Final project

Lian Fu: REINFORCE with Baseline, One-Step Actor-Critic
Deepa Rukmini Mahalingappa:REINFORCE with Prioritized Sweeping, Monte Carlo Tree Search

December 2024

1 REINFORCE with Baseline

1.1 Brief description of the methods

According to the Policy Gradient Theorem:

$$\nabla J(\theta) \propto E\left[\sum_{t=0}^{\infty} \gamma q^{\pi_\theta(S_t,A_t)} \frac{\partial \ln\left(\pi\left(S_t, A_t; \theta\right)\right)}{\partial \theta}\right] \tag{1}$$

It's very time-consuming to calculate the $q(s,a)$, so the unbiased estimator G_t can be used as an alternative. Just as what shows in Monte-Carlo, though G is unbiased, it will bring variance to the estimates. Then a baseline is introduced to reduce the variance.The variance of estimates can be decreased by introducing a correlated variable, now we are trying to get the estimates of $q(s,a)$, and since $v(s) = \sum_a \pi(s,a)q(s,a)$, the correlated variable is $v(s)$. Take an example, after an episode, we get $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$, now we want the estimates of $q(s,a)$, inspired by Monte Carlo, we use G as estimates, to decrease variance, new estimates $G_t - v(s_t)$ will be introduced.

$$\nabla J(\theta) \propto E\left[\sum_{t=0}^{\infty} \gamma (G_t - v(s_t)) \frac{\partial \ln\left(\pi\left(S_t, A_t; \theta\right)\right)}{\partial \theta}\right] \tag{2}$$

Above is the update rule of REINFORCE with Baseline, though we don't know what $v(s)$, we can construct an estimate of it $\hat{v}_w^\pi(S_t)$. Here are some different ways to approximate value function, in this project we implement two methods on the domain cart-pole:

1.1.1 Feature Construction:Fourier Features

The rule of Fourier features is:

$$\hat{v}_w(s) = w^T \phi(s) = \sum_{i=1}^{d} w_i \phi(s) \tag{3}$$

Theoretically, as d increases, it can approximate any smooth value functions. In the Cart pole domain, the observation space is 4 dimension. we choose to use order-4 Fourier basis. Params setting: $\gamma = 0.99$, $\alpha_w = 2e-4$, $\alpha_\theta = 2e-5$.

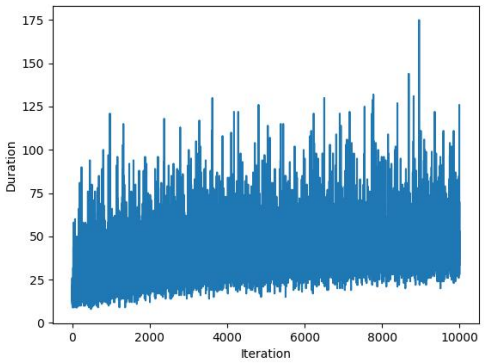

Figure 1: Feature construction approximation- order-4 Fourier basis

The trend of the graph shows that though the duration increases, the speed of convergence is so slow, which will cost much time. It may be because the approximation of value function is not accurate enough. While if we increase the number of Fourier basis, the number of features will grow exponentially. So it seems doesn't work well.

1.1.2 Feature Construction:Neural Network

Neural network can work well when approximating non-linear function. Params setting: number of hidden layer:1, number of neurons of hidden layer: 128, $\alpha_w = 1e-4$, $\alpha_\theta = 1e-3$. The growth of duration is much faster compared

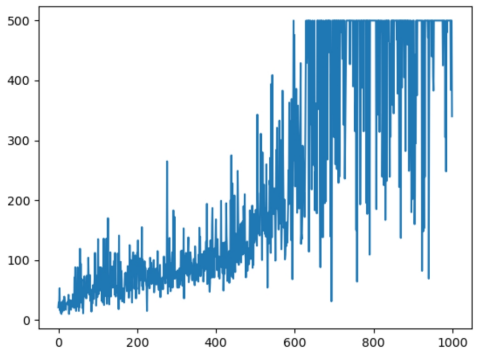

Figure 2: Feature construction approximation- neural network

to Fourier Features. Based on the experiment result, we choose to use neural network as the approximation of value function.

1.2 Pseudocode

Algorithm 1 REINFORCE with Baseline

Input: α_θ and α_w, hidden layer, a list contains neurons of each hidden layer
for episode = 1,2,...,1000 **do**
    Initialize the $S_0 = [x : 0, v : 0, \theta : 0, \hat{\theta} : 0]$ for cart pole and $S_0 = [\theta_1, \theta_2, \hat{\theta_1}, \hat{\theta_2}]$ each params is initialized uniformly between -0.1 and 0.1 for acrobot.
    $S_t \leftarrow S_0$
    step $\leftarrow 0$
    Initialize empty Lists Actions and States
    bool terminate $\leftarrow$ False
    while terminate == False and step < MAX STEPS **do**
        Add S_t to the List States
        $A_t \leftarrow$ choose an action according to the possibility given by PolicyNeuralNetwork(S_t)
        S_{t+1}, terminate, $R_t \leftarrow$ step(S_t,A_t)
        Add A_t to the List Actions
        step $\leftarrow$ step + 1
    end while
    for t = 0,1,..., step **do**
        $G \leftarrow \sum_{k=t+1}^{step} \gamma^{k-t-1} R_k$
        $\delta \leftarrow G - \hat{v}(S_t, w)$
        Update value function network: $mse(G, v(S_t)).backward()$
        Update policy network: $-\delta \cdot \ln \pi(A_t \mid S_t, \theta).backward()$
    end for
end for

1.3 Tune the parameters

First we set the learning rate $\alpha_\theta = 0.1$ and $\alpha_w = 0.1$, and soon we get the error report that value network gets *nan* output, which may cause by the large learning rate α_w, then we decrease α_w by half, until network doesn't output *nan*. Then we found that, if the α_θ is too large, the parameters of policy network will not be able to updat properly because of the gradient explosion. And the duration will stay the same. We decrease the α_θ to $1e-4$ and the output of each episode shows the reasonable trend.

1.4 Results

Params for domain Cart-pole: $\alpha_\theta = 1e-4$, $\alpha_w = 1e-3$, hidden layer = [128], the plot of duration of each episode and number of the episode shows below:

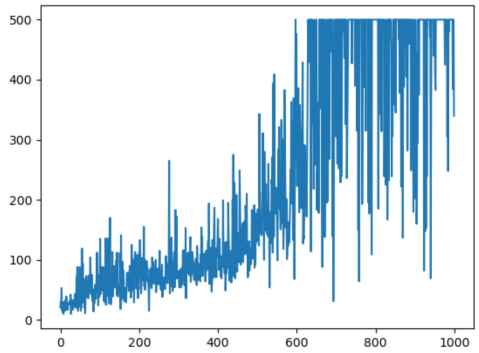

Figure 3: REINFORCE with Baseline: Cart-Pole

Params for domain Acrobot: $\alpha_\theta = 1e-4$, $\alpha_w = 1e-3$, hidden layer = [128], the plot of duration of each episode and number of the episode shows below:

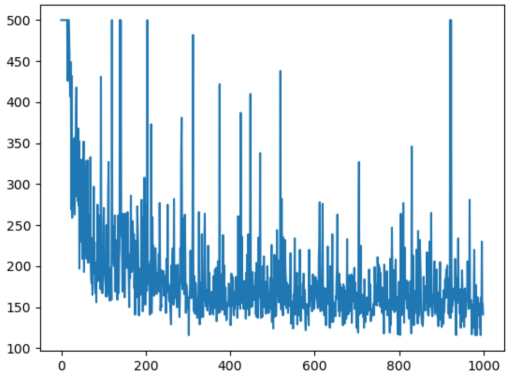

Figure 4: REINFORCE with Baseline: Acrobot

2 One-Step Actor-Critic algorithm

2.1 Brief description of the methods

In the REINFORCE with Baseline, the estimates sets a baseline for subsequent return,but the baseline is made prior to the transition's action and thus cannot

be used to assess that action. As what shows in TD learning, though one-step return introduces bias, it is often superior to the actual return in terms of its variance and computational congeniality. We will still use the update rule introduced in REINFORCE with Baseline, but instead of updating the weights of network after generating the whole episode, the weights will be updated at the same time episode generates.

2.2 Pseudocode

Algorithm 2 REINFORCE with Baseline

Input: α_θ and α_w, hidden layer, a list contains neurons of each hidden layer
for episode = 1,2,...,1000 **do**
    Initialize the $S = [x : 0, v : 0, \theta : 0, \hat{\theta} : 0]$ for cart pole and $S = [\theta_1, \theta_2, \hat{\theta_1}, \hat{\theta_2}]$ each params is initialized uniformly between -0.1 and 0.1 for acrobot.
    step $\leftarrow 0$
    bool terminate $\leftarrow$ False
    while terminate == False and step $<$ MAX STEPS **do**
        $A_t \leftarrow$ choose an action according to the possibility given by PolicyNeuralNetwork(S_t)
        S_{t+1}, terminate, $R_t \leftarrow$ step(S_t,A_t)
        $\delta \leftarrow R_t + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$
        estimate $\leftarrow R_t + \gamma\hat{v}(S_{t+1}, w)$
        Update value function network: $mse(estimate, v(S_t)).backward()$
        Update policy network: $-\delta \cdot \ln \pi(A_t \mid S_t, \theta).backward()$
        $S_t \leftarrow S_{t+1}$
        step $\leftarrow$ step + 1
    end while
end for

2.3 Tune the parameters

I use the same strategy as what described in REINFORCE with Baseline. The parameters of domain Acrobot: $\alpha_\theta = 1e-3$, $\alpha_w = 1e-3$. The parameters of domain Cart-Pole: $\alpha_\theta = 1e-4$, $\alpha_w = 1e-4$.

2.4 Results

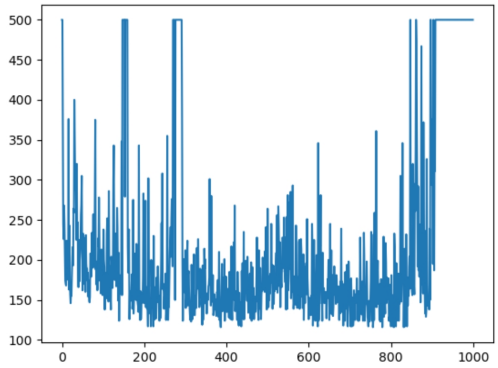

Figure 5: One-Step Actor-Critic algorithm: Acrobot

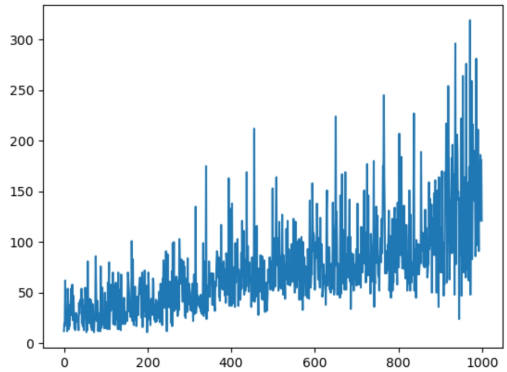

Figure 6: One-Step Actor-Critic algorithm: Cart-Pole

3 REINFORCE with Prioritized Sweeping

3.1 Brief description of the methods

This project implements the Prioritized Sweeping algorithm, a model-based reinforcement learning method, in a discretized Acrobot environment. The environment is modeled as a Markov Decision Process (MDP) with discrete states and actions, where the agent receives a penalty at each step and a large reward for reaching the goal state. The agent's learning is guided by Q-learning, which updates the action-value function using the Bellman equation:

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R(S,A) + \gamma \max_{A'} Q(S',A') - Q(S,A) \right)$$

where α is the learning rate and γ is the discount factor. In addition to Q-learning, Prioritized Sweeping is employed to improve learning efficiency. This method focuses planning on state-action pairs with high temporal difference (TD) errors, calculated as:

$$\delta(S, A) = \left| R(S, A) + \gamma \max_{A'} Q(S', A') - Q(S, A) \right|$$

The model is updated with the transitions and rewards, and the Q-values are refined through planning based on the priorities in a priority queue. After learning, the policy is evaluated by running the agent in the environment and computing the total reward achieved over several episodes.

4 Pseudocode for Methods

4.1 Q-learning Update

Algorithm 3 Q-learning Update Rule

1: **Input:** Current state S, action A, reward $R(S, A)$, next state S'
2: **Output:** Updated Q-value $Q(S, A)$
3: Update Q-value using the Bellman equation:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R(S, A) + \gamma \max_{A'} Q(S', A') - Q(S, A) \right)$$

4.2 Prioritized Sweeping Planning

4.3 Policy Evaluation

5 Hyperparameter Tuning

The performance of the Prioritized Sweeping algorithm depends on several key hyperparameters, including the learning rate (α), discount factor (γ), epsilon decay rate, and the number of planning steps (n) for Prioritized Sweeping.

5.1 Learning Rate (α)

The learning rate controls the rate at which Q-values are updated. We tested values between 0.01 and 0.5, and settled on $\alpha = 0.1$ as it provided a good balance between stability and learning speed.

Algorithm 4 Prioritized Sweeping Planning

Input: Model M, Q-table Q, priority queue PQ, learning rate α, discount factor γ, planning steps n
Output: Updated Q-table after planning
for each state-action pair (S, A) in the model **do**
    Compute the temporal difference error:

$$\delta(S, A) = \left| R(S, A) + \gamma \max_{A'} Q(S', A') - Q(S, A) \right|$$

    if $\delta(S, A) > \theta$ **then**
        Add (S, A) to priority queue PQ with priority $-\delta(S, A)$
    end if
end for
for each planning step i from 1 to n **do**
    if priority queue PQ is not empty **then**
        Dequeue (S, A) from PQ
        Update Q-value using Q-learning update rule:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R(S, A) + \gamma \max_{A'} Q(S', A') - Q(S, A) \right)$$

        for each predecessor (S', A') of (S, A) **do**
            Compute the temporal difference error $\delta(S', A')$ for predecessor
            if $\delta(S', A') > \theta$ **then**
                Add (S', A') to priority queue PQ with priority $-\delta(S', A')$
            end if
        end for
    else
        Break
    end if
end for

Algorithm 5 Policy Evaluation

Input: Agent's learned Q-table Q, number of episodes
Output: Average total reward from evaluation
Initialize variable total_rewards $\leftarrow$ []
for each episode from 1 to number of episodes **do**
    Initialize state S randomly
    Initialize episode_reward $\leftarrow 0$
    while episode is not done **do**
        Choose action A based on policy $\pi(S) = \arg\max_A Q(S, A)$
        Take action A, observe reward $R(S, A)$, and next state S'
        Add reward to episode total: episode_reward $\leftarrow$ episode_reward + $R(S, A)$
        Set $S \leftarrow S'$
    end while
    Append episode_reward to total_rewards
end for
Return the average of total_rewards

5.2 Discount Factor (γ)

The discount factor determines the importance of future rewards. We experimented with values between 0.8 and 1.0, and chose $\gamma = 0.99$ to prioritize long-term rewards without compromising immediate learning.

5.3 Epsilon Decay Rate

Epsilon governs the exploration-exploitation trade-off. We started with $\epsilon = 1.0$ and used a decay rate of 0.99, which allowed sufficient exploration initially and gradually shifted towards exploitation.

5.4 Number of Planning Steps (n)

The number of planning steps, n, affects how often the agent updates its policy using the model. We tested values between 5 and 20 and selected $n = 10$, balancing efficiency and computational cost.

5.5 Summary of Final Hyperparameters

The following values were used for the final model:

- $\alpha = 0.1$, $\gamma = 0.99$
- Epsilon decay rate = 0.99
- $n = 10$

These choices provided effective learning performance in the Acrobot environment.

Experiment Setup

- **State Size:** 6 (discretized state space)
- **Target Height:** 5 (goal state)
- **Reward Structure:**
  - Penalty of -1 for each step.
  - No penalty (reward 0) when the goal state is reached.
  - Additional penalty proportional to the number of steps taken: $-0.1\times$ current steps.
- **Maximum Steps:** 50
- **Episodes:** 500

6 Learning Progress

The agent's learning progress over 500 episodes is visualized below. The reward of 0 indicates the agent successfully reached the goal state during its actions.

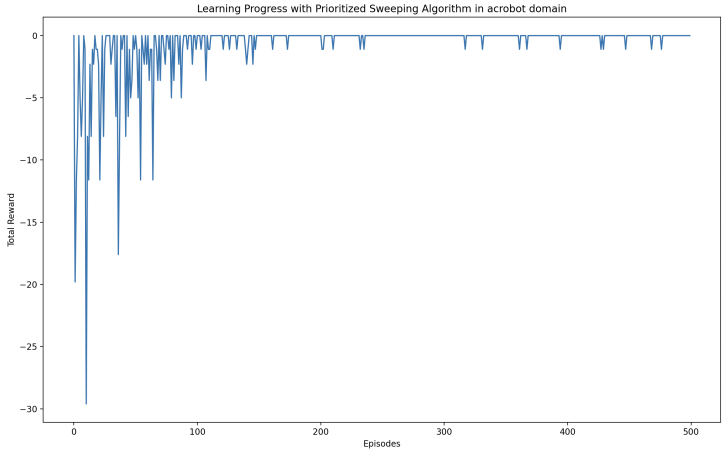

Figure 7: Learning Progress with Prioritized Sweeping Algorithm in the Acrobot domain.

7 Policy Performance

The histogram below shows the distribution of total rewards achieved by the learned policy across 100 evaluation episodes. The agent consistently reached the goal state.

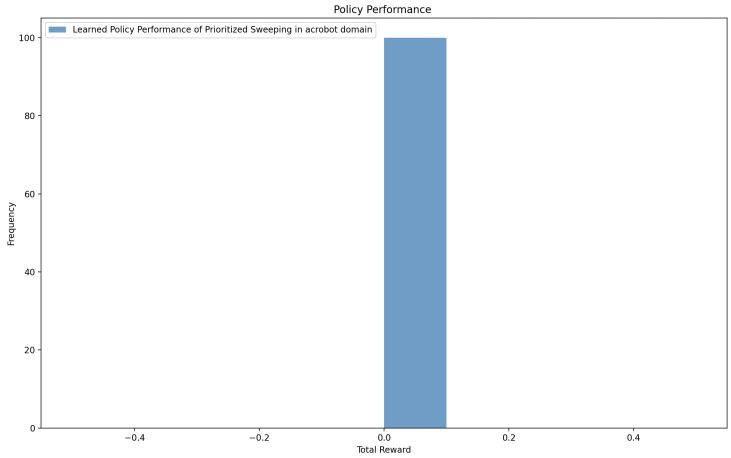

Figure 8: Learned Policy Performance in the Acrobot domain.

8 Conclusion

The Prioritized Sweeping algorithm effectively learned a policy to reach the goal state in the Acrobot environment, as indicated by the consistent reward of 0. The learning curve demonstrates rapid convergence, and the evaluation confirms the reliability of the learned policy.

Monte Carlo Tree Search (MCTS) for Cat vs Monster Problem

9 REINFORCE with Monte Carlo Tree Search (MCTS) Algorithm

9.1 Brief description of the methods

In the "Cat vs Monster" domain, the cat attempts to escape the grid while avoiding the monster. The state space includes the positions of both the cat and the monster on a grid. The game ends when:

- The cat reaches the edge of the grid and escapes,
- The monster catches the cat by moving into the same position.

MCTS consists of four main steps: Selection, Expansion, Simulation, and Backpropagation. These steps are repeated for a specified number of iterations to refine the decision-making policy.

9.2 Selection

Starting from the root node, the algorithm traverses the tree by selecting child nodes based on an exploration-exploitation trade-off. The selection strategy is

based on the Upper Confidence Bound for Trees (UCT) formula:

$$UCT(s,a) = \frac{Q(s,a)}{N(s,a)} + C\sqrt{\frac{\ln N(s)}{N(s,a)}}$$

Where: - $Q(s,a)$ is the value (e.g., reward or win rate) of taking action a from state s. - $N(s,a)$ is the number of times action a has been taken from state s. - $N(s)$ is the number of times state s has been visited. - C is the exploration constant that controls the balance between exploration and exploitation.

The child node with the highest $UCT(s,a)$ is chosen for the next step.

9.3 Expansion

Once a leaf node is reached, the algorithm expands the tree by adding a new child node corresponding to an unexplored action. This child node is added to the search tree for further exploration.

9.4 Simulation

From the expanded node, the algorithm performs a random simulation to estimate the outcome. The simulation is run by choosing random actions until a terminal state is reached (either the cat escapes or is caught).

The result of the simulation is denoted as R, where:

$$R = \begin{cases} +1 & \text{if the cat escapes} \\ -1 & \text{if the cat is caught} \end{cases}$$

9.5 Backpropagation

The result of the simulation R is propagated back to all nodes in the path from the expanded node to the root. For each node, the visit count $N(s)$ and value $Q(s)$ are updated as follows:

$$N(s) \leftarrow N(s) + 1$$
$$Q(s) \leftarrow Q(s) + \frac{R}{N(s)}$$

10 Pseudocode for MCTS algorithm in Cat vs Monster domain

Hyperparameter Tuning for MCTS Algorithm

Algorithm 6 MCTS Algorithm

Input: Initial state s_0, number of iterations N
Output: Best state after N iterations
root ← Node(s0)
for i = 1 to N **do**
    node ← root
    while node is not terminal and node is fully expanded **do**
        node ← bestChild(node)
    end while
    if node is not terminal **then**
        node ← expand(node)
    end if
    currentState ← node.state
    while currentState is not terminal **do**
        action ← random choice from legal actions of currentState
        currentState ← takeAction(currentState, action)
    end while
    result ← 1 if currentState has winner 'cat', else -1
    node ← parent of node
    while node is not null **do**
        update(node, result)
        result ← -result
        node ← parent of node
    end while
end for
return bestChild(root)

Algorithm 7 Best Child Selection

Input: node, exploration weight c
Output: best child node
for each child in node.children **do**
    score ← $\left(\frac{\text{child.value}}{\text{child.visits}+\epsilon}\right) + c \times \sqrt{\frac{\log(\text{node.visits}+1)}{\text{child.visits}+\epsilon}}$
end for
return child with maximum score

Algorithm 8 Expansion of Node

```
Input: node
Output: expanded child node
for each legal action in node.state do
    newState ← takeAction(node.state, action)
    if newState is not already in node.children then
        childNode ← Node(newState, parent=node)
        add childNode to node.children
        return childNode
    end if
end for
return null
```

Algorithm 9 State Update

```
Input: node, result
Output: updated node
node.visits ← node.visits + 1
node.value ← node.value + result
```

11 Hyperparameter Tuning for MCTS Algorithm

In the Monte Carlo Tree Search (MCTS) implementation for the Cat vs Monster game, we optimized the agent's performance by tuning several hyperparameters: the number of MCTS iterations, the exploration weight (c), and the number of training episodes. The number of iterations was tested with values of 100, 500, and 1000, with 500 providing a good balance between decision-making and computational cost. The exploration weight (c) was tested at 0.1 (higher exploitation), 1.0 (balanced), and 2.0 (higher exploration), with $c = 1.0$ leading to the best performance.

Lastly, the number of training episodes was varied between 100, 200, and 500, and 200 episodes were selected as the optimal number, as performance improvements diminished beyond this value. The final selected hyperparameters were 500 MCTS iterations, an exploration weight of 1.0, and 200 training episodes, providing a good trade-off between performance and efficiency.

12 Experimental Results for MCTS Algorithm

12.1 MCTS Learning Progress

The following graph shows the learning progress of the MCTS agent over the training episodes. It tracks the improvement in the agent's performance as training progresses.

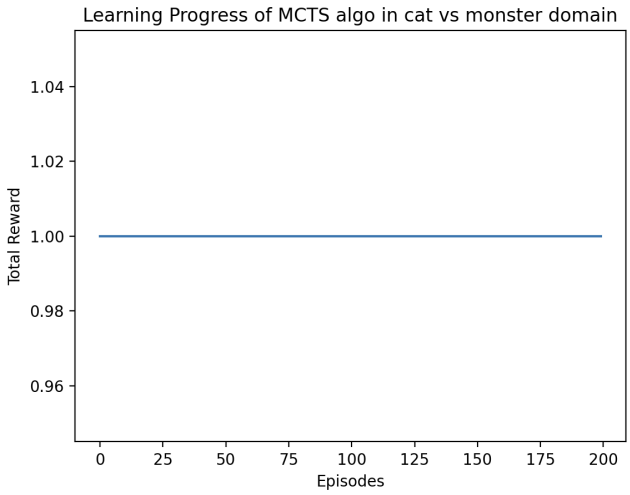

Figure 9: MCTS Learning Progress

12.2 Policy Performance of MCTS

The following graph illustrates the policy performance of the MCTS algorithm in the Cat vs Monster domain. This shows the agent's win rate and total reward across different episodes of training.

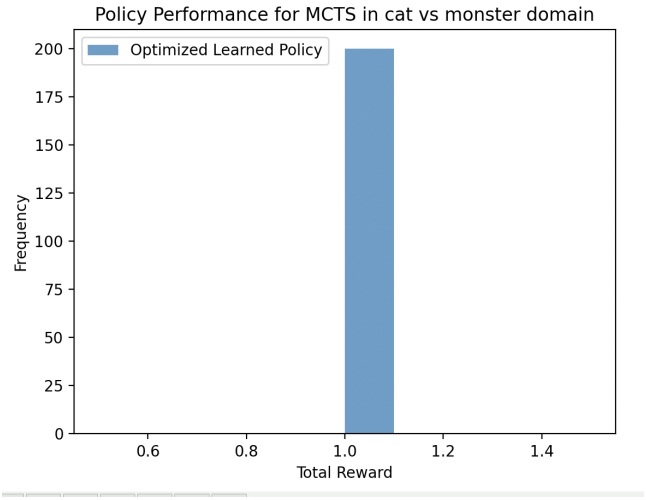

Figure 10: Policy Performance of MCTS in Cat vs Monster Domain