COMPSCI 687: Reinforcement Learning Final Project

Samuel Kovaly
Riddho Ridwanul Haque

December 9, 2021

Abstract

For this project, we primarily went with option #2 as described in our Final Project description. Our goal was to implement RL algorithms that were not covered in class and evaluate them on existing MDPs. To this end, we implemented One-Step Actor Critic, Proximal Policy Optimization (PPO) and Monte Carlo Tree Search (MCTS) on the 687-Gridworld and Mountain Car domains. Furthermore, we implemented different exploration strategies on the 687-Gridworld domain for an agent to learn the model of the world, and tested how Value Iteration would perform on the learned model in lieu of the actual one. Finally, we implemented a new environment, which we call the 'Meal Planner' domain, that models the problem of solving package queries from Database Tables, a research interest of the DREAM Lab at UMass Amherst, as a Markovian Decision Process (MDP) and used the Value Iteration Algorithm to form optimal meal plan packages.

1 Introduction

In this project, our goal was to explore the domains and applications of reinforcement learning beyond what was already covered in class. To do so, we learnt, implement, tuned and evaluated three RL algorithms:

1. The One-Step Actor-Critic Algorithm
2. The Proximal Policy Optimization Algorithm (PPO)
3. The Monte Carlo Tree Search Algorithm (MCTS)

All three of these algorithms were experimented on two different environments, which were previously described in class:

1. The 687-Gridworld Domain: A domain in which the agent starts from the top-left corner of a grid world and aims to move to the bottom right goal state while avoiding obstacles and water states in its path. The transitions of the agent in the environment are stochastic, meaning an action can lead to multiple outcomes. All attributes of the environment were kept exactly as described in the classes during the execution of all three algorithms.

2. The Mountain Car Domain: A continuous environment that models the struggles of a car that aims to generate enough power to escape from a valley in a mountainous region. All attributes of the environment were kept as is during the execution of all algorithms except for the Monte Carlo Tree Search, for which the start position, instead of being stochastic, was fixed to -0.5.

In addition to implementing these algorithms, we also experimented the following approaches which involve learning the model of the world and using MDP-s to model research problems in domains outside of reinforcement learning:

1. Learning 687-Gridworld: Assuming the agent does not have a model of the environment, we experimented on methods by which the agent can learn the model through exploration first, and then use the learnt model to find optimal policies using the Value Iteration algorithm.

2. The Meal Planner Environment: Consider the case of an aspiring athlete who wants to plan what meals she can take in the upcoming few days. To ensure her fitness, she wants to keep her consumption of total calories consumed within a fixed bound, and minimize the amount of saturated fat she takes. This problem exemplifies a wider class of problems in Data Management

research known as Package Queries [1], where the goal is to extract a set of tuples from a database table where the total 'package' of tuples returned must adhere to certain global constraints (for example, the total calories consumed if the athlete takes all the meals in the package must be within 800-1000 calories) and must be optimal with respect to certain objectives (here, the total fat content in the meal plan should be minimized). The athlete may also give 'repetition constraints' to ensure that the same meal is not repeated more times than she desires in the package. This may be enforced to ensure diversity in the meals she takes. While package queries have traditionally been solved using a mix of integer linear programming and divide and conquer approaches, we model the problem as a Markovian Decision Process (MDP) in this project. Details on our formulation and implementation of the environment are given in Section 4.

The remainder of this write-up is organized as follows. Section 2 details the contributions of each member, Section 3 describes the new algorithms that we implemented, Section 4 details our newly formed Meal Planner Environment, Section 5 shows the results of our experiments and our choice of hyperparameter settings for each algorithm, and Section 6 contains our conclusive remarks.

2 Contributions

The project is the result of a joint contribution by Samuel Kovaly and Riddho Ridwanul Haque. Samuel implemented, tuned and executed the One-Step Actor Critic and Proximal Policy Optimization Approaches on both the 687-Gridworld and the Mountain Car domains, while Riddho did the same for Monte-Carlo Tree Search. Riddho also implemented and experimented on the model-learning approaches and formulated and implemented the Meal Planner Domain. The same pattern was followed in this writeup, with each individual writing the sections they primarily contributed to.

3 Algorithms Used

In this section, we provide a brief overview of the RL approaches we used in our project.

3.1 Policy Gradient Algorithms

The Policy Gradient Theorem $\frac{\delta J(\theta)}{\delta\theta} = \sum_s d^\theta(s) \sum_a \frac{\delta\pi(s,a_i,\theta)}{\delta\theta} q^{\pi_\theta}(s,a)$ had been the groundwork for many successful recent RL algorithms and it has become the standard in deep reinforcement learning and AI in the context of games. I implemented two popular Policy Gradient algorithms: the One Step Actor Critic and a one step variant of Proximal Policy Optimization.

For both, I used the softmax policy with a single θ parameter matrix along with a linear regression state value function that uses a separate w vector for it's parameters. For both algorithms I used the standard one step temporal difference error signal with my value function for bootstrapping.

I evaluated the algorithms with some simple metrics. In general I evaluated them on the basis of actions per episode. This means that as the agent learns through experience, it should take less time (less actions) to finish episodes. For some of the experiments, I searched over hyper parameters and used a table to store the results and for some I used plots of actions over episodes. The score I use for the tables is the "final mean performance" which is the average number of actions needed to complete an episode (reach the goal). I only included the episodes near the end of the agent's learning process in this metric since it is a "final performance metric", so it should not include all the episodes used in training.

I evaluated them on the Gridworld and Mountain Car domains. For the Gridworld I used the last 20 episodes of it's learning process to evaluate it's performance (actions per episode) and for the Mountain Car I used it's last 50 episodes. My goal was to get these values as low as possible. I did not consider how much time it took for the two algorithms to converge as I implemented them differently and there could be bias in my code as far as computation speed. First I start with my evaluation of the One Step Actor Critic.

3.2 One-Step Actor Critic

For One Step Actor Critic, I implemented, using Numpy, what was discussed in class. Namely I Implemented the softmax policy $\pi(s, a_i, \theta) = \frac{\exp(\theta_{a_i}^T\ \phi(s))q}{\sum_{a'}\exp(\theta_{a'}^T\ \phi(s))}$ where $\frac{\delta \ln(\pi(s,a_i,\theta))}{\delta\theta_{a_i}} = [1 - \pi(s, a_i, \theta)]\phi(s)$ and $\frac{\delta \ln(\pi(s,a_i,\theta))}{\delta\theta_{a_j}} = -\pi(s, a_i, \theta)\phi(s)$ for $a_j \neq a_i$. θ then was $|A|$ x *state_features* in size where *state_features* refers to the vector length of my state features $\phi(s)$. Each mdp used different state features which I will discuss later. For learning, I used $\theta \leftarrow \theta + \alpha^\theta \delta_t \Delta \ln(\pi(s, a_i, \theta))$ as seen in the One Step Actor Critic pseudo code.

The value function was a simply linear regression on my state features $\phi(s)$. Specifically it is characterized by: $\hat{v}_w(s) = w^T\phi(s) = \sum_{i=1}^d w_i\phi_i(s)$. To learn, I used: $w \leftarrow w + \alpha^w \delta_t \Delta \hat{v}(s, a)$ as seen in the One Step Actor Critic pseudo code.

3.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO), a policy gradient algorithm by Open AI [4], formulates the problem as having a certain objective function that needs maximization. This loss is given by:

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k(a|s)}} A^{\pi_{\theta_k}}(s, a), \mathit{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k(a|s)}}, 1 - \epsilon, 1 + \epsilon\right) A^{\pi_{\theta_k}}(s, a)\right)$$

This uses many ideas. The first is that stochastic gradient ascent is used to maximize this objective function for some number of epochs after seeing a sufficient number of training examples. The examples come from the agent interacting with it's environment. Instead of updating the policy and value function after every action, PPO saves the collected data and updates it in a mini-batch manner using this objective function.

Another important idea here is importance sampling given by $\frac{\pi_\theta(a|s)}{\pi_{\theta_k(a|s)}}$ where the idea is that the distribution of action, state and reward data collected using the old policy π_{θ_k} is not the same as the distribution of them during the learning steps. The current policy is updated multiple times with this old data which can cause diverging behavior. Thus important sampling uses the ratio between the two probabilities to make updates in a better way.

Most importantly is PPO's novel contribution, which is their clip functionality. This clipping makes sure the policy does not radically change from the old policy during the stochastic gradient ascent phase. If the policy changes too much, then the clip ensures the gradient becomes 0 for that state and action. Changing the policy too radically during stochastic gradient ascent can cause diverging behavior that the policy might never recover from. PPO introduces three new hyper parameters, the clip ϵ, how many timesteps between learning T and the number of learning epochs K.

I implemented PPO using Numpy. I used the same vector w parameters for my value function and θ for my softmax policy as before in my One Step Actor Critic formulation. I also did not change the state features $\phi(s)$ for either mdp. The only differences thus are the use of batch updates and the new loss formulation. For the Advantage value that PPO needs, I again went with the one step td error that is calculated from the value function.

For the PPO specific hyper parameters, I formulated the T timesteps between learning as *rollout_episodes* where learning could only happen between episodes. I kept this value set to 1 since episodes contained many training examples in their own right. For the clip, I set it to 0.2 as the PPO paper showed that this was a good value to use.

3.4 Monte Carlo Tree Search

The Monte Carlo Tree Search Algorithm aims to build a search tree rooted at the starting state, and build up the tree incrementally based on the agent's interactions with the world. Starting from the root, the algorithm executes the following four steps in sequence:

- **Selection:** A path is charted out from the root to a leaf by picking the child with the highest value of UCB1 (upper confidence bound applied on trees)[2]. UCB1 is determined as follows:

$$UCB1 = \bar{X}_i + C \times \sqrt{\frac{2 \times log(N)}{n}} \tag{1}$$

Here, $\bar{X}_i$ represents the average of the simulations rolled out from a node or one of the nodes in its subtree, N represents the number of simulations rolled out from the parent and n represents the number of simulations rolled out from the child. C is a hyperparameter, increasing which increases exploration and vice-versa.

- **Expansion:** The leaf is expanded, and the possible actions that can arise from it lead to different children. In my implementation, I kept two types of nodes in the tree-state nodes and action nodes, where the state nodes are parent to the action nodes, and the action nodes are parent to the state nodes. When an action node is expanded, it is done on the basis of interacting with the environment and observing the next state based on the action represented by the action node, and the state represented by its parent node. If the environment takes the agent to a previously unseen state, a new child is added to the action node.
- **Rollout:** Rollouts are random monte-carlo simulations run from one of the children created during the expansion step,
- **Backpropagation:** The return from the rollout is then backpropagated along all the nodes from the leaf node where the rollout began to the root.

Each iteration of MCTS computes these four steps in sequence, and the pseudocode of this approach is given in Algorithm 1.

Algorithm 1 Monte_Carlo_Tree_Search

Input An environment E, number of iterations N, control parameter C

```
1: rootNode ← Create node based on starting state
2: for i ← {1, 2, 3, ..., N} do
3:     leaf ← selection(rootNode, C)
4:     child ← expand(leaf)
5:     value ← rollout(child, E)
6:     backpropagate(child, value)
7: end for
```

3.5 Value Iteration Algorithm

The value iteration algorithm was taught in class, and the exact same version of the algorithm was applied on the learned model of the 687-Gridworld and the Meal Planner Environment. Since the algorithm was covered in class and in our previous assignments, we do not describe it in detail here.

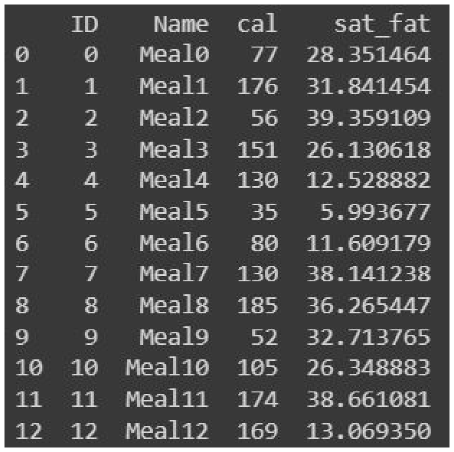
4 The Meal Planner Environment

Referring to the Meal Planning scenario for the athlete described in the introduction, given a database table consisting of many meals, with attributes referring to the calories consumed if the athlete chooses each meal, and the saturated fat content within that meal, suppose the athlete requires a package of meals-such that the sum of the calories within the meals included in the package is between 800 and 1000 calories, the same meal is not repeated more than p times, and the sum of the saturated fat content of the meals included in the package is minimized. Referring to the package query syntax specified in [1], the athlete can specify a Package Query for her requirements as follows:

```
SELECT PACKAGE(*) AS P
FROM MEALS M REPEAT 1
SUCH THAT SUM(P.cal) BETWEEN 2000 AND 2500
MINIMIZE SUM(P.sat_fat)
```

In this query, the maximum number of repetitions is set to 1, meaning that a meal can be repeated at most once, that is, be included at most twice in the package. While the user is free to change the specifics of the query according to their requirements, we use this query to demonstrate how the problem of evaluating this query can be solved using an MDP.

For reference as to what the MEAL table looked like in our dataset, consider the snapshot given in Figure 1. In our experiments, for simplicity of implementation, the amount of calories in each meal was considered to have integer values. The table had a total of 50 tuples, with the data randomly generated. Let us suppose U(x, y) is a random number generator which generates a random integer with uniform probability distribution from the range [x, y]. The 'cal' column of the table was generated using the random generator $U(50, 200) - U(0, 1) * U(0, 20) + U(0, 1) * U(0, 20)$. The fat content was set randomly by picking a continuous value from the range [5, 50].



	ID	Name	cal	sat_fat
0	0	Meal0	77	28.351464
1	1	Meal1	176	31.841454
2	2	Meal2	56	39.359109
3	3	Meal3	151	26.130618
4	4	Meal4	130	12.528882
5	5	Meal5	35	5.993677
6	6	Meal6	80	11.609179
7	7	Meal7	130	38.141238
8	8	Meal8	185	36.265447
9	9	Meal9	52	32.713765
10	10	Meal10	105	26.348883
11	11	Meal11	174	38.661081
12	12	Meal12	169	13.069350

Figure 1: Snapshot of the MEALS Table

The MDP was implemented using the following specifications:

- **State:** There are two state variables, *pos*, representing the index of the meal for which an action will been taken in that state, and *rem_cal*, representing the further amount of calories that can be taken into the package.

- **Action:** An action corresponds to the act of not including a meal in a package, or including the meal in the package 1 or more times in the package respectively. Given that the number of repetitions is constrained to p, as specified in a query, an action represents a number in the set $\{0, 1, 2, ..., 1 + p\}$, representing the number of times a meal is included in the package.

- **Transition:** Transitions in the environment are deterministic, if the meal at index *pos* and calorie amount c is included a times in a package, then, in the next state, an action will have to be taken for the immediate next meal-meal numbered $pos + 1$, and the remaining amount of calories will reduce by $a \times c$, becoming $rem\_cal - a \times c$.

$$p((pos, rem\_cal), a) = (pos + 1, rem\_cal - a \times c) \tag{2}$$

- **Starting State:** In the starting state, an action will be chosen for the first meal (numbered 0), and the amount of calories allowed to be consumed further is 1000. Thus, the starting state is (0, 1000).

- **Terminal State:** Decision making terminates if an action has been chosen for all meals. Since there were 50 tuples in the dataset in our case, this is represented by our state variables if $pos = 50$. We also terminate if the value of *rem_cal* becomes negative, indicating that the actions taken have results in a prohibitive amount of calorie intake.

- **Rewards:** If a meal with fat content f is taken a times, the reward given is equal to $-a \times f$. Since the agent will seek to maximize the reward, the negative factor will lead it to minimize the total fat intake. To ensure that the total calorie intake is constrained within 1000, if a transition leads to a state with negative *rem_cal*, this indicates that the calorie intake is higher than 1000, and very low reward of -10000000 (in other words, a very high punishment) is given. If $pos = 50$, that is, actions has been taken for all meals, if the *rem_cal* is over 200, indicating that less than 800 calories have been consumed, the same low reward of -10000000 is given. If $pos = 50$ and the *rem_cal* is within [0, 200], indicating an acceptable amount of calories have been consumed due to the actions taken on the meals, a reward of 0 is given.

- **Discount Factor:** Since there's no succession of meals here, we are choosing which meals to include, and not which ones to take first or later, we do not have any discount factor in our environment. In other words, the discount factor of our environment γ is set to 1.0.

The Meal Planner domain was implemented in Python 3, with a Meal Planner class returning the starting states, terminal states, rewards and transitions according to the definitions specified above. The environment consisted of a total of 50,000 states and 1+p actions, where p, representing the number of repetitions of a meal allowed, was empirically varied from 0 to 4.

Since the model was known in advance, I used the Value Iteration Algorithm to find the optimal policy. To construct the package from the optimal policy, a trajectory starting from the starting state to a terminal one was charted out using the actions specified by the optimal policy. All meals for which the optimal action in the trajectory was a non-zero value were included in the package and the quantity of those meals were set to be equal to the optimal action.

5 Experimental Results

5.1 Results with the One-Step Actor Critic Approach

Actor Critic on Gridworld

For the Gridworld MDP, I chose the features $\phi(s)$ to be a vector of length 25 (5x5) to represent the grid world. I filled this vector with 0s everywhere except I put a 1 for where the agent currently is located. I used the formula $y * self.state\_size\_y + x = 1$ to map the agent's location in the 2d Gridworld array to it's corresponding index in the 1d feature vector. This feature vector was given to both the actor and the critic.

On an interesting side note, this formulation is very similar to the tabular case for the value function because essentially when the regression value is calculated by: $\hat{v}_w(s) = w^T\phi(s) = \sum_{i=1}^{d} w_i\phi_i(s)$, only one weight is actually used since all but one of the state features are 0. This weight then acts as the tabular value of that state where the agent currently is located.

My first experiment was to find good learning rates for both my policy and my value function. I ran a sweep of potential values and table 1 reflects these results. For each, I ran the algorithm 25 times and took the average performance as the result. I could afford to do this since gridworld is very small and only takes a few seconds for a single itertaion. For many of the learning rates there was little difference but the values of 0.25 for the value function alpha and 0.5 for the policy alpha scored the best with a mean final performance of 10.540 and standard deviation 2.388.

α_w vs α_θ	0.05	0.1	0.25	0.5	1.0
0.05	41.730	31.583	20.383	16.107	12.930
0.1	23.017	16.437	12.600	11.363	11.727
0.25	15.967	12.860	10.753	**10.540**	11.763
0.5	15.103	12.327	11.180	10.723	11.700
1.0	15.497	12.870	11.693	20.403	23.750

Table 1: Actor Critic Gridworld Value Function vs Policy Learning Rates. The left most column represents the value function learning rates α_w while the first row represents the policy learning rates α_θ. The best results found were for $\alpha_w = 0.25$ and $\alpha_\theta = 0.5$ with a mean final performance of 10.540 and standard deviation 2.388

With these learning rates I then did 50 iterations and graphed the averages along with the standard deviation as seen in figure 2. The red line represents the most optimal time achievable (in this case, 8 actions for the Gridworld). Since the policy and value function are always changing (especially with a high learning rate), then we would expect the algorithm to never find the exact optimal policy, which is what we see here. The One Step Actor Critic only needs about 50 episodes on average to converge which is quite fast.

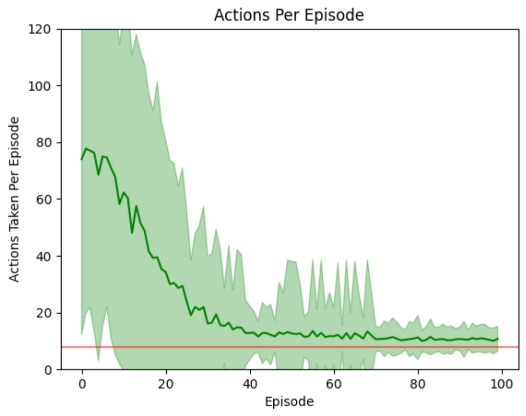


Figure 2: Actor Critic Gridworld actions over episodes with $\alpha_w = 0.25$ and $\alpha_\theta = 0.5$. Averaged over 50 runs.

Actor Critic on Mountain Car

For the Mountain Car MDP, I chose $\phi(s)$ to be the fourier basis where $\phi(s) = [1, \cos(1\pi x), \cos(2\pi x), \ldots, \cos(M\pi x), \cos(1\pi v), \cos(2\pi v), \ldots \cos(M\pi v)]^\top$. M is the order parameter which controls the complexity and length of this basis and thus it controls the size of the θ parameter matrix and the w vector.

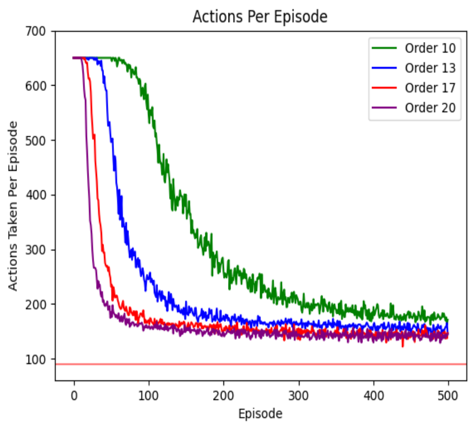
The order M in the fourier basis allows the model to better represent the mdp state and thus find a better policy. I found that the order and the learning rates were correlated in terms of performance. This means that lower learning rates needed higher order which makes sense since the higher the order, the more parameters there are to learn. Unfortunately I did not have the compute power to do a hard search over all possible learning rates and order configurations (which for example could be as high as $4\alpha_w$ x $4\alpha_\theta$ x $4M = 64$ configurations). Each hyper parameter configuration would need at least 10 iterations of the algorithm to find a statistically significant result which mean 640 iterations. Since each iteration takes anywhere from 30 seconds to a minute, on my computer, obviously I could not wait this long.

Instead I decided to traverse only 4 order values on pre-selected learning rates in order to see how the model complexity affected learning. I chose $\alpha_w = 0.0001$ and $\alpha_\theta = 0.001$. I swept over the order values [10, 13, 17, 20] with 30 iterations each and graphed them side by side as seen in figure 3a. Table 2 shows the means as well as the standard deviations. The best order has it's own separate graph (see figure 3b) where we can also see it's standard deviation over the episodes.

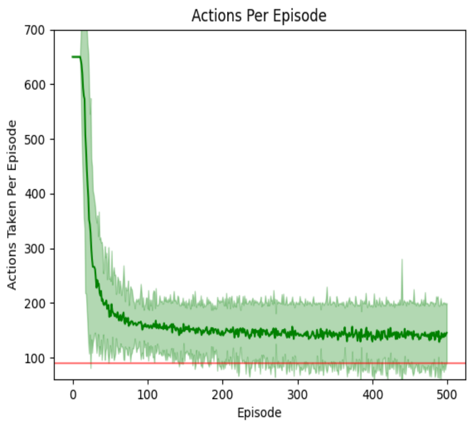
Order	Performance Mean Final	Performance Std Final
10	175.978	32.774
13	154.618	25.778
17	143.639	29.029
20	**140.558**	30.309

Table 2: Actor Critic Mountain Car Order comparison with set learning rates of $\alpha_w = 0.0001$ and $\alpha_\theta = 0.001$. Averaged over 25 runs.

Order 20 was found to be the best but not much better than order 17. This hints that complexity has diminishing returns in this context. The optimal performance of 90 actions per episode was achievable by the model but not consistently. It never seemed to learn to repeat the actions that gave it this return. With more time this phenomenon could be examined in more detail,



(a) Mountain Car Actions over Episodes for four order hyper parameters. Order 20 was found to be the best. Averaged over 25 runs.



(b) This was the top performing Actor Critic model on the Mountain Car domain. with $\alpha_w = 0.0001$, $\alpha_\theta = 0.001$ and order $= 20$. It's standard deviation is very high all throughout the training sessions.

5.2 Results with the Proximal Policy Optimization Approach

PPO on Gridworld

For PPO on Gridworld, I used an learning epoch size of 20 and then performed a sweep over possible learning rates for my models. I used 25 iterations as before and took the mean results. These are shown in table 3. I did not reuse the 1.0 learning rate as before because this caused overflow issues in my code so 0.75 was used instead.

α_w vs α_θ	0.05	0.1	0.25	0.5	0.75
0.05	66.422	61.504	53.858	44.044	40.294
0.1	59.088	50.034	29.576	17.360	13.666
0.25	38.000	18.728	11.352	10.692	**10.676**
0.5	25.748	13.766	11.076	10.774	10.832
0.75	22.012	13.000	11.072	10.796	11.270

Table 3: PPO Gridworld Value Function vs Policy Learning Rates. The best results found were for value alpha = 0.25 and policy alpha = 0.75 with a mean final performance of 10.676 and standard deviation 2.522. Interestingly, low learning rates were very destructive.

With a seemingly optimal value function alpha of 0.25 and a policy alpha of 0.75, I performed another 50 iterations and graphed the results along with the resulting standard deviation as seen in figure 4. The standard deviation starts out high but quickly settles to a low number and remains stable until the end. The mean number of actions to complete an episode is achieved a good score, although still not completely optimal. It would seem that policy gradients can't achieve perfect optimality without extreme hyper parameter tuning.

PPO on Mountain Car

For PPO on the Mountain Car domain, I again reused the same softmax policy and linear value function. I also reused the fourier basis state features. Since I found the best fourier order to be 20, I keep that value fixed in this section. There were too many hyper parameters to sweep over so I fixed the policy and value function learning rates and swept only over the learning epochs. PPO on the mountain car domain surprisingly seemed to need different learning rates than the actor critic. I chose to use $\alpha_w = 0.0.001$ and $\alpha_\theta = 0.01$ although other good options do exist. I swept over the learning epoch values [5,8,10,12] and used the same metrics as before to evaluate them. I averaged over 20 runs and the results are given in table 4 and figures 5a and 5b.

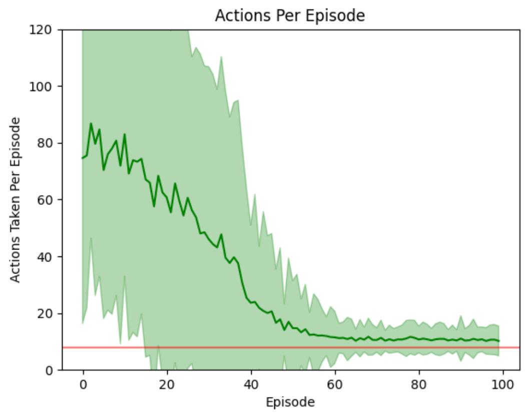
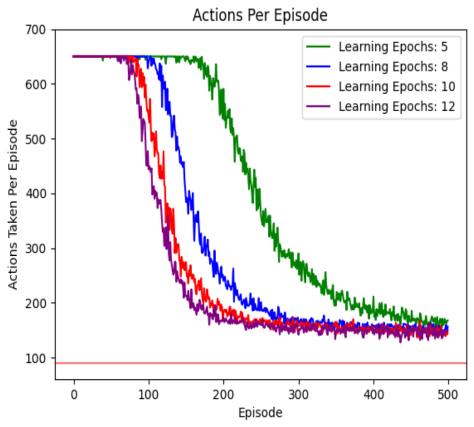


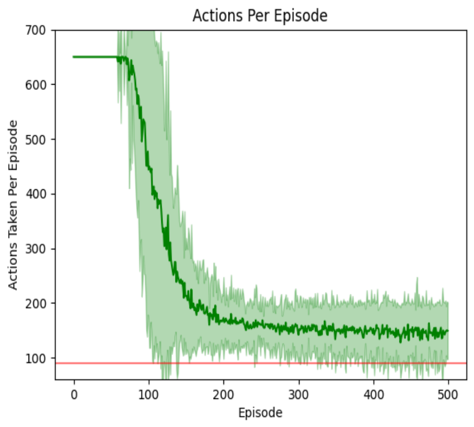
Figure 4: PPO on Gridworld best performing model with $\alpha_w = 0.25$, $\alpha_\theta = 0.75$ clip $= 0.2$, rollout episodes $= 1$ and 20 epochs. Averaged over 50 runs.

Learning Epochs	Performance Mean Final	Performance Std Final
5	166.758	24.339
8	152.824	25.778
10	147.769	26.707
12	**146.006**	28.614

Table 4: PPO Mountain Car Learning Epochs comparison with set learning rates of $\alpha_w = 0.001$ and $\alpha_\theta = 0.01$. Averaged over 20 runs.



(a) Mountain Car Actions over Episodes for four possible learning epoch values. A learning epoch value of 12 was found to be the best, although it's standard deviation was the highest. Averaged over 20 runs.

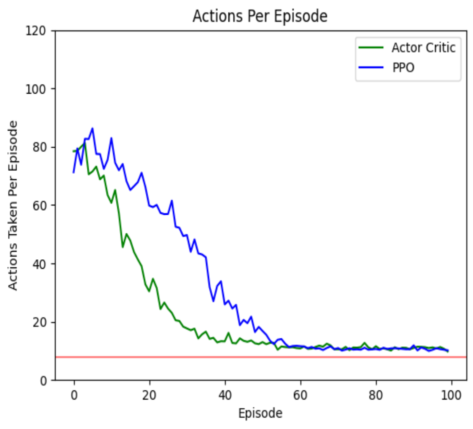


(b) This was the top performing PPO model on the Mountain Car domain. with $\alpha_w = 0.001$, $\alpha_\theta = 0.01$ and learning epochs $= 12$.

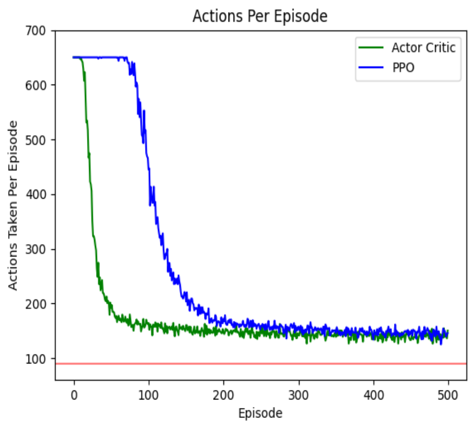
PPO on Mountain Car did not beat the Actor Critic performance of 140 but it's very possible that better hyper parameters exist for tuning on this mdp. It's unlikely that increasing the learning epochs would result in a better performance as is seen in the diminishing returns in the table. It's standard deviation was about the same as the Actor Critic model which is interesting.

Actor Critic vs PPO

In conclusion for the policy gradient algorithms, both are successful on these simple mdps but the Actor Critic algorithm achieves better results, and converges faster, than the PPO algorithm as seen in the direct comparison of these two in figures 6a and 6b. Hyper parameter selection could be the reason for this. The final results for Gridworld are: Actor Critic final performance mean: 10.893 and final performance std: 2.802 and PPO final performance mean: 10.665 and final performance std: 2.663. For the Mountain Car: Actor Critic final performance mean: 140.285 and final performance std: 29.065 and PPO final performance mean: 145.366 and final performance std: 27.859.



(a) Actor Critic vs PPO on Gridworld mdp. Averaged on 50 runs.



(b) Actor Critic vs PPO on Mountain Car mdp. Averaged over 25 runs.

5.3 Results with the Monte Carlo Tree Search Algorithm Approach

I implemented the Monte Carlo Tree Search Algorithm on both the 687-Gridworld and the Mountain Car domains. The two major tunable hyperparameters for MCTS is the value of c, which is used to control the algorithm's preference of exploitation over its tendency towards exploration, and the number of iterations it will be run for. In general, the more simulations that are run by Monte Carlo Tree Search, the better it approximates the expected returns. However, the number of iterations cannot be made arbitrarily large without paying heavy computational costs.

5.3.1 MCTS on GridWorld

MCTS worked relatively well on the gridworld domain, although the value of c needed careful tuning which relied heavily on empirical observations. The value of the number of iterations for which MCTS was run was confined to 2 million iterations, due to computational constraints. Perhaps a higher number of iterations would give even better results. The value of c was 0.1 for the trial that gave us the best results. Since the goal of MCTS is not to estimate the value function of all states, but instead to chart out an optimal sequence of actions from the starting state, to understand if the resulting simulations actually give better results as the amount of experiences increases, we plotted the relation between the number of iterations and the average of the returns accumulated at the root.

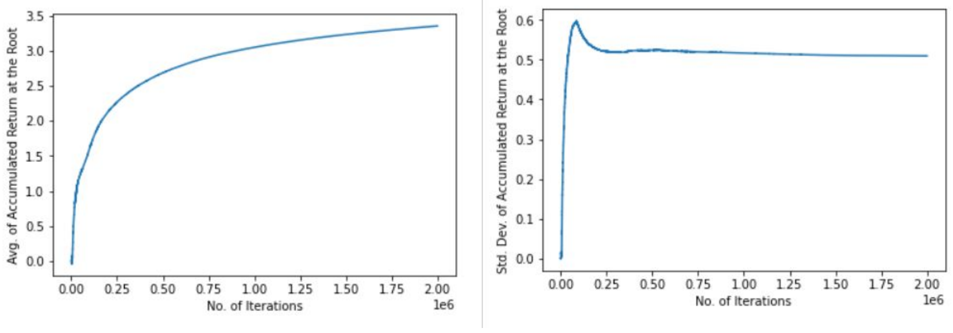


Figure 7: Learning Curve of MCTS on the Gridworld Domain (based on 5 trial runs for c=0.1)

The results of these experiments, plotted in Fig. 7 by averaging over 5 trial runs of the algorithm (a higher number of runs could not be scheduled due to compuational constraints), shows that the accumulated average indeed gets better with time, and approaches the actual expected return of 4 in the 687-Gridworld domain with an increase in the number of iterations. The standard deviation also decreases with an increasing number of iterations, showing that the algorithm slowly converges towards a true estimate of the value of the starting state.

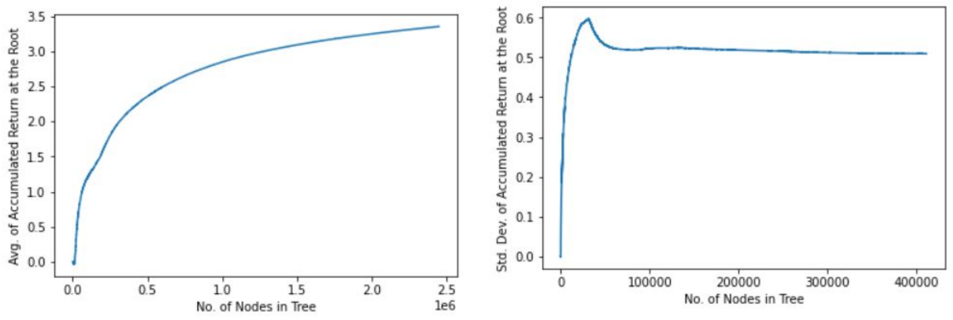


Figure 8: Change of Accumulated Return with Tree Size in the GridWorld Domain

As the number of iterations grow, so does the number of nodes in the tree due to repeated calls of the expand() procedure. A broader tree represents better knowledge of the state-action space by the agent, resulting in better estimates as seen in Fig. 8.

5.3.2 MCTS on Mountain Car

It was decidedly harder to run MCTS on the Mountain Car domain, mainly because the MCTS is designed to handle sequential decision making problems and adversarial search problems in game theory, but not necessarily continuous domains. I discretized the space into having 500 bins for both state variables, since MCTS is well-equipped for handling large state spaces. Even finer discretizations might have helped, but the number of iterations needed to train the model would be prohibitively large. Under our constraints, the best empirical results I obtained was for c=0.005 and discretization into 500 bins per state variable.

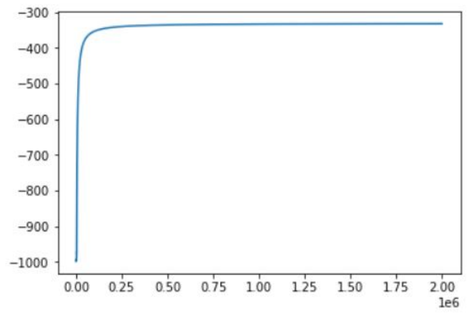


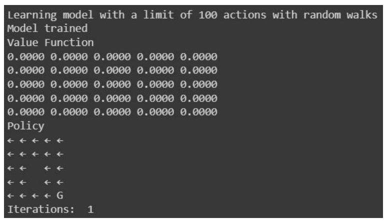
Figure 9: Learning Curve of MCTS on the Mountain Car Domain (for c=0.005), with the Y-axis denoting the average of the accumulated return at the root, and the X-Axis denoting the number of iterations

The experimental result, shown in Fig. 9, shows that the final accumulated result after 2 million iterations is -332, which is still a long way off from the optimal result of -60.

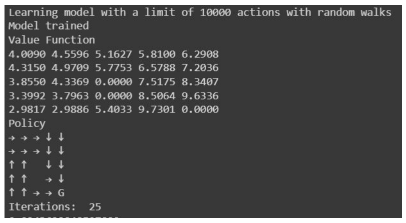
5.4 Experiments on Learning a Model of 687-Gridworld

We used three different approaches to learn a model of the 687-Gridworld, and evaluated the results after running Value Iteration on the learned models. The model learning approaches used were:

1. **Random Walks:** We placed the agent on the starting state of 687-Gridworld, and asked it to walk, i.e. take a sequence of random actions, and observe the resulting state after each action. The model was learnt based on these interactions, and the total number of steps that could be used to learn the model were kept limited during our experiments.



```
Learning model with a limit of 100 actions with random walks
Model trained
Value Function
0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000
Policy
← ← ← ← ←
← ← ← ← ←
← ←   ← ←
← ←   ← ←
← ← ← ← G
Iterations:  1
```



```
Learning model with a limit of 10000 actions with random walks
Model trained
Value Function
4.0090 4.5596 5.1627 5.8100 6.2908
4.3150 4.9709 5.7753 6.5788 7.2036
3.8550 4.3369 0.0000 7.5175 8.3407
3.3992 3.7963 0.0000 8.5064 9.6336
2.9817 2.9886 5.4033 9.7301 0.0000
Policy
→ → → ↓ ↓
→ → → ↓ ↓
↑ ↑   ↓ ↓
↑ ↑   → ↓
↑ ↑ → → G
Iterations:  25
```

Figure 10: Difference of the estimated optimal policy and value function when the model is learnt using 100 samples vs. 10000 samples

Fig. 10 shows how the policies learnt by value iteration based on a model learnt from 10,000 steps is significantly better than one learnt from just 100.

2. **Random Steps:** Instead of walking, i.e. executing a sequence of actions from the starting state, in this approach, we placed the agent on any state chosen at random and asked it to take just one step (instead of a sequence) by executing an action chosen at random.
3. **Steps in States with Min. Visits:** Instead of picking which state to place the agent in at random, we pick the state for which the minimum amount of data has been collected, and place the agent there. Instead of picking the action at random, the agent executes the action for which the least amount of data has been collected.

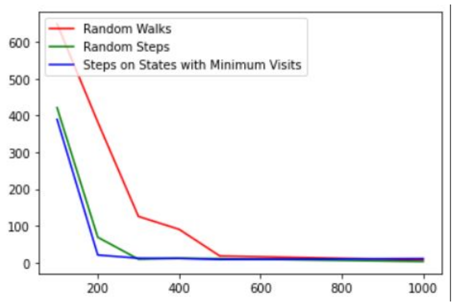
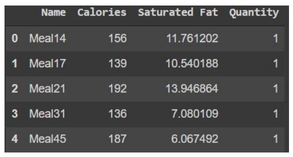


Figure 11: Mean Squared Error of Value Iteration vs The number of steps used to learn the model

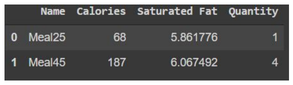
Fig. 11 shows that the three approaches get progressively more effective at quickly getting an accurate model of the environment, with the optimal value function for the learnt model quickly converging to that of the original model as the number of steps taken by the agent for learning increases. The reason for the third approach working the best among the three approaches is that it distributes the steps more uniformly among the state-action pairs.

5.5 Experiments on the Meal Planner Domain

Experiments of applying the Value Iteration algorithm on the Meal Planner domain gave interesting results, with a couple of sample meal plan packages being shown in Fig. 12.



	Name	Calories	Saturated Fat	Quantity
0	Meal14	156	11.761202	1
1	Meal17	139	10.540188	1
2	Meal21	192	13.946864	1
3	Meal31	136	7.080109	1
4	Meal45	187	6.067492	1



	Name	Calories	Saturated Fat	Quantity
0	Meal25	68	5.861776	1
1	Meal45	187	6.067492	4

Figure 12: Meal Plans Generated with 0 and 3 repetitions allowed

The value iteration algorithm converges quite quickly on this environment, as represented by the learning curves in Fig. 13.

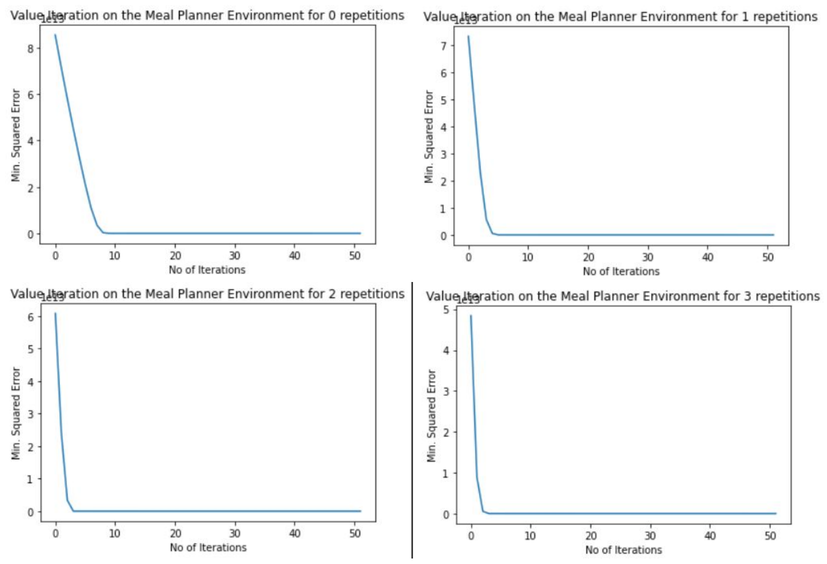


Figure 13: Learning Curves for Value Iteration with 0, 1, 2 and 3 allowable repetitions

6 Conclusion:

In this project, we have implemented three new RL algorithms on the Mountain Car and 687-Gridworld domains. We discovered that policy gradient methods are sensitive to their hyper parameters and thus require custom tuning for each mdp they are applied to. PPO for instance had too many hyper parameters to explore all of them.

Our observations show that policy gradient approaches are usually more effective in continuous domains than approaches that were built for discrete models, like the MCTS. While Ba et. al. [3] reported slightly better results on the Mountain Car domain by adding Hierarchical Optimistic Optimization on the original MCTS approach, that remains beyond the scope of this project.

Furthermore, we have implemented the value iteration algorithm on some learnt models, and we have observed that model learning can be rather effective especially when the state space is small, as seen in the case of 687-Gridworld. We have also proposed a new environment which represents a data management research problem as an MDP. While our experiments gave interesting results on the relatively small toy dataset we created for this project, scaling up these RL algorithms to work on large database tables remains an interesting challenge.

References

[1] Brucato, Matteo, et al. “Scalable package queries in relational database systems.” arXiv preprint arXiv:1512.03564 (2015).

[2] Kocsis, L., and Szepesvari, C. 2006. Bandit based Monte-Carlo planning. In European conference on machine learning, 282–293. Springer.

[3] Ba, Seydou, et al. “Monte Carlo Tree Search with Scalable Simulation Periods for Continuously Running Tasks.” arXiv preprint arXiv:1809.02378 (2018).

[4] Schulman, John, et al. ”Proximal Policy Optimization Algorithms” arXiv preprint arXiv:1707.06347 (2017).