Reinforcement Learning Project

Sai Vineeth Kumar Dara (sdara@umass.edu)
Rohith Siddhartha Bheemreddy (rbheemreddy@umass.edu)

December 8, 2023

1 Introduction

For the final RL 687 project, we implemented two algorithms - True Online Sarsa(λ) algorithm and the One-Step Actor-Critic algorithm. And we had tested these algorithms on three different MDPs - the 687-Grid World MDP, the Cartpole MDP, and the Mountain Car MDP.

The model dynamics for the 687-GridWorld MDP and the Cartpole MDP are same as the ones we used in the homeworks (HW3 and HW2 respectively). And the model dynamics for the MountainCar are taken from the gym library documentation here.

The pseudo codes for the algorithms and the corresponding experimental results are shown in later sections.

2 Algorithms

2.1 True Online Sarsa (λ) algorithm

We implemented the true online version of Sarsa(λ) algorithm. The pseudo for the same is given below: (pseudo code is taken from section 12.7 of Sutton & Barto's RL 2nd edition book).

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

Figure 1: True Online Sarsa (λ) pseudo code

2.2 One-Step Actor-Critic algorithm

Actor-Critic Algorithms are the algorithms that learn approximations to both policy and value function are called actor-critic methods, where actor is a reference to a learned policy and critic refers to learned state-value function. One-Step Actor-Critic Algorithm combine both elements of policy-based (actor) and value-based (critic) methods to improve learning efficiency. Update is performed at every step to update actor's policy and critic's value. Actor is responsible for determining the agent's actions based on the current policy. It selects actions that maximize expected cumulative rewards over time. The critic evaluates the state pairs, providing feedback on the goodness of the chosen actions. Algorithm is fully online and incremental, with states, actions, and rewards processed as they occur and then never visited.

One-Step actor we use parameterized policy with parameters θ where is perform gradient updates to search for policy parameters that maximize expected returns and it tends to guarantee convergence. This algorithm follows biased estimates of the gradient where we update the policy parameters at every step which induces the bias. TD Error is calculated before we update gradient estimate. This algorithm is an extension or modification of REINFORCE algorithm which used MonteCarlo return whereas in One-Step Actor Critic we follow TD return. In other words, One-step actor-critic method replaces the full return of REINFORCE with one-step return, Hence the name One-step Actor-Critic Algorithm.

TD Error (δ) is calculated using the following equation where S is the current state, S' is the next state, $\hat{v}$ is the value estimate.

$$\delta = R + \gamma V(S', w) - V(S, w)$$

State (w) and Policy (θ) parameters are updated at every step using the following equation, where α^w, α^θ are learning rates for updating the parameters of critic (value function) and the actor (policy) respectively.

$$w = w + \alpha^w \delta \nabla V(S, w)$$

$$\theta = \theta + \alpha^\theta \delta \nabla \ln(\pi(A|S, \theta))$$

Since the action space is discrete in the MDP's choosen. We can use softmax function to convert raw action preferences into a probability distribution over discrete actions and the function is also differentiable which is crucial to perform gradient updates. Softmax policy is defined as follows, where θ_i represents parameters associated with action i and s is the state:

$$\pi(a|s, \theta) = \frac{e^{\theta_i^T s}}{\sum_j e^{\theta_j^T s}}$$

Derivative of softmax function w.r.t θ:

$$\frac{\partial \pi(a|s, \theta)}{\partial \theta_i} = \pi(a|s, \theta) \cdot (1 - \pi(a|s, \theta)) \cdot s$$

Update when the same action is chosen:

$$\theta_i \leftarrow \theta_i + \alpha^\theta \delta \cdot (1 - \pi(a|s, \theta)) \cdot s$$

Update when the different actions are chosen:

$$\theta_i \leftarrow \theta_i - \alpha^\theta \delta \cdot \pi(a|s, \theta) \cdot s$$

Below is the pseudocode to One-Step Actor-Critic Algorithm from RL Book:

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 $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Figure 2: One-step Actor-Critic pseudo code

3 Algorithm implementation

In this section, we will describe the methods we implemented in the code and their description.

3.1 Methods corresponding to MDPs

getInitState()

The function returns the initial state for an episode. This is called at the start of each episode. The function logic depends on the MDP we are working with.

- For GridWorld MDP, the function uniformly selects one of the 23 valid states (non-wall) and returns it.
- For Cartpole MDP, the function returns the tuple/array (0,0,0,0) meaning all the state components, x - horizontal position, v - the cart velocity, ω - the pole angle, $\dot{\omega}$ - the pole's angular velocity are zeros.
- For MountainCar MDP, the function returns the tuple (x, v) where x, the position of the car is assigned a uniform random value in [-0.6 , -0.4] and v - the starting velocity of the car is assigned to 0.

isTerminalState(s)

The function takes input the state of the MDP and returns True if the state is a terminal state and False if the state is a non-terminal state. Again, the function logic depends on the MDP we are working with.

$$
\begin{aligned}
&if\ \ terminalcondition: \\
&\qquad return\ \ True \\
&else: \\
&\qquad return\ \ False
\end{aligned}
$$

- For Grid World MDP, the Goal state is the only terminal state, all other states are non-terminal. So when s is (4,4) - goal state, then the function returns $True$, for every other state, the function returns $False$.
- For Cartpole MDP, the termination conditions are as follows: horizontal position, $x < -2.4$ or $x > 2.4$, pole angle, $\omega < -\pi/15$ or $\omega > \pi/15$, time steps > 500 which is not dependent on the state.
- For Mountain car, the termination conditions are as follows: position of the car, $x \geq 0.5$.

getFeatureVec(s, a, M)

The function is used to construct the state-action feature vector for a given state s and action a. Here we use Fourier features to represent the state s. The function therefore also takes as input M, which is the order of Fourier Basis. If k is the number of elements of state s, the number of features in the state representation will be $(M+1)^k$. The number of state features grow exponentially with the number of elements of state s. To form state-action feature vector we take $|A|$ such state feature length vectors and concatenate them to form a state-action feature vector. The range of values in the feature vector that correspond to the given action will be set to the state features and the rest of the values will be set to zeros. This is similar to one hot encoding but instead of one we have state features.

getAction(s, w, epsilon, M)

The function returns the action to be chosen when the agent is in state s, following ϵ-greedy policy. Since we are following ϵ-greedy we will find the q values corresponding to all actions at state s and choose an action based on epsilon-greedy policy probabilities.

$$\pi(s,a) = \begin{cases} \frac{1-\epsilon}{|\mathcal{A}^*|} + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a \in \mathcal{A}^* \text{ where } \mathcal{A}^* = \arg\max_{a \in \mathcal{A}} \hat{q}(s,a). \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise,} \end{cases}$$

getGreedyAction(s, w, M)

The function returns the greedy action the agent chooses when it is in state s. We will find the q values corresponding to all the actions at s and choose an action with best $q(s,a)$ value. If there are multiple actions with best $q(s,a)$, we will uniformly pick one action of those best actions.

$$\pi(s,a) = \begin{cases} \frac{1}{|\mathcal{A}^*|} & \text{if } a \in \mathcal{A}^* \text{ where } \mathcal{A}^* = \arg\max_{a \in \mathcal{A}} \hat{q}(s,a). \\ 0 & \text{otherwise,} \end{cases}$$

getNextState(s, a)

The function takes the current state, action and returns the next state the agent goes to. The exact logic will depend of the MDP and the state dynamics.

getReward(s, a, s')

The function takes as input the current state, action and the next state and gives output the reward for this transition. Again the exact reward depends on the MDP we are working on.

3.2 Methods corresponding to True Online Sarsa(λ) Algorithm

getInitWeight()

The function initializes the weights and returns them to the true online sarsa algorithm. The function is called at the start of every run of the true online sarsa algorithm. There are different strategies to initialize weights. We can initialize randomly, or with zeros or optimistically. The size of the weight vector depends on the size of the state-action feature vector.

runEpisode()

The function basically simulates one episode of the true online sarsa algorithm. The logic of this function is same as the pseudo code given in the true online sarsa introduction.

runSarsa()

The function basically simulates one run of the true online sarsa algorithm. The logic of this function is same as the pseudo code given in the true online sarsa introduction.

runSarsaNtimes()

The function simulates N runs of the true online sarsa algorithm. At the end of N runs, the results are averaged across the N runs and are plotted. The results will be shown in the later sections.

tuningParameters()

The function handles the hyper parameter tuning of the algorithm. We run the algorithm for a range of values of the parameter and based on the outputs, we choose the parameter value that worked best. We do this for all the parameters and at the end, we select the parameter values that gave best results.

3.3 Methods corresponding to One-Step Actor-Critic Algorithm

getInitWeight()

Function returns the weights by initializing them, which are further used in One-Step Actor-Critic algorithm. This function is called at start of every run of the algorithm. There are different strategies to initialize weights. We can initialize randomly, or with zeros or optimistically. The size of the weight vector depends on the size of the state feature vector.

runEpisode()

The function basically simulates one episode of the One-step Actor-Critic algorithm. The logic of this function is same as the pseudo code given in section 2.2.

runOAC()

The function basically simulates one run of the One-step Actor-Critic algorithm. The logic of this function is same as the pseudo code given in section 2.2

runOAC_Ntimes()

The function simulates N runs of One-step Actor-Critic algorithm. At the end of N runs, the results are averaged across the N runs and are plotted. The results will be shown in the later sections.

tuningParameters()

The function handles the hyper parameter tuning of the algorithm. We run the algorithm for a range of values of the parameter and based on the outputs, we choose the parameter value that worked best. We do this for all the parameters and at the end, we select the parameter values that gave best results.

4 Experimental Setup and Results

4.1 True Online Sarsa (λ)

We tested the algorithm on three MDPs - the 687-Grid World MDP, the Cartpole MDP, and the Mountain Car MDP. Below we show the hyperparameter tuning and the algorithm results for each of these MDPs.

For searching the hyperparameters, I used the following method: First, based on general RL training practices, I fixed the hyperparameters to a set of values. Starting with these as base assumptions, I targeted one hyperparameter at a time. I checked the performance for different values of that hyperparameter keeping the rest of the hyperparameters same. By comparing their learning curve graphs, I selected the hyperparameter value that worked best. I similarly tuned for all the parameters and found the set of parameters that worked best.

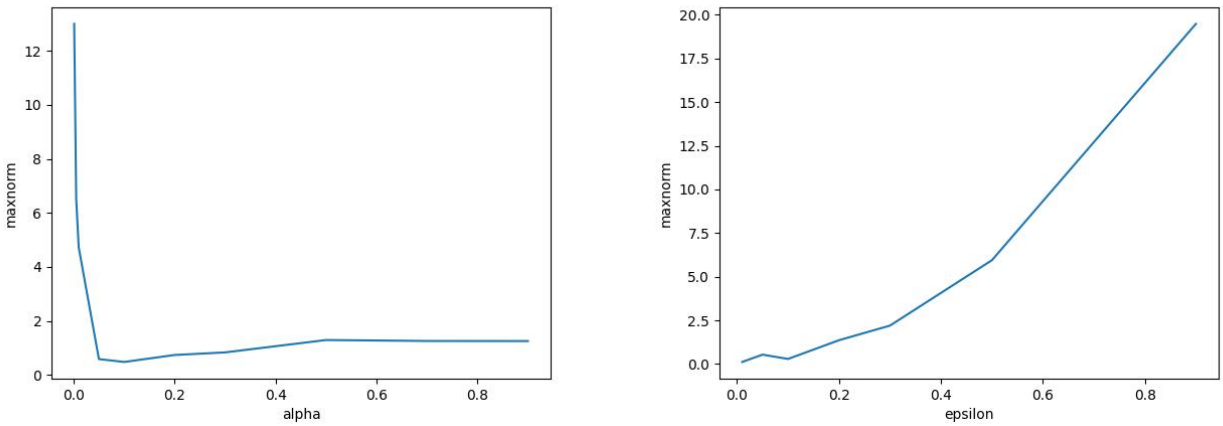
4.1.1 687 - Grid World

The model dynamics of the grid world are taken same as HW3 assignment. We used One Hot Encoding for state feature vector representation that is the cell corresponding to the state s and action a is set to 1 and every other value in the vector is set to 0. This is a simple representation for MDPs with discrete states and actions.

In order to measure the performance of the algorithm and fine-tune the parameters, we found the average value function and calculate max norm with the optimal value function. Since the optimal value function changes with γ, we kept γ as constant through out the fine tuning process. And we used the optimal value function that we found by Value Iteration algorithm in our hw3 assignment to find the max norm. Below we show the fine tuning of parameters and the final results.

alpha - step size:
Parameter alpha played a crucial role in the model performance. We analysed the performance for the following set of alpha values: *0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.5, 0.7, 0.9* and found the best alpha to be *0.1*. The maxnorm vs alpha curve is shown below. For higher values of alpha the algorithm converges faster but not to the optimal value function (higher maxnorm). We found using decaying helped the algo to converge to a better solution. We decayed the alpha by $\alpha = \alpha - \alpha/1000$. We found the value *1000* by trying out different values in the range *500 - 2500*.
epsilon - exploration parameter:
Epsilon is exploration parameter. We used epsilon-greedy policy to select actions. Ideally we want to explore a lot at first and exploit as the algorithm runs more and more episodes. We analysed the performance for the following set of epsilon values: *0.01, 0.05, 0.1, 0.2, 0.3, 0.5, 0.9* and found the best epsilon to be *0.1*. The maxnorm vs epsilon curve is shown below. We try to keep epsilon in bounds for balanced exploration-exploitation. We found using decaying helped the algo to converge to a better solution. We decayed the epsilon by $\epsilon = \epsilon - \epsilon/500$. We found the value *500* by trying out different values in the range *500 - 2500*.

Initial weights
How we initialise the weights also matter when it comes to how fast and how good the convergence happens. There are multiple strategies to initialize weights - randomly, with zeros or optimistically. Here we found initializing optimistically worked better. We tried different values of initial weight values and found 20 to be a good choice.



(a) alpha (b) epsilon

Figure 3: Tuning curves for alpha and epsilon

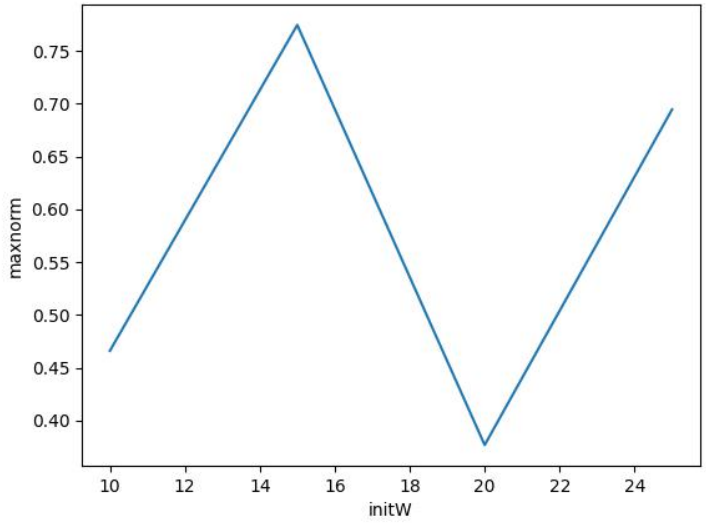


Figure 4: initial weight value parameter tuning

lambda - trace decay rate

We analysed the performance for the following set of lambda values: *0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 1* and found the best lambda to be *0.3*.

Final result:

We now show the performance plots and greedy policy learned by the algorithm by using the tuned parameters. Note that the plots and the values are shown for gamma = 0.9 and taking the optimal value function learned by value iteration algorithm for computing MSE. These graphs are plotted by running the algorithm 20 times and taking average over all the runs. On average the algo took 5531 episodes and got a maxnorm of 0.1392.

Plots interpretation: We can observe from the total actions vs episodes graph that initially the agents takes a lot of actions per episode indicating it is learning more. Once it learns enough, the episodes start to end quicker than before so the total actions per episode decrease. Also we can see from the MSE graph that the Mean Square Error between the optimal Value function and value function learned by the algorithm decreases and converges to almost 0. This indicates that the value function learned is very close to actual value function. Also final greedy pol-

icy is very much in accordance with the general expected agent behavior like trying to reach goal state by avoiding water state, leaving water state and going to goal state.

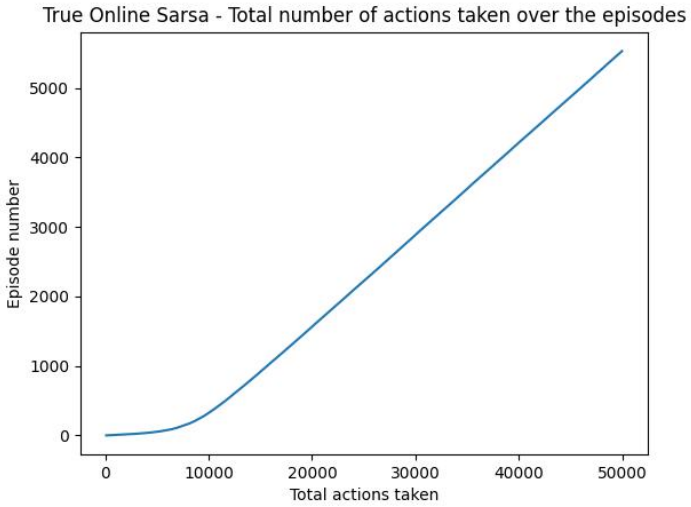


Figure 5: Total actions taken over the episodes

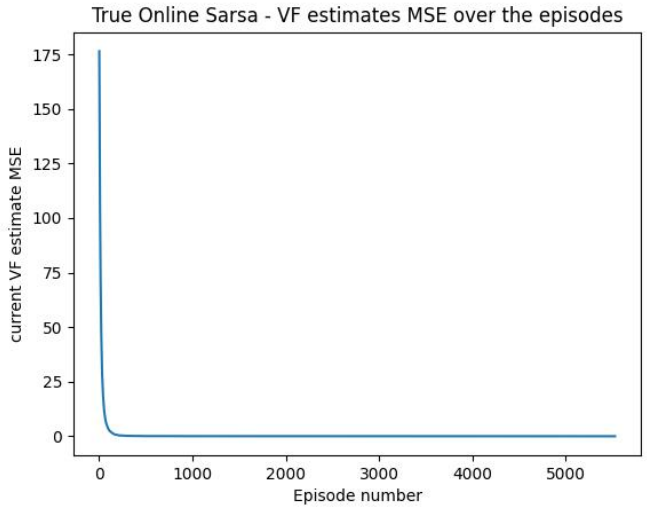


Figure 6: Value function estimate MSE over the episodes

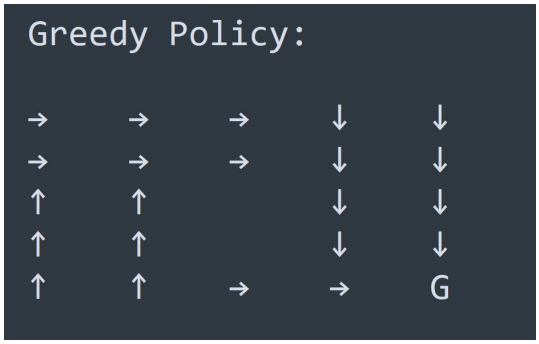
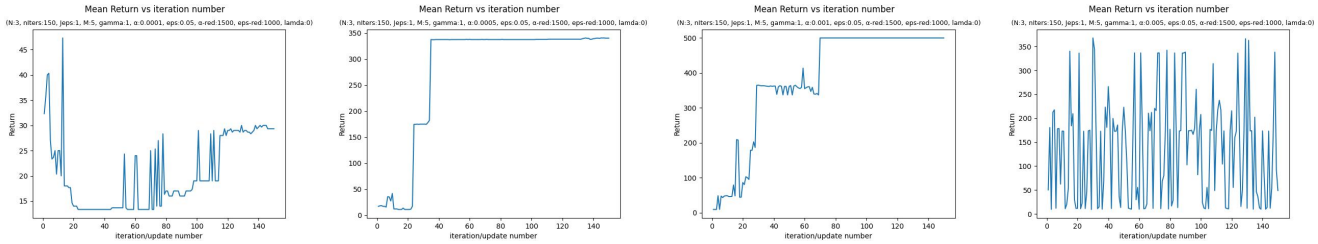


Figure 7: Greedy policy

4.1.2 Cartpole MDP

The model dynamics of the Cartpole MDP are taken same as HW2 assignment. We used Fourier features to represent states and get state-action feature vector. In order to measure the performance of the algorithm and fine-tune the parameters, we plotted learning curves. By comparing these learning curves, we selected the best hyperparameters. After every episode of the algorithm, we calculated average return using the latest weights and stored them for all the episodes. We plotted this reward as a function of episode graph averaged again over N runs of the algorithm. This plot helps us understand how the agent is learning and performing over the episodes. Below we show the fine tuning of parameters and the final results.

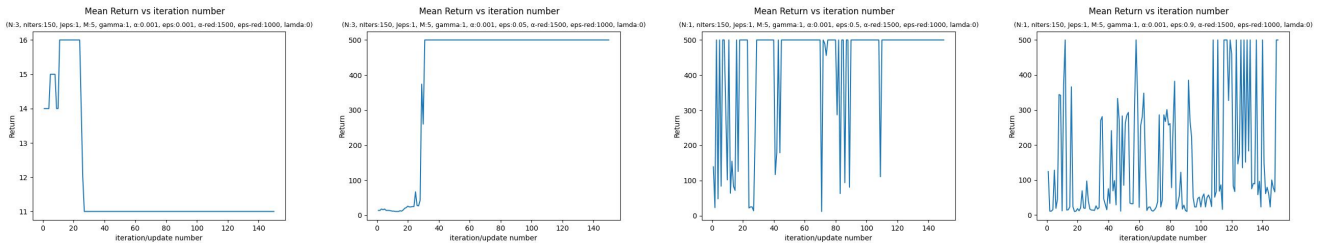
alpha - step size:
Parameter alpha played a crucial role in the model performance. We analysed the performance for the following set of alpha values: *0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.5* and found the best alpha to be *0.001*. For higher values of alpha the algorithm seems to become unstable. We found using decaying helped the algorithm to converge to a better solution. We decayed the alpha by $\alpha = \alpha - \alpha/1000$. We found the value *1000* by trying out different values in the range *100 - 2500*.



$\alpha = 0.0001$ $\alpha = 0.0005$ $\alpha = 0.001$ $\alpha = 0.005$

Figure 8: Cartpole alpha parameter tuning, best $\alpha = 0.001$

epsilon - exploration parameter:
Epsilon is exploration parameter. We used epsilon-greedy policy to select actions. Ideally we want to explore a lot at first and exploit as the algorithm runs more and more episodes. We analysed the performance for the following set of epsilon values: *0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.5, 0.9* and found the best epsilon to be *0.05*. We try to keep epsilon in bounds for balanced exploration-exploitation. We tried decaying the epsilon but it did not help much in this scenario.



$\epsilon = 0.001$ $\epsilon = 0.05$ $\epsilon = 0.5$ $\epsilon = 0.9$

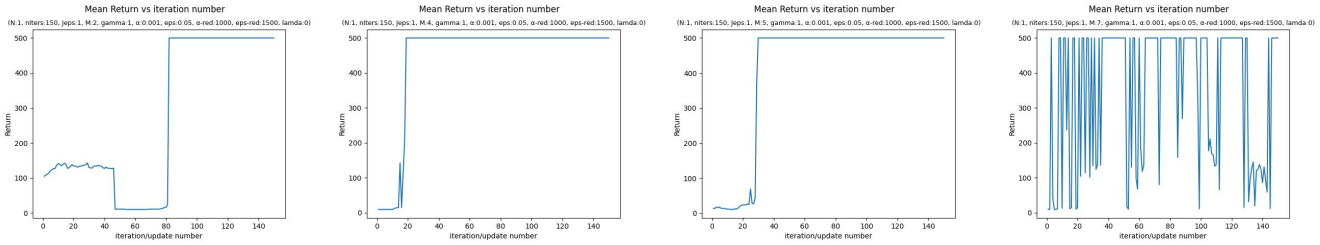
Figure 9: Cartpole epsilon parameter tuning, best $\epsilon = 0.05$

Initial weights

How we initialise the weights also matter when it comes to how fast and how good the convergence happens. There are multiple strategies to initialize weights - randomly, with zeros or optimistically. Here we found initializing randomly worked better. The weight values are taken from range [0,1] randomly.

M - Order of Fourier Basis

The state representation of cartpole has 4 dimension. So number of features in state representation will be $(M+1)^4$. We analysed the performance for the following set of M values: *1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20* and found the best M to be *4*.

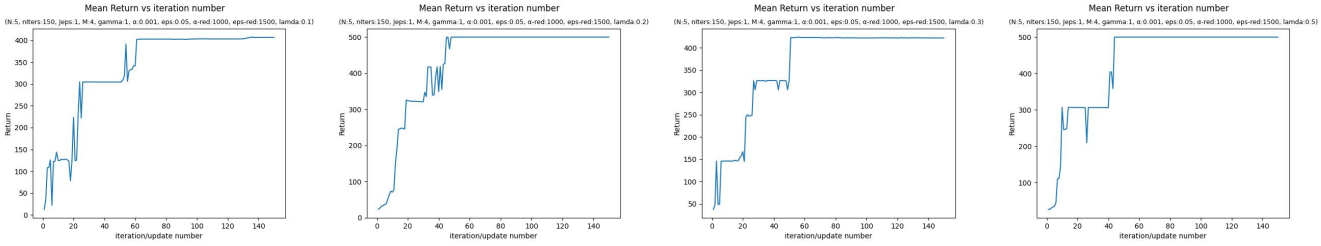


$M = 2$ $M = 4$ $M = 5$ $M = 7$

Figure 10: Cartpole M parameter tuning, best $M = 4$

lambda - trace decay rate

We analysed the performance for the following set of lambda values: *0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 1* and found the best lambda to be *0.2*.



$\lambda = 0.1$ $\lambda = 0.2$ $\lambda = 0.3$ $\lambda = 0.5$

Figure 11: Cartpole λ parameter tuning, best $\lambda = 0.2$

Final result:

We now show the performance plots obtained by using the tuned parameters. Note that the plots and the values are shown for gamma = 1. We ran the algorithm 10 times plotted the learning curves by taking average over all

the runs. We plotted the Mean return over the episodes, return and standard deviation over the episodes and total actions taken over the episodes plots.

Plots interpretation: We can observe from the plots that the average total returns eventually converges to 500. This indicates that the agent is learning and eventually learned to balance the pole until the time expires. The total actions vs episodes graph indicates the same. During the initial episodes, the agent takes very few total actions per episode, but once it starts to learn balancing the pole, it takes more and more actions per episode, indicating it is learning to balance well.

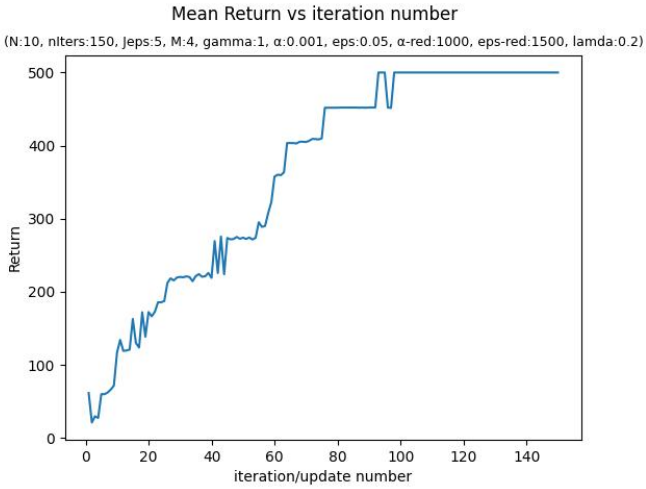


Figure 12: Returns as a function of episodes averaged over 10 runs

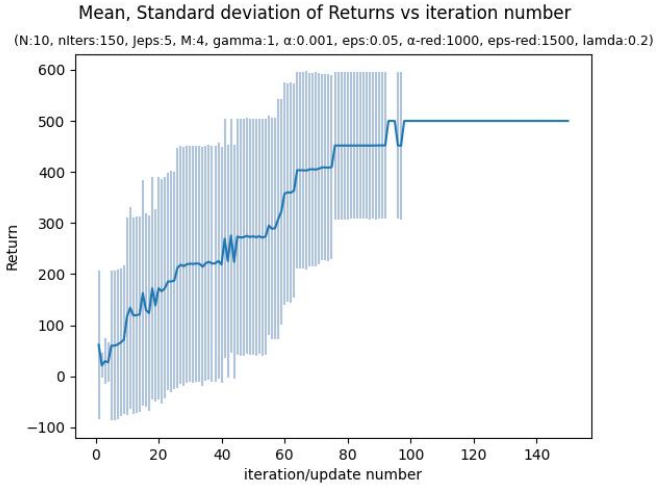


Figure 13: Returns and standard deviation as a function of episodes averaged over 10 runs

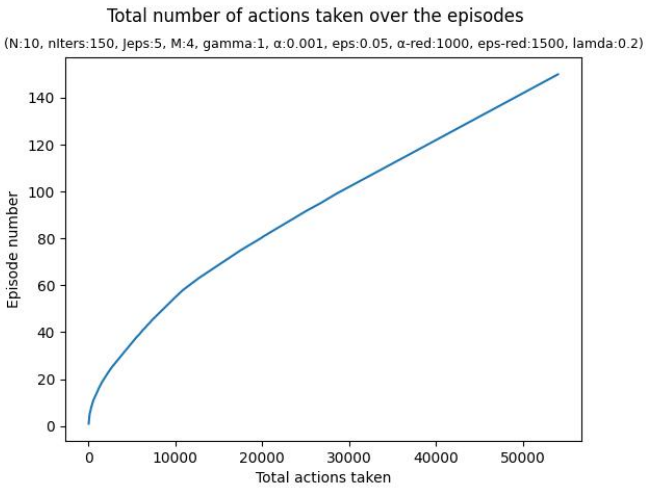
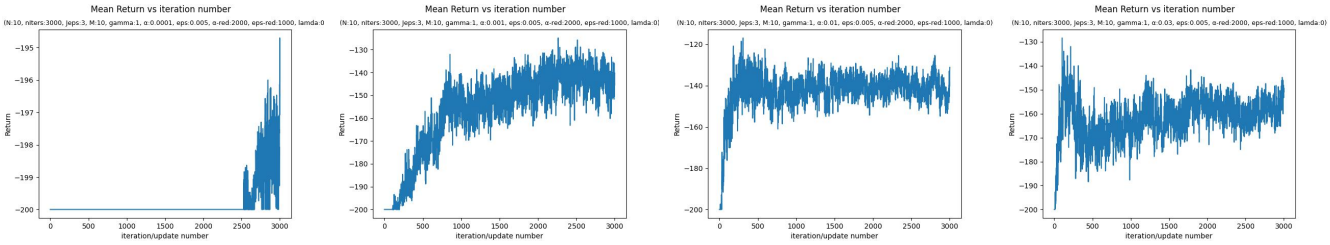


Figure 14: Total Actions taken over the episodes averaged over 10 runs

4.1.3 Mountain Car MDP (Extra Credit)

The model dynamics of the Mountain Car MDP are taken from the gym library documentation. The fine tuning process is same as Cartpole MDP. Since both are continuous state space domains with discrete actions, we used the similar Fourier features as Cartpole to represent states and get state-action feature vector. In order to measure the performance of the algorithm and fine-tune the parameters, we plotted learning curves. By comparing these learning curves, we selected the best hyperparameters. After every episode of the algorithm, we calculated average return using the latest weights and stored them for all the episodes. We plotted this reward as a function of episode graph averaged again over N runs of the algorithm. This plot helps us understand how the agent is learning and performing over the episodes. Below we show the fine tuning of parameters and the final results.

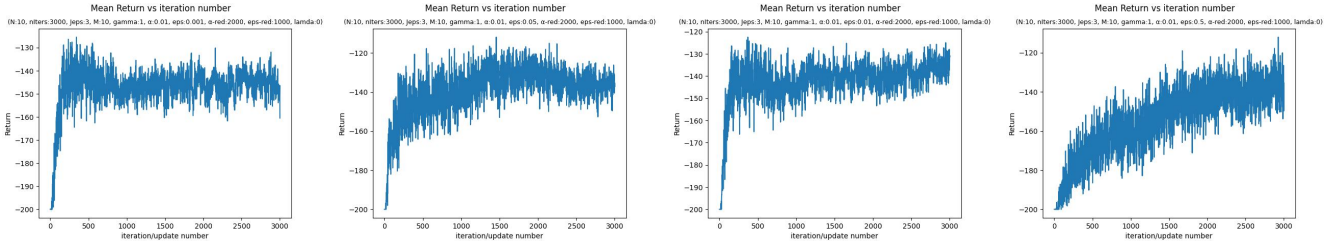
alpha - step size:
We analysed the performance for the following set of alpha values: *0.0001, 0.0005, 0.001, 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.2, 0.5* and found the best alpha to be *0.01*. For higher values of alpha, the weights starts to explode and the algorithm seems to become unstable. We found using decaying helped the algorithm to converge to a better solution. We decayed the alpha by $\alpha = \alpha - \alpha/1000$. We found the value *1000* by trying out different values in the range *100 - 2500*.



$\alpha = 0.0001$ $\alpha = 0.001$ $\alpha = 0.01$ $\alpha = 0.03$

Figure 15: Mountain car α parameter tuning, best $\alpha = 0.01$

epsilon - exploration parameter:
Epsilon is exploration parameter. We used epsilon-greedy policy to select actions. Ideally we want to explore a lot at first and exploit as the algorithm runs more and more episodes. We analysed the performance for the following set of epsilon values: *0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.9* and found the best epsilon to be *0.05*. We try to keep epsilon in bounds for balanced exploration-exploitation. We tried decaying the epsilon but it did not help much in this scenario.



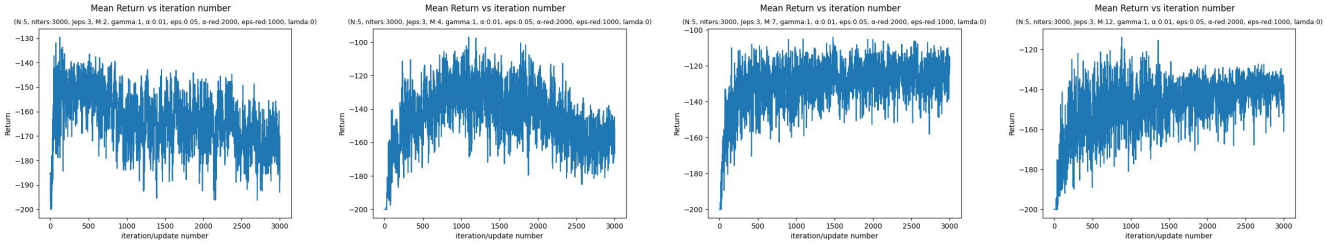
$\epsilon = 0.001$ $\epsilon = 0.05$ $\epsilon = 0.01$ $\epsilon = 0.5$

Figure 16: Mountain car ϵ parameter tuning, best $\epsilon = 0.05$

Initial weights
There are multiple strategies to initialize weights - randomly, with zeros or optimistically. Here we found initializing randomly worked better. The weight values are taken from range [0,1] randomly.

M - Order of Fourier Basis

The state representation of Mountain car has 2 dimension. So number of features in state representation will be $(M+1)^2$. We analysed the performance for the following set of M values: *1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20* and found the best M to be *7*.

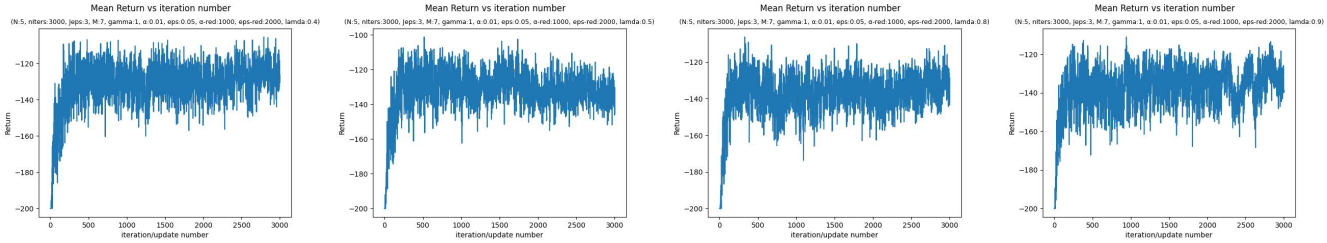


$M = 2$ $M = 4$ $M = 7$ $M = 12$

Figure 17: Mountain car M parameter tuning, best $M = 7$

lambda - trace decay rate

We analysed the performance for the following set of lambda values: *0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 1* and found that the value of lambda did not have a significant affect on the learning curve. But out of all, *0.8* seems to work better.



$\lambda = 0.4$ $\lambda = 0.5$ $\lambda = 0.8$ $\lambda = 0.9$

Figure 18: Mountain car λ parameter tuning, best $\lambda = 0.8$

Final result:

We now show the performance plots obtained by using the tuned parameters. Note that the plots and the values are shown for gamma = 1. We ran the algorithm 10 times plotted the learning curves by taking average over all the runs. We plotted the Mean return over the episodes, return and standard deviation over the episodes and total actions taken over the episodes plots.

Plots interpretation: We can observe from the plots that the average total returns eventually converges to a value less than -200 somewhere between -130 to -140. This indicates that the agent is learning and eventually learned to reach the goal before the time expired (expiration time = 200).

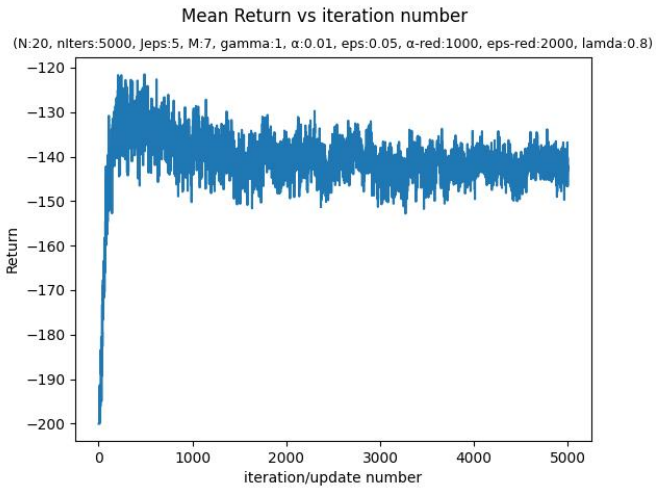


Figure 19: Returns as a function of episodes averaged over 20 runs

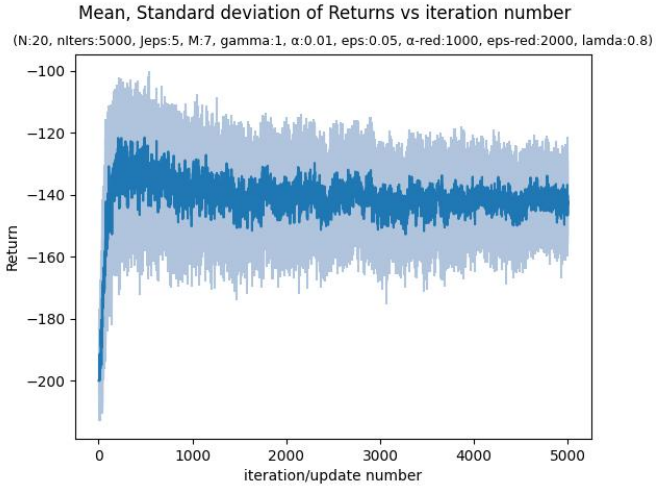


Figure 20: Returns and standard deviation as a function of episodes averaged over 20 runs

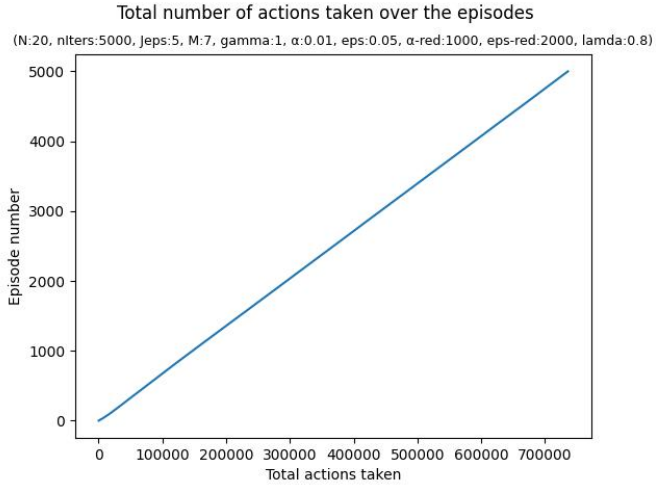


Figure 21: Total Actions taken over the episodes averaged over 20 runs

4.2 One-Step Actor-Critic