Reinforcement Algorithms on Mountain Car and Gridworld MDPs

Satwik Goyal and Kunj Dhedia

1 One-step Actor Critic

(i) The One-step Actor Critic algorithm is a policy gradient method that estimates the optimal policy. The method learns an actor θ that decides which action to pick given a state and a critic that evaluates the perfomance of the actor. The policy is represented using a differentiable parameterization $\pi(\cdot, \cdot; \theta)$ - for instance, softmax and the state value function is represented using differentiable approximator $\hat{v}_w(s)$. Since the algorithm is episodic, for each transition from state s to state s', it computes the TD error and updates the parameters θ and w to maximize the average return. This method specifically, computes the one-step return using the estimated value of the second state s', when discounted and added to the reward, which provides an estimate with lower variance.

(ii) Pseudocode for One-step Actor Critic

Algorithm 1 One-step Actor Critic

step size $\alpha^w \in (0, 1]$
step size $\alpha^\theta \in (0, 1]$
$w \leftarrow 0$
$\theta \leftarrow 0$
for `each episode` **do**
  $s \leftarrow d_0$
  while s `is not terminal` **do**
    $a \leftarrow \pi(s, \cdot; \theta)$
    Execute action a, observe reward r and next state s'
    $\delta \leftarrow r + \gamma \hat{v}^\pi_w(s') - \hat{v}^\pi_w(s)$
    $w \leftarrow w + \alpha^w \delta \nabla \hat{v}^\pi_w(s)$
    $\theta \leftarrow \theta + \alpha^\theta \delta \nabla \ln(\pi(s, a; \theta))$
    $s \leftarrow s'$
  end while
end for

(iii) Hyper-parameter tuning

(a) Learning Rate (α) - If the learning rate is set too low, the training will progress very slowly as the algorithm will be making making very tiny updates to the weights in the network. In addition, the learning might get stuck in a local minima. However, if the learning rate is set too high, it can cause undesirable divergent behavior in your loss function. Consequently, I fine-tuned α such that it is not too low or not too high.

(b) Dimension of State Representation - If the number of dimensions is too low, the representation might not capture enough information about the state. However, if the number of dimensions is too high, the number of learnable parameters in the model increases, increasing the computational requirement. I performed a randomized search over the dimensions that represent the states and picked the one that worked best.

(iv) Experimental Results -

(a) 687-GridWorld

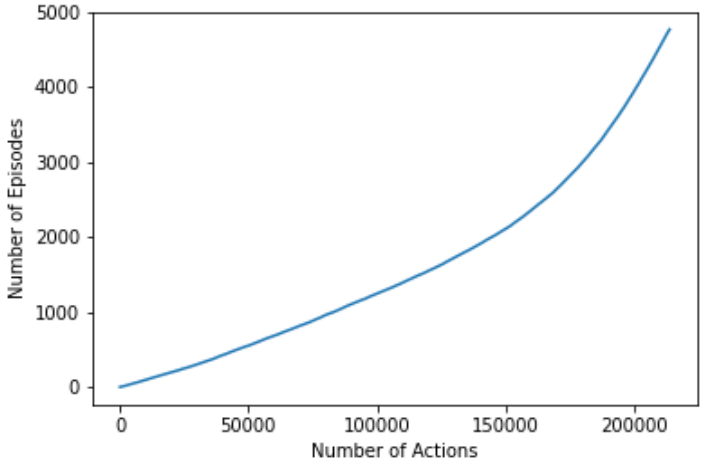

Figure 1: Learning Curve - Number of Episodes vs Number of Actions

Since this graph has an increasing slope, it indicates that as the agent takes more and more actions (and thus executes more learning updates), it requires fewer actions/timesteps to complete each episode.

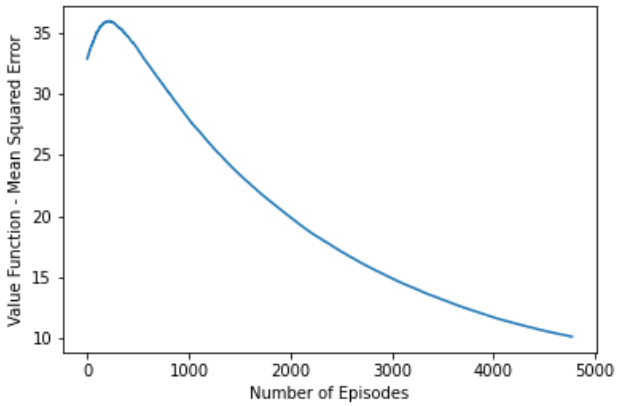

Figure 2: Learning Curve - Avg MSE vs Number of episodes computed over 20 iterations

As the algorithm learns the optimal policy, the mean squared error between the estimate value function and the optimal value function decreases.

(b) Mountain Car

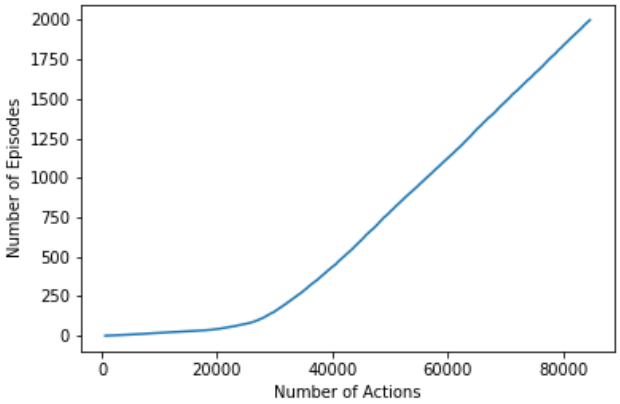

Figure 3: Learning Curve - Number of Episodes vs Number of Actions

Since this graph has an increasing slope, it indicates that as the agent takes more and more actions (and thus executes more learning updates), it requires fewer actions/timesteps to complete each episode.

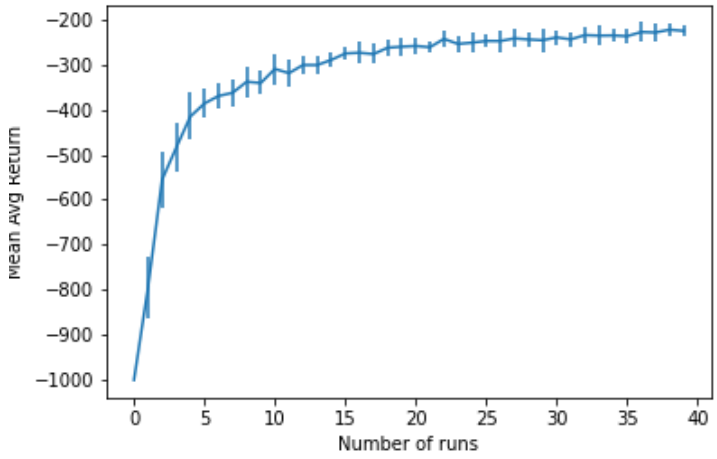

Figure 4: Learning Curve - Mean Avg Return vs Number of Runs, computed over 20 iterations

Since the Mountain Car MDP is episodic — not because there is a terminal state that is guaranteed to be reachable under any policy, but because we have defined a timeout, the "episode" ends after 1000 steps, with a -1 reward at each time step. Consequently, the average return is -1000 at first and then reduces as the policy converges to the optimal policy.

2 True Online SARSA(λ) using Tabular Features

(i) SARSA(λ) uses Eligibility Traces to unify SARSA and Monte Carlo methods to make a more general method. SARSA(λ) works like SARSA methods when λ approaches 0 and works liked Monte Carlo methods as λ approaches 1. True Online SARSA(λ) is algorithmically very similar to SARSA(λ), however, it follows the underlying ideas of forward view more closely (van Seijen et al., 2016). In SARSA(λ), the algorithm approximates the forward view for for optimally small time-steps, however, in True Online SARSA(λ) the algorithms exactly computes the weight vectors for any time-step (van Seijen & Sutton, 2014).

(ii) Pseudocode for True Online SARSA(λ) using Tabular Features

Algorithm 2 True Online SARSA(λ) using Tabular Features

step size $\alpha > 0$
trace decay rate $\lambda \in [0, 1]$
small $\epsilon > 0$
initialize $q(s, a)$ for all (s, a)
for `each episode` **do**
  $s \leftarrow d_0$
  $a \leftarrow$ action using $\epsilon-$greedy
  $e(s, a) \leftarrow 0$ for all (s, a)
  $Q_{old} \leftarrow 0$
  while s `is not terminal` **do**
    Obtain next state s' and Reward r
    $a' \leftarrow$ action using $\epsilon-$greedy
    $\Delta Q \leftarrow q(s, a) - Q_{old}$
    $Q_{old} \leftarrow q(s', a')$
    $\delta \leftarrow r + \gamma q(s', a') - q(s, a)$
    $e(s, a) \leftarrow (1 - \alpha)e(s, a) + 1$
    for all (s, a): **do**
      $q(s, a) \leftarrow q(s, a) + \alpha(\delta + \Delta Q)e(s, a)$
      $e(s, a) \leftarrow \gamma\lambda e(s, a)$
    end for
    $s \leftarrow s'$
    $a \leftarrow a'$
  end while
end for

Note 1: This algorithm was adapted from "True Online Temporal-Difference Learning" by Richard S. Sutton, Marlos C. Machado, Patrick M. Pilarski, A. Rupam Mahmood, and Harm van Seijen.

Note 2: For continuous states we used bins to discretize them.

(iii) Hyper-Parameter Tuning

(a) Step Size α: It is the learning rate/step size at which the weight vectors learn/are updated. In general α should be proportional to $\frac{1}{d}$ where d is the size of the weight vectors. To tune α we used Randomized Search among values that were proportional to 0.1 for both the MDPs and obtained the optimal values. It is because when the learning rate is very high and the weight vector will have values with very high magnitudes that results in no learning.

(b) Trace Decay Rate λ: It determines the trade off between bias and variance. We used randomized search for λ. Optimal λ is usually around when the trade off between bias and variance is right for the MDP.

(c) Exploration Technique: We tried several different techniques to explore different actions. For Gridworld initializing action-values optimistically worked the best. For Mountain Car decaying $\epsilon-$greedy with $\epsilon = 1$ with a decay of 0.01 per iteration until $\epsilon = 0.05$ worked the best.

(d) Number of bins: For the number of bins for continuous states we empirically obtained the optimal value using Randomized Search.

(iv) Gridworld

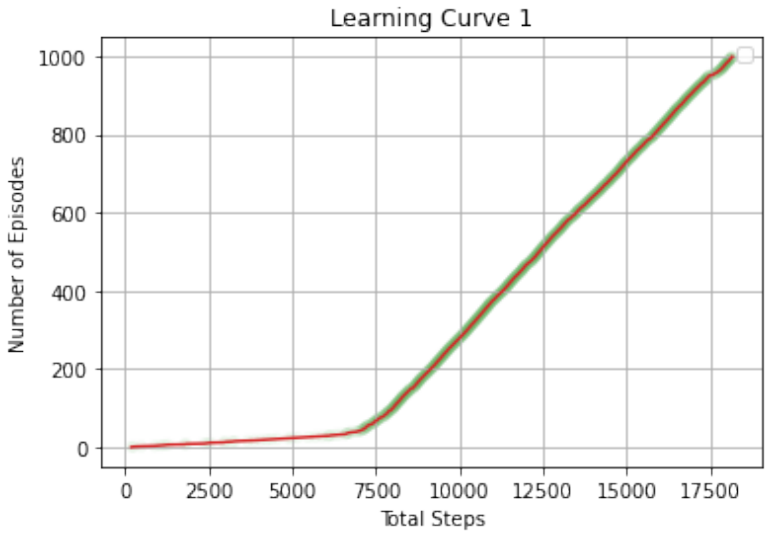

Figure 5: Learning Curve: Number of Episode v/s Total Steps

Since the graph has a positive increasing gradient we can conclude that the agent is learning. A significant increase in gradient is observed between episode 50 and episode 150.

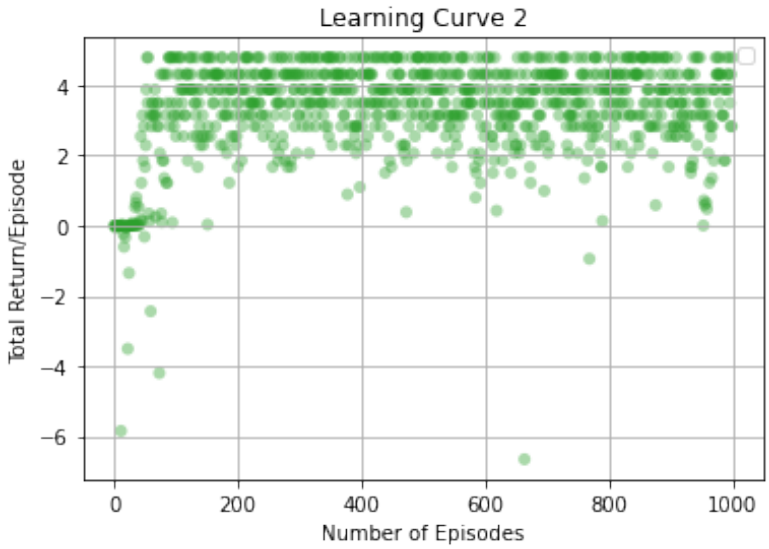

Figure 6: Learning Curve: Total Return per Episode v/s Number of Episode

In this graph we can observe that the total return at the end of the episode starts to converge at the optimal, however, due to high variance and stochastic transition functions we still observe that the total return defers from episode to episode.

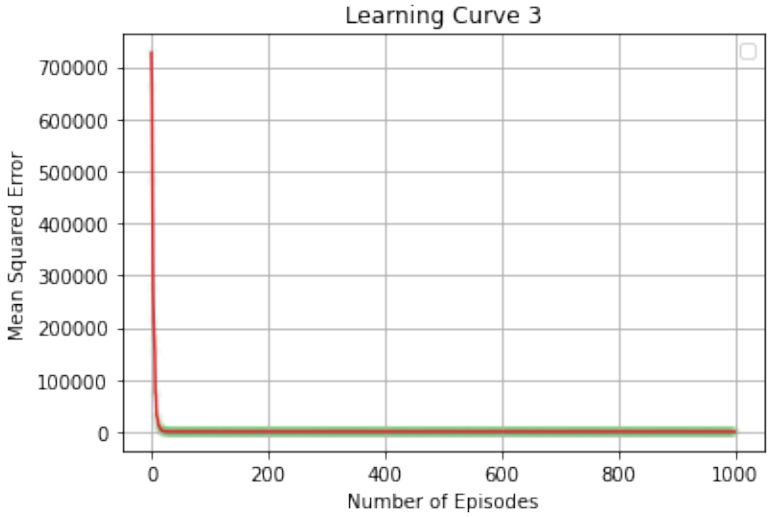

Figure 7: Learning Curve: Mean Squared Error v/s Number of Episode

In this graph we observe that the Mean Squared Error between optimal value-function and estimated value function decreases significantly. At the end of the algorithm the mean squared error was 0.3204084201347057.

Optimal Policy Estimated by the Algorithm

	0	1	2	3	4
0	→	→	→	→	↓
1	→	→	→	→	↓
2	↑	↑		→	↓
3	↑	←		→	↓
4	↑	←	→	→	G

Mountain Car

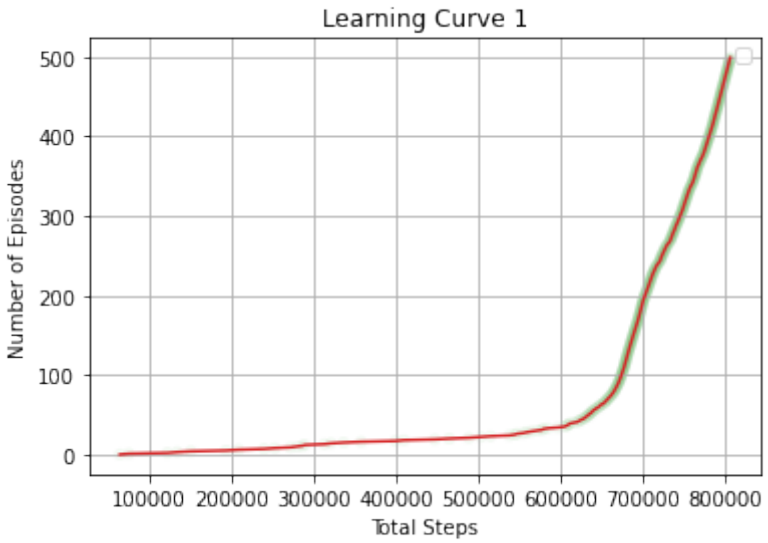

Figure 8: Learning Curve: Number of Episode v/s Total Steps

Since the graph has a positive increasing gradient we can conclude that the agent is learning. A significant increase in gradient is observed between episode 40 and episode 200.

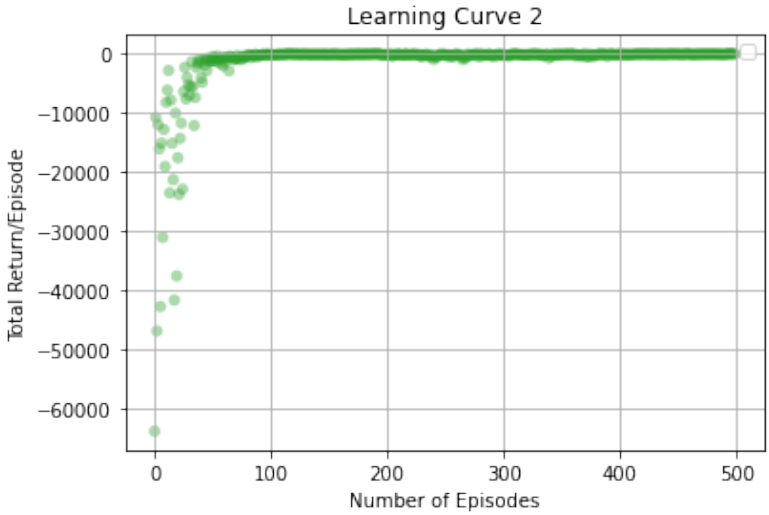

Figure 9: Learning Curve: Total Return per Episode v/s Number of Episode

In this graph we can observe that the total return at the end of the episode starts to converge at the optimal.

3 Prioritized Sweeping

(i) Most episodic algorithms learn from simulated transitions that start from a state selected uniformly at random. The updates at each time step can be much more efficient if simulated transitions and updates are focused on particular state–action pairs. With most mdps, at the beginning of the second episode, only the state–action pair leading directly into the goal has a non-zero value; the values of all other pairs are still zero causing redundant updates. Prioritized Sweeping takes a non-episodic approach by working back not just from goal states but from any state whose value has changed. For instance, if the value of a particular state changes, the value for state-action pairs leading to that state would also change. Consequently, the prioritized sweeping algorithm maintains a priority queue of state-action pairs whose value is expected to change based on the values of states that have changed, prioritized by how large the change was.

(ii) Pseudocode for Prioritized Sweeping

Algorithm 3 Prioritized Sweeping

initialize $Q(s,a)$ for all $s \in S$ and $a \in A$, PQueue to empty
while $True$ **do**
  $s \leftarrow d_0$
  $a \leftarrow policy(s,Q)$
  $Q_{old} \leftarrow Q(s,a)$
  $P^{\hat{r}}_{\hat{s}} :=$ `probability of` $(\hat{s},\hat{r})$ `in Model(s, a)`
  $Q(s,a) = \sum_{\hat{s},\hat{r}} P^{\hat{r}}_{\hat{s}}(\hat{r} + \gamma \max_{\hat{a}} Q(\hat{s},\hat{a}))$
  $p \leftarrow Q(s,a) - Q_{old}$
  if $p > \theta$ **then**
    insert s into PQueue with priority p
  end if
  $updates \leftarrow 1$
  while `PQueue is not empty and` $updates \leq k$ **do**
    $s \leftarrow$first(PQueue)
    for `all` $(\bar{s},\bar{a})$ `predicted to lead to s` **do**
      $Q_{old} \leftarrow Q(\bar{s},\bar{a})$
      $P^{\hat{r}}_{\hat{s}} :=$ `probability of` $(\hat{s},\hat{r})$ `in Model`$(\bar{s},\bar{a})$
      $Q(\bar{s},\bar{a}) = \sum_{\hat{s},\hat{r}} P^{\hat{r}}_{\hat{s}}(\hat{r} + \gamma \max_{\hat{a}} Q(\hat{s},\hat{a}))$
      $p \leftarrow Q(\hat{s},\hat{a}) - Q_{old}$
      if $p > \theta$ **then**
        $updates \leftarrow updates + 1$
        insert $\hat{s}$ into PQueue with priority p
      end if
    end for
  end while
end while

Note: In order to account for stochastic environments (687-GridWorld), I implemented an extension of prioritized sweeping. Instead of updating each pair with a sample update, it was updated with an expected update that is taking into account all possible next states and their probabilities of occurring as described in the pseudocode above.

(iii) Hyper-parameter tuning

(a) Epsilon Decay - Since the algorithm initializes and underestimates the action-value function to zero, epsilon is initially set to 1 to allow exploration. Over multiple iterations of the algorithm, the epsilon is decayed to 0.1 and stays constant thereon.

(b) Maximum iterations (k) - This specifies the number of preceeding states that will be popped out from PQueue during each iteration. Since γ quantifies the importance of future rewards, collecting rewards after a certain number of transitions is redundant (nearly zero). Consequently, k is governed by the γ of the mdp.

(iv) Experimental Results -

(a) 687-GridWorld

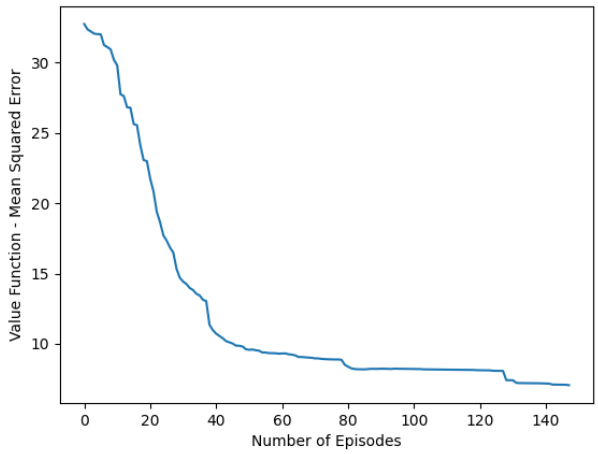

Figure 10: Learning Curve - Avg MSE vs Number of iterations computed over 20 runs

As the algorithm learns the optimal policy, the mean squared error between the estimate value function and the optimal value function decreases.

Average Value function, computed over 20 runs

3.9155	4.4251	5.0215	5.6806	6.3692
4.2331	4.8593	5.6131	6.4505	7.2804
3.7644	4.2461	0.0000	7.3661	8.3417
3.3206	3.6998	0.0000	8.3017	9.5631
2.9048	2.5595	4.7286	8.8949	0.0000

Average Policy

→	→	→	↓	↓
→	→	→	↓	↓
↑	↑		↓	↓
↑	↑		→	↓
↑	←	→	→	G

4 SARSA(λ) using Linear Approximation

(i) Using TD (Temporial Difference) and Monte Carlo methods, the agents can learn directly from their raw environment without the knowledge of any distributions. TD Methods are considered to have high biases whereas Monte Carlo methods are considered to have high variances. TD(λ) uses Eligibility Traces to unify TD and Monte Carlo methods to make a more general method. TD(λ) works like TD methods when λ approaches 0 and works liked Monte Carlo methods as λ approaches 1. In SARSA(λ), the algorithm approximates the forward view for for optimally small time-steps.

(ii) Pseudocode for True Online SARSA(λ) using Tabular Features

Algorithm 4 SARSA(λ) using Linear Approximation

input a feature function $f : S \to R^d$ such that f : terminal $= 0$
step size $\alpha > 0$
trace decay rate $\lambda \in [0, 1]$
small $\epsilon > 0$
weight vector size $d \in Z_+$
initialize $w \in R^{d \times |A|}$
for `each episode` **do**
  $s \leftarrow d_0$
  $x \leftarrow f(s)$
  $a \leftarrow$ action using $\epsilon-$greedy
  $e \leftarrow \mathbf{0}$ such that $e \in R^{d \times |A|}$
  while s `is not terminal` **do**
    Obtain next state s' and Reward r
    $x' \leftarrow f(s')$
    $a' \leftarrow$ action using $\epsilon-$greedy
    $\delta \leftarrow r + \gamma w[a'] \cdot x' - w[a] \cdot x$
    $e \leftarrow \gamma \lambda e$
    $e[a] \leftarrow e[a] + x$
    $w \leftarrow w + \alpha \delta e$
    $x \leftarrow x'$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end while
end for

Note: This algorithm was adapted from Phillip Thomas' Lecture 24 of Computer Science 687.

(iii) Hyper-Parameter Tuning

(a) Step Size α: It is the learning rate/step size at which the weight vectors learn/are updated. In general α should be proportional to

$\frac{1}{d}$ where d is the size of the weight vectors. To tune α we used Randomized Search among values that were proportional to 0.01 and obtained the optimal values. It is because when the learning rate is very high and the weight vector will have values with very high magnitudes that results in no learning.

(b) Trace Decay Rate λ: It determines the trade off between bias and variance. We used randomized search for λ. Optimal λ is usually around when the trade off between bias and variance is right for the MDP.

(c) Exploration Technique: We tried several different techniques to explore different actions. For Mountain Car decaying $\epsilon-$greedy with $\epsilon = 1$ with a decay of 0.01 per iteration until $\epsilon = 0.05$ worked the best.

(d) Weight vector size d: For the size of weight vectors we empirically obtained the optimal value using Randomized Search.

(iv) Mountain Car

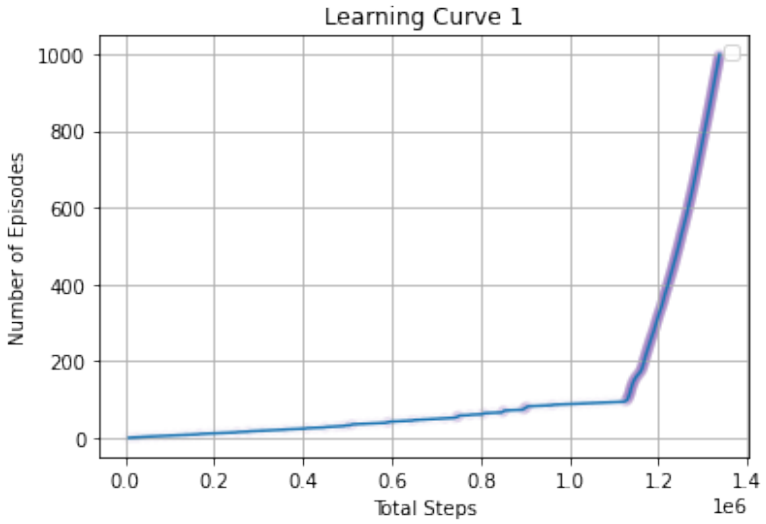

Figure 11: Learning Curve: Number of Episode v/s Total Steps

Since the graph has a positive increasing gradient we can conclude that the agent is learning. A significant increase in gradient is observed between episode 80 and episode 200.

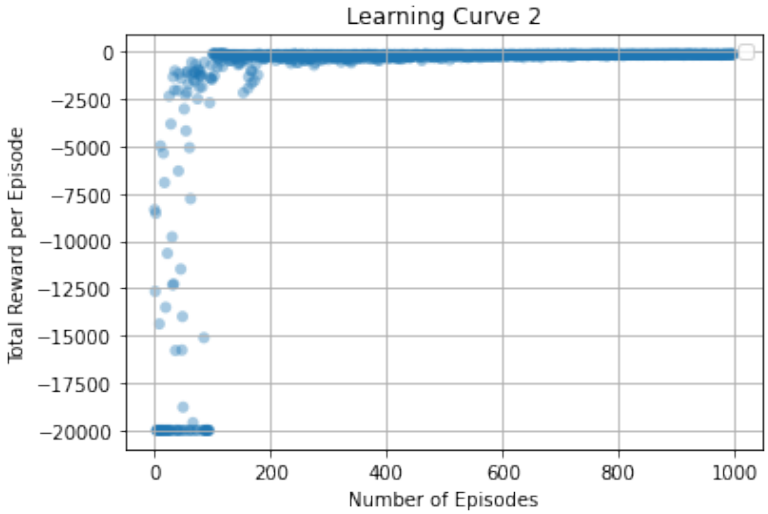

Figure 12: Learning Curve: Total Return per Episode v/s Number of Episode

In this graph we can observe that the total return at the end of the episode starts to converge at the optimal.