RL project report

Shravanthi Murugesan, Shivani Anilkumar

December 2024

1 Introduction

In this report, we explore and evaluate three Reinforcement Learning algorithms: REINFORCE with Baseline, Prioritized Sweeping and One-Step Actor-Critic. These algorithms represent different approaches to learning optimal policies. The report investigates the performance of these algorithms across multiple domains including both discrete and continuous environments with the aim of understanding their behaviour under various conditions. The results of the experiments are analyzed, highlighting the impact of different design decisions and hyperparameters on the learning process and agent performance. A detailed discussion of each algorithm and domain is provided in the subsequent sections.

2 Algorithms Implemented

2.1 REINFORCE with Baseline

The REINFORCE with baseline algorithm is a Monte Carlo method, requiring the completion of an entire episode to calculate the actual return. This return is then used to update the policy parameters. As a variation of the policy gradient algorithm, it aims to find the optimal policy by iteratively updating the parameter vector θ, which defines the probability of taking an action A in a given state S.

Although REINFORCE with baseline does not directly rely on value function estimates to determine the optimal policy, it leverages these estimates as a baseline to train the model. Specifically, the estimate of the value function is subtracted from the calculated total returns to reduce variance while maintaining an unbiased estimate of the policy gradient. This variance reduction facilitates more stable learning.

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

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

Figure 1: Pseudo-code for REINFORCE with baseline

2.1.1 Description of the methods used

The algorithm was initially evaluated in two environments:

1. Cats versus Monsters

   The environment is implemented with the same state and action spaces as in Assignment 3. The reward function, transition probabilities, and initial state distribution are identical to those described in Assignment 3.
   Since the states and actions are discrete, we store the policy parameters in a tensor and use a softmax function to convert the policy parameters at a given state into a probability distribution over discrete actions, determining the probability of selecting each action in that state.

   - Initialization
     - Theta (weights for policy parameters): initialized to zero
     - w (weights for baseline parameters): intialized to zero
     - γ: 0.925
   - Environment description:
     - **States**: This problem consists of a 5×5 environment where each state $s = (r, c)$ describes the current coordinates/location of a cat.
     - **Actions**: There are four actions: AttemptUp (AU), AttemptDown (AD), AttemptLeft (AL), and AttemptRight (AR).
     - **Dynamics**: This is a *stochastic* MDP:
       - With 70% probability, the cat moves in the specified direction.
       - With 12% probability, the cat gets confused and moves to the right with respect to the intended direction.
       - With 12% probability, the cat gets confused and moves to the left with respect to the intended direction.
       - With 6% probability, the cat gets sleepy and decides not to move.
       - The environment is surrounded by walls. If the cat hits a wall, it gets scared and does not move.
       - There are four *Forbidden Furniture* locations in this environment: one in $(2, 1)$, one in $(2, 2)$, one in $(2, 3)$, and one $(3, 2)$. If the cat touches a Forbidden Furniture, it gets paralyzed and remains in its current state. The cat cannot go on the furniture.
       - There are two *Monsters*: one in $(0, 3)$ and one in $(4, 1)$.
       - There is a *Food state* located at $(4, 4)$.
     - **Rewards**: The reward is always -0.05, except when transitioning to (entering) the Food state, in which case the reward is 10; or when transitioning to (entering) a state containing a Monster, in which case the reward is -8.
     - **Terminal State**: The Food state is terminal. Any actions executed in this state always transition to s_∞ with reward 0.
     - **Initial State**: The cat wakes up in one of the starting states. The starting states include all the states defined in the state space, except for the monster states, the forbidden furniture state, and the food state.
   - Methods used:
     - `get_next_state()`: Takes the current state and intended action as input and returns the next state that the agent transitions to based on the transition probability.
     - `get_reward()`: Returns the reward obtained by the agent based on the state it transitioned to.
     - `softmax()`: Takes the weights of the policy parameters at the current state and transforms them into a probability distribution for selecting an action in that state.
     - `state_to_index()`: A utility function to convert the state (r, c) to an integer that can be used to index into the weights of the baseline parameter tensor w.
     - `reinforce_baseline()`: Runs the updates for 50,000 episodes.
       - Runs the episode by choosing an action based on the action probability distribution obtained from the current θ values.

* After the episode terminates (i.e., the agent reaches the terminal state), we calculate the every-visit Monte Carlo return obtained at every state.
* The returns calculated are then used to calculate the advantage, which is the difference between the return obtained and the estimated value function.
* We then calculate the policy loss and use that to update the policy parameters.
* The baseline loss is calculated as the distance between the actual value obtained at every time step and the estimated value.
* The gradients are used to update both the policy and baseline parameters using the Adam optimizer.

2. Inverted Pendulum

The environment implemented is similar to the inverted pendulum domain discussed in assignment 2. Since the states and actions are continuous, we use neural networks to parametrize both the policy and the baseline. To select an action, the neural network, parameterized by θ, predicts the mean and standard deviation of a Gaussian distribution based on the current state. The predicted mean and standard deviation define the parameters of this distribution. An action is then sampled from the Gaussian distribution, and the agent executes the sampled action.

- Initialization
  - γ: 1
- Environment Description:
  - **State**: $s = (\omega, \dot{\omega})$, where $\omega \in [-\pi, \pi]$ is the angle of the pendulum relative to the upright position and $\dot{\omega}$ is pendulum's angular velocity. Angles are in radians and $\omega = 0$ means that the pendulum is upright.
  - **Actions**: $a \in [-2, 2]$, where a is a continuous value representing the torque applied to the pivot as the agent attempts to rotate the pendulum clockwise or counterclockwise.
  - **Dynamics**: The dynamics are *deterministic*: taking action a in state s always produces the same next state, s'. Thus, $p(s, a, s') \in \{0, 1\}$. To characterize the dynamics, we first need to define the following constants:

$$\begin{aligned} G &= 10 && \texttt{(gravity)} \\ L &= 1 && \texttt{(length of the pendulum)} \\ M &= 1 && \texttt{(mass of the pendulum)} \\ \texttt{MAX\_SPEED} &= 8 && \texttt{(maximum angular velocity)} \\ \texttt{MAX\_TORQUE} &= 2 && \texttt{(maximum torque)} \\ \texttt{dt} &= 0.05 && \texttt{(time step)} \end{aligned}$$

The next state, $s_{t+1} = (\omega_{t+1}, \dot{\omega}_{t+1})$, can be computed as follows:

$$\begin{aligned} \ddot{\omega}_{t+1} &\leftarrow \frac{3G}{2L}\sin(\omega_t) + \frac{3a_t}{ML^2} \\ \dot{\omega}_{t+1} &\leftarrow \texttt{clip}\big(\dot{\omega}_t + \ddot{\omega}_{t+1}\,\texttt{dt},\ \texttt{-MAX\_SPEED},\ \texttt{MAX\_SPEED}\big) \\ \omega_{t+1} &\leftarrow \omega_t + \dot{\omega}_{t+1}\,\texttt{dt} \end{aligned}$$

where $\texttt{clip}(x, \texttt{min}, \texttt{max})$ is a function that clips the value x so that it stays in the closed interval $[\texttt{min}, \texttt{max}]$.
  - **Terminal States**: Episodes terminate after 200 time steps.
  - **Rewards**: $R_t = -\Big((\omega_{\text{normalized}})^2 + 0.1(\dot{\omega}_t)^2 + 0.001(a_t)^2\Big)$.

    Where, $\omega_{\text{normalized}} = \Big((\omega_t + \pi)\,\texttt{mod}\,(2\pi)\Big) - \pi$, $\texttt{mod}$ is the modulo operator used to get the remainder of a division.

- **Initial State**: $S_0 = (\Omega_0, \dot{\Omega}_0)$, where Ω_0 is an initial angle drawn uniformly at random from the interval $[\frac{-5\,\pi}{6}, \frac{5\,\pi}{6}]$ and $\dot{\Omega}_0$ is an initial angular velocity drawn uniformly at random from the interval $[-1, 1]$.

- Methods used
  - The `Pendulum` class implements the environment and has two methods: `step()` and `reset()`. The `step()` method takes the action as an input and outputs the next state along with the corresponding reward. The `reset()` method re-initializes the pendulum at the beginning of every episode.
  - The `PolicyNetwork` class implements the neural network that is being used to parameterize the policy.
  - The `ValueNetwork` class implements the neural network that is being used to parameterize the value function.
  - Initially, an episode is generated using the `generate_episode()` function. Once we generate an episode, we use the rewards vector to compute the Monte Carlo return at every state using the `compute_returns()` function.
  - Using the returns calculated, we calculate the advantage and the corresponding gradients and accordingly update the PolicyNetwork and the ValueNetwork
  -

3. Mountain Car The mountain car domain has actions that are discrete but the state space is continuous. The environment implemented is as described below.

   - Environment Description:
     - State Space:
       - position of the car along the x-axis: -1.2 to 6.0 units
       - velocity: -0.07 to 0.07 units
     - Action: Three discrete actions
       - Accelerate to the left [0]
       - No acceleration [1]
       - Accelerate to the right [2]
     - Transition Dynamics

$$velocity_{t+1} = velocity_t + (action - 1) * force - cos(3 * position_t) * gravity$$

$$position_{t+1} = position_t + velocity_{t+1}$$

       where,
       $force = 0.001$
       $gravity = 0.0025$
     - Reward: The agent is given a negative reward of -1 at every time step until it reaches the flag. It is given a reward of on reaching the flag.
     - Initial State: The agent's position is a uniform random value in [-0.6 , -0.4]. The starting velocity of the car is always assigned to 0.
     - There are two terminating conditions for the episode:
       - The agent's position is ≥ 0.5.
       - The number of time-steps in the episode is ≥ 200.

   We have used neural networks for parameterizing the policy and the baseline. The policy neural network takes the current state as an input and outputs a logits vector on which we apply the softmax function to get the probability distribution of picking an action in the current state. The temperature parameter is set to 0.995 to control the level of exploration. Reducing the temperature makes the resulting policy more deterministic.

2.1.2 Hyper-parameter Tuning

Hyper-parameters: $\gamma, \alpha_w, \alpha_\theta$

- γ, Discount Factor: As the value of γ decreases, the agent places less emphasis on distant rewards, thereby requiring less time to train. This is because the impact of future rewards on the current policy update is reduced. On the other hand, a higher γ ensures that the agent considers distant rewards, allowing it to plan for long-term outcomes during the current time step. The value of γ was chosen through trial and error to balance the trade-off between immediate and future rewards effectively.
  - Cats versus Monsters: The gamma value chosen was $\gamma = 0.925$. This is because lower values of gamma do not account for the monster states effectively and hence the policy obtained is not optimal.
  - Inverted Pendulum: The γ value chosen is 1.
  - Mountain Car: The γ value chosen is 0.99.
- α_w, Learning Rate for Value Function: This parameter controls how much the return at the current time step influences the update of the weights in the baseline network. A well-tuned α_w ensures stable learning of the value function while preventing overfitting to recent updates.
  A value of 0.001 was chosen as the learning rate for the value function since it facilitated accurate learning. Lower values of α_w would significantly slow down the training process, while higher values of α_w resulted in drastic updates to the policy, leading to unexpected deviations in the learning curve.
- α_θ, Learning Rate for Policy Parameters: This parameter dictates the step size for updating the policy parameters, θ. It determines the magnitude of change in the policy parameter vector during each iteration. Choosing an appropriate α_θ is crucial for stable convergence; a value too high can cause divergence, while a value too low may result in slow learning.

2.1.3 Experimental Results

1. Cats versus Monsters
   The learning curve depicting the number of time-steps taken by the agent to complete an episode plotted against the episode number.

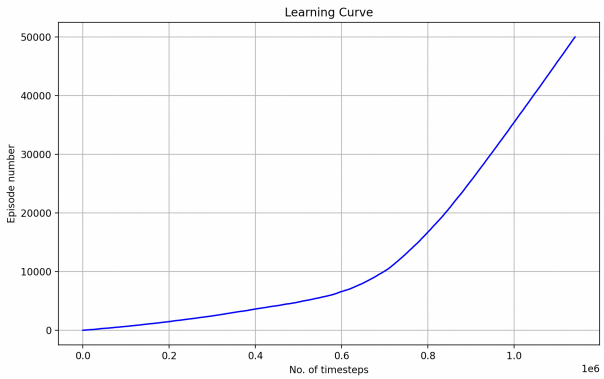

Figure 2: Learning curve - No. of time steps vs Episode number

The reward earned by the agent every 100th episode plotted against the episode number.

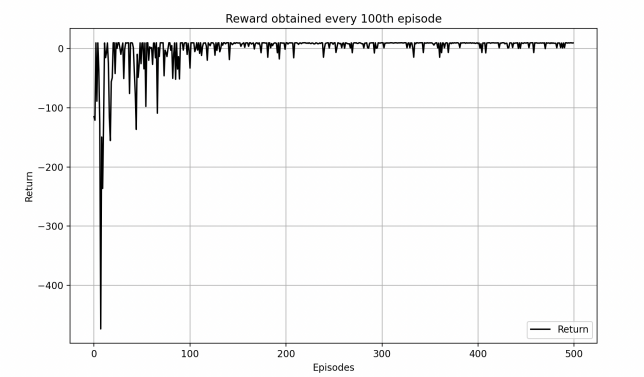

Figure 3: Return earned every 100th episode

The estimated value function at the end of running 50,000 episodes.

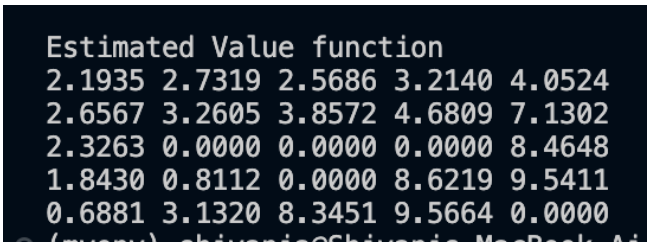

Figure 4: Estimated value function

2. Inverted Pendulum
   Return obtained at every time-step, averaged over 20 runs of the algorithm along with the standard deviation.

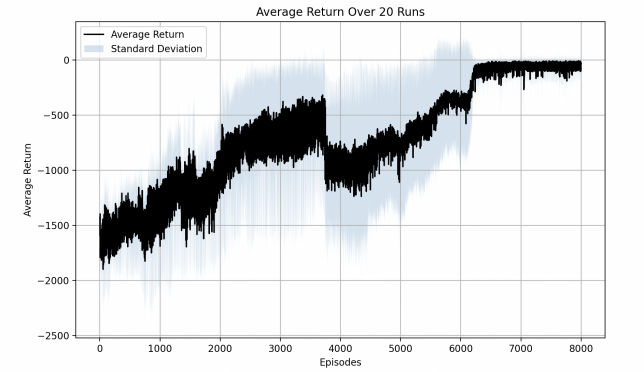

Figure 5: Learning curve

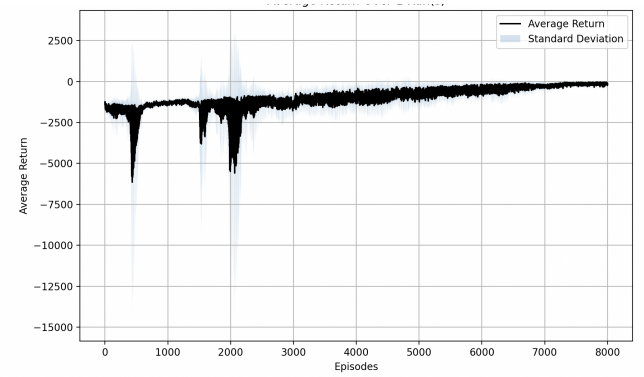

Figure 6: Learning curve

During some runs, the agent tended to explore drastically in the initial stages, leading to very low rewards during the early training period. This behavior highlights the fact that REINFORCE with baseline can sometimes be unstable, as it relies on running entire episodes and calculating the Monte Carlo reward without bootstrapping. However, it is important to note that the agent was still able to learn an optimal policy to maximize rewards. As observed in the plot, the rewards approach zero by the end of 8,000 iterations in one run.

3. Mountain Car
   Learning curve depicting the total return obtained in every episode plotted against the episode number.

   We can see from the curve that the agent is initially exploring its environment during which the returns obtained vary greatly and once the agent learns the optimal policy, it exploits the policy to maximize rewards after convergence.

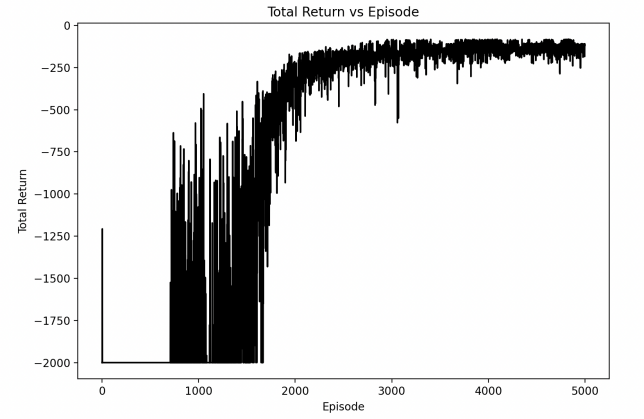

Figure 7: Learning curve

2.2 Prioritized Sweeping

2.2.1 Overview of the Algorithm

This approach focuses on state-action pairs that are likely to have most significant impact on the overall value function. The priority of a state action pair is calculated based on the magnitude of its potential change. This is calculated as the difference between current Q value and the expected Q value:

$$P = |R + \gamma max_a Q(S', a) - Q(S, A)|$$

Where,

- R is the reward observed
- γ is the discount factor, controlling the agent's consideration of future rewards
- $max_a Q(S', a)$ is the maximum possible Q value of the next state
- Q(S,A) is the current Q value for the state-action pair

The priority P reflects how much the Q-value of a state-action pair is expected to change. Larger changes in the Q-value of a state-action pair are more likely to have a significant impact on the agent's policy, and therefore, these pairs are prioritized for updates. The idea is to focus computational resources on the state-action pairs that will lead to the most substantial policy improvements. Once the priority P for a state-action pair is calculated, it is compared against a threshold θ. Only those state-action pairs with priorities greater than θ are inserted into the priority queue. The threshold θ acts as a filter to ensure that only state-action pairs that are expected to undergo a significant change are updated. Once the priority for each state-action pair is calculated and those with priority greater than θ are added to the queue, the priority queue is used to manage the updates. The state-action pair with the highest priority is selected for updating its Q-value. This ensures that the most impactful changes are addressed first, speeding up the convergence of the algorithm.

After updating the Q-value of the selected state-action pair, the effect of this change on its predecessor states (states that lead to the current state S) is evaluated. If the Q-value update is significant enough to affect the value of any predecessor state-action pairs, these pairs are also added to the priority queue with their updated priority values.

This update propagation process continues recursively: each time a state-action pair is updated, the changes propagate backward through the model, affecting predecessor pairs and leading to further updates. The update process stops when no further significant changes (as determined by the threshold θ) are observed in the state-action pairs.

The pseudocode of the algorithm is shown below in Figure 8.

Prioritized sweeping for a deterministic environment

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

Figure 8: Pseudocode of Prioritized Sweeping

2.2.2 Description of the methods used

1. Cats Vs Monsters: The environment used in this experiment is described in detail in Section 2.1.1.

2. DynaMaze

- **States**: This problem consists of a 6×9 grid maze with 54 states. Each state $s = (r, c)$ represents the agent's current coordinates (row r, column c) in the maze.
- **Actions**: The agent has four possible actions: *left*, *up*, *right*, and *down*.
- **Obstacles**: There are several blocked cells in the maze that prevent movement they are: (1, 2), (2, 2), (3, 2), (0, 7), (1, 7), (2, 7), (4, 5)
- **Dynamics**: The environment is deterministic:
  - Each action moves the agent to the corresponding neighboring state in the direction specified by the action, provided the movement is not blocked by an obstacle or the edge of the maze.
  - If the agent attempts to move into a blocked position or off the edge of the maze, it remains in the current state.
- **Rewards**:
  - The reward for each transition is 0, except when the agent reaches the goal state G (0,8), where the reward is +1.
  - Upon reaching the goal state, the agent resets to the start state S (0,2) to begin a new episode.
- **Terminal State**: The goal state $G = (0, 8)$ is terminal. Once the agent reaches the goal, the episode terminates and the agent returns to the start state $S = (2, 0)$ for the next episode.
- **Initial State**: The agent deterministically starts each episode at the start state $S = (2, 0)$.
- **Maze Layout**: The maze is a 6×9 grid, with the following configuration:

.	.	.	.	.	.	.	X	G
.	.	X	.	.	.	.	X	.
S	.	X	.	.	.	.	X	.
.	.	X	.	.	.	.	.	.
.	.	.	.	.	X	.	.	.
.	.	.	.	.	.	.	.	.

  - S is the start state at $(2, 0)$.
  - G is the goal state at $(0, 8)$.
  - X represents blocked (obstacle) positions.

3. Inverted Pendulum: The environment used in this experiment is described in detail in Section 2.1.1.

2.2.3 Experimental Results

1. Cats Vs Monsters

   Prioritized Sweeping was implemented for this domain. The parameters which were tuned are:

   - α: The learning rate, extent to which controls how much the Q-value of a state-action pair is updated during the prioritized backup process, based on the difference between the current and expected values.
   - ϵ: The exploration parameter which controls the randomness in action selection

   Initially, a range of values for both α and ϵ were selected. The parameters were systematically varied to identify the combination that resulted in the best performance, measured by the Mean Squared Error (MSE) between the value function learned by the agent and the optimal value function. The algorithm was run for 500 episodes using each combination and this was repeated 20 times. The MSE was calculated at the end of each episode, and the number of steps (or actions) taken per episode was also recorded. The MSE values between learned value function and optimal value function for the top 4 combinations are shown in Table 1.

   From the results, it is clear that $\epsilon = 0.2$ with $\alpha = 0.05$ yields the lowest MSE, indicating that this combination results in the most accurate learned value function in comparison to the optimal value function. ϵ=0.2 strikes a balance between exploration and exploitation, allowing the agent to learn effectively.

Alpha (α)	Epsilon (ϵ)	MSE
0.05	0.2	1.7315
0.05	0.5	2.3502
0.05	0.1	2.3919
0.1	0.2	2.3928

Table 1: MSE Results for Different Parameter Combinations in Prioritized Sweeping

Though the combination of ϵ=0.2 and α=0.1 is close to this combination, it resulted in a higher MSE. This could be due to the higher learning rate resulting in larger updates making the learning process unstable

With a low value for ϵ (0.05), the agent exploits its learned policy, leading to a reasonable MSE of 2.3015. As ϵ increases to 0.5, the agent's performance degrades (MSE = 2.3502), the increased exploration disrupts the learning process and prevents convergence to an optimal solution. ϵ=0.2 provides the right balance between exploration and exploitation allowing closest convergence to optimal solution.

Two learning curves were plotted for each parameter combination, the first curve shows the the number of steps taken per episode (or the number of actions per episode) as the number of episodes increases. This gives insight into the efficiency of the agent's learning process. The second curve shows the MSE between the learned value function and the optimal value function, providing a measure of how close the agent's learned value function is to the optimal solution. Both curves were averaged over 20 runs of the algorithm, with each run consisting of 500 episodes. The learning curves of the best parameters are shown below in Figures 9 and 10.

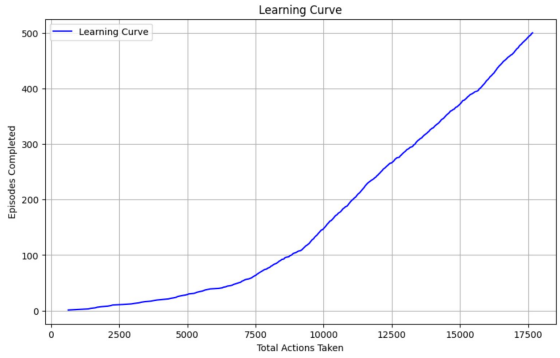

Figure 9: Learning Curve 1: ϵ=0.2, α=0.05

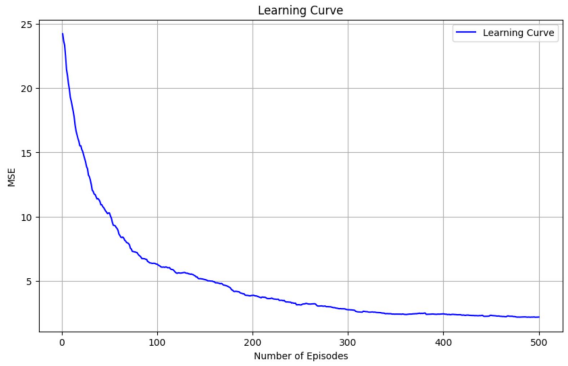

Figure 10: Learning Curve 2: ϵ=0.2, α=0.05

The value function identified using the best parameters ϵ=0.2 and α=0.05 is shown below in Figure 11.

```
The value function estimate after 500 episodes is:

2.4717  5.1809  1.2364  6.4862  4.7640
5.2044  4.7691  6.0603  6.0139  8.3510
4.7596  X       X       X       9.1722
0.3796  -0.0608 X       9.1997  9.9942
-0.1381 4.3284  4.8172  9.9997  G
```

Figure 11: Value function identified

The greedy policy learned using the best parameters ϵ=0.2 and α=0.05 is shown below in Figure 12.

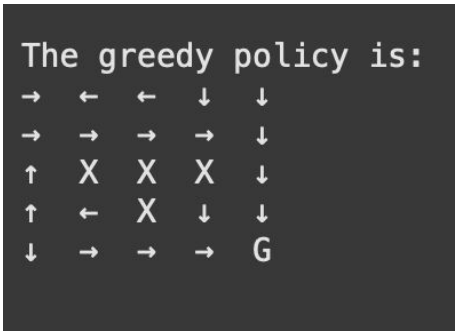

Figure 12: Greedy Policy learned

2. Dyna Maze

   Prioritized Sweeping was implemented for this domain, with various values of the exploration parameter ϵ and the learning rate α tested to identify the optimal parameter combination. The algorithm was run for 50 episodes per trial and 20 trials were done, and the number of steps (or actions) taken per episode was recorded. For each parameter combination, a plot of the number of steps per episode versus the number of episodes was generated. The plots are shown below as Figure 13, 14, 15.

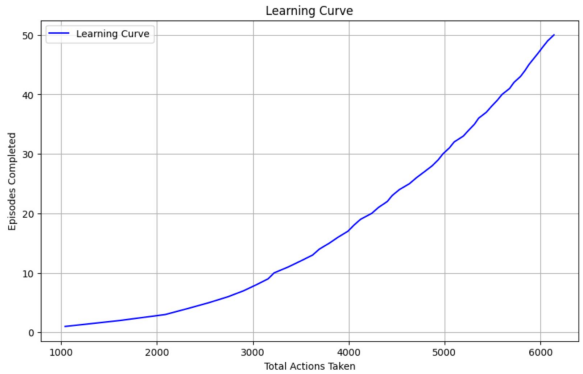

Figure 13: Learning Curve 1: ϵ=0.05, α=0.05

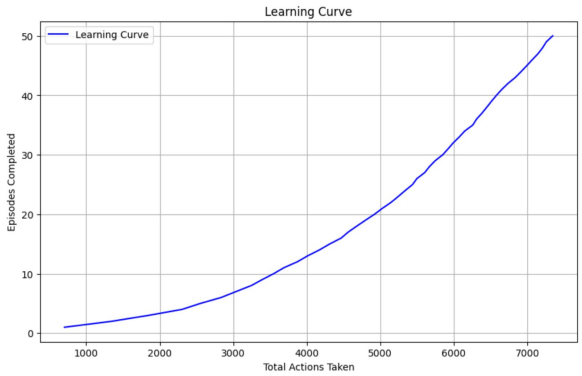

Figure 14: Learning Curve : ϵ=0.5, α=0.05

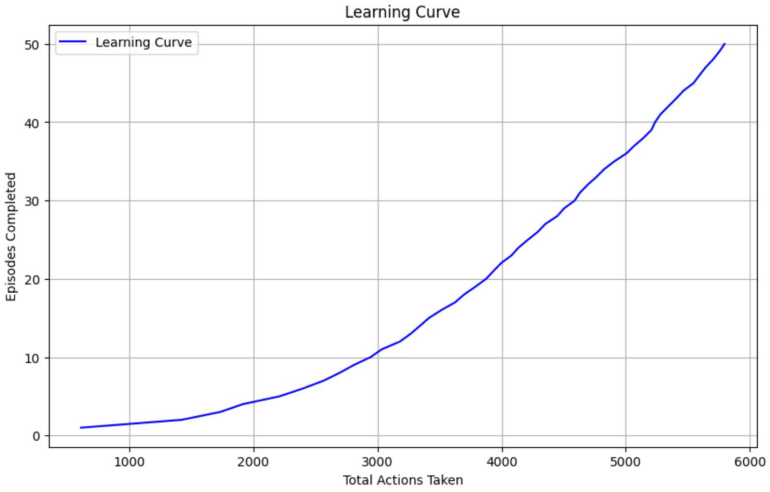

Figure 15: Learning Curve: ϵ=0.2, α=0.05

The analysis of these plots revealed that ϵ=0.2 provided the best balance between exploration and exploitation, requiring fewer than 6000 total steps. When ϵ=0.05, the agent's limited exploration led to a suboptimal solution, requiring a more number of steps (between 6000 and 7000) to converge. When ϵ=0.5, excessive exploration resulted in a high number of steps (over 7000) as the agent struggled to converge due to the increased randomness in its actions. The greedy policy learned with the best parameters (ϵ=0.2, α=0.05) is shown below in Figure 16.

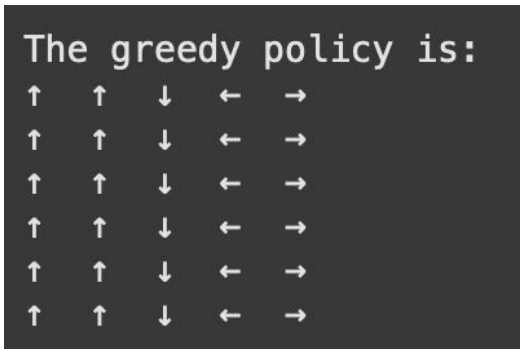

Figure 16: Greedy Policy Learned

3. Inverted Pendulum
   The states in the Inverted Pendulum domain are continuous, to apply it to Prioritized Sweeping, the states and actions were discretized. Different discretization levels were explored for both states (ω and $\dot{\omega}$) with values of 40, 50, and 60, and for actions with bin sizes of 5, 10, and 15. In addition, various values of the exploration parameter (ϵ) and the learning rate (α) were tested to assess their impact on the agent's performance. However, the agent's learning performance did not improve as expected, with the total return remaining consistently low across episodes. The learning curve is shown in Figure 17.

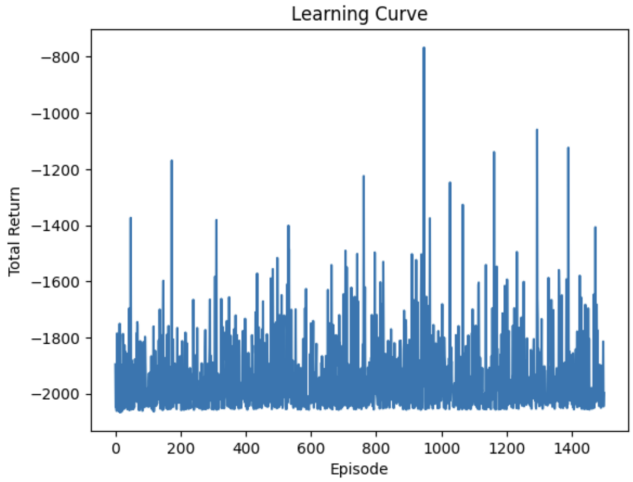

Figure 17: Learning Curve

2.3 One Step Actor Critic method

2.3.1 Overview of the Algorithm

This algorithm has two key components: the actor and the critic. The actor component defines the policy, it is responsible for selecting actions based on the current policy parametrized by θ. The critic estimates the value function, it helps evaluate the actions chosen by the actor. In this algorithm, the critic uses one-step return also known as TD(0) return, this is the immediate reward plus the value function estimate of the next state

and is used instead of the return from the current state. The actor's policy is updated using Policy gradient theorem. The critic's estimate helps compute a more accurate gradient of the expected return with respect to the policy parameters. The agent interacts with the environment by observing its current state S_t and selecting an action A_t based on its policy $\pi(a|s,\theta)$, which is parameterized by θ. After taking action A_t, the agent transitions to a new state S_{t+1} and receives a reward R_{t+1}. The critic computes the one-step return $G_{t:t+1}$, which is a combination of the reward and the estimated value of the next state S_{t+1}. This return is given by:

$$G_{t:t+1} = R_{t+1} + \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$$

Where, $\hat{v}(S_{t+1}, w)$ is the estimated value of the next state and $\hat{v}(S_t, w)$ is the estimated value of the current state. The policy update is then carried out by the actor using the policy gradient method. The policy parameters θ are updated based on the one-step return aimed to increase the return.

$$\theta_{t+1} = \theta_t + \alpha \cdot (G_{t:t+1} - \hat{v}(S_t, w)) \cdot \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$$

This update helps the policy improve by moving in the direction that maximizes the expected return. The critic's value function is updated using the semi-gradient TD(0) method. This learning process allows the critic to more accurately estimate the value function, which in turn aids the actor in selecting better actions.
The pseudocode of the algorithm is shown below in Figure 18.

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 18: Pseudo-code of One step Actor Critic

2.3.2 Hyperparameter Tuning

- α_w: The learning rate for the value function estimate is set when initializing the Adam optimizer. In our implementation, we use a learning rate of 0.001 to ensure that the weights are not updated significantly based on the returns of a single episode, while still allowing for smooth learning.
- α_θ: The learning rate for the policy function estimate is set when initializing the Adam optimizer. In our implementation, we use a learning rate of 0.001 to ensure that the weights are not updated significantly based on the returns of a single episode, while still allowing for smooth learning.

2.3.3 Experimental Results

- Cats versus Monsters domain
  The environment used in this experiment is described in detail in Section 2.1.1.

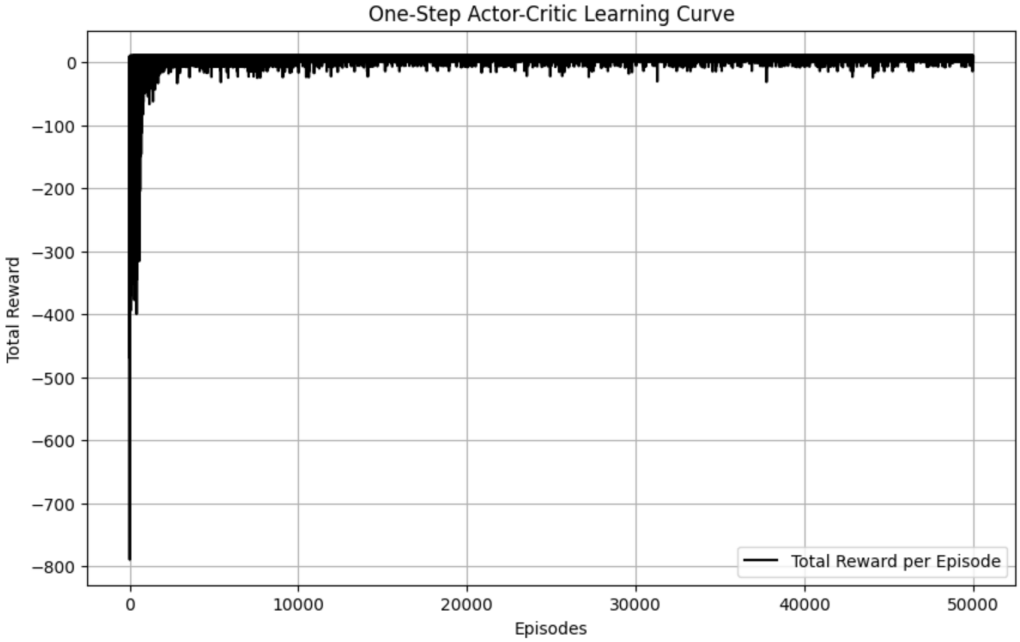

Figure 19: Learning curve for the One Step Actor Critic algorithm