

---

# REINFORCE WITH BASELINE & MONTE-CARLO TREE SEARCH

---

ON CARPOLE, CS687-GRIDWORLD & MOUNTAIN CAR AND MULTI-ARMED BANDITS



**Jay Sinha**

University of Massachusetts, Amherst  
jsinha@umass.edu



**Saishradha Mohanty**

University of Massachusetts, Amherst  
saishradhamo@umass.edu

December, 2023

## ABSTRACT

We present implementations of REINFORCE with Baseline and Monte-Carlo Tree Search algorithms on three MDPs: Cartpole, CS687-Gridworld and Mountain Car. For extra-credits, we have implemented a yet unexplored MDP: Mountain Car and we present different algorithms: Epsilon Greedy, Epsilon Decreasing Greedy, Upper Confidence Bound (UCB) and Thompson sampling performance analysis on multi-armed bandits.

## 1 MDPs

### 1.1 Cartpole

The Cartpole problem [Barto et al. \[1983\]](#) is a finite-horizon MDP RL environment which simulates a pole balanced on a movable cart. The pole is joined via an un-actuated joint which allows free rotation. The goal is to apply horizontal forces (+10/ - 10 Newton depending on the direction) on the cart to keep the pole upright for as long as possible.

In this setup, the cart moves along a finite frictionless track, while the pole's angle and tipping rate dictate the cart's motions prevent it from falling. The agent can thus push the cart left or right in discrete time steps, observing the pole angle and velocities to decide next moves. The overall aim is to stabilize the pole upright through actions 'Left' or 'Right' at each time step.

The observation space tracks critical state variables - cart position and velocity, pole angle and tipping rate. Reward of +1 is provided every time step for keeping the balance stable. Episodes terminate if the pole falls beyond 12 degrees or if the cart reaches the track edges, capped at 500 steps max.

The Cartpole problem encapsulates the exploration-exploitation trade-off within an environment where dynamics are unknown and effects are delayed. Successfully addressing this challenge demands the ability to contemplate how current actions influence future stability by adapting to changing observations.

**Implementation:** We have implemented the Cartpole problem as outlined in Homework 2. For the Reinforce with Baseline algorithm's stopping condition, we have set the running average to be 480 (out of 500 maximum possible reward).

### 1.2 CS687-Gridworld

Gridworld is an indefinite-horizon MDP and is a 5x5 grid where each state represents the agent's coordinates. The agent can take four actions: AttemptUp, AttemptDown, AttemptLeft, and AttemptRight. This stochastic MDP involves probabilities for successful movements, slight veering, temporary breaks, and constraints like walls, obstacles in (2, 2) and (3, 2), a Water state at (4, 2), and a Goal state at (4, 4). The agent cannot enter obstacle states, and hitting walls or obstacles keeps the agent in its current state. The task is to compute the optimal policy for navigating this complex grid-based scenario. If the agent transitions to water state, it gets a reward of -10 whereas if it transitions into the Goal State, it gets a reward of +10. For any other transitions, it receives a reward of 0.

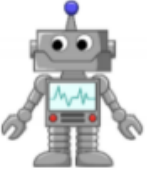

<b>Start</b> State 1	State 2	State 3	State 4	State 5
State 6		State 8	State 9	State 10
State 11	State 12	Obstacle	State 13	State 14
State 15	State 16	Obstacle	State 17	State 18
State 19	State 20		State 22	<b>End</b> State 23

Figure 1: CS687-Gridworld MDP

**Implementation:** We have implemented the gridworld problem as outlined in Homework 3. For the Reinforce with Baseline algorithm's stopping condition, we have set the running average to be 9.975 (out of 10 maximum possible reward). We have chosen to set 1000 max steps as a timeout for gridworld to avoid running into indefinite episode steps.

### 1.3 Mountain Car

The Continuous Mountain Car problem [Barto et al. \[1983\]](#) is a continuous-state finite MDP RL environment which simulates a car situated in a valley between two hills. The car, lacking sufficient power, aims to reach a higher hilltop by maneuvering against gravity where the car's throttle control determines the car's movement.

The car's state space involves continuous variables such as position and velocity, affecting the car's actions. The car can apply throttle actions, pushing in the direction of the positive or negative throttle, with a continuous range of force (+1/ - 1 Newton). The goal is to navigate the car to reach the higher hilltop by appropriately applying throttle control while overcoming the force of gravity.

Each time step provides a reward of -1, with an additional reward (100) upon reaching the goal state (hilltop). However, episodes conclude if the car reaches the goal or after a predefined maximum step limit. The MDP also terminates after 1000 steps which makes this a definite-horizon MDP.

Addressing the Continuous Mountain Car problem challenges the agent to learn a continuous policy and navigate the environment efficiently. This involves adapting to continuous action spaces, understanding the long-term consequences of throttle control, and comprehending delayed effects on reaching the goal state, all for achieving the objective of reaching the hilltop.

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

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

Loop forever (for each episode):  
   Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot|\cdot, \theta)$   
   Loop for each step of the episode  $t = 0, 1, \dots, T-1$ :

$(G_t)$

$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$   
 $\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$   
 $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$   
 $\theta \leftarrow \theta + \alpha^{\theta} \gamma^t \delta \nabla \ln \pi(A_t | S_t, \theta)$

Figure 2: REINFORCE with Baseline Pseudo-code

**Implementation:** We have implemented Mountain Car as outlined in OpenAI Gym’s [Continuous Mountain Car](#). For the Reinforce with Baseline algorithm’s stopping condition, we have set the running average to be 90 (out of 100 maximum possible reward).

## 2 Algorithms

### 2.1 Reinforce with Baseline

#### 2.1.1 Description

Reinforce with Baseline is an extension of the Reinforce algorithm which aims to reduce gradient estimate variance. In the Reinforce algorithm, the policy network parameters get updated proportional to the policy’s log-probability of taken actions multiplied by the resultant discounted reward from an episode. However, this gradient estimate can have high variance. Like the Reinforce algorithm, Reinforce with Baseline also samples trajectories and then updates the policy network’s parameters to select a better action given a state.

Reinforce with Baseline introduces a baseline function that subtracts a state-dependent baseline: the value function from the returns before multiplying by the log-probabilities. This difference alone does not change the expected gradient value, but can significantly reduce its variance. Both Reinforce and Reinforce with Baseline are examples of Monte Carlo Policy Gradient algorithms.

#### 2.1.2 Implementation

**Implementation for Cartpole:** We have implemented a Neural Network with one hidden layer with 128 hidden units. The network takes the state as an input and outputs the probability of actions and the value of the state. We have separate optimizers for the policy head and value head in the network. The hyperparameters of the network and the optimal hyperparameter search process are as follows:

- $\gamma$ : Value for  $\gamma$  is set to be 0.99.
- **Number of Hidden Units:** For this, we simply selected 128 without any search for the optimal value.
- **Optimizer:** For this, we selected among Adam & AdamW and found Adam to be optimal.
- **Learning Rate:** At first, we tried the optimal hyperparameter setting ratio recommended in the RL book between the policy and the value network of  $10^3$  but found that to be very unstable and non-converging. After this we decided to set the learning rate as the same. Then we started a grid-search in the space of  $[e^{-2}, e^{-3}, e^{-4}, e^{-5}]$ . For values of  $e^{-2}$ ,  $e^{-4}$  and  $e^{-5}$ , we did not see the parameters converging till after 1000 episodes. For the value of  $e^{-4}$ , we ran another grid-search in  $[e^{-3}, 2e^{-3}, 5e^{-3}, 8e^{-3}]$  and selected  $5e^{-3}$  as our final learning rate.

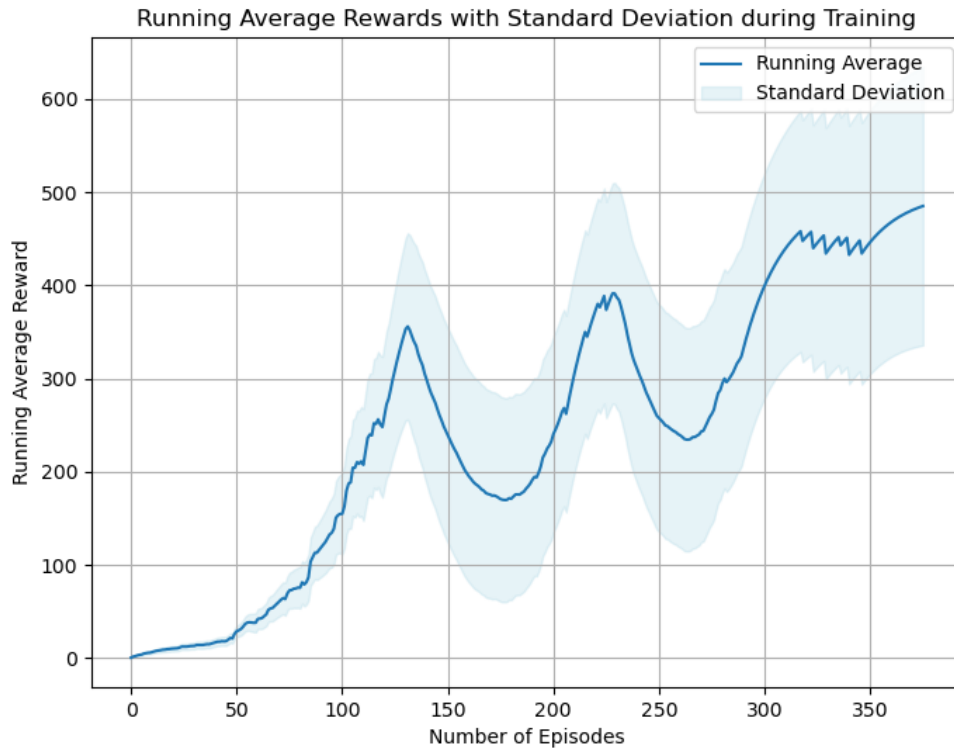


Figure 3: REINFORCE with Baseline during Training for Cartpole

**Implementation for CS687-Gridworld:** We have implemented the same network for the gridworld too with the same set of hyperparameters.

**Implementation for Mountain Car:** We have implemented the same network for Mountain Car too with the same set of hyperparameters.

### 2.1.3 Results

#### Cartpole:

- For Cartpole, the algorithm converges to the set 480 running average reward value after 376 episodes.
- After recognizing the trained model, we run 50 episodes in evaluation mode. The average reward obtained is 500 which is the maximum possible reward as shown in Figure 6.

#### CS687-Gridworld:

- For gridworld, the algorithm converges to the set 9.975 running average reward value after 404 episodes.
- After recognizing the trained model, we run 50 episodes in evaluation mode. The average reward obtained is 10 which is the maximum possible reward as shown in Figure 7.

#### Mountain Car:

- For Mountain Car, the algorithm converges to the set 92.721 running average reward value after around 2000 episodes.
- After recognizing the trained model, we run 50 episodes in evaluation mode. The average reward obtained is 92.922 which is the maximum possible reward as shown in Figure 8.

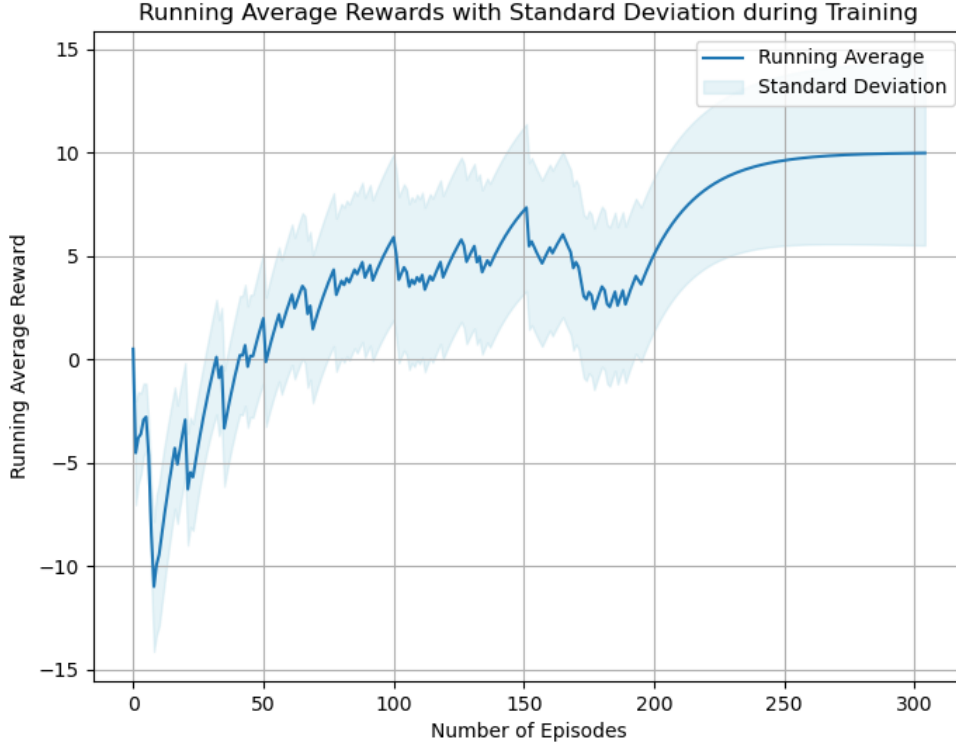


Figure 4: REINFORCE with Baseline during Training for Gridworld

## 2.2 Monte-Carlo Tree Search

### 2.2.1 Description

Monte Carlo Tree Search (MCTS) is a decision-making algorithm employed in scenarios with complex state spaces. Unlike conventional search algorithms, MCTS samples trajectories or paths within a tree structure, gradually expanding and refining the search space to identify optimal decisions.

The algorithm comprises four primary steps: selection, expansion, simulation, and backpropagation. During selection, MCTS traverses the tree to balance exploration and exploitation. It then expands the tree by adding nodes to simulate possible future states. Subsequently, it conducts simulations or rollouts to evaluate potential outcomes, followed by updating node statistics to refine decision-making through backpropagation.

A key strength of MCTS lies in its ability to iteratively update its search tree by employing the results of simulated play-outs, effectively guiding decision-making in complex domains. MCTS's adaptive nature allows it to navigate vast state spaces, making it a powerful approach for strategic decision-making in games, planning, and various AI applications.

### 2.2.2 Implementation

**Implementation for Cartpole:** We have implemented the Monte Carlo Tree Search algorithm from scratch. This algorithm operates by considering the best-selected node from the Monte Carlo tree generated during exploration, primarily based on the best action available. In our implementation of MCTS, we have not utilized any specialized optimization techniques to enhance the algorithm's efficiency. The fine-tuning of the hyperparameters for this algorithm was achieved through manual search and experimentation. The hyperparameters of the algorithm are as follows:

- $\gamma$  : Value for  $\gamma$  is set to 0.9.

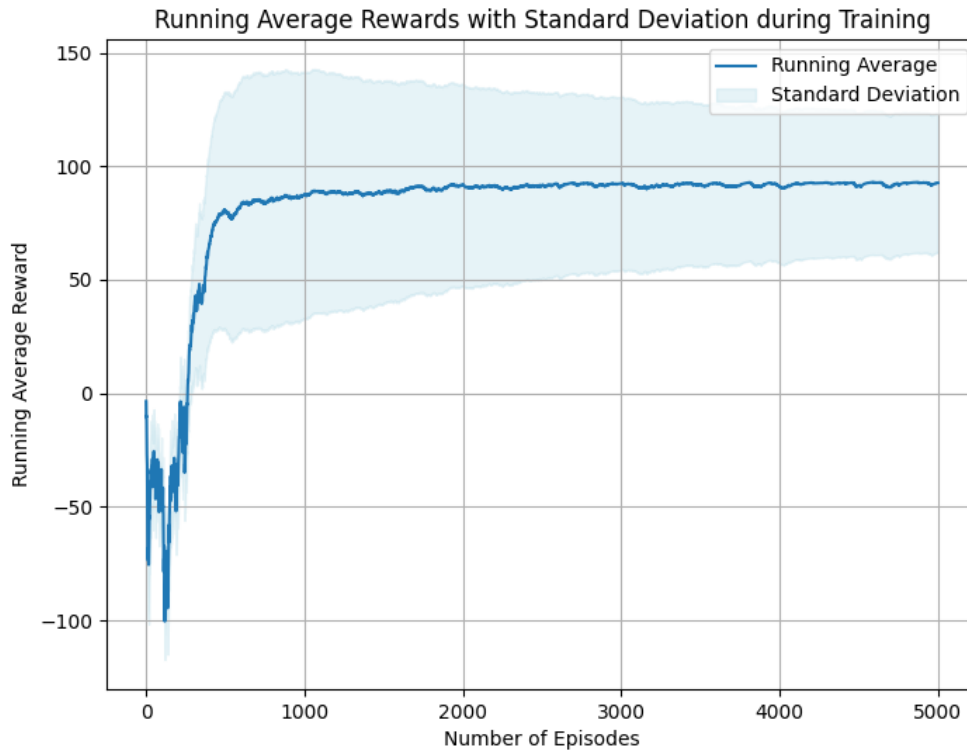


Figure 5: REINFORCE with Baseline during Training for Mountain Car

- **Rollouts:** This parameter dictates the quantity of Monte-Carlo simulations executed on each chosen node throughout exploration to approximate the value of the node or state. We hypothesized that increasing the number of rollouts would enhance the accuracy of value estimates for each node. To validate this assumption, we conducted experiments with varying numbers of rollouts within the range of [10, 20, 30, 40, 50]. Consequently, we settled that employing 50 rollouts, paired with 1000 iterations per episode, formed an effective combination for policy learning.
- **c:** This parameter regulates the level of exploration while selecting nodes during the selection phase of MCTS. It contributes to the Upper Confidence Bound (UCB) score, which aids in choosing the subsequent best node or state. Our exploration involved a manual search within the range [0.2, 0.4, 0.6, 0.8]. Lower values of 'c' resulted in excessive exploration, leading to a deep tree and sluggish convergence. Therefore, we settled on  $c = 0.8$  to strike a balance between exploration and exploitation.

**Implementation for CS687-Gridworld:** We have implemented the MCTS algorithm for the gridworld too with different number of rollouts.

- **Rollouts:** Similar to that in the previous MDPs, we conducted experiments with the same space of rollouts. Surprisingly, we found that utilizing 20 rollouts, in combination with 1000 iterations per episode, was adequate for value estimation. Even with 50 rollouts, the behavior remained the same.

**Implementation for Mountain Car:** We have implemented the MCTS algorithm for Mountain Car with different number of rollouts.

- **Rollouts:** Similar to that in the previous MDPs, we conducted experiments with the varying space of rollouts [5, 10, 15]. This time 15 rollouts were found to be best for value estimation in combination with 1000 iterations per episode.

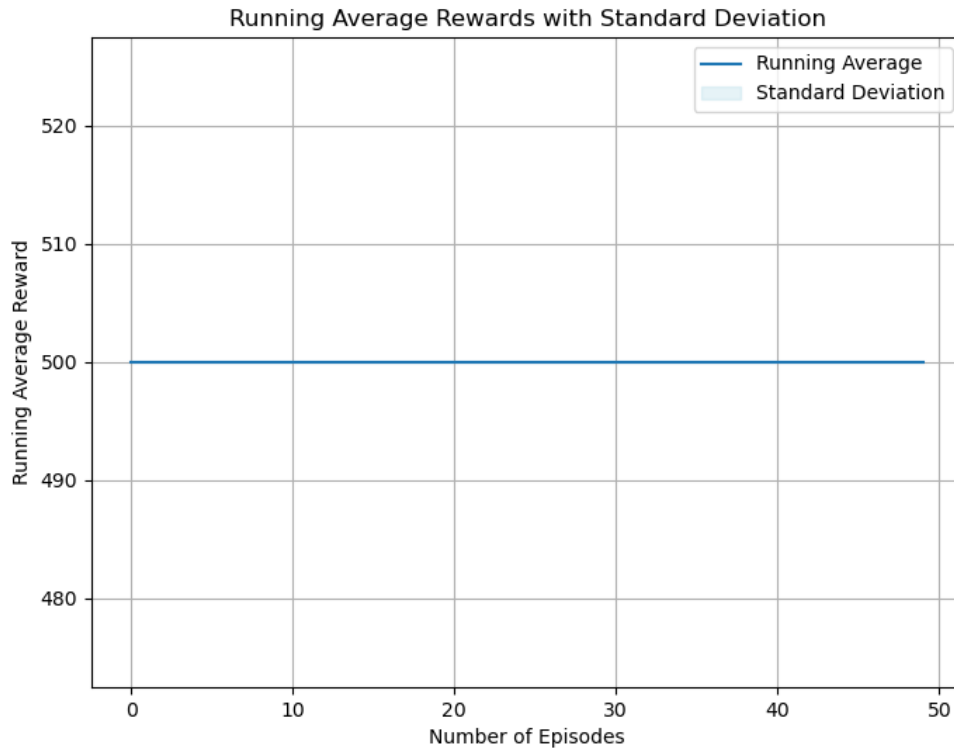


Figure 6: REINFORCE with Baseline Test Episodes for Cartpole

### 2.2.3 Results

#### Cartpole:

- For Cartpole, the algorithm converges to the set 486.2 running average reward value after 256 episodes as shown in Figure 10.
- The orange line is the running average and the blue line is the true reward returned per episode.

#### CS687-Gridworld:

- For CS687-Gridworld, the algorithm fails to converge using MCTS even after experimenting with a range of set of episodes, iterations per episode and rollouts per iteration. The unconverged learning curve is shown in Figure 11.
- The orange line is the running average and the blue line is the true reward returned per episode.

#### Mountain Car:

- We ran our analysis for 10 episodes on MCTS and observed the rewards outline in Figure 12.
- While we used our own implementation of mountain car, we found it to be very time-consuming to run MCTS given the number of rollout and iterations which limited our experiments to a smaller value of rollout.
- Additionally, we swapped with OpenAI's Gym implementation of Mountain Car and it was still time-consuming to run MCTS with rollouts higher than 15.
- The orange line is the running average and the blue line is the true reward returned per episode.

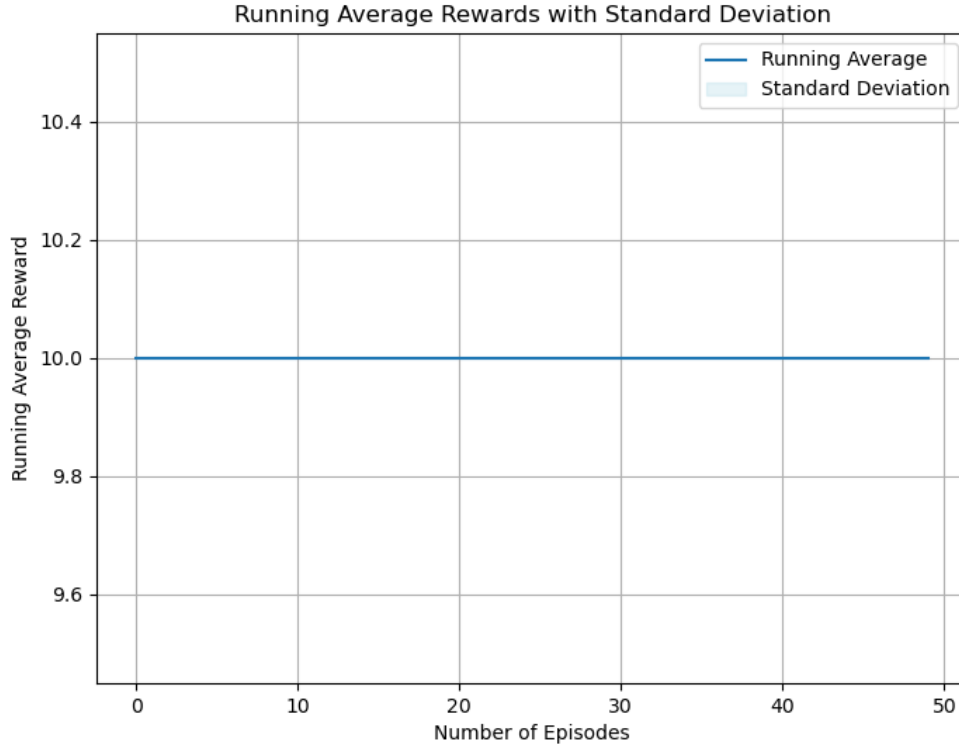


Figure 7: REINFORCE with Baseline Test Episodes for Gridworld

### 3 Extra Credit

#### 3.1 Multi-Armed Bandits

The multi armed bandit problem considers facing multiple slot machines (arms) with unknown, random reward rates across. The overall objective is to gain the maximum rewards under a given time limit with sequential pulls. Since, the knowledge of the reward distribution is unknown, the user can try different arms randomly at start (explore) and then start selecting the arm that has the highest possible reward (exploit). However, every time the user chooses to explore, it has to take into account the loss taken for not exploiting the best known arm. This is called exploration-exploitation trade-off.

The overall aim of algorithms meant for MAB is to strike a balance between exploiting the seemingly best machine based on current statistics and exploring other machines to gather more accurate estimations of their reward probabilities. This is the essence of the exploration-exploitation trade-off inherent in the multi-armed bandit problem. Algorithms designed for multi-armed bandits seek to navigate this trade-off efficiently. They adapt strategies that gradually shift from exploration to exploitation as more information is gathered, aiming to optimize cumulative rewards over repeated actions, despite the uncertainty surrounding each machine's true reward probability.

**Implementation:** For implementation, we have chosen a Bernoulli Bandit with 10 arms. Basically, each arm  $i$  has a probability  $Pr_i$  with which it will give a reward of 1. In the bandit, we keep track of the best arm. For reproduction purposes, we have set seed as 42 such that the reader can replicate the plots and results we have shown in this report. The highest rewarding arm is Arm 2 with probability = 0.9507 and we set  $N$  as 2000 which means we will make 2000 pulls.

We apply three algorithms mentioned in the RL book -  $\epsilon$ -Greedy,  $\epsilon$ -Decreasing Greedy and Upper Confidence Bound (UCB). We have also implemented Thompson sampling as it was very interesting of a concept.



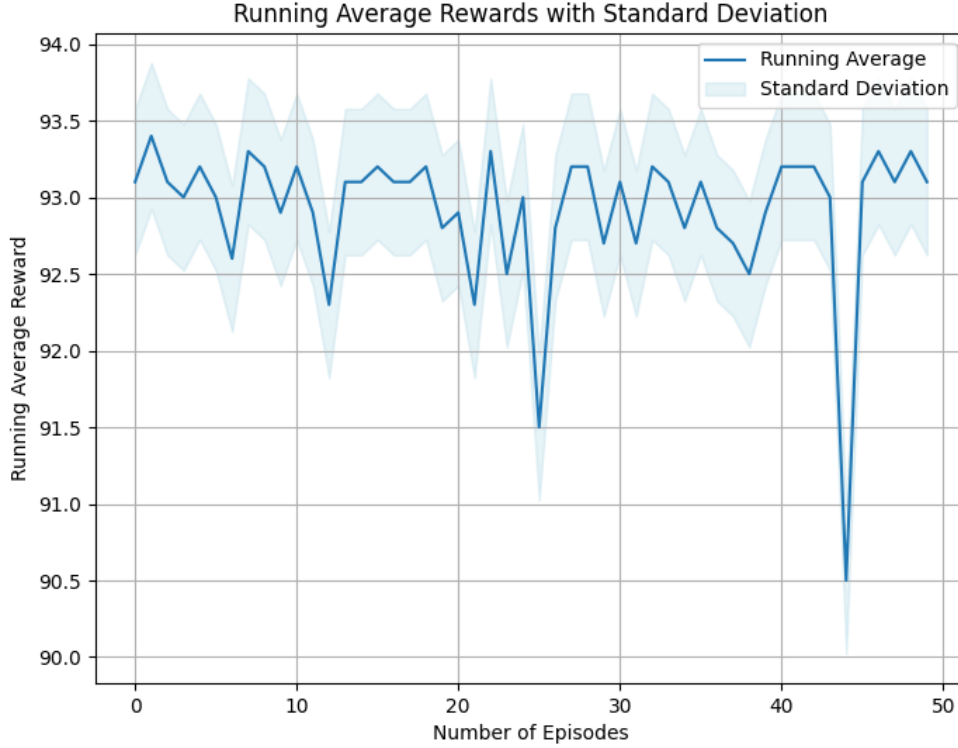


Figure 8: REINFORCE with Baseline Test Episodes for Mountain Car

### 3.1.1 $\epsilon$ -Greedy Algorithm

Epsilon greedy is a very fundamental algorithm for the exploration-exploitation trade-off in multi-armed bandits where it operates by allowing the user to set a parameter,  $\epsilon$ , prior to implementation. This epsilon denotes the probability of exploration versus exploitation. Basically, the algorithm at a given time step, with  $1 - \epsilon$  probability, chooses the arm with the highest average payoff yet and with  $\epsilon$  probability, it chooses an arm uniformly at random which helps in exploration. By tuning  $\epsilon$ , it balances leveraging arms that appear best so far, against gathering more data on uncertain options even if some pull squander payouts. By maintaining this balance, this algorithm aims to optimize cumulative rewards over a given period of time steps, leveraging both known successful actions and exploration to discover potentially better ones. However, determining the appropriate epsilon value poses a challenge, as setting it too high or too low can lead to either overlooking the best machine or wasting turns on under-performing options.

**Implementation:** For implementation, we have selected the value of  $\epsilon$  as 0.1. We experimented with  $\epsilon$  values: [0.01, 0.1, 0.25, 0.5] and found this to be the best value.

### 3.1.2 $\epsilon$ -Decreasing Greedy Algorithm

The  $\epsilon$ -decreasing greedy algorithm offers a simple extension to the previously discussed Epsilon-greedy algorithm. Intuitively, we introduce a decay rate  $d$ , which decreases the value of epsilon over time, prioritizing more exploration at the starting time steps and prioritizing more exploitation in the later time steps. As arm statistics get refined after more time steps, reliance on the realized best choice increases to maximize exploitation gains. However, choice of value for epsilon and decay rate still remain as a major decision factor.

**Implementation:** For implementation, we have selected the value of  $\epsilon$  as 0.1 and decay rate at 0.95. We experimented with decay rate values: [0.99, 0.95, 0.9, 0.8] and found this to be the best value.

---

**Algorithm 2** The UCT algorithm.

---

```

function UCTSEARCH( $s_0$ )
  create root node  $v_0$  with state  $s_0$ 
  while within computational budget do
     $v_l \leftarrow \text{TREEPOLICY}(v_0)$ 
     $\Delta \leftarrow \text{DEFAULTPOLICY}(s(v_l))$ 
    BACKUP( $v_l, \Delta$ )
  return  $a(\text{BESTCHILD}(v_0, 0))$ 

function TREEPOLICY( $v$ )
  while  $v$  is nonterminal do
    if  $v$  not fully expanded then
      return EXPAND( $v$ )
    else
       $v \leftarrow \text{BESTCHILD}(v, Cp)$ 
  return  $v$ 

function EXPAND( $v$ )
  choose  $a \in$  untried actions from  $A(s(v))$ 
  add a new child  $v'$  to  $v$ 
    with  $s(v') = f(s(v), a)$ 
    and  $a(v') = a$ 
  return  $v'$ 

function BESTCHILD( $v, c$ )
  return  $\arg \max_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c \sqrt{\frac{2 \ln N(v)}{N(v')}}$ 

function DEFAULTPOLICY( $s$ )
  while  $s$  is non-terminal do
    choose  $a \in A(s)$  uniformly at random
     $s \leftarrow f(s, a)$ 
  return reward for state  $s$ 

function BACKUP( $v, \Delta$ )
  while  $v$  is not null do
     $N(v) \leftarrow N(v) + 1$ 
     $Q(v) \leftarrow Q(v) + \Delta(v, p)$ 
     $v \leftarrow \text{parent of } v$ 

```

Figure 9: Monte Carlo Tree Search with UCT Pseudo-code



Figure 10: MCTS Learning Episodes for Cartpole

### 3.1.3 Upper Confidence Bound (UCB) Algorithm

Upper Confidence Bound or UCB algorithm calculates an upper confidence bound for each arm based on its history of rewards and the number of times it has been pulled. At the core, it selects at each time step the arm that maximizes this estimated upper confidence bound. Unlike Epsilon Greedy or Epsilon Decreasing Greedy, UCB dynamically adjusts the exploration-exploitation trade-off by prioritizing the exploitation of potentially rewarding arms while considering uncertainties in their estimated means. The selection of an arm in UCB follows a formula that incorporates the estimated value of an action (based on past rewards), a term accounting for uncertainty or variance in these estimates, and a parameter  $c > 0$  controlling the degree of exploration. This equation emphasizes the trade-off between exploiting known rewarding actions and exploring actions that might offer even better rewards.

UCB algorithm selects action  $A$  at time  $t$  such that:

$$A_t = \arg \max_a \left[ Q_t(a) + c \sqrt{\frac{\ln(t)}{N_t(a)}} \right]$$

where  $Q_t(a)$  is the estimated q-value of arm/action  $a$ ,  $t$  is the total number of pulls we have made till now,  $N_t(a)$  is the number of pulls we have made on arm  $a$  and  $c$  is an exploration hyper-parameter.

Table 1: Total Rewards from Algorithms

Algorithm	Total Rewards
$\epsilon$ -Greedy	1478
$\epsilon$ -Decreasing Greedy	1903
UCB	1605
Thompson Sampling	1875

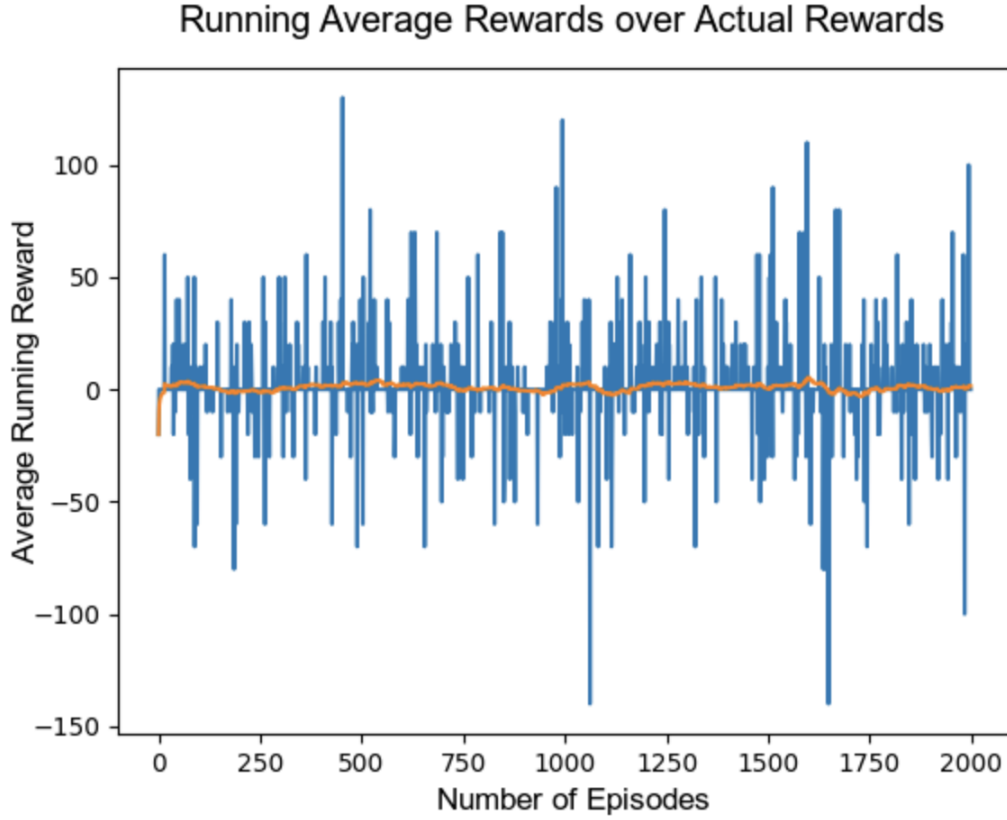


Figure 11: MCTS Learning Episodes for CS687-Gridworld

**Implementation:** For implementation, we have selected the value of  $c$  as 2. This is the standard value of the exploration parameter recommended in the text book.

### 3.1.4 Thompson Sampling Algorithm

While UCB algorithm focuses on reducing the uncertainty associated with each arm's reward value, Thompson sampling [Russo et al. \[2020\]](#) is a probabilistic algorithm that leverages probability matching to estimate the true value distribution of each arm. This algorithm also dynamically adapts the exploration-exploitation trade-off. It uses Bayesian inference to continuously update values about the distribution of rewards for each arm. By sampling from these distributions and selecting the machine with the highest sampled value, it combines exploration (sampling from uncertain distributions) and exploitation (favoring machines with higher estimated rewards), dynamically adjusting as more data is gathered. Through repeated time steps, the estimated value distribution converges towards the true values, while the variance of these distributions diminishes.

**Implementation:** For Bernoulli Bandit, we use the Beta distribution as the  $Q_t(a)$  function for arm  $a$  is the success probability of a Bernoulli distribution. We use the value  $\alpha$  and  $\beta$  for the Beta distribution as 1. At any given time step  $t$ , let's say we select the action  $a$ . If selecting action  $a$  gives reward of 1, then we increase the value of  $\alpha$  by 1 and if it gives a reward of 0, we increase the value of  $\beta$  by 1. At any given point, we can obtain the probability estimate by using the values of  $\alpha$  and  $\beta$  by doing:

$$\Pr(R_t = 1 | A_t = a) = \frac{\alpha_a}{\alpha_a + \beta_a}$$

At any time step  $t$ , we select the arm  $a$  by:

$$A_t = \arg \max_a [\Pr(R_t = 1 | A_t = a)]$$

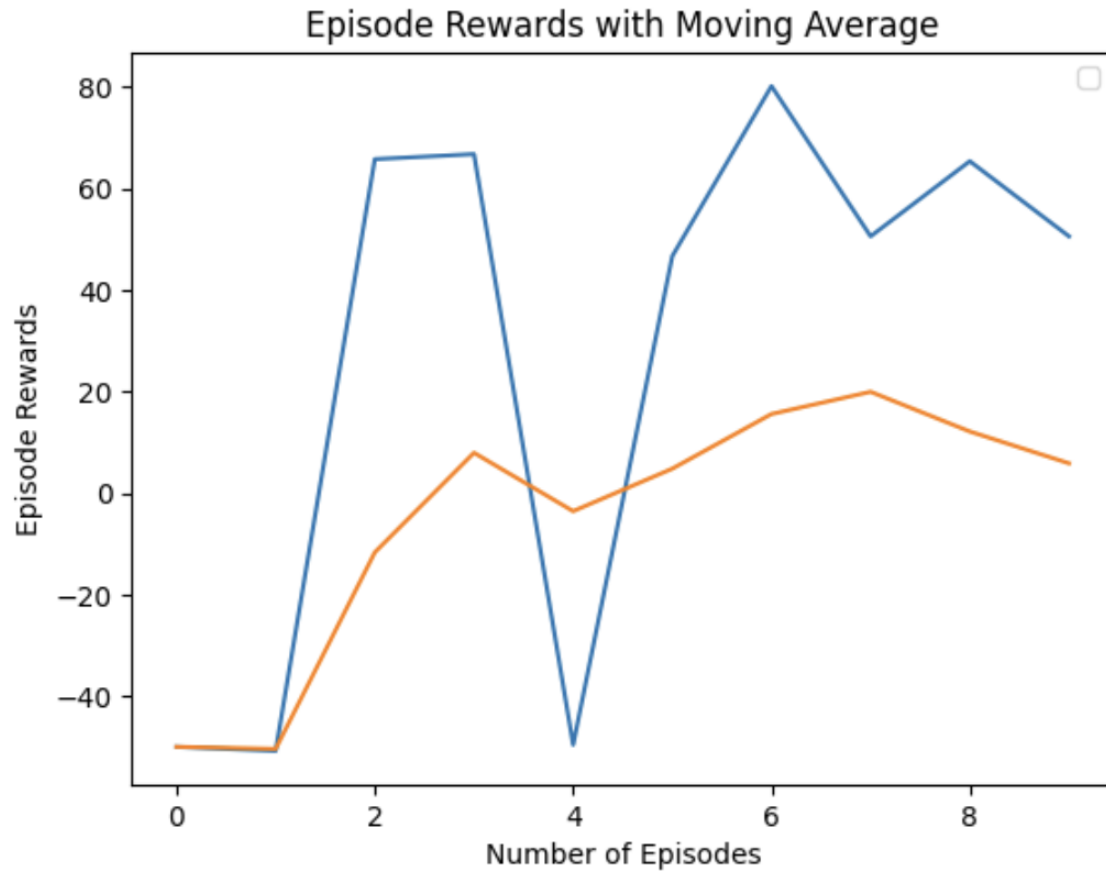


Figure 12: MCTS Learning Episodes for Mountain Car

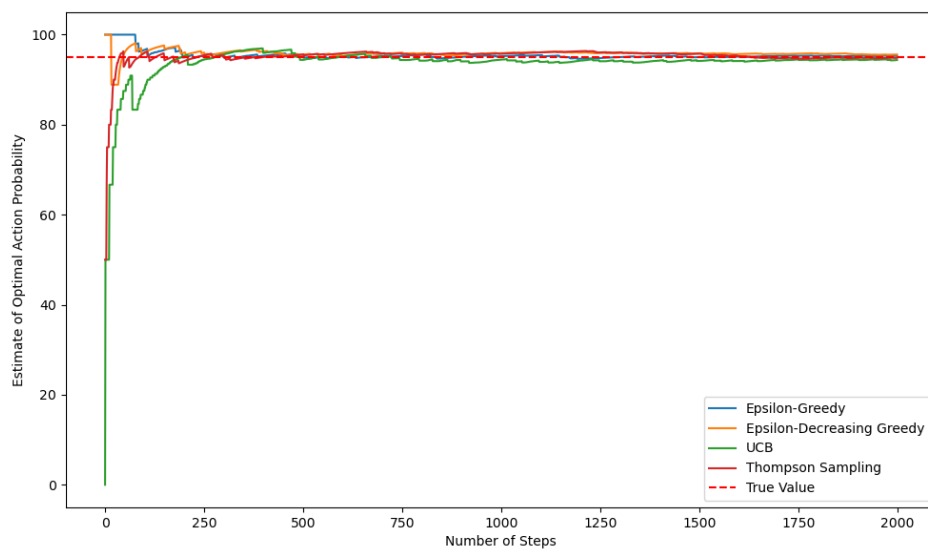


Figure 13: % Optimal Arm

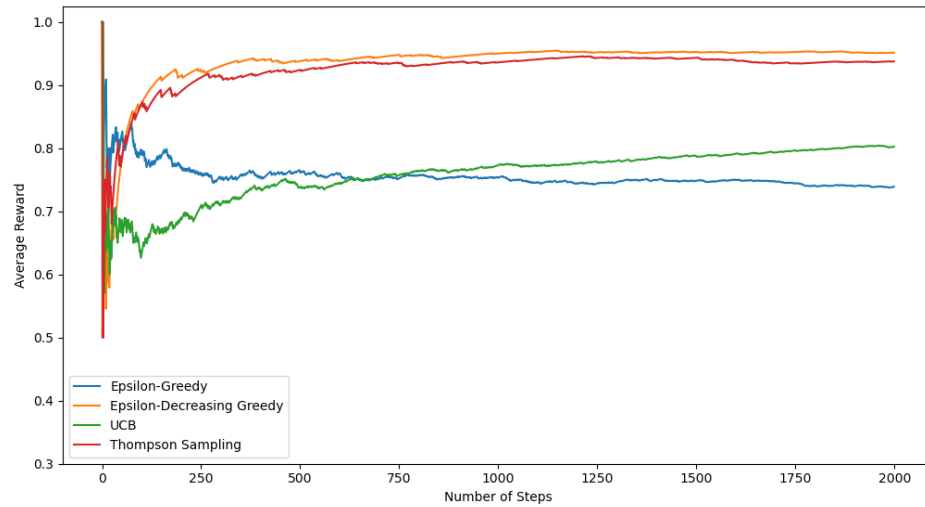


Figure 14: Average Rewards

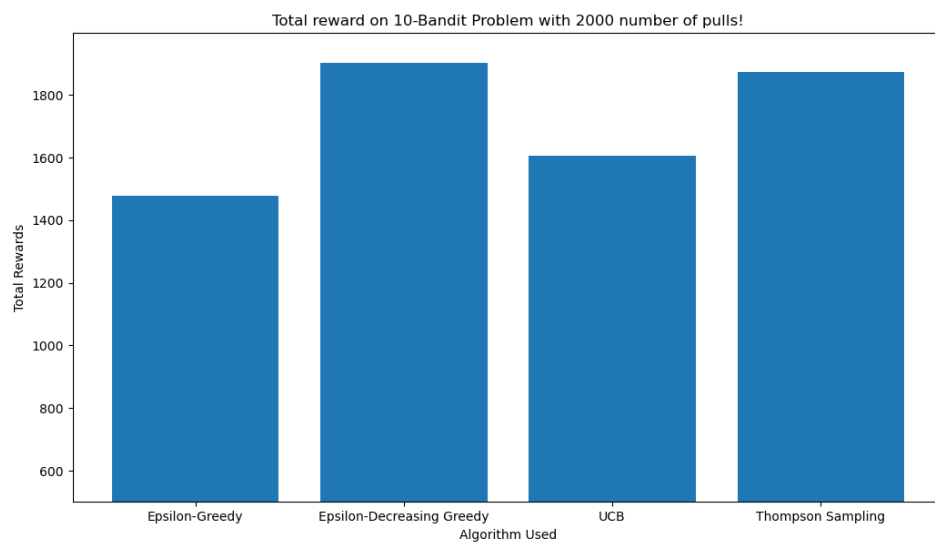


Figure 15: Total Rewards at the end of 2000 pulls.

## References

Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi:[10.1109/TSMC.1983.6313077](https://doi.org/10.1109/TSMC.1983.6313077).

Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A tutorial on thompson sampling, 2020.