the Acrobat agent are -1 for each timestamp that the agent has not been able to swing the rod to the required height. So, the total score is negative of the total time taken before the agent learns to swing the rod correctly.

The learning rate for policy parameters is 1e-3 and learning rate for state value parameters is set to 1e-2, again. The graph in figure 3 shows the Training Score w.r.t number of episodes generated by Monte Carlo simulations.

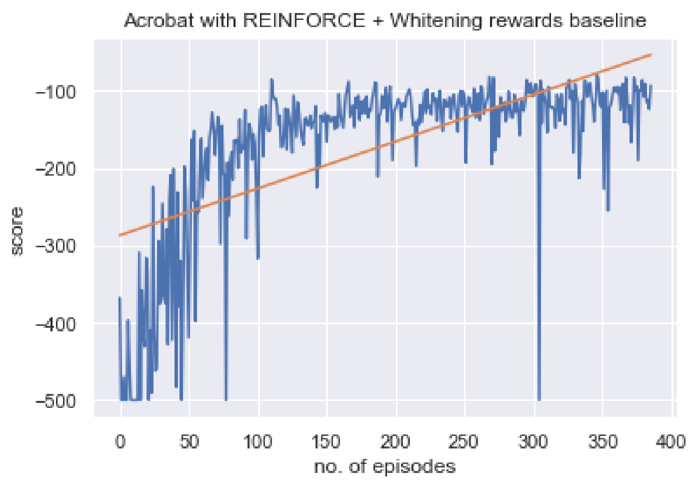


Figure 3: Training Score of REINFORCE with Whitening Rewards as Baseline for Acrobat environment.

Testing the policy: we get an average score of -109.08 for Acrobat, i.e. the agent is able to swing the acrobat up correctly after approximately 100 timestamps. The average leaderboard score value for this is -120.

3 Episodic Semi-Gradient n-Step SARSA

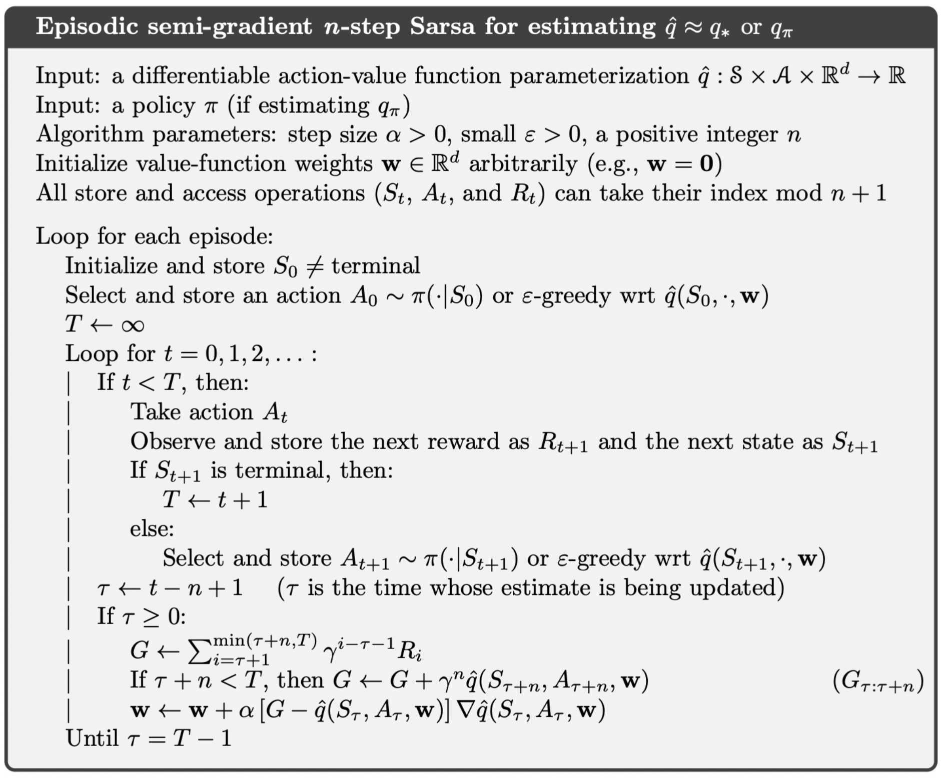
For this algorithm, the update equation of Episodic Semi-Gradient SARSA is changed to incorporate only returns from n-steps within a generated episode. The n-step discounted sum of returns is given by the following formula:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1}) \tag{1}$$

Thus, the final update equation looks like this:

$$w_{t+n} = w_{t+n-1} + \alpha[G_{t:t+n} - \hat{q}(S_t, A_t, w_{t+n-1}]\nabla\hat{q}(S_t, A_t, w_{t+n-1} \tag{2}$$

The pseudocode shown in figure 4 (refer Section 10.2 of Barto and Sutton's RL Book) is implemented in Python. The code for the same can be found in the Algorithms/N-Step_SARSA.ipynb file.



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

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy $\pi$ (if estimating $q_\pi$)
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 $\varepsilon$-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \dots$ :
  |  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 $\varepsilon$-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  |  $\tau \leftarrow t - n + 1$    ($\tau$ 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 $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$        ($G_{\tau:\tau+n}$)
  |      $\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{q}(S_\tau, A_\tau, \mathbf{w})] \nabla\hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T - 1$
```

Figure 4: Pseudocode for Epsiodic n-step Semi-Gradient SARSA Algorithms

3.1 Setup

We used Sutton Tiling Encoding [1] for linear value function approximation. The Tiling approach is commonly employed for online learning algorithms for domains that have continuous state/action spaces.

The idea behind using this approach is that a simple linear function approximator for say Mountain Car problem where the state is represented by a 2D vector (x, v), would be something like $w_1 * x + w_2 * v$ where the weights need to be learned. The problem with this is that it doesn't take into account correlations between the features, for example for very low values of x (at the top of the hill), the velocity will be high. One solution would be to use a polynomial function as an approximator. Another option is the Tiling Encoding, which maintains the linear property of the approximator.

The hyperparameters which required to be finetuned are:

- N: the number of steps to consider returns from
- Gamma: Discount parameter to determine the trade-off between far-off vs immediate rewards
- Epsilon: The exploration factor: a high epsilon encourages the agent to choose actions that look sub-optimal according to the current policy, thus ensuring exploration happens.
- Alpha/Learning Rate: step-size for gradient learning updates

[1]More details about this can be found in Section 1.7.5 (Feature Construction for Linear Methods) of https://michaeloneill.github.io/RL-tutorial.html or Section 9.5.4 of Stton and Barto's book: http://www.incompleteideas.net/book/RLbook2018.pdf#page=239. The documentation and reference manual for the Tile Coding Software is provided http://incompleteideas.net/tiles/tiles3.html and for the purpose of this project, we have taken reference from the same

3.2 Experiments and Results

With the Tiling Estimator as a Linear function approximation for the Q value function of the MDP, we can decouple the update and estimate_Q functions (provided by the Tiling library) from the SARSA Learning algorithm. The step-size for the Q function updates (alpha) is kept a high value at first as initially we need to incorporate the estimates, however bad they may be, but as learning progresses, we fractionally decrease the alpha value with the number of tilings to ensure that the optima is not skipped due to a large stepping value. The parameters for the Tiling function are set to the following:

- step_size = 0.5 initially but decayed by a factor of the number of tiles after the first few iterations. A high value in the beginning is good for faster learning, but it has to be adjusted during the process to ensure that the optimal point is not skipped.
- n = 8: Number of tilings per state dimension. Should be a power of 2.
- k = 8: Dimension of tiles

3.2.1 Results on MountainCar:

For the MountainCar domain, the agent which is born stuck in a valley, has to learn how far to back up the first hill to gain enough momentum to reach the flag at the top of the other hill. The rewards are accumulated by penalizing the agent with a -1 every timestamp that the agents objective (goal to reach the flag) has not been met. So, the total score is negative of the total time taken before the agent reaches the flag at the mountain top.

For this environment, N is set to 5, $\gamma = 0.95$, $\epsilon = 0.05$. The graph in figure 5 shows the number of time steps taken by the agent vs the number of episodes that have lapsed. We see that the after the first few iterations, the number of timesteps in each episode becomes less and stabilize when a good policy has been found by exploration.

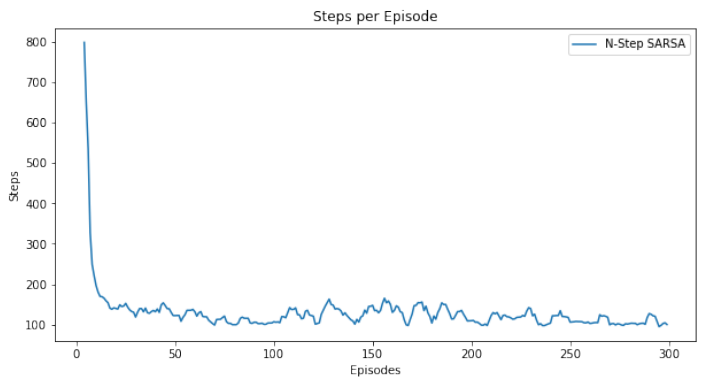


Figure 5: Timesteps per Episode for Episodic Semi-Gradient n-Step SARSA with Mountain Car environment

The graph in figure 6 shows the total Returns vs the number of episodes. We observe that the returns per episode are very bad at first, it takes a long time to reach the goal state (and each timestamp the agent is penalized with -1). As the learning progresses, the returns get better and converge to somewhere around -100.

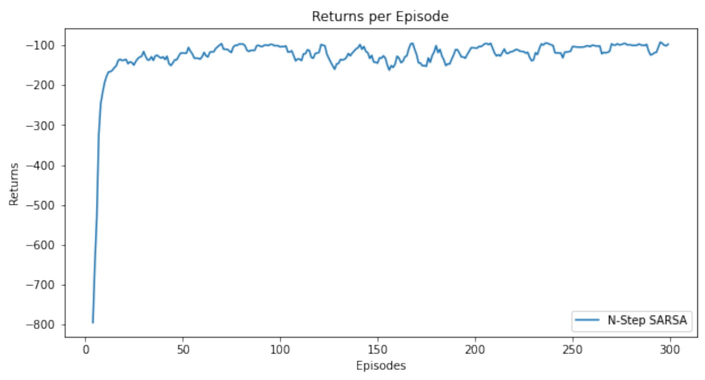


Figure 6: Returns per Episode for Episodic Semi-Gradient n-Step SARSA with Mountain Car environment

3.2.2 Results on Acrobat:

The Acrobat system consists of 2 joins and 2 rods, such that the rods can swing about hinged at these joints. The aim is for the agent to learn how much thrust to apply on the first joint so that the second rod swings and reaches above a horizontal level. The score/rewards for the Acrobat agent are -1 for each timestamp that the agent has not been able to swing the rod to the required height. So, the total score is negative of the total time taken before the agent learns to swing the rod correctly.

For this environment, N is set to 5, $\gamma = 0.9$, $\epsilon = 0.15$. The graph in figure 7 shows the number of time steps taken by the agent vs the number of episodes that have lapsed. We see that the after the first few iterations, the number of time steps in each episode becomes less and stabilize when a good policy has been found by exploration.

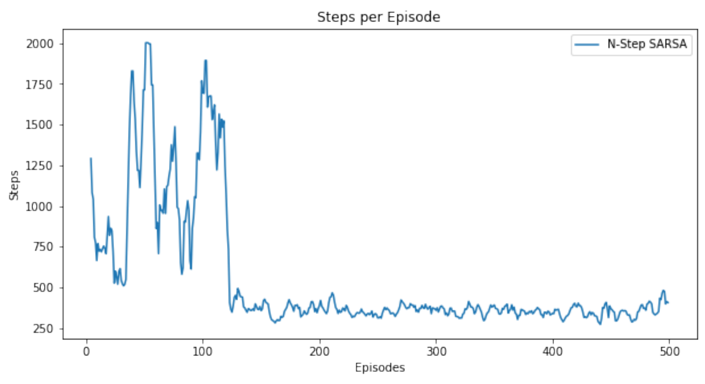


Figure 7: Timesteps per Episode for Episodic Semi-Gradient n-Step SARSA with Acrobat environment

The graph in figure 6 shows the total Returns vs the number of episodes. We observe that the returns per episode are very bad at first, it takes a long time to reach the goal state (and each timestamp the agent is penalized with -1). As the learning progresses, the returns get better and converge to somewhere around -250.

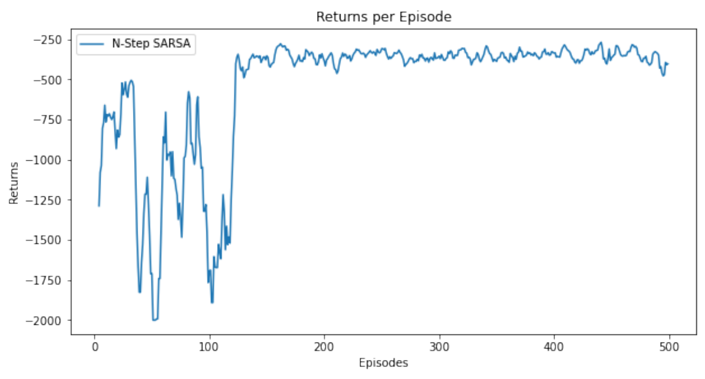


Figure 8: Returns per Episode for Episodic Semi-Gradient n-Step SARSA with Acrobat environment

4 Extra Credit: Prioritized Sweeping

Prioritized Sweeping is a form of Tabular Q-Learning algorithm, which aims to optimizing the number of updates to the Q-function at each iteration of the learning process. The motivation behind involving priorities of updates is that we know tabular updates are only relevant for those state-action pairs that are a predecessor in the path to reach states that were updated in the previous iteration. As the learning progresses, the pool of state-action pairs that need to be updated also increases, but the algorithm is efficient as it skips unnecessary updates.

Since this is a tabular method for Control and Planning, we needed environments with discrete states and action spaces. There are 2 approaches to this:

1. Setup a Gridworld environment: OpenAI Gym doesnt offer any gridworld/maze navigating agents. Thus, we implemented our own Gridworld environment from scratch, by overriding OpenAI environment classes and structured it in a generalizable format. The code for the same is provided in code file
2. Discretize the continuous State/Action spaces for environments already available (like CartPole, MountainCar etc.) by applying techniques like binning.

4.1 Setting up the Environments

4.1.1 Setting up 687 GridWorld

We have implemented the following generalizable and extendable code for generating a GridWorld Environment. To ensure this, the size $\mathbb{N}$ of the $\mathbb{N}$x$\mathbb{N}$ Grid is a user-defined variable, along with the map of obstacles and water (negative reward that the agent should avoid) states. The state space of this MDP is the numbered set of grid cells identified by the x and y coordinate. The action space consists of 4 values signifying the actions: Up, Down, Left, Right. The rewards are +10 for reaching the goal/terminal state, -10 for entering a water state and 0 for all other states.

We have assumed the agent follows an ϵ-soft policy for each episode. The learning task is to find the optimal policy for the agent to follow to reach from start position ([0,0]) to destination (bottom-right last cell).

Similar to 687-GridWorld, the agent is sometimes victim to failures: like breaking down and not moving, or veering horizontally (left or right) to the direction it was moving while taking an action. All these properties have been modelled into this MDP within the step() function. The code for the same can be seen in the Environments/GridWorld/GridEnv.py file.

4.1.2 Setting up MountainCar

We have overridden the OpenAI Gym defined MountainCar model for the purposes of Discretizing the continuous state values into bins. This is necessary because the Prioritized Sweeping algorithm is a form of tabular Q Learning. We used the OpenAI Gym provided methods env.step() and env.reset() to perform the simulations on the Car, and wrote code for assigning the continuous state values (x: horizontal position, v: velocity) into discrete bins. This introduces another hyperparameter, namely the number of bins to divide the 2 state variables into. The code for the same can be seen in the Environments/MountainCar/MCEnv.py file.

4.2 Implementing Prioritized Sweeping

The pseudo-code shown in figure 9 (refer Section 8.4 of Sutton's RL Book) was implemented in python. The code for the same can be seen in the Algorithms/PrioritizedSweeping.py file.

```
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 $P$
```

Figure 9: Pseudocode for Prioritized Sweeping Algorithm for Tabular Q-Learning

A dictionary needs to be maintained to store the model, and another one to record predecessors (state, action) pairs for each state we encounter in an episode. The Q function is initialized with random numbers, and we keep track of number of episodes completed by updating a counter whenever we reach the terminal state. The action selection is governed by a ϵ-soft policy.

The hyperparameters that need to be finetuned are as follows:

- n: the max number of predecessors to consider for Q-value updates (after extracting from the Priority queue) for each state.
- Epsilon: The epsilon parameter for the ϵ-soft policy
- Gamma: The discount parameter for the MDP, defines how important rewards in the distant future are compared to immediate rewards
- Theta: The parameter to threshold the update values to be inserted into the Priority Queue. Updates to Q value that are smaller than this theta parameter, are not inserted into the Priority Queue. The update amount for the Q-function is given by the following equation:

$$\big|R + (\gamma * \max_a Q[S', a] - Q[S, A])\big| \tag{3}$$

- Alpha: The learning rate, i.e. how much importance to give to estimated updates, needs to be chosen carefully: too large and estimated values of Q-func are given too much importance, too small and very small updates are made, thus hindering efficient learning.

4.3 Experiments and Results

The above algorithm's working is demonstrated by experimenting on the 687 GridWorld and MountainCar domain. The results of the same are described in detail in the subsequent sections:

4.3.1 Results on 687-GridWorld

The parameters are set to the following values: gamma=0.9, theta=1e-5, n=10 (at most 10 predecessors will be considered for updates for each state that is encountered), alpha=0.2 and decays by 75% every 250 iterations, epsilon=0.3 and decays by 0.05 every 250 iterations. The algorithm is run for 3000 iterations and the results are discussed below.

The optimal Value function and Policy are known to us for this environment and shown below:

4.0187	4.5548	5.1575	5.8336	6.4553
4.3716	5.0324	5.8013	6.6473	7.3907
3.8672	4.3900	0.0000	7.5769	8.4637
3.4182	3.8319	0.0000	8.5738	9.6946
2.9977	2.9309	6.0733	9.6946	0.0000

→	→	→	↓	↓
→	→	→	↓	↓
↑	↑	O	↓	↓
↑	↑	O	↓	↓
↑	↑	→	→	G

Since, we have the ground truth available (as shown above), we record the MSE of the estimated value function (calculated from the Q function using the equation $v(s) = \sum_{a \in \mathbb{A}} q(s,a)$) with the above shown optimal value function for each iteration. This can be seen in figure 10 as decreasing as the number of iterations progresses. We reach a Mean Squared Error value of 2.2 w.r.t the optimal value function for 687-GridWorld.

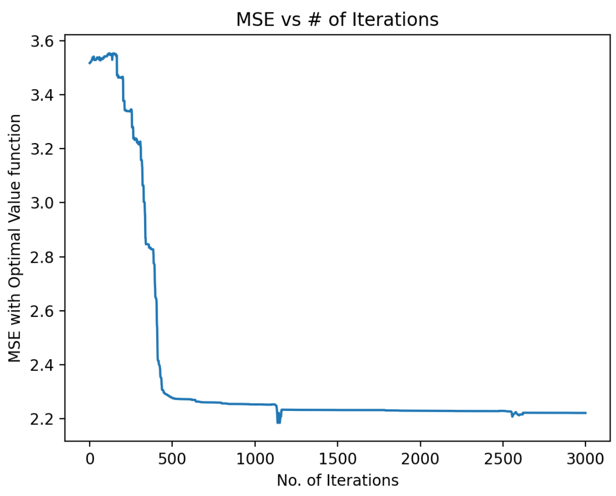


Figure 10: Mean Squared Error of the learned value function vs the optimal value function for 687-GridWorld vs the number of iterations completed.

We also record the number of updates per episode shown in figure 11. As the training progresses, the updates are lesser, as we reach close to the optimal policy.

The optimal policy learned by the Prioritized Sweeping algorithm is shown below:

→	→	→	↓	↓
→	→	→	↓	↓
↑	↑	O	↓	↓
↑	↑	O	↓	↓
→	↑	→	→	G

4.3.2 Results on MountainCar:

For the MountainCar domain, the agent which is born stuck in a valley, has to learn how far to back up the first hill to gain enough momentum to reach the flag at the top of the other hill. The rewards are accumulated by penalizing the agent with a -1 every timestamp that the agents objective (goal to reach the flag) has not been met. So, the total score is negative of the total time taken before the agent reaches the flag at the mountain top.

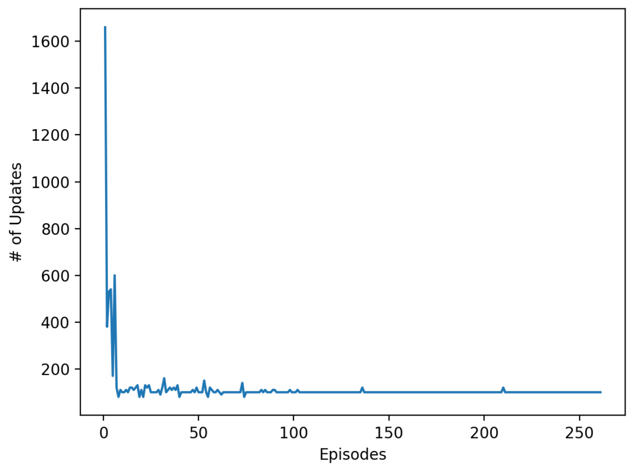


Figure 11: The number of Q-value updated made for 687-GridWorld vs the number of episodes completed.

The hyperparameters are set as follows for this environment: number of bins for x = 6, number of bins for v = 3, gamma=0.9, theta=1e-5, n=5 (at most 5 predecessors will be considered for updates for each state that is encountered), alpha=0.05 and decays by a factor of 0.9 every 5 iterations, epsilon=0.25 and decays by 0.05 every 5 iterations. The algorithm is run for 1000 iterations. The learning curves are shown below:

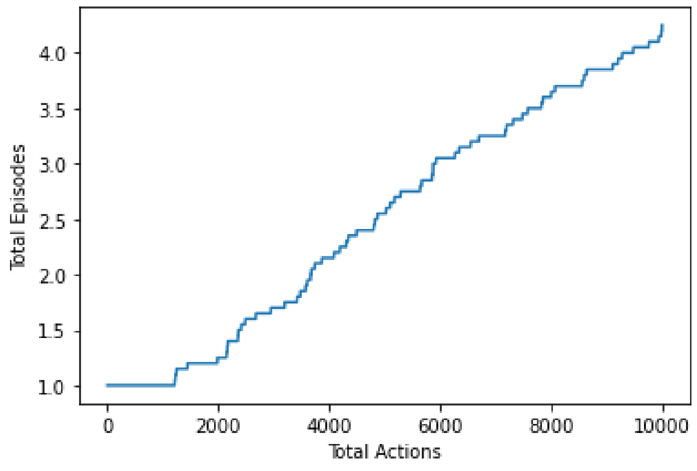


Figure 12: Total episodes vs cumulative number of actions taken till then

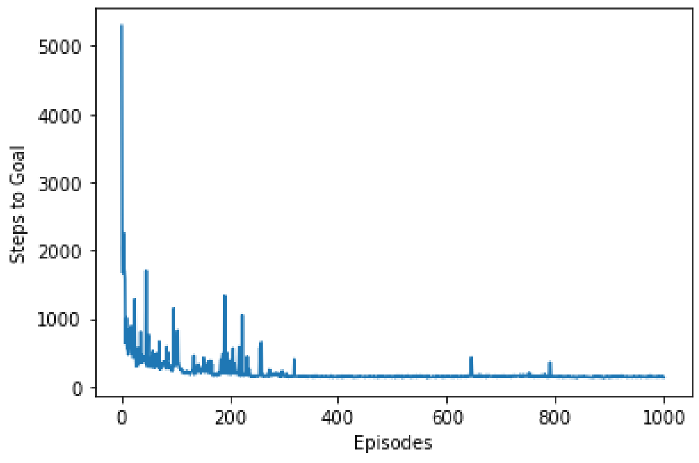


Figure 13: Number of steps to reach Goal position vs Number of Episodes

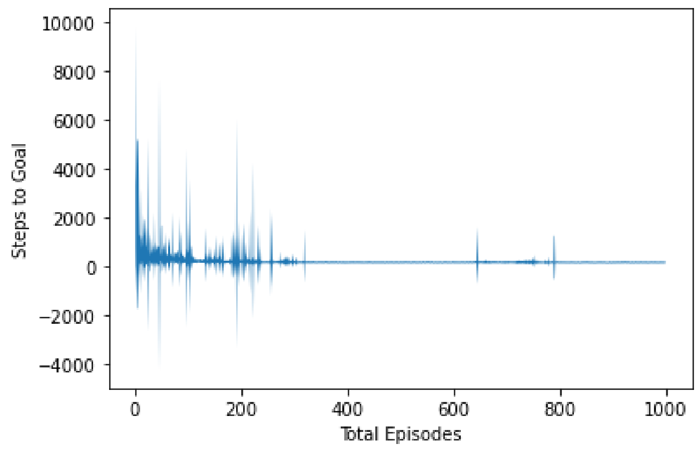


Figure 14: Standard Deviation of steps to reach Goal position w.r.t Number of Episodes

5 Work Distribution

REINFORCE With Baseline: Akshay Sharma
Episodic Semi-Gradient: Pragya Prakash
Prioritized Sweeping with Gridworld: Akshay Sharma
Prioritized Sweeping with MountainCar: Pragya Prakash
Report Writing: Pragya Prakash, Akshay Sharma