COMPSCI 687: Reinforcement Learning Fall 2023

CS687: Final Project

Group: Abhishek Sureddy, Akshay Kumar Sureddy

1 Algorithm - 1: REINFORCE with BaseLine

In this section we are going to discuss the intricacies in the implementation of REINFORCE with BaseLine algorithm.

1.1 Introduction: Why REINFORCE with BaseLine?

As a part of the course project, we wanted to explore How Policy Gradient Methods work, so we chose this algorithm to implement. For estimating the value function and policy function, we used a neural network as neural networks are a popular tool to learn and approximate any generic objective function. The algorithm implemented here was referred from the RL book Sutton and Barto and partial ideas from online resources link1 and link2.

1.2 Pseudocode/ Algorithm

The pseudo-code of our implementation of REINFORCE with BaseLine is shown in Algorithm 1.

- **PolicyNetwork**
  - This is a neural network with one hidden layer of 128 nodes and a ReLu activation function, we apply a softmax on the output layer.
  - It takes state space as input and outputs the probabilities of various at that state.
- **ValueNetwork**
  - This is a neural network with one hidden layer of 128 nodes and ReLu activation function.
  - It takes state space as input and outputs the value function at that state.
- We tried varying the neurons in the hidden layer. If we were using fewer neurons, then the model was underfitting, and for a higher number of neurons and more layers, the model was overfitting, so we used a single hidden layer with 128 neurons.
- To initialize the weights of the neural network, we used $w \sim \mathcal{N}(0,1)$ and $w \sim \mathcal{U}nif(0,1)$. The Normal initialization of weights of the neural network worked well.
- To implement the neural network, we were using PyTorch, so we could just use the 'backward' method to calculate the gradients of the loss function and update it.

Algorithm 1 Reinforce with Baseline

Input:

1) *env*: Initialize the MDP, used cart pole or Acrobot here.
2) *n_states*: Number of states in the initialized MDP/ environment.
3) *n_actions*: Number of actions possible in the MDP.
4) γ: Discount factor
5) *num_episodes*: Number of episodes the algorithm should run for.
6) α: Learning rate for updating the parameters of Policy Function estimator Neural Network
7) β: Learning rate for updating the parameters of Value function estimator Neural Network.

policy $\leftarrow$ PolicyNetwork(*action_input_size*, *action_output_size*) // Initialize policy estimator
value $\leftarrow$ ValueNetwork(*value_input_size*, *value_output_size*) // Initialize value function estimator
for *episode in range(num_episodes)* **do**
  state $\leftarrow$ *env.reset*() // Re-initialize the MDP to initial state
  states $\leftarrow$ [ ] // To track the sequence of states the agent traversed.
  actions $\leftarrow$ [ ] // To track the sequence of actions the agent took.
  rewards $\leftarrow$ [ ] // To track the sequence of rewards the agent received.
  score $\leftarrow 0$ // Total reward for the episode.
  while *True* **do**
    $\pi(state, \theta) \leftarrow policy_\theta(state)$ // probabilities of taking each action at current state
    action $\leftarrow$ *random.choice*($\pi(state, \theta)$) // Sample an action according to policy, $\pi(state, \theta)$
    // Take sampled action and reach next state and receive reward
    next_state, *reward*, *is_terminal*, _ $\leftarrow$ *env.step*(*action*)
      // Append current state, action taken and reward received
    states.append(*state*)
    actions.append(*action*)
    rewards.append(*reward*)
    score $\leftarrow$ *score* + *reward*
    state $\leftarrow$ *next_state*
    if *is_terminal* **then**
      break
  for *t= 0,1,..., T-1 of an episode* **do**
    // Update the parameters: **Note:** These updates are performed in the code by calling the .backward function of PyTorch network
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{(k-t-1)} * reward[k]$
    $\delta \leftarrow G - value_w(states[t])$ // here w is an internal parameter of neural network
    // w - weights of the neural network for Value function approximation
    $w \leftarrow w + \alpha * \delta \nabla value_w(states[t])$
      // θ - weights of the neural network for policy approximation
    $\theta \leftarrow \theta + \beta \gamma^t \delta \nabla ln\, \pi(actions[t] \mid states[t], \theta)$
return *policy*

1.3 Evaluating MDPs

1. **Cart Pole Environment**:

   - **Description:** Here, a pole is attached to the cart by a frictionless pivot, the pole is placed upright on the cart and the goal is to balance it by applying forces either left or right.
   - **Action Space:** push cart to the left or right.

- **Observation Space/ State space:** $(x, v, \theta, \dot{\theta})$ and the limits are:
  position of cart: $x \in (-4.8, 4.8)$, cart velocity: $v \in (-\infty, \infty)$, pole angle: $\theta \in (-24°, 24°)$, pole angular velocity: $\dot{\theta} \in (-\infty, \infty)$.
- **Starting State:** $(x, v, \theta, \dot{\theta}) \sim \mathcal{U}nif(-0.05, 0.05)$
- **Rewards:** A reward of +1 is given for every timestep including the terminal state.
- **Episode End:** Episode terminates if:
  - Pole angle is not in between $-12°$ and $12°$.
  - Cart Position is not in between -2.4 and 2.4.
  - Episode length is greater than 500.

  The writeup for this environment was mostly referred from gym documentation of the Cartpole environment (link).

2. **Acrobot**

- **Description:** The Acrobot system consists of two links connected linearly with an actuated joint. The goal is to swing the free end above a target height by applying torque.
- **Action Space:**
  - 0: Apply -1 torque to the actuated joint (torque in N m)
  - 1: Apply 0 torque to the actuated joint
  - 2: Apply 1 torque to the actuated joint
- **Observation Space/ State space:** An array (shape: (6,)) representing joint angles (cosine and sine) and angular velocities.
- **Rewards:** - Goal reward: 0, Termination state reward: -1, Reward threshold: -100
- **Starting State:** Initial joint angles and velocities uniformly initialized between -0.1 and 0.1.
- **Episode End:** Termination condition: -cos(theta1) - cos(theta2 + theta1) > 1.0
  Truncation condition: Episode length > 500

The writeup for this environment was mostly referred from gym documentation of Acrobot environment (link)

1.4 Hyper-parameter tuning And Observations

1. **Cart Pole Environment:** The hyperparameters taken into consideration are α, β and γ. We ran the Hyper-parameter tuning by performing a grid search on:

- $\alpha : \{0.001, 0.005, 0.01\}$
- $\beta : \{0.001, 0.005, 0.01\}$
- $\gamma : \{0.8, 0.9, 0.99\}$

(a) **Varying α setting:** Here α is the learning rate parameter to update the weights of Policy Neural Network. When experimented with varying α settings, we observed that when the learning rate was low, the algorithm converged and stabilized at a high reward value. The same can be depicted by the green curve shown in figure 1. Even the learning curves for slightly higher values of α seem to converge because we were decaying, we can also see that slightly higher learning rates took slightly more episodes to converge/ stabilize, this might be because of the nature of the non-linear function (here log loss) on the surface given by policy neural network, i.e. higher learning rates might have taken big steps to update the gradient, hence more oscillations in the initial steps. As the algorithm is prone to bias and variance, to establish a trade-off, after trying different Neural architectures, we have chosen a single hidden layer network with 128 nodes. Also the batch update using memory helps stabilize the model. These plots were plotted by running the algorithm 10 times and averaging the results, and it is evident that the algorithm still suffers from some variance.

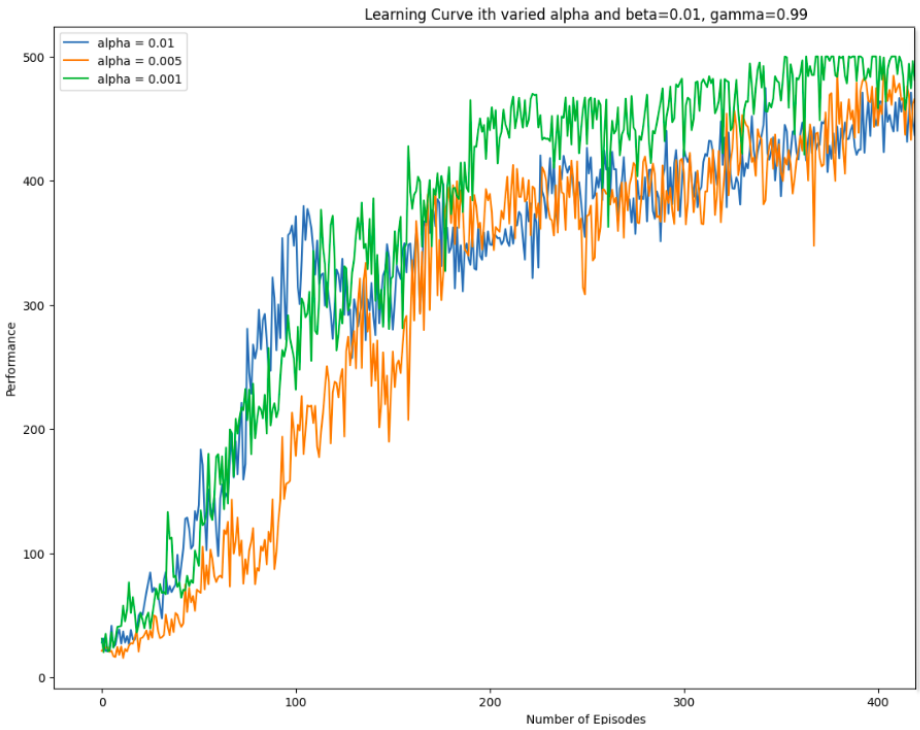

Figure 1: Reinforce with Baseline for Varied α setting for CartPole MDP

(b) **Varying β setting:** Here β is the learning rate parameter to update the weights of Value Function Neural Network. In this setting, when the learning rate was set to a very low value the performance of the algorithm increased very slowly. But when the learning rate was set to a moderate value, it's performance increased and stabilized. The same is depicted in the figure 2.

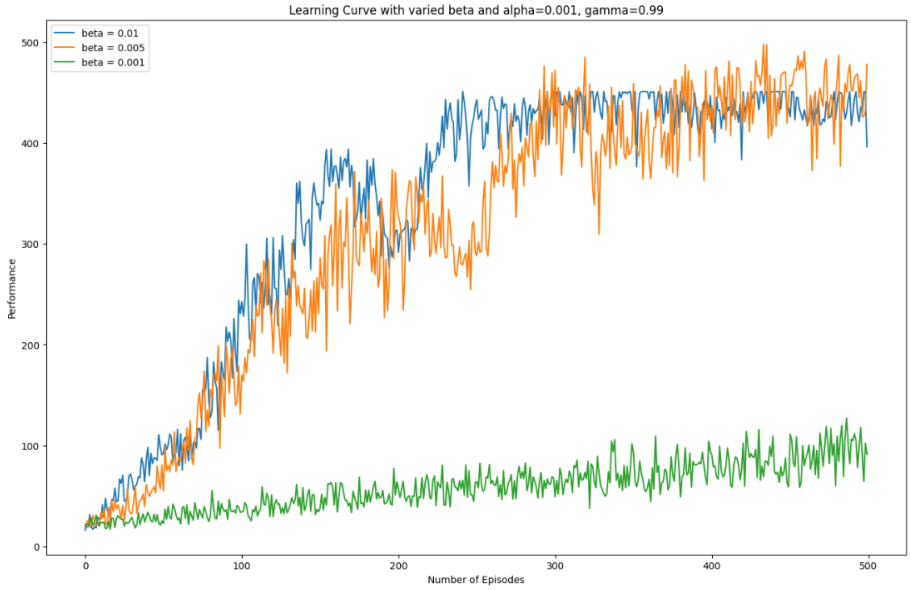

Figure 2: Reinforce with Baseline for Varied β setting

(c) **Varying γ setting:** Here, γ is the discounting parameter for calculating the discounted return. In Cart Pole setting, setting γ to a low value means, we are not giving high enough rewards for the agent to get the pole balanced for a longer duration of time, So the agent might learn to take actions to balance the pole well for short duration of time, not caring much about balancing it for longer duration, resulting in

less amount of time for which the pole is vertical. Hence, higher value of γ is encouraged in this setting, the same can be depicted in the figure 3. In the figure, we can see that, the maximum performance (time for which the pole is balanced) achieved by the blue curve (very high γ), is significantly higher compared to that of lower γ settings.

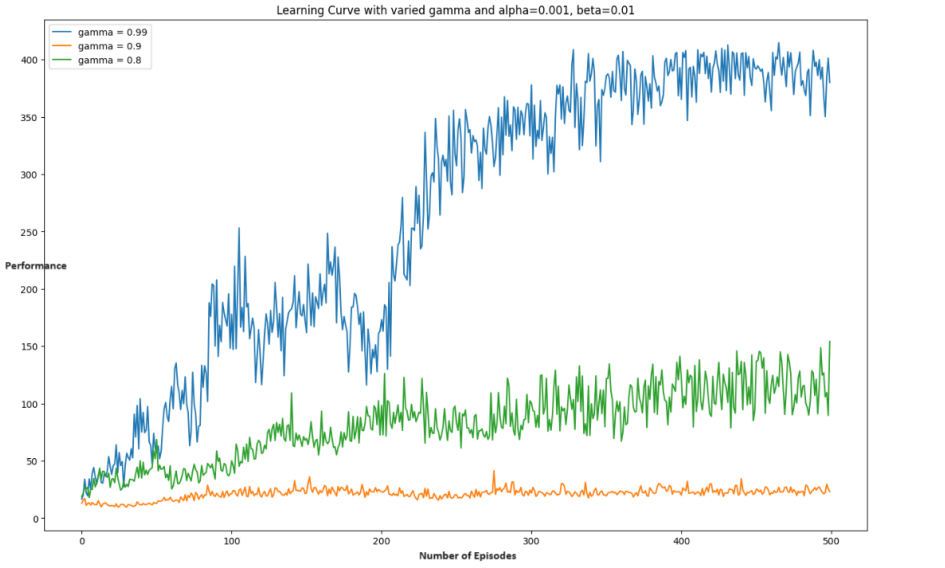

Figure 3: Reinforce with Baseline for Varied γ setting

(d) **Best Hyper Parameter setting:** The best Hyper parameter setting was when $\alpha = 0.001, \beta = 0.01, \gamma = 0.99$, we used a single hidden layer for policy network and value network. The weights of the network were initialized using Normal Random initialization. The Performance learning curve for the same is shown in figure 4,

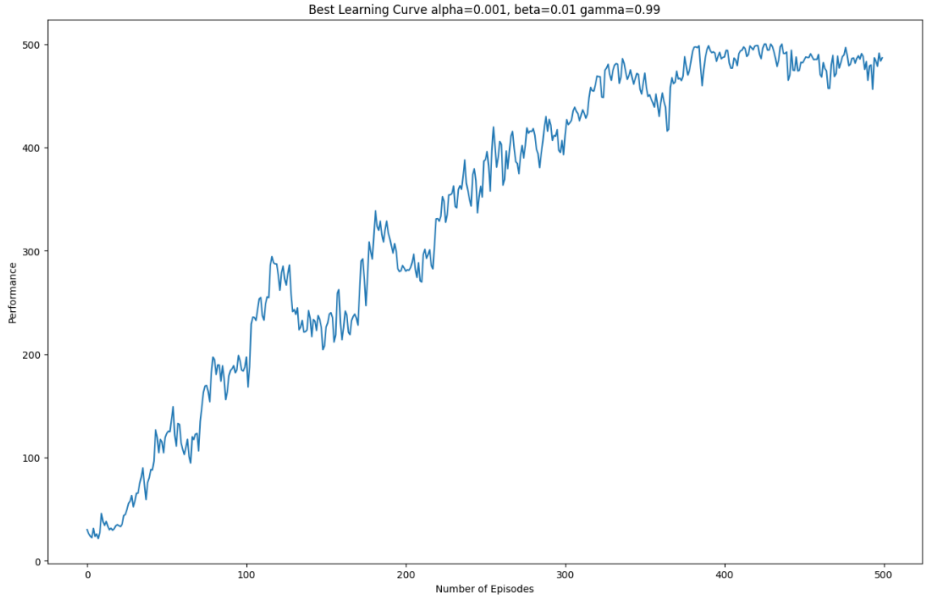

Figure 4: Reinforce with Baseline for Best parameter setting

2. **Acrobot Environment:** The hyperparameters taken into consideration are α, β and γ. We ran the Hyperparameter tuning by performing a grid search on:

- $\alpha : \{0.001, 0.005, 0.01\}$

- $\beta : \{0.001, 0.005, 0.01\}$
- $\gamma : \{0.8, 0.9, 0.99\}$

(a) **Varying α setting:** Here α is the learning rate parameter to update the weights of Policy Neural Network. When experimented with varying α settings, we observed that when the learning rate was moderate, i.e. $\alpha = 0.005$, the algorithm converged and stabilized at a high reward value. The same can be depicted by the orange curve shown in figure 5. The learning curves for higher value of α seem to converge to a local minima because we were decaying α, this might be because of the nature of the non-linear function (here log loss) on the surface given by policy neural network. Also, for a very low learning rate, the algorithm's performance doesn't seem to update much after a few episodes because the learning rate used might have been very low to update the network. As the algorithm is prone to bias and variance, to establish a trade-off, after trying different Neural architectures, we have chosen a single hidden layer network with 128 nodes. Also, the batch update using memory helps stabilize the model. These plots were plotted by running the algorithm 10 times and averaging the results, and it is evident that the algorithm still suffers from some variance.

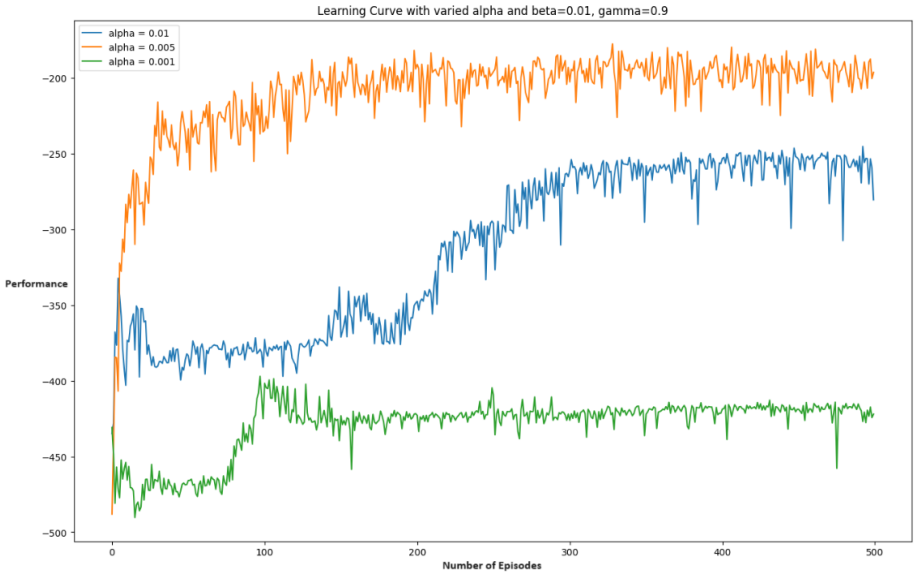

Figure 5: Reinforce with Baseline for Varied α setting for Acrobot

(b) **Varying β setting:** Here β is the learning rate parameter to update the weights of Value Function Neural Network. Even In this setting, when the learning rate was set to a moderate value of 0.005 the performance increased and stabilized. The same is depicted in the figure 6.

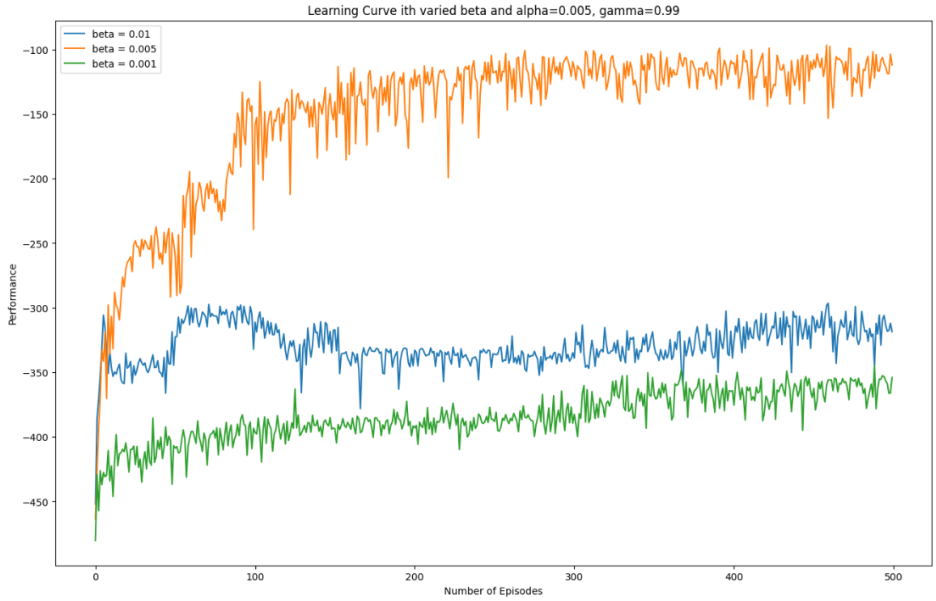

Figure 6: Reinforce with Baseline for Varied β setting for Acrobot

(c) **Varying γ setting:** Here, γ is the discounting parameter for calculating the discounted return. In this setting, setting γ to a low value means, we are not penalizing the agent for taking a longer duration to get to a specified height. Hence, a higher value of γ is encouraged in this setting, the same can be depicted in the figure 7. In the figure, we can see that, the maximum performance (time for which the pole is balanced) achieved by the blue curve (very high γ), is significantly higher compared to that of lower γ settings.

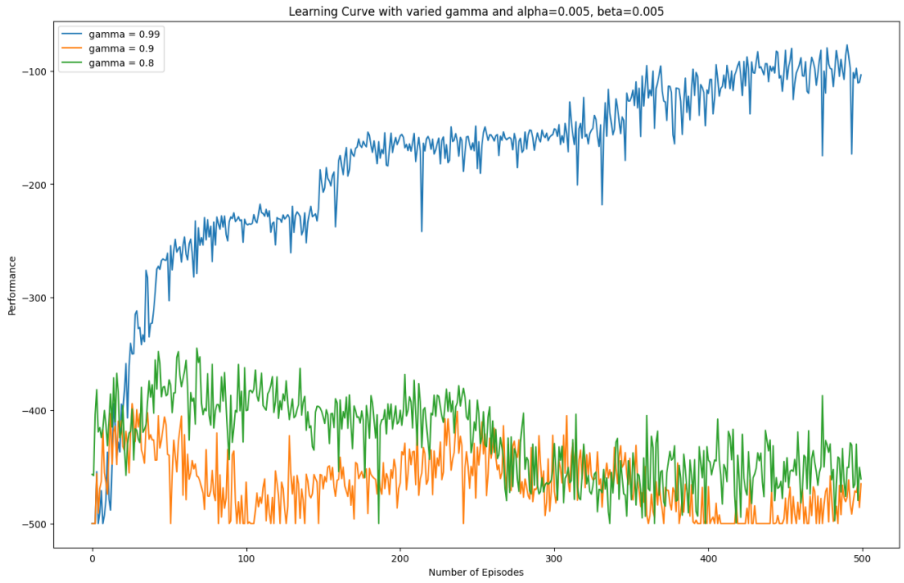

Figure 7: Reinforce with Baseline for Varied γ setting for Acrobot

(d) **Best Hyper Parameter setting:** The best Hyper parameter setting was when $\alpha = 0.005, \beta = 0.005, \gamma = 0.99$, we used a single hidden layer for policy network and value network. The weights of the network were initialized using Normal Random initialization. The Performance learning curve for the same is shown in figure **??**,

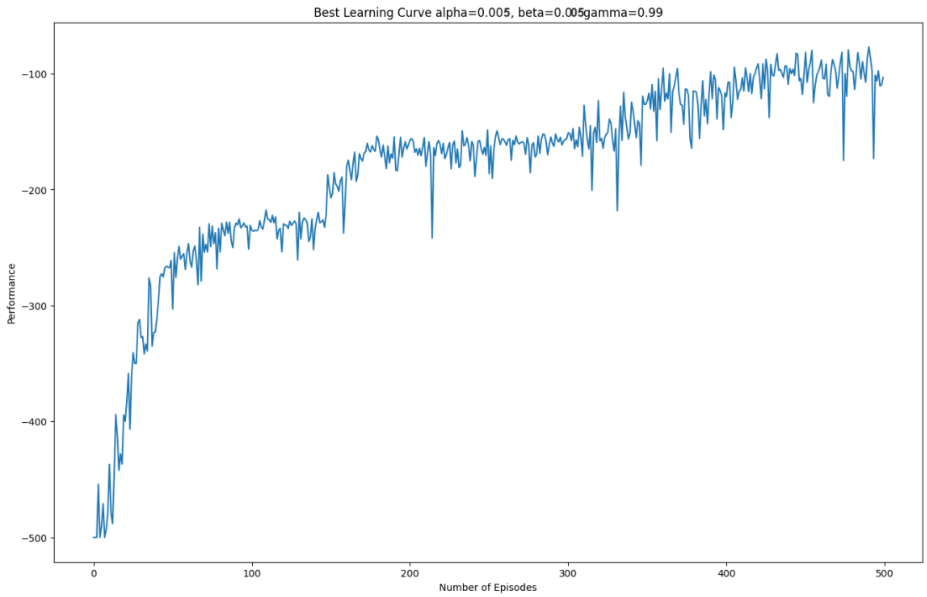

Figure 8: Reinforce with Baseline for Best parameter setting

1.5 Challenges Faced

- The initial code was written from scratch by referring the REINFORCE with baseline algorithm from Sutton and Barto.
- This version of the code was converging to the maximum sometimes, but is had very high variance and the results were not consistent.
- It took a significant amount of time to try out various various Neural network weight initializations, architectures hyperparameter tuning and setting decay values for α and β. Finally, we were able to find the range of parameters for which the algorithm converges/ learns.
- We have tried deeper networks to compare the results, but the algorithm was taking too long for each episode to run. So went ahead with a single-layered neural network following the occum-Razor principle of preferring the simple approximation functions. Also, to reduce the variance of the model, we used the averaging technique, i.e. running the algorithm multiple times and taking its average.

2 Algorithm - 2: D-QNN

2.1 Pseudocode/ Algorithm

Algorithm 2 Q-Learning with Neural Network Initialization

Initialize Q function using the neural network algorithm

Initialize Q_{off} using the neural network as the off-policy Q function for exploration

Initialize empty list Memory for future sampling and batch update
for *each episode in Total_number_of_episodes* **do**
    Initialize S
    for *each step of episode* **do**
        Choose A from S using epsilon-greedy policy derived from Q
        Execute action A, and observe reward R, next state S', and store S, A, R, S' in Memory
        Sample a batch of batch_size S, A, R, S' from Memory
        for *each* S, A, R, S' *in batch* **do**
            $Q\_value \leftarrow Q(S)$
            $Q_{\text{off\_value}} \leftarrow Q_{\text{off}}(S')$
            if S' *is not terminal* **then**
                target_value $\leftarrow R + \gamma \cdot Q_{\text{off\_value}}$
            else
                target_value $\leftarrow R$
            loss += MSE(Q_value, target_value)
        Back Propagate and revise the weights of Q
        $S \leftarrow S'$
        if *episode % 10 == 0* **then**
            Update weights of Q_{off}
    while S *is not terminal* **do**
        Continue until the terminal state is reached

2.2 Brief Description of methods

Why Deep Q-Learning?

- As a part of the project, we wanted to try out different class of algorithms and compare their results. Thus, Q-Learning with Neural Networks as Q value function, which is categorized as Off-policy TD Control is chosen.
- Also, as Reinforce-Baseline uses policy and value networks, we wanted to try out the State Action Value Network too.

Deep Q-Learning

- The algorithm implemented here was referred from the book Sutton and Barto and partial ideas from Pytorch Reinforcement Learning.
- The algorithm uses a Neural Network which estimates Q values by taking state space as input.
- The algorithm also uses something called off-policy, a second network, used for computation of next state estimate, it helps in exploration.
- We also used a Memory list, used for sampling batch and updating during each step in the episode.

Q Network using Pytorch Neural Networks

- The Network was implemented using Pytorch's nn.linear and relu functions.

- It takes state space as input and outputs the q value corresponding to all the actions.

Q Learning Algorithm using Neural Network, Off policy and Memory

- The algorithm starts by initializing two Q-functions: Q and Q_{off}. These functions are represented using neural networks. Q is the main Q-function that will be updated during the learning process. Q_{off} is an off-policy Q-function used for exploration.
- An empty list Memory is initialized to store experiences (S, A, R, S') for later sampling and batch updates. This is part of experience replay, a technique that helps stabilize and improve the learning process.
- For each step in the episode, select an action using epsilon-greedy and execute the action to get the next state and reward. Store the state, action, reward, and next state in the memory list.
- Compute Q_{value} for S and $Q_{offvalue}$ for S' the batch and compute the target value.
- Using the target value compute the loss, then back propagate.
- Assign S = S'.
- Terminate if S is terminal.

2.3 Evaluating MDPs

1. **Cart Pole Environment**:

   - **Description:** Here, a pole is attached to the cart by a frictionless pivot, the pole is placed upright on the cart and the goal is to balance it by applying forces either left or right.
   - **Action Space:** push cart to the left or right.
   - **Observation Space/ State space:** $(x, v, \theta, \dot{\theta})$ and the limits are:
     position of cart: $x \in (-4.8, 4.8)$, cart velocity: $v \in (-\infty, \infty)$, pole angle: $\theta \in (-24°, 24°)$, pole angular velocity: $\dot{\theta} \in (-\infty, \infty)$.
   - **Starting State:** $(x, v, \theta, \dot{\theta}) \sim \mathcal{U}nif(-0.05, 0.05)$
   - **Rewards:** A reward of +1 is given for every timestep including the terminal state.
   - **Episode End:** Episode terminates if:
     - Pole angle is not in between $-12°$ and $12°$.
     - Cart Position is not in between -2.4 and 2.4.
     - Episode length is greater than 500.

   The writeup for this environment was mostly referred from gym documentation of the Cartpole environment (link).

2. **Acrobot**

   - **Description:** The Acrobot system consists of two links connected linearly with an actuated joint. The goal is to swing the free end above a target height by applying torque.
   - **Action Space:**
     - 0: Apply -1 torque to the actuated joint (torque in N m)
     - 1: Apply 0 torque to the actuated joint
     - 2: Apply 1 torque to the actuated joint
   - **Observation Space/ State space:** An array (shape: (6,)) representing joint angles (cosine and sine) and angular velocities.
   - **Rewards:** - Goal reward: 0, Termination state reward: -1, Reward threshold: -100
   - **Starting State:** Initial joint angles and velocities uniformly initialized between -0.1 and 0.1.
   - **Episode End:** Termination condition: -cos(theta1) - cos(theta2 + theta1) > 1.0
     Truncation condition: Episode length > 500

   The writeup for this environment was mostly referred from gym documentation of Acrobot environment (link)

2.4 Hyper-parameter tuning And Observations

1. **Cart Pole Environment:** The hyperparameters taken into consideration are α, ϵ and γ. We ran the Hyperparameter tuning by performing a grid search on:

   - $\alpha : \{0.001, 0.005, 0.01\}$
   - $\epsilon : \{0.1, 0.2, 0.3\}$
   - $\gamma : \{0.8, 0.9, 0.99\}$

   (a) **Varying α setting:**

   - Here α is the learning rate parameter to update the weights of Q (State Action Value) Network.
   - When experimented with varying α settings, with $\alpha = 0.01, 0.005, 0.001$ we observed that when the learning rate was low, the algorithm converged slowly, but is relatively stable or consistent.
   - The convergence was quick with a higher Learning rate, but there is a tradeoff with stability. The same can be depicted by the blue curve shown in figure 9.
   - As the blue and orange curves reach the end, their oscillations are reduced. This is because of the damping of the learning rate for every 25 epochs by a factor of 10%.
   - As the Q-learning algorithm is prone to bias and variance, to establish a trade-off, after trying different Neural architectures, we have chosen a single hidden layer network with 128 nodes. And also the batch update using memory helps stabilize the model.
   - These plots were plotted by running the algorithm 10 times and averaging the results, and it is evident that the algorithm still suffers from some variance.
   - For further analysis, $\alpha = 0.01$ was considered as it gave higher rewards and early convergence.

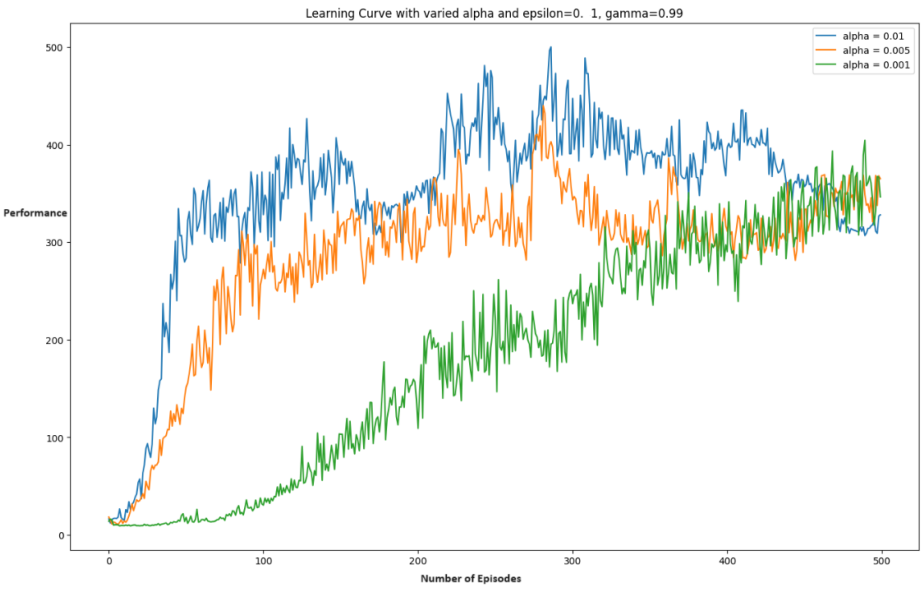

Figure 9: Deep Q-Learning for Varied α setting, averages for 10 iterations

   (b) **Varying ϵ setting:**

   - Here ϵ is the exploration parameter used in selecting the actions, in order to facilitate exploration of all available states and choose the best gradually.
   - In our algorithm, we are decaying the ϵ for every episode by multiplying it with 0.995. This is because the agent needs to reduce its exploration as it gradually learns the best policy.
   - When experimented with varying initial ϵ settings, with $\epsilon = 0.1, 0.2, 0.3$ we observed that the algorithm performed well with higher ϵ, evident from the orange curve.
   - This is because the higher ϵ is facilitating the agent by providing good exploration and thus giving a good start.

- For further analysis, $\epsilon = 0.4$ was considered as higher values of ϵ gave good results.

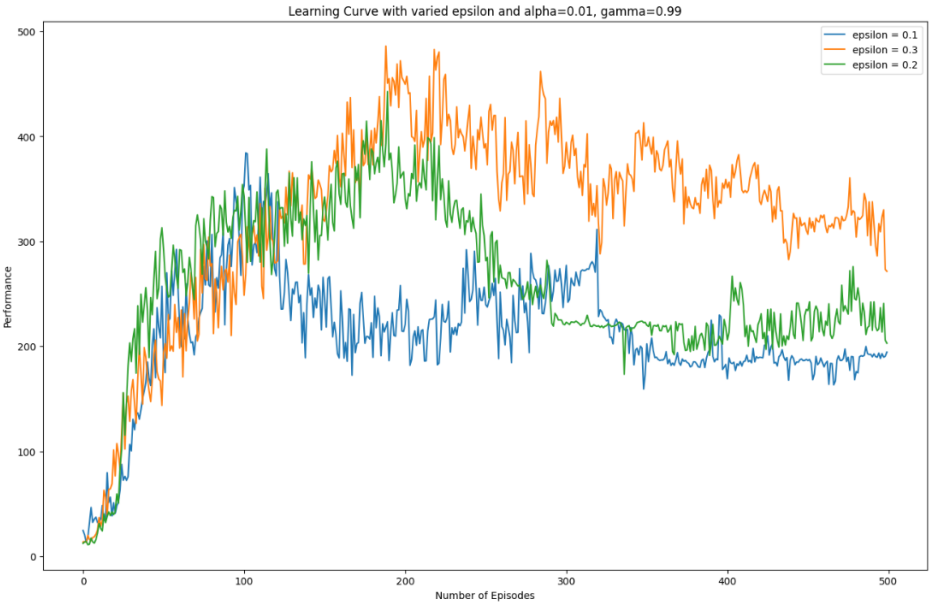

Figure 10: Deep Q-Learning for Varied ϵ setting, averages for 10 iterations

(c) **Varying γ setting:**

- Here, γ is the discounting parameter for calculating the discounted return.
- In Cart Pole setting, setting γ to a low value means, we are not giving high enough rewards for the agent to get the pole balanced for a longer duration of time, So the agent might learn to take actions to balance the pole well for short duration of time, not caring much about balancing it for longer duration, resulting in less amount of time for which the pole is vertical.
- Hence, higher value of γ is encouraged in this setting, the same can be depicted in the figure 11.
- In the figure, we can see that, the maximum performance (time for which the pole is balanced) achieved by the blue curve (very high γ), is significantly higher compared to that of lower γ settings.
- For further analysis, $\gamma = 0.99$ was considered as higher values of γ gave good results.

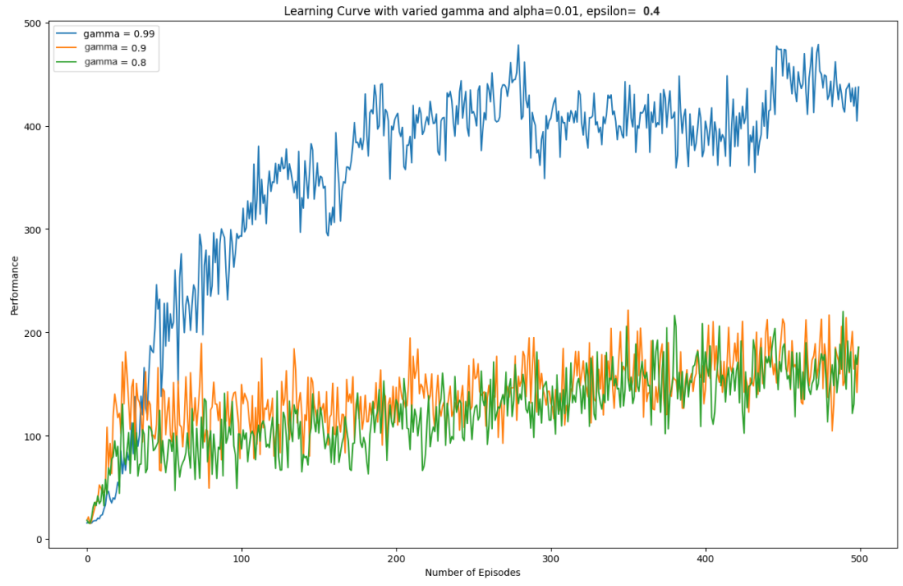

Figure 11: Deep Q-Learning for Varied γ setting, averaged for 10 iterations

(d) **Best Hyper Parameter setting:**

- The best hyperparameters turned out to be $\alpha = 0.01$, $\epsilon = 0.4$, $\gamma = 0.99$.
- Apart from the above-mentioned hyperparameter tuning, various approaches were tried out for finalizing the Neural Network Architecture, weight initialization, Learning rate decay, ϵ decay etc..
- Network architectures like 1,2,3 hidden layers with nodes count 64,128,256 were tried out.
- Weight initialization methods like Lecun, random uniform initialization, and Kaiming normal were tried.
- Network with 1 hidden layer of 128 nodes and Lecun (default) weight initialization provided decent start for the algorithm, thus were used throughout the whole experimentation process.
- A learning rate decay with step size of 25 was used.
- ϵ decay of 0.5% per epoch was fixed.

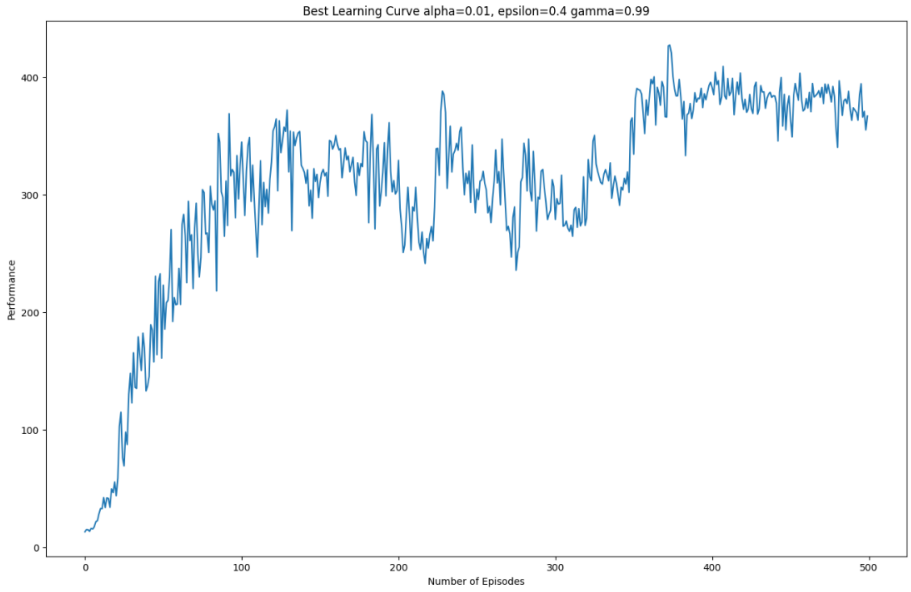

Figure 12: Deep Q-Learning with best parameter setting averaged 5 iterations

The best setting results were lost after γ analysis and these were the results generated when run later using same set of parameters for 5 iterations, thus show a little instability compared to the best result in γ comparision.

2. **Acrobot Environment:** The hyperparameters taken into consideration are α, ϵ and γ. We ran the Hyperparameter tuning by performing a grid search on:

- $\alpha : \{0.001, 0.005, 0.01\}$
- $\epsilon : \{0.1, 0.2, 0.3\}$
- $\gamma : \{0.8, 0.9, 0.99\}$

(a) **Varying α setting:**

- Here α is the learning rate parameter to update the weights of Q (State Action Value) Network.
- When experimented with varying α settings, with $\alpha = 0.01, 0.005, 0.001$ we observed that when the learning rate was low, the algorithm performed well, it was able to improve its policy gradually, whereas higher values of *alpha* were not able to perform well, they are failing to learn consistently and are falling at some point and are being stuck at local optima.
- Although the algorithm was able to converge with $\alpha = 0.001$, there is a lot of variance around -100. The same can be depicted by the blue curve shown in figure 13.

- As the Q-learning algorithm is prone to bias and variance, to establish a trade-off, after trying different Neural architectures, we have chosen a single hidden layer network with 128 nodes. And also the batch update using memory helps stabilize the model.
- These plots were plotted by running the algorithm 10 times and averaging the results, and it is evident that the algorithm still suffers from some variance.
- For further analysis, $\alpha = 0.001$ was considered as it was performing well.

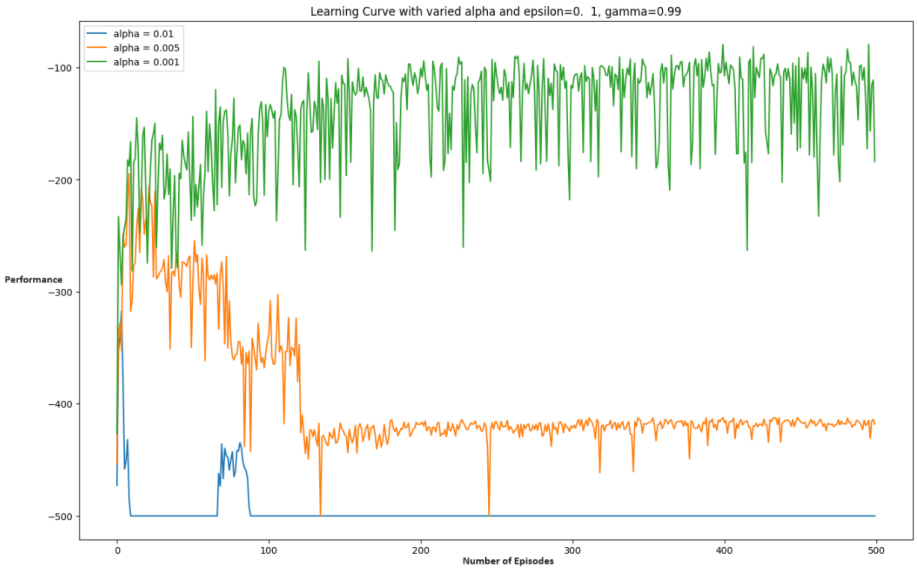

Figure 13: Deep Q-Learning for Varied α setting, averages for 10 iterations

(b) **Varying ϵ setting:**

- Here ϵ is the exploration parameter used in selecting the actions, in order to facilitate exploration of all available states and choose the best gradually.
- In our algorithm, we are decaying the ϵ for every episode by multiplying it with 0.995. This is because the agent needs to reduce its exploration as it gradually learns the best policy.
- When experimented with varying initial ϵ settings, with $\epsilon = 0.1, 0.2, 0.3$ we observed that the algorithm similar for all values of ϵ, but as the curve reaches the end, higher ϵ has slightly better rewards.
- This is because the higher ϵ is facilitating the agent by providing good exploration and thus helping the agent to choose the best actions in the future too.
- For further analysis, $\epsilon = 0.4$ was considered as higher values of ϵ gave good results.

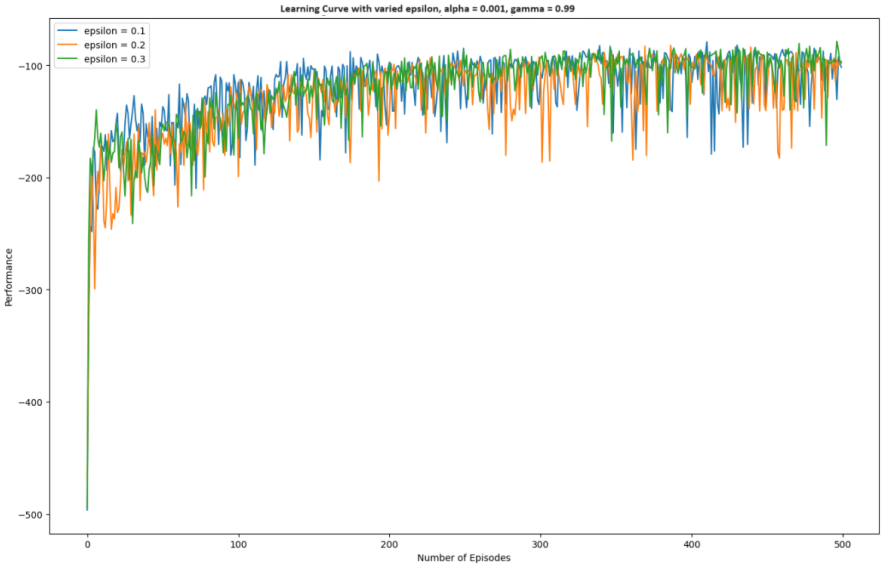

Figure 14: Deep Q-Learning for Varied ϵ setting, averages for 10 iterations

(c) **Varying γ setting:**

- Here, γ is the discounting parameter for calculating the discounted return.
- In Acrobot setting, setting γ to a low value means, we are not penalizing the agent enough to reach the designated height, So the agent might learn to take random actions all the time, not caring much about reaching the designated height.
- Hence, higher value of γ is encouraged in this setting, the same can be depicted in the figure 15.
- In the figure, we can see that, the maximum performance (reaching designated height quickly) achieved by the blue curve (very high γ), is significantly higher compared to that of lower γ settings.
- For further analysis, $\gamma = 0.99$ was considered as higher values of γ gave good results.

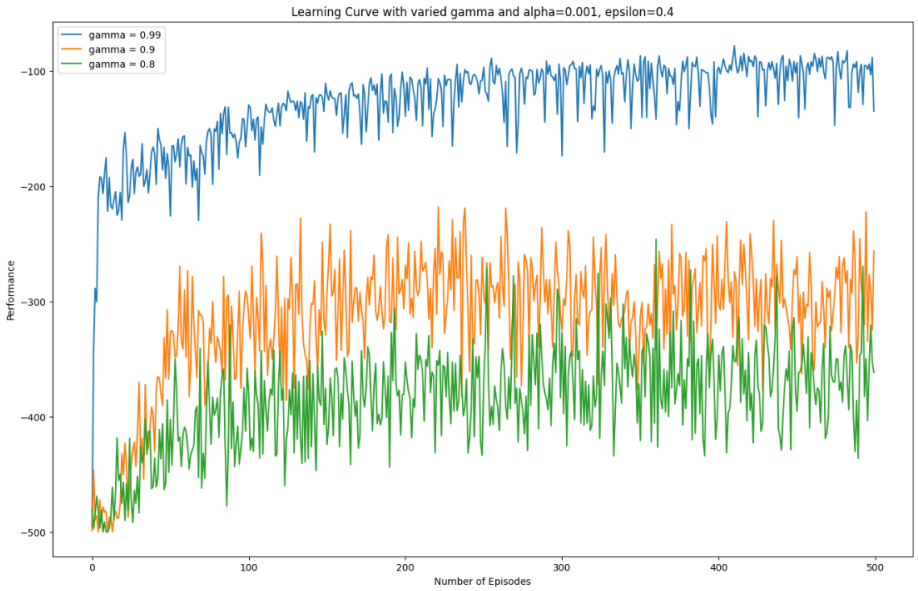

Figure 15: Deep Q-Learning for Varied γ setting, averaged for 10 iterations

(d) **Best Hyper Parameter setting:**

- The best hyperparameters turned out to be $\alpha = 0.001$, $\epsilon = 0.4$, $\gamma = 0.99$.

- Apart from the above-mentioned hyperparameter tuning, various approaches were tried out for finalizing the Neural Network Architecture, weight initialization, Learning rate decay, ϵ decay etc..
- Network architectures like 1,2,3 hidden layers with nodes count 64,128,256 were tried out.
- Weight initialization methods like Lecun, random uniform initialization, and Kaiming normal were tried.
- Network with 1 hidden layer of 128 nodes and Lecun (default) weight initialization provided decent start for the algorithm, thus were used throughout the whole experimentation process.
- A learning rate decay with step size of 25 was used.
- ϵ decay of 0.5% per epoch was fixed.

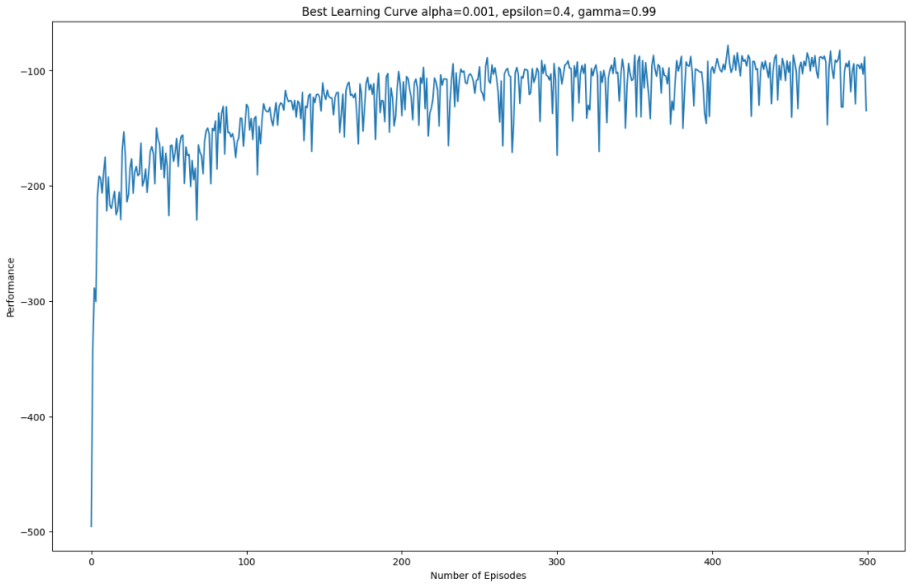

Figure 16: Deep Q-Learning with best parameter setting averaged 5 iterations

.

2.5 Challenges Faced

- The initial code was written from scratch by referring the Q-Learning algorithm from Sutton and Barto.
- This version of code was converging to the maximum sometimes, but is having very high variance and the results were not consistent.
- It took a significant amount of time to try out various Network tuning approaches, hyperparameter tuning and setting decay values for α and ϵ.
- Later, after implementing the Memory list and using batch sampling update, the algorithm started to stabilize and achieve optima consistently.
- We have tried deeper networks to compare the results, but the algorithm was taking too long for each episode to run.
- Achieving to optimal set of hyperparameters was really tough, although it appears simple in the report. Sometimes, starting with some parameters raised a question about if the implementation is really correct.

3 Algorithm - 3: Episodic Semi-Gradient N-step SARSA

As a part of the course project, we wanted to explore How On-policy Control Methods work, so we decided to implement this algorithm.

3.1 Pseudocode/ Algorithm

Semi-Gradient Sarsa N step (Episodic)

Inputs:

- n: Number of steps for bootstrapping
- `env`: Environment
- `action_input_size`: Size of the input for the action-value network
- `action_output_size`: Number of possible actions
- γ: Discount factor
- `num_episodes`: Number of episodes to run
- α: Learning rate
- ϵ: Exploration parameter
- ϵ_{end}: Minimum exploration parameter
- `threshold`: Stopping criterion for average reward

Outputs:

- `episode_states`: List of states encountered in each episode
- `episode_actions`: List of actions taken in each episode
- `episode_rewards`: List of rewards received in each episode
- `episode_scores`: List of cumulative rewards for each episode

Algorithm Steps:

1. Initialize State-Action Value Network.
2. Loop through episodes from 1 to `num_episodes`:
   (a) Reset the environment (`state = env.reset()`).
   (b) Initialize empty lists `S`, `A`, `R`, `Q` to store states, actions, rewards, and action values for each time step
   (c) Set initial score to 0 (`score = 0`).
   (d) Initialize time step counter `t` to 0.
   (e) Sample an action using epsilon-greedy policy and store in `action`.
   (f) Append initial state `state`, action `action`, and action value `action_vals[action]` to lists `S`, `A`, and `Q`.
   (g) Set `T` to infinity.
   (h) While True:
      i. Increment time step `t`.
      ii. If `t < T`:
         A. Take action in the environment (`next_state, reward, done, _ = env.step(action)`).
         B. Append reward to list `R`, next state to list `S`, and action to list `actions`.
         C. Update score (`score += reward`).
         D. If episode is done, set `T = t + 1`.
         E. Else, sample the next action using epsilon-greedy policy.
         F. If `T_small + n < T`, append the next action and its value to lists `A` and `Q`.
      iii. Set `T_small` to `t - n + 1`.
      iv. If `T_small >= 0`:

A. Calculate the return `G` using rewards from time step `T_small + 1` to `min(T_small + n, T)`.
B. If `T_small + n < T`, add the discounted value of the next state-action value.
C. Compute the loss and perform a gradient descent step.
D. If episode is done, print the loss.

v. If `T_small == T - 1`, break the loop.

(i) Update epsilon for the next episode (`epsilon = max(epsilon_end, epsilon * 0.995)`).

(j) Append lists `states`, `actions`, and `rewards` to the respective episode lists.

(k) Print episode information.

(l) If average reward over last 100 episodes exceeds threshold and has low standard deviation, break the loop.

3. Return the episode information lists.

3.2 Evaluating MDPs

1. **Cart Pole Environment**:

   - **Description:** Here, a pole is attached to the cart by a frictionless pivot, the pole is placed upright on the cart and the goal is to balance it by applying forces either left or right.
   - **Action Space:** push cart to the left or right.
   - **Observation Space/ State space:** $(x, v, \theta, \dot{\theta})$ and the limits are:
     position of cart: $x \in (-4.8, 4.8)$, cart velocity: $v \in (-\infty, \infty)$, pole angle: $\theta \in (-24°, 24°)$, pole angular velocity: $\dot{\theta} \in (-\infty, \infty)$.
   - **Starting State:** $(x, v, \theta, \dot{\theta}) \sim \mathcal{U}nif(-0.05, 0.05)$
   - **Rewards:** A reward of $+1$ is given for every timestep including the terminal state.
   - **Episode End:** Episode terminates if:
     – Pole angle is not in between $-12°$ and $12°$.
     – Cart Position is not in between -2.4 and 2.4.
     – Episode length is greater than 500.

   The writeup for this environment was mostly referred from gym documentation of the Cartpole environment (link).

2. **Acrobot**

   - **Description:** The Acrobot system consists of two links connected linearly with an actuated joint. The goal is to swing the free end above a target height by applying torque.
   - **Action Space:**
     - 0: Apply -1 torque to the actuated joint (torque in N m)
     - 1: Apply 0 torque to the actuated joint
     - 2: Apply 1 torque to the actuated joint
   - **Observation Space/ State space:** An array (shape: (6,)) representing joint angles (cosine and sine) and angular velocities.
   - **Rewards:** - Goal reward: 0, Termination state reward: -1, Reward threshold: -100
   - **Starting State:** Initial joint angles and velocities uniformly initialized between -0.1 and 0.1.
   - **Episode End:** Termination condition: -cos(theta1) - cos(theta2 + theta1) > 1.0
     Truncation condition: Episode length > 500

   The writeup for this environment was mostly referred from gym documentation of Acrobot environment (link)

3.3 Hyper-parameter tuning And Observations

1. **Cart Pole Environment:** The hyperparameters taken into consideration are α, ϵ and γ. We ran the Hyperparameter tuning by performing a grid search on:

   - $\alpha : \{0.001, 0.01\}$
   - $\epsilon : \{1, 0.5\}$
   - $N : \{1, 5, 10\}$

   (a) **Varying α setting:**

   - Here α is the learning rate parameter to update the weights of Q (State Action Value) Network.
   - When experimented with varying α settings, with $\alpha = 0.001, 0.01$ we observed that when the learning rate was low, the algorithm was getting better rewards, evident from the blue curve.
   - This is because, as α increases, the algorithm becomes divergent due to bigger steps of update. And lo α helps in converging rather than diverging
   - For further analysis, $\alpha = 0.001$ was considered as it gave higher rewards.

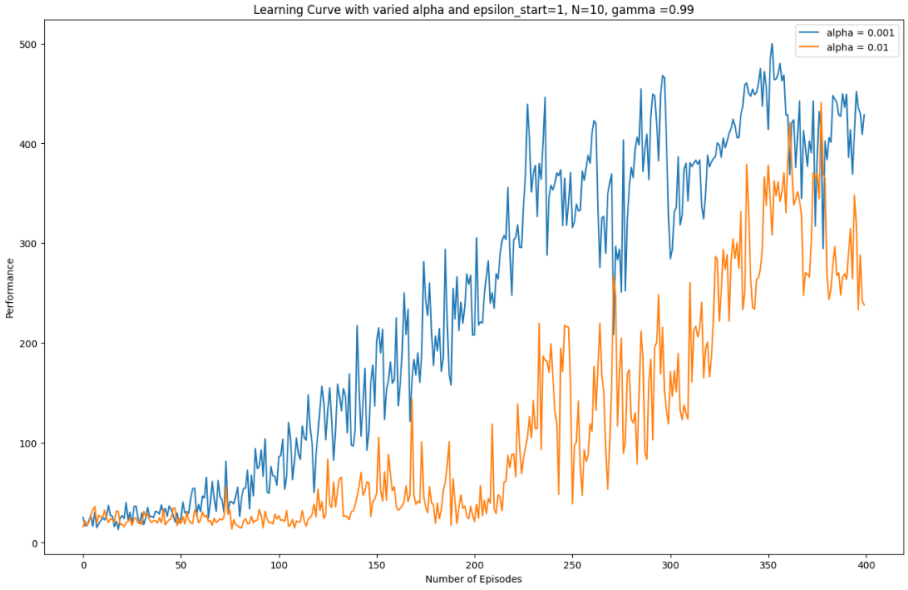

Figure 17: Semi-gradient N step sarsa (episodic) for Varied α setting, averages for 10 iterations

   (b) **Varying ϵ setting:**

   - Here ϵ is the exploration parameter used in selecting the actions, in order to facilitate exploration of all available states and choose the best gradually.
   - In our algorithm, we are decaying the ϵ for every episode by multiplying it with 0.99. This is because the agent needs to reduce its exploration as it gradually learns the best policy.
   - When experimented with varying initial ϵ settings, with $\epsilon = 1, 0.5$ we observed that the algorithm performed well with higher ϵ, evident from the blue curve.
   - This is because the higher ϵ is facilitating the agent by providing good exploration and thus giving a good start.
   - For further analysis, $\epsilon = 1$ was considered as higher values of ϵ gave good results.

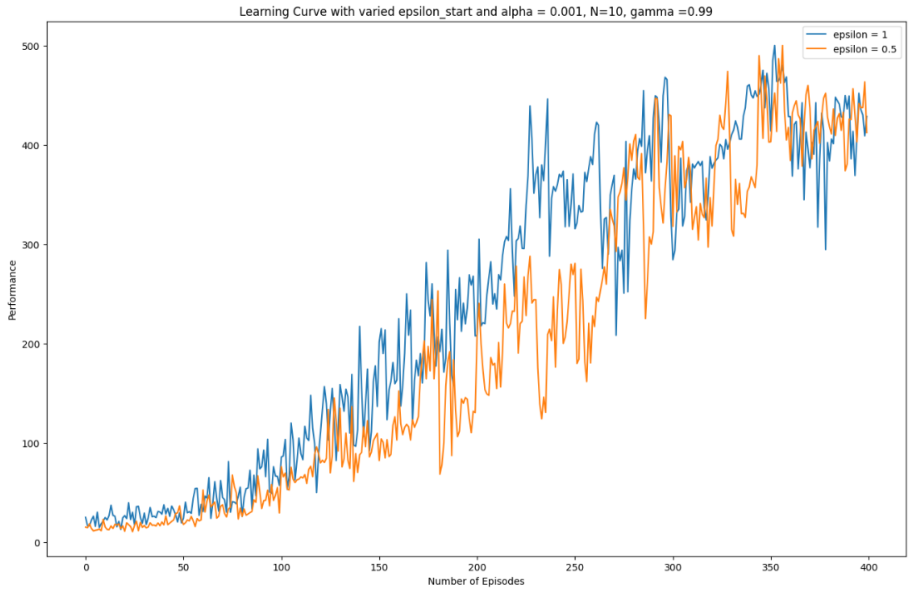

Figure 18: Semi-gradient N step sarsa (episodic) for Varied ϵ setting, averages for 10 iterations

(c) **Varying N setting:**

- Here, N is the number of steps for bootstrapping.
- In Cart Pole setting, setting N to a high value means, the agent sees the reward for n steps in the future. Thus, as N increases the bias decreases.
- This is evident from the graph. The blue and orange curves are performing better than black curve.
- Hence, higher value of N is encouraged in this setting.

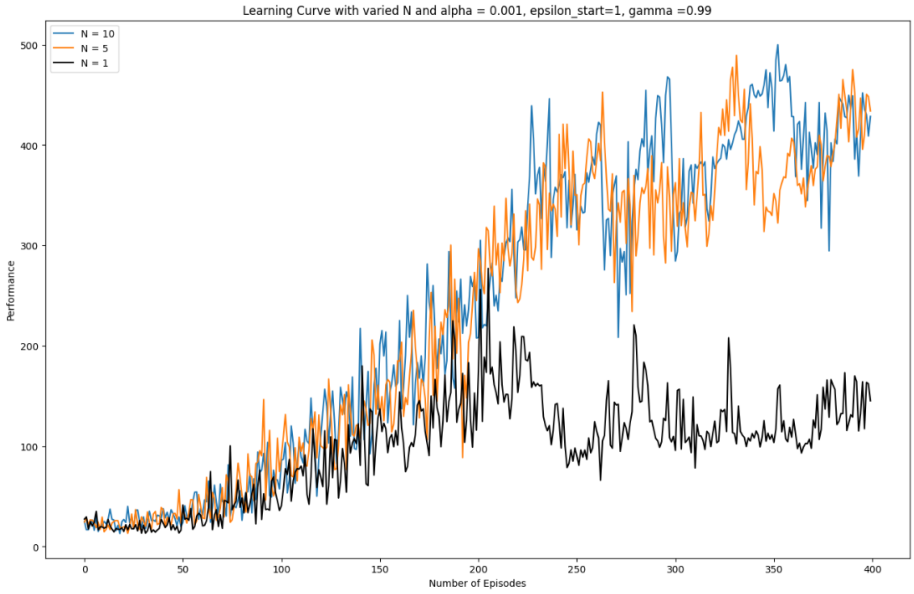

Figure 19: Semi-gradient N step sarsa (episodic) for Varied N setting, averages for 10 iterations

2. **Acrobot Environment:** The hyperparameters taken into consideration are α, ϵ and γ. We ran the Hyperparameter tuning by performing a grid search on:

- $\alpha : \{0.01, 0.001\}$
- $\epsilon : \{0.5, 1\}$

- $N : \{1, 5, 10\}$

(a) **Varying α setting:**

- Here α is the learning rate parameter to update the weights of Q (State Action Value) Network.
- When experimented with varying α settings, with $\alpha = 0.001, 0.01$ we observed that when the learning rate was low, the algorithm was getting better rewards, evident from the blue curve.
- This is because, as α increases, the algorithm becomes divergent due to bigger steps of update. And lo α helps in converging rather than diverging
- For further analysis, $\alpha = 0.001$ was considered as it gave higher rewards.

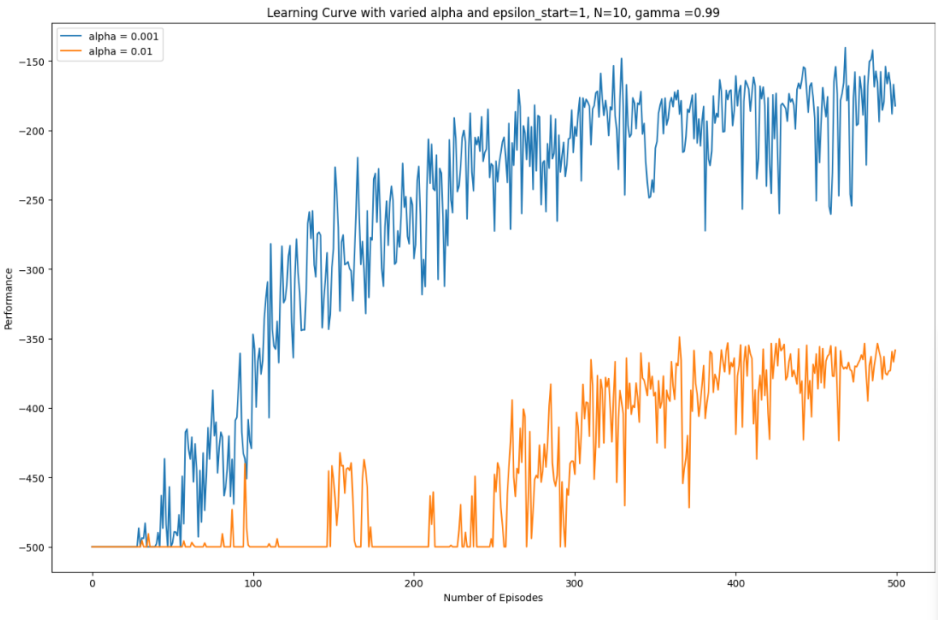

Figure 20: Semi-gradient N step sarsa (episodic) for Varied α setting, averages for 10 iterations

(b) **Varying ϵ setting:**

- Here ϵ is the exploration parameter used in selecting the actions, in order to facilitate exploration of all available states and choose the best gradually.
- In our algorithm, we are decaying the ϵ for every episode by multiplying it with 0.99. This is because the agent needs to reduce its exploration as it gradually learns the best policy.
- When experimented with varying initial ϵ settings, with $\epsilon = 0.5, 1$ we observed that the algorithm performed well for lower values of ϵ.
- This is because the higher ϵ might be hampering the learning process at later stages due to exploration.

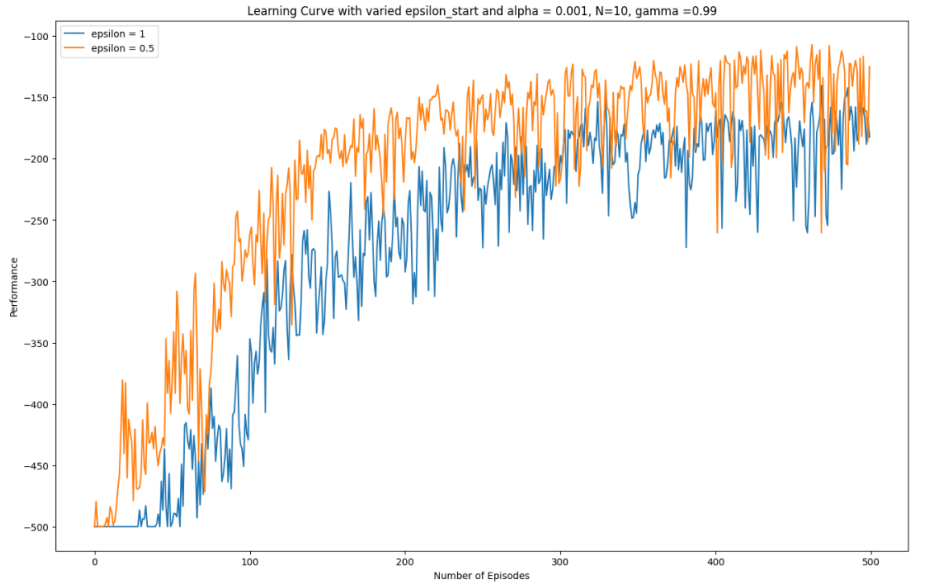

Figure 21: Semi-gradient N step sarsa (episodic) for Varied ϵ setting, averages for 10 iterations

(c) **Varying N setting:**

- Here, N is the number of steps for bootstrapping.
- In Cart Pole setting, setting N to a high value means, the agent sees the reward for n steps in the future. Thus, as N increases the bias decreases.
- This is evident from the graph. The blue and orange curves are performing better than black curve.
- Hence, higher value of N is encouraged in this setting.

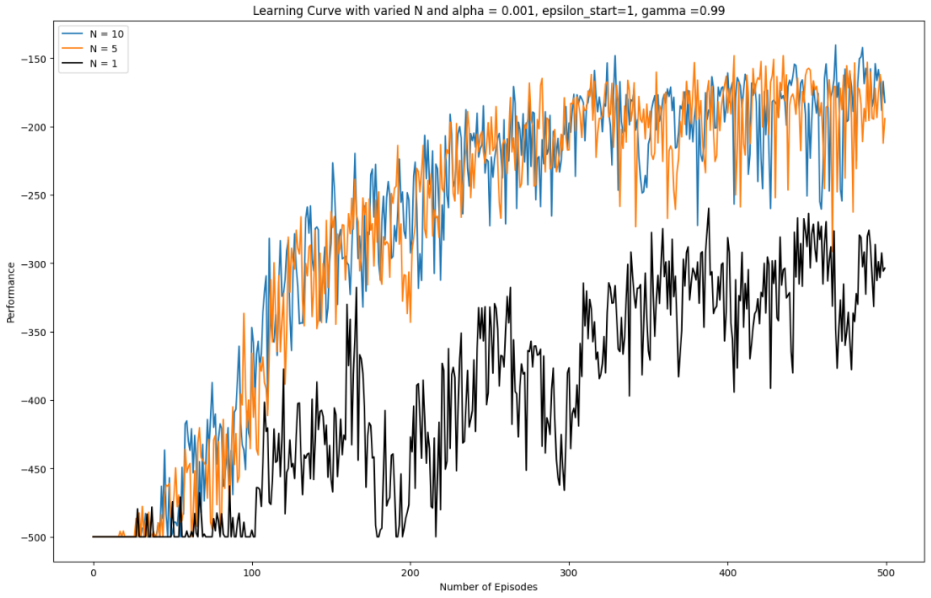

Figure 22: Semi-gradient N step sarsa (episodic) for Varied N setting, averages for 10 iterations

.

3.4 Challenges Faced

- The initial code was written from scratch by referring to Episodic semi-gradient N-step SARSA.

- This version of code was converging to the maximum sometimes, but is having very high variance and the results were not consistent.
- It took a significant amount of time to try out various Network tuning approaches, hyperparameter tuning for N, setting decay values for α and ϵ.
- We have tried deeper networks to compare the results, but the algorithm was taking too long for each episode to run.
- Achieving to optimal set of hyperparameters was really tough, although it appears simple in the report.

4 Conclusion

In this section, we compare the performance of the above 3 algorithms on the MDPs used:

- Cartpole MDP

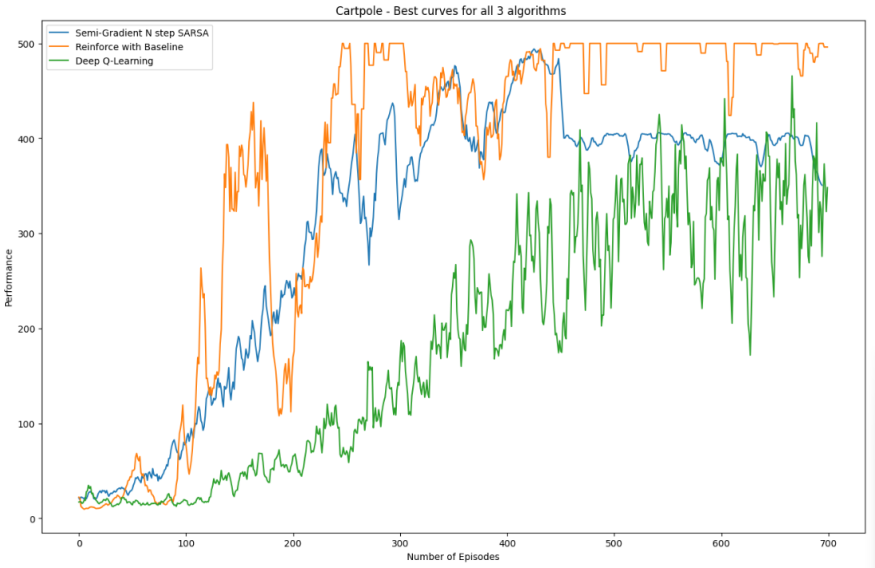

Figure 23: Moving Average of Performance for various algorithms on Cartpole MDP

- Acrobot MDP

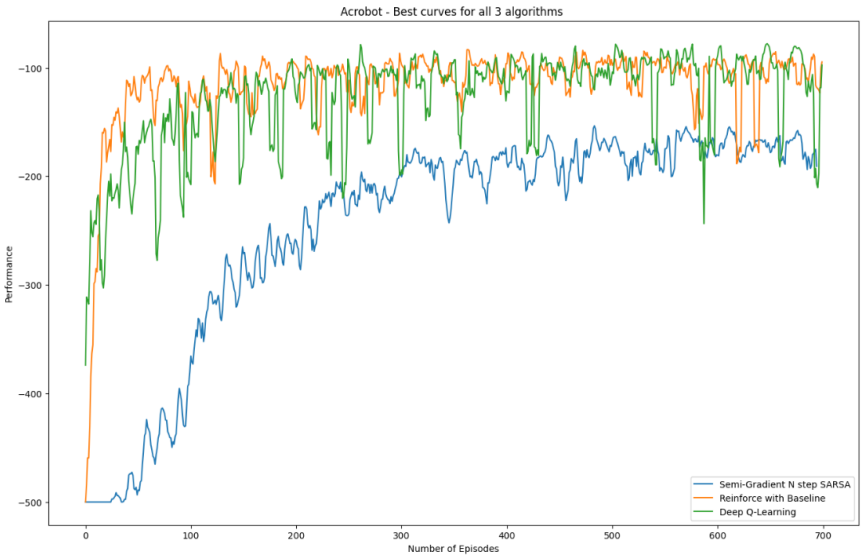

Figure 24: Moving Average of Performance for various algorithms on Acrobot MDP

From the figures 23 and 24, we see that

- It can be inferred that REINFORCE with Baseline converges quickly relative to other algorithms in both the cases.
- Reinforce with baseline is performing better at the end, although there is some variance observed.
- Deep Q-Learning (DQNN), exhibits high variance.
- For Acrobot, Deep Q-Learning is relatively slightly performing better than Reinforce with baseline, but has high variance compared to Reinforce with baseline.
- N-step semi-gradient SARSA, approached the maximum return for Cartpole, but couldn't converge there and fell back to sub-optimal policy.
- For Acrobot, N-step semi-gradient SARSA was not able to converge to the optima. It was stuck at some sub-optimal policy.
- Overall Reinforce with baseline seems to have better performance both the MDPs.

5 Contributions

1. Both of us contributed equally to the project.
2. The code for REINFORCE with Baseline, Analysis for CartPole and Acrobot and Hyper-parameter tuning for this algorithm was done by Abhishek.
3. The code for Deep Q-Learning, in-depth analysis and hyper parameter tuning on Cartpole and Acrobot was done by Akshay.
4. The code for Episodic Semi-gradient N-step SARSA, was implemented by Akshay, it's in-depth analysis was done by Abhishek.
5. Apart from this, we also implemented self driving car MDP, but were not able to come up with the suitable reward function for it and ended up with poor results. We also implemented Actor-critic and Monte-Carlo Tree search algorithms, but because of time constraint, we were not able to perform extensive analysis and mention it in the report.
6. At every point of time, we both have continuously brainstormed on the algorithms and whenever someone gets stuck. Thus making both of us aware of the project components fully.

6 References

- Sutton and Barto RL book
- Gym documentation for using the existing MDP environments: gym documentation.
- Neural Network Weight initializations: https://pytorch.org/docs/stable/nn.init.html
- https://goodboychan.github.io/python/reinforcement_learning/pytorch/udacity/2021/05/12/REINFORCE-CartPole.html
- https://github.com/spro/practical-pytorch/blob/master/reinforce-gridworld/reinforce-gridworld.ipynb
- Pytorch DQN implementation