Reinforcement Learning - Coursework 1

Constantin Eulenstein

November 2020

1 Question 1: Understanding of MDPs

1.1 Exercise 1a)

My CID is 01804832. Therefore, my trace is $\tau = s_3 1 s_2 0 s_2 0 s_2 0 s_2 0 s_1 3 s_0 2$

1.2 Exercise 1b)

According to my trace, one starts at state s_3 and collects a reward of 3 when departing from it. Afterwards one self loops in state s_2 three times, that means three times one is departing from state s_2, collecting a reward of 0 and ending up in s_2. Finally, one departs from s_2 to s_1 (collecting again a reward of 0). From s_1, one leaves for s_0, collecting a reward of 3, and ends up in s_0. Since one is not departing from s_0, one doesn't collect the reward of 2.

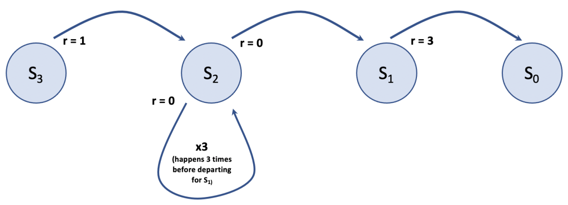


Figure 1: MDP graph for trace found in exercise 1a)

1.3 Exercise 1c)

1.: The transition matrix gives the corresponding probabilities of arriving at one state given an action (in our case only one possible action). We know that the transition matrix is of size 4x4 since there are 4 states in set S and our naive agent experienced these 4 states. As a naive RL agent, seeing only this one trace, I would infer from the data and the graph that the probability of going anywhere from state s_0 is 0 for all elements in the first row, because our trace ends in state s_0 (further discussion of state type of s_0 in Exercise 1c.3); so $p_{0i} = 0$ for i $\in [0, 1, 2, 3]$. Being in state s_1 a naive RL agent would infer from the trace that one can only end up in s_0, which is why $p_{1i} = 0$ for i $\in [1, 2, 3]$ and $p_{10} = 1$. The same holds for our first state s_3: $p_{3i} = 0$ for i $\in [0, 1, 3]$ and $p_{32} = 1$, since we go from s_3 only to s_2 in our trace. Lastly, we observe that from state s_2 we depart three times in a row and end up again in s_2, while the fourth time we depart and end up in s_1. From a frequentist approach we would therefore assume that on average it takes us four trys to leave s_2 to s_1 and, therefore $p_{21} = 1/4$, while $p_{22} = 3/4$ ($p_{20} = p_{20} = 0$). The Transition matrix is therefore $\mathbf{P} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0.25 & 0.75 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.

2.: Since we are considering a "no-choice" environment our reward function $r(s, a, s')$ becomes $r(s, s')$. While we might end up in different states with different probabilities (see s_2), there is only one action we can take. The reward function is independent of the transition matrix and is an immediate reward when departing. Also since the reward is collected whenever departing a specific state the reward functions are independent of the successor states and $r(s, s')$ becomes just $r(s)$. These rewards we can read from our trace and they should be $r(0) = 2$, $r(1) = 3$, $r(2) = 0$ and $r(3) = 1$, which corresponds to $\mathbf{R}^*$. However, a naive RL agent might argue that there can only be a reward for transitions that are actually possible (he observed) and the reward matrix becomes not $\mathbf{R}^* = \begin{bmatrix} 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$, but $\mathbf{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.

3.: The value of a state is the expected collected reward that one can get in the future, being in this state right now. The value function is the following: $V(s) = \sum_{s' \in S} P_{ss'}(R_{ss'} + \gamma V(s'))$. γ is given as 1 and we could start calculating the value at s_0 and from there calculate each predecessor. However, since we do not know whether s_0 is a transient or terminal state, we can't compute the value function for s(t=0) without taking any assumptions. Let's first assume that s_0 is a terminal state, which means s_0 has a value of 0 ($V(s_0) = 0$) since we can not depart from there and collect a reward. Then $V(s_1) = 3$ (only immediate reward), $V(s_2) = 3$ (immediate reward of 0 + γ $V(s_1)$) and $V(s_3) = 4$ (immediate reward of 1 + γ $V(s_2)$). To compute this I back-propagated the value function (bellman equation) under the assumption that the value of state s_0 is 0 because of termination. Under the assumption that s_0 is a transient state, we would of course have a different value function $V(s_3)$. Wherever we would transition to from state s_0, we would get a closed loop system or an infinite self-connection in s_0. This would basically mean, that our system can not be exited and, since we have a discount factor of 1 (no discount), the agent would be stuck in the system indefinitely and collect rewards accordingly. Consequently, under the assumption that s_0 transitions to a new state, all our states' value functions would be infinite. A third option would be that our agent only terminates in s_0 with some probability and, otherwise, departs. However, to compute the value functions under this assumption, we would have to be aware of the full transition matrix, which a naive RL agent is obviously not. A last option would be that there is a cut-off after six steps, and this is why our agent ends in state s_0.

2 Question 2: Understanding of Grid Worlds

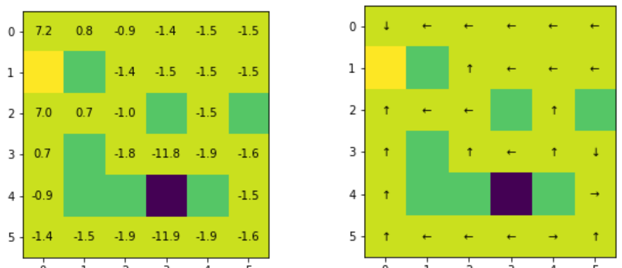
2.1 Exercise 2a)

My personal CID number is 01804832. Consequently, the reward state is s_1, $p = 0.7$ and $\gamma = 0.35$.

2.2 Exercise 2b)

1.: After building the grid (see code in appendix), I initiated a random starting policy, for which the probability to perform action a ($a \in \mathcal{A} = [N, E, S, W]$) is equal to 0.25. I used this initial uniform policy as well as a threshold of 0.001 and my gamma of 0.35, to perform the policy iteration algorithm (in code method policy_iteration_algorithm) (and the iterative policy evaluation algorithm which is implied in this algorithm). I am aware that value iteration might be computationally more efficient, but since our grid world is small, I chose this algorithm for its comprehensiveness. I used a greedy policy, which always performs the action with currently highest state action value per state. Whenever a new policy for all states is found, another method policy_evaluation performs the iterative policy evaluation algorithm. The algorithm converges towards the optimal policy, which obtains the optimal value function $V^*(s)$ for every state.

2. and 3.:



(a) Found Values (b) Found Policy

Figure 2: Found values of optimal policy and optimal policy (reward state s_1 in yellow, penalty state s_{11} in dark purple, walls in green, remaining states in light green.

4.: Since our optimal policy $\pi^*(s, a)$ for a specific state s is deterministic, our optimal value function reduces to $V^*(s) = max_{a \in \mathcal{A}} Q^*(s, a)$. The p influences our transition matrix $\mathcal{P}^a_{ss'}$, s.t. our desired action, e.g. go north, results in moving north with probability p and moving in any of the other three possible directions with probability $\frac{1-p}{3}$. In my particular grid world p is 0.7, so there is a high chance (70%) that the chosen action will actually result in performing the action. Now if we would assume $p = 0.25$, our agent would move randomly inside the grid independent of the chosen action ($max_{a \in \mathcal{A}} Q^*(s, a)$ is the same for any action a). Therefore, it would not matter which policy one chooses since the result of that action would be random due to our environment (all possible policies become optimal). For any $p > 0.25$ we will find an optimal policy, however, the lower the p, the more the algorithm will prefer safer actions over potentially faster, but unsafer, actions (e.g. prefer a successor state that has more neighbors with lower values over a successor state with one neighbor with high value and one with very low value). With a $p < 0.25$ the chosen action is actually the one with the lowest probability of completion (the other three have same, but higher, probability). Consequently, the optimal policy would always take the action that points towards the successor state with the worst value function (the worst possible successor state), s.t. the agent would actually end up there with the lowest probability.

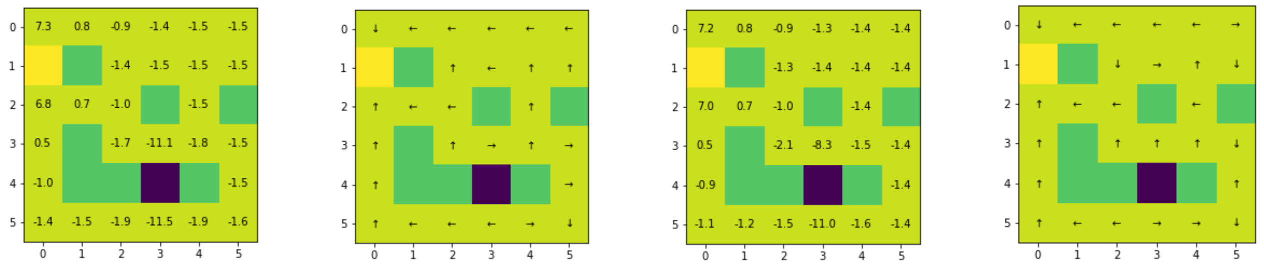
Our discount factor γ influences to what extent we give weight to immediate rewards compared to later ones (value of successor state). If we had a p of 1, the gamma would not matter (as long as its not 0) since we would always choose the action which leads us (with p=1) to the successor state with the highest value. With a $p < 1$ a γ close to 0 (< 0.5) would be very myopic, which means it's short sighted. The agent e.g. would give much higher importance to states that are close to him as compared to ones that are far away. For γ close to one, the agent is very far-sighted. So it will always keep the goal of ending up in the reward state in mind. Also for large γ the V(s) differ far more between neighboring states compared to systems with small γ. In our grid (with a γ of 0.35), one can observe that far away of the positive reward state s_1, our agent chooses to move into the wall (state s_{24}) since it gives high importance to immediate rewards ($\rightarrow$ not getting into s_{11}), instead of long-term rewards ($\rightarrow$ getting into s_1). However, such occurrences can only happen for a $p \neq 1$ (if there is a stochasticity for the actions) because otherwise the agent would be stuck going into the wall.

2.3 Exercise 2c)

1.: To solve the problem I used a Monte Carlo (MC) iterative optimisation algorithm (method monte_carlo_iterative_optimisation() in the code). The MC algorithm starts with a uniform policy and a state action value function which is initiated to only zeros. It generates episodes by calling another function generate_episode() (according to our current policy - uniform in the beginning - and the grid noise due to p of 0.7). Each episode is generated with a random starting point in the grid, which is chosen with a probability of 1/27, and ends when an absorbing state is reached. This ensures that all the states in our grid will be visited. I am using a first-visit MC algorithm, so I only calculate the forward discounted return for the first appearance of state action pairs (s,a) per episode. This discounted return is utilised to update our state action value function Q(s,a) ($Q(s,a) = Q(s,a) + \alpha\,[R - Q(s,a)]$). The learning rate α decides how much we weigh newly gained insights (returns). I chose a constant alpha of 0.002, which smooths the learning significantly. Due to this relatively small alpha our agent will learn comparatively slowly, but I am letting my agent run through the grid 40000 times, which compensates that. Further, I chose an ϵ that fulfills the GLIE condition and decays per iteration. The ϵ is 0.9999^i, where i is the current iteration. After 40000 iterations this epsilon becomes around 0.02. The slowly decaying ϵ costs computational power, however, since the grid is comparatively small, it also gives a favorable gradual decay which allows for sufficient exploration. Therefore at the beginning of the training when our Q(s,a) approximation is still bad, our policy is more random (exploration) and as we further train, random behavior gets inefficient, and we want to use our Q(s,a) to decide which action to take (exploitation). If we would now increase our iterations to infinity our value function would converge towards the optimal value function found using dynamic programming. Overall, I realized empirically that increasing my iterations and increasing my initial ϵ (never being more than 1, in my case 0.9999), while decreasing α leads to better results, but also higher run-time.

For the following sections, I want to show my thinking, but using 40000 iterations and running this multiple times, will be too computationally exhaustive. Therefore, I decrease the iterations to 600 and decaying ϵ to 0.994^i (to ensure small enough epsilon after hundreds of iterations), while increasing α to 0.025. This will not give a result as good as with the parameters stated before, but still improves substantially. You can see the value function and policy of this setting in figure 3 c) and d).

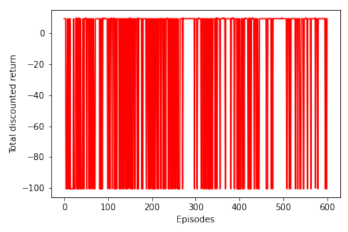
2.:



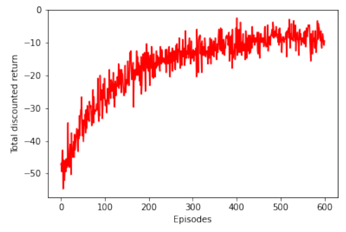
(a) Values (40000 iterations) (b) Policy (40000 iterations) (c) Values (600 iterations) (d) Policy (600 iterations)

Figure 3: Value function and policy found by Monte Carlo Online optimisation; 40000 iterations, $\alpha = 0.002$, decaying $\epsilon = 0.9999^i$ (a and b); 600 iterations, $\alpha = 0.025$, decaying $\epsilon = 0.994^i$ (c and d)

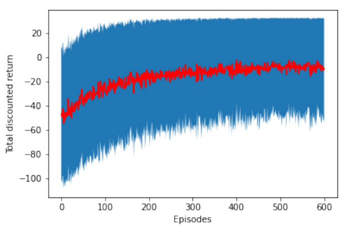
3.:



(a) Discounted total return per episode (1 run)



(b) Averaged discounted total return per episode (250 runs)



(c) Averaged discounted total return per episode with standard deviation (250 runs)

Figure 4: Averaged backwards discounted total returns per episode over the number of episodes (using Monte Carlo Online optimisation with 600 iterations per run, $\alpha = 0.025$ and decaying $\epsilon = 0.994^i$)

For showing the learning curve, I plot the total backwards discounted return. That means that the absorbing state reward will be added to all previous collected rewards backwards discounted. E.g. taking 3 steps and ending up in the penalty state, the total return becomes $-100 - 1\gamma - 1\gamma^2 = -100.4725$. We can see this in figure 4 a), where many times the total backwards discounted return is either around -100 or around slightly below 10 for the reward state. One can already in 4 a) see that over the episodes our agent ends up less in the penalty state and more in the reward state (less spikes towards -100 in second half of plot). Due to our low γ our improvement in discounted total return over encountered episodes trained, will be highly dependent on where our agent ends up (and not how long it took him to end up there). To show how our agent is improving, we need to run the algorithm multiple times for 600 iterations and average over the experienced backwards discounted return per specific episode. I decided to use 250 runs, because it smooths our curve while still letting room for comparison of the extent of dynamics (zigzag), see figure 5) . In figure 4 b), one can observe how our learning curve is increasing in fact, and our agent is getting better with the number of episodes he has already been trained. One can also see, how it seems to be converging towards a value of roughly -10, which is certainly not optimal (in fact it is converging towards the optimal value function for iterations $\rightarrow \infty$ because of GLIE). This is due to the fact that our agent has not been trained long enough. For our initial parameter setting we would have observed way better results. In figure 4 c), we can also observe that the standard deviation is decreasing until convergence, which shows us that our agent is becoming more confident, and chooses less often to go towards the penalty state, as it did in the beginning (standard deviation is decreasing from roughly +/-50 to about +/-35).

4.: In general, it is the case that if $\alpha = 0$, then the Q(s,a) estimate is never updated by the agent and if $\alpha = 1$, then the final value estimate for each Q(s,a) is always equal to the last return that was experienced by the agent. For ϵ it is the case that the larger ϵ is, the more likely you are to pick one of the non-greedy actions and vice versa. An ϵ of 1, results in a policy that is always random.

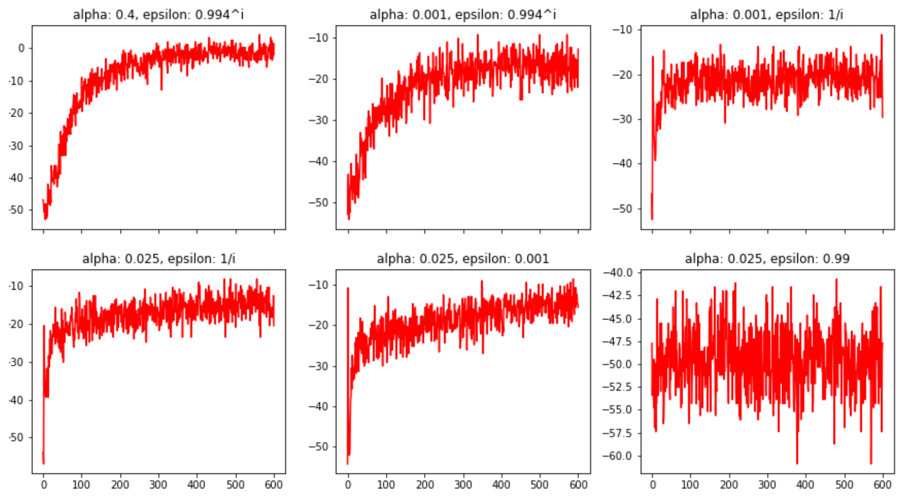


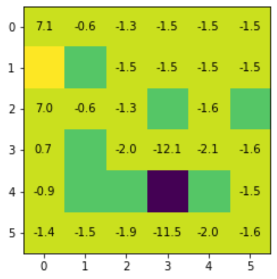
Figure 5: Averaged discounted total returns per episode over the number of episodes for different α and ϵ (using Monte Carlo Online optimisation with 600 iterations per run)

First of all, in figure 5 one can see that an ϵ of 0.994^i leads to a less steep learning rate because the first part of the learning is very exploration-orientated (see first two plots). A very low alpha in combination, leads to an even flatter curve and consequently worse returns after 600 runs. An ϵ of $1/i$ performs similar to a very small ϵ (almost deterministic), where we can see a fast rise with few exploration and then a flatter rise depending on the size of α. In the last plot, one can see how a very high ϵ leads to a terrible return due to randomness (independent of α). One can conclude the following: A low α, produces flatter curves because of slower learning and might not get to convergence in few iterations. High α leads to a fast rise (due to weighing new insights heavily), but might be not very robust since it might be very sensitive to noise and potentially bad policies. For very low ϵ, our policy becomes almost deterministic. This leads to a very quick rise of our learning curve, but may lead to no convergence to our optimal total return, since the agent might not explore all possible actions per state. Whereas, a very high ϵ will lead to a policy that is always almost random. To be honest, analyzing the first plot in 5, might lead to the assumption that it gives the best results (which might be true for 600 iterations), but analyzing its value functions, they differ heavily to the ones we found using DP. Therefore, I assume that for larger iterations, this agent will not further improve and potentially even diverge.

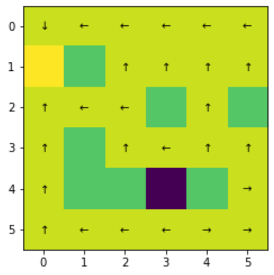
2.4 Exercise 2d)

1.: For the Temporal Difference reinforcement learner, I implemented the SARSA algorithm with a decaying ϵ of $1/i$, where i is the iteration and also a decaying α of 1/N(s,a) (method sarsa() in code). Once again the initial state per episode is found the same way as with MC before. N(s,a) is a counter that counts how many times the state action pair has been visited over all episodes (iterations). By gradually decaying alpha , we adjust our Q(s,a) heavily in the first visits and gradually less later, which ensures that with ongoing episodes our Q(s,a) is less influenced by any immediate reward which might be caused by noise. By decaying it per each specific (s,a), we ensure equal decaying per pair occurence and not per iteration. I am initializing my Q(s,a) in the beginning to -1s since that is the penalty per step, except for the terminal states, which are initialized to 0. In order to achieve the results in figure 6, I am letting the agent train over 40000 episodes (iterations). Running this algorithm an infinite number of times ensures convergence, since our algorithm fulfills both the ϵ-GLIE condition and the Robbins-Monroe Theorem for α. The main advantage of implementing SARSA is that it both samples and bootstraps and, therefore, does not need to be aware of the entire episode (it's truly online).

2.:



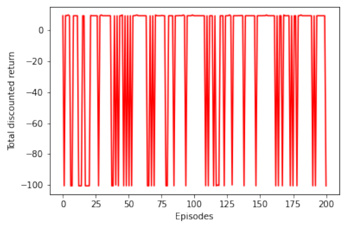
(a) Found Values



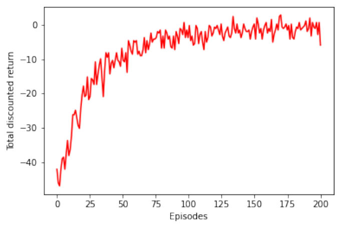
(b) Found Policy

Figure 6: Value function and policy found by Sarsa optimisation in 40000 iterations with decaying $\alpha = 1/N(s, a)$ and decaying $\epsilon = 1/i$

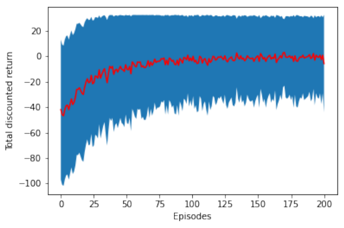
3.:



(a) Discounted total return per episode (1 run)



(b) Averaged discounted total return per episode (250 runs)



(c) Averaged discounted total return per episode with standard deviation (250 runs)

Figure 7: Averaged backwards discounted total returns per episode over the number of episodes (using Sarsa with 200 iterations per run, decaying $\alpha = 1/N(s, a)$ and decaying $\epsilon = 1/i$

For the learning curve I show the backwards discounted total returns of the SARSA learner with decaying $\alpha = 1/N(s, a)$ and decaying $\epsilon = 1/i$ over 200 iterations (episodes), since my SARSA learner is converging way faster than the MC learner (it is quicker becoming more deterministic as ϵ decreases and, therefore, spends less time randomly exploring the grid in the beginning).

Once again, as before in section 2c)3. for MC, one can see the backwards discounted return is either around -100 or +10, if plotting only one run (figure 7 a)), which is again due to our low γ and backwards discounting. Also, one can discover that the second half of the plot (after 100 episodes), does more often end up at +10 than in the first half. Many of the -100s in the end might be due to the noise in our system. When plotting the average total discounted return per episode over 250 runs (250 because of same reasons as with MC), one can see in figure 7 b) how the learning curve seems to converge towards a total discounted return of roughly 0 after being trained on only a few episodes (way faster than with MC, mainly due to different ϵ). Figure 7 c) shows that, as with MC, the standard deviation reduces significantly in the first couple of episodes (from about +/-50 to around +/- 30).

4.:

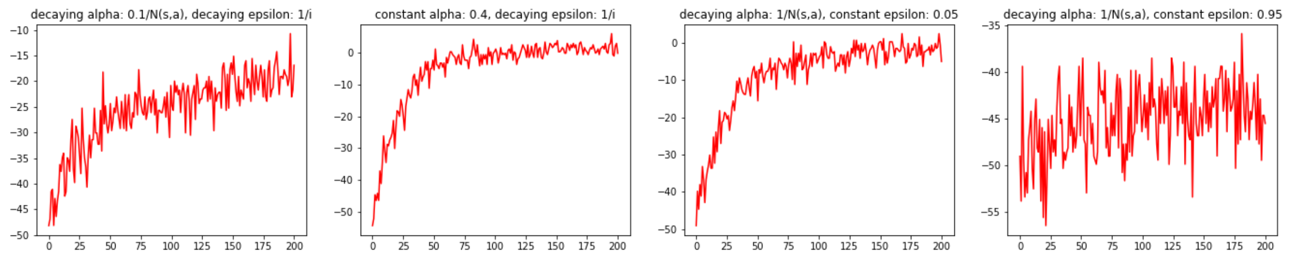
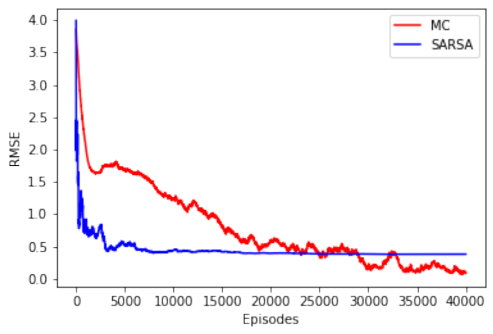


Figure 8: Averaged backwards discounted total returns per episode over the number of episodes for different α and ϵ (using SARSA optimisation with 200 iterations per run)

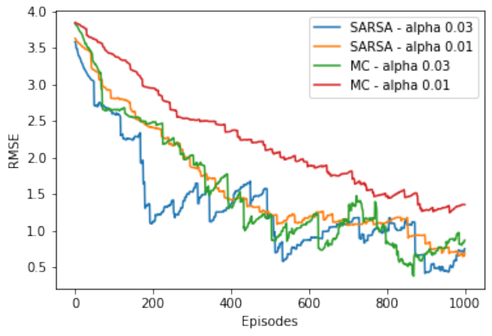
I created a comparison of different α and ϵ in figure 8. In general, α and ϵ have similar effects as already discussed for MC in section 2c). However, now we are also looking at adjustments of decaying α. Keeping all the settings as they were in my algorithm, but decreasing the constant of our decaying α, e.g. to 1/10N(s,a), leads to slower learning. Our agent takes more episodes until it converges but, is also, especially in the beginning, less influenced by maybe ending up in the penalty state due to uncertainty in the environment (p =0.7). Using a large constant α lets our curve rise quicker, but might be not as good in converging as compared to a smaller α, since it will even after multiple episodes trained, update our Q(s,a) heavily (sensitive to noise). Keeping α as 1/N(s,a) but using a small constant ϵ is very similar to the results in 7 b), since with constant ϵ we are only exploring less in the early runs and a bit more in later ones (but its not GLIE, so will not converge towards optimum when iterations $\rightarrow \infty$). A large ϵ leads to a lot of randomness and no convergence.

2.5 Exercise 2e)

1.:



(a) RMSE over 400000 episodes SARSA ($\alpha = 1/N(s,a), \epsilon = 1/i$) and MC ($\alpha = 0.002, \epsilon = 0.9999^i$)

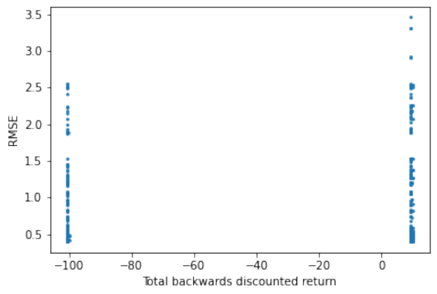


(b) RMSE over 1000 episodes for different α using MC and SARSA with constant $\epsilon = 0.035$

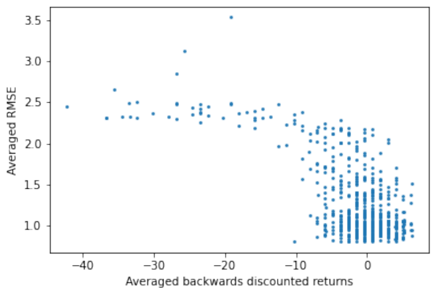
Figure 9: RMSEs between found V(s) of RL algorithms and V(s) found through Dynamic programming

In figure 9, one can see how my RL agent trained using MC takes way more time to reduce the RMSE compared to my algorithm trained using SARSA. This is mainly due to the different approaches regarding ϵ, where the MC agent is exploring way more in the beginning than the SARSA agent. However, one can also see how the RMSE for MC reduces even more than for SARSA after around 30000 iterations. This shows that my MC algorithm is approximating the optimal value function better than my SARSA algorithm when running both for 40000 iterations. In figure 9 c), I am also portraying how for constant $\epsilon = 0.035$ MC and SARSA algorithms with varying α behave differently. One can see that for the same constant α the RMSE for SARSA is reducing way quicker. Also for higher α one can see how the reduction occurs not as steadily and more bouncy, since new insight are weighted higher due to higher learning rate.

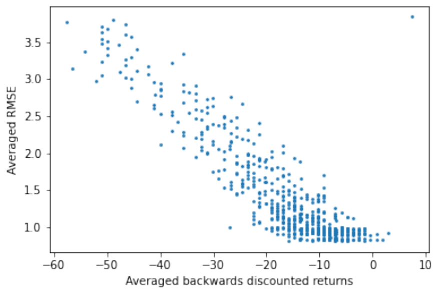
2.:



(a) SARSA ($\alpha = 1/N(s,a), \epsilon = 1/i$): RMSE over Total Backwards Discounted Return (600 episodes)



(b) SARSA ($\alpha = 1/N(s,a), \epsilon = 1/i$): Averaged RMSE over Averaged Total Backwards Discounted Return (600 episodes - 100 runs)



(c) MC ($\alpha = 0.025, \epsilon = 0.994^i$): Averaged RMSE over Averaged Total Backwards Discounted Return (600 episodes - 100 runs)

Figure 10: RMSEs over Total Backwards Discounted Return on an episode by episode basis (each dot corresponds to one episode)

Figure 10 a) shows that plotting the RMSE over the total backwards discounted return does not give us a lot of insight, since the return will either be around -100 or +10. This is the case for both SARSA and MC. Therefore, I ran the algorithms 100 times training the agent each time with 600 episodes. Then I took the averaged total backwards discounted return per episode (average over the 100 times for each episode) and the averaged RMSE and plot these. Now one can see how, when training over 600 episodes, the Sarsa (figure 10 b)) learner is getting better way quicker, since there are less dots with more negative returns than compared to MC. Also the SARSA algorithm achieves better returns in general (dots are further to the right). In both plots (figure 10 b) and c)) one can infer how with decreasing RMSE, the total discounted return increases (bad returns are mostly corresponding to high RMSE). This implies that, in general, it is very important to have a good value function estimate to obtain a good return. One can also infer from the plots that for very low RMSEs there is still some spread, which might be due to noise (p=0.7).

3.: Both MC and TD unfortunately do not learn my policy perfectly in 40000 episodes, but both are improving it significantly. Whether to choose MC or TD depends on the problem, since MC samples from entire episodes and TD bootstraps from a 1-step sample. If one can not know the entire episode TD is the better choice. However, for our problem, I experienced that, using TD, the learning occurs generally quicker for appropriate α and ϵ (compare 9 c)). Altering parameters, will have significant influence on the performance for both. MC seems to perform better with an ϵ-GLIE policy that decays more slowly, such that there is more time for exploration. Further, especially for MC one seems to have to use very low α in order to converge at a good total discounted return, which in turn necessarily leads to more iterations needed, since the learning rate is small. For huge iterations ($>$ 100.000), I experienced that MC sometimes performs better than TD, but MC also seems to be more dependent on a well chosen ϵ. In summary, when fulfilling the GLIE-condition for MC and the GLIE-condition as well as Robbins-Monroe-Theorem for TD, both will learn correctly over a sufficient number of times. But, for our problem, I believe, that one can quicker achieve a more correct result using TD (Sarsa).

3 Appendix - Code

```
import numpy as np
import matplotlib.pyplot as plt

####### Grid world class that includes all the necessary methods for building thee grid as well as
    algorithms for DP, MC and SARSA #######
class GridWorld(object):
    def __init__(self):

        ### Attributes defining the Gridworld #######

        # probability that desired action occurs
        p = 0.7

        # Shape of the gridworld
        self.shape = (6,6)

        # Locations of the obstacles
        self.obstacle_locs = [(1,1),(2,3),(2,5),(3,1),(4,1),(4,2),(4,4)]

        # Locations for the absorbing states
        self.absorbing_locs = [(1,0),(4,3)]

        # Rewards for each of the absorbing states
        self.special_rewards = [10, -100] #corresponds to each of the absorbing_locs

        # Reward for all the other states
        self.default_reward = -1


        # Action names
        self.action_names = ['N','E','S','W']

        # Number of actions
        self.action_size = len(self.action_names)


        # Randomizing action results: [1 0 0 0] to no Noise in the action results.
        self.action_randomizing_array = [p, (1-p)/3, (1-p)/3 , (1-p)/3]

        ############################################



        #### Internal State ####


        # Get attributes defining the world
        state_size, T, R, absorbing, locs = self.build_grid_world()

        # Number of valid states in the gridworld (there are 27 of them)
        self.state_size = state_size

        # Transition operator (3D tensor)
        self.T = T

        # Reward function (3D tensor)
        self.R = R

        # Absorbing states
        self.absorbing = absorbing

        # The locations of the valid states
        self.locs = locs

        # Starting location
        self.starting_loc = self.get_starting_loc()

        #indizes of absorbing states
        self.absorbing_indizes = [self.loc_to_state(state_id,self.locs) for state_id in
            self.absorbing_locs]

        # Number of the starting state
        self.starting_state = self.loc_to_state(self.starting_loc, locs);

        # Locating the initial state
        self.initial = np.zeros((1,len(locs)))
        self.initial[0,self.starting_state] = 1


        # Placing the walls on a bitmap
        self.walls = np.zeros(self.shape);
        for ob in self.obstacle_locs:
            self.walls[ob]=-20

        # Placing the absorbers on a grid for illustration
        self.absorbers = np.zeros(self.shape)
        for ab in self.absorbing_locs:
            self.absorbers[ab] = -1

        # Placing the rewarders on a grid for illustration
        self.rewarders = np.zeros(self.shape)
        for i, rew in enumerate(self.absorbing_locs):
            self.rewarders[rew] = self.special_rewards[i]

        #Illustrating the grid world
        self.paint_maps()
        ################################


    ####### Getters ###########

    def get_transition_matrix(self):
        return self.T

    def get_reward_matrix(self):
        return self.R

    #get random starting location with probability of 1/27
    def get_starting_loc(self):
        possible_starts = [state for state in self.locs if state not in self.absorbing_locs]
        index = np.random.choice(range(len(possible_starts)))
        return possible_starts[index]

    ########################


    ########### Internal Helper Functions #####################
    def paint_maps(self):
        plt.figure()
        plt.subplot(1,3,1)
        plt.imshow(self.walls)
        plt.subplot(1,3,2)
        plt.imshow(self.absorbers)
        plt.subplot(1,3,3)
        plt.imshow(self.rewarders)
        plt.show()

    def build_grid_world(self):
        # Get the locations of all the valid states, the neighbours of each state (by state number),
        # and the absorbing states (array of 0's with ones in the absorbing states)
        locations, neighbours, absorbing = self.get_topology()

        # Get the number of states
        S = len(locations)

        # Initialise the transition matrix
        T = np.zeros((S,S,4))

        for action in range(4):
            for effect in range(4):

                # Randomize the outcome of taking an action
                outcome = (action+effect+1) % 4
                if outcome == 0:
                    outcome = 3
                else:
                    outcome -= 1

                # Fill the transition matrix
                prob = self.action_randomizing_array[effect]
                for prior_state in range(S):
                    post_state = neighbours[prior_state, outcome]
                    post_state = int(post_state)
                    T[post_state,prior_state,action] = T[post_state,prior_state,action]+prob


        # Build the reward matrix
        R = self.default_reward*np.ones((S,S,4))
        for i, sr in enumerate(self.special_rewards):
            post_state = self.loc_to_state(self.absorbing_locs[i],locations)
            R[post_state,:,:]= sr

        return S, T,R,absorbing,locations

    def get_topology(self):
        height = self.shape[0]
        width = self.shape[1]

        index = 1
        locs = []
        neighbour_locs = []

        for i in range(height):
            for j in range(width):
                # Get the locaiton of each state
                loc = (i,j)

                #And append it to the valid state locations if it is a valid state (ie not absorbing)
                if(self.is_location(loc)):
                    locs.append(loc)

                    # Get an array with the neighbours of each state, in terms of locations
                    local_neighbours = [self.get_neighbour(loc,direction) for direction in
                        ['nr','ea','so', 'we']]
                    neighbour_locs.append(local_neighbours)

        # translate neighbour lists from locations to states
        num_states = len(locs)
        state_neighbours = np.zeros((num_states,4))

        for state in range(num_states):
            for direction in range(4):
                # Find neighbour location
                nloc = neighbour_locs[state][direction]
                # Turn location into a state number
                nstate = self.loc_to_state(nloc,locs)

                # Insert into neighbour matrix
                state_neighbours[state,direction] = nstate;


        # Translate absorbing locations into absorbing state indices
        absorbing = np.zeros((1,num_states))
        for a in self.absorbing_locs:
            absorbing_state = self.loc_to_state(a,locs)
            absorbing[0,absorbing_state] =1

        return locs, state_neighbours, absorbing



    def loc_to_state(self,loc,locs):
        #takes list of locations and gives index corresponding to input loc
        return locs.index(tuple(loc))


    def is_location(self, loc):
        # It is a valid location if it is in grid and not obstacle
        if(loc[0]<0 or loc[1]<0 or loc[0]>self.shape[0]-1 or loc[1]>self.shape[1]-1):
            return False
        elif(loc in self.obstacle_locs):
            return False
        else:
             return True

    def get_neighbour(self,loc,direction):
        #Find the valid neighbours (ie that are in the grif and not obstacle)
        i = loc[0]
        j = loc[1]

        nr = (i-1,j)
        ea = (i,j+1)
        so = (i+1,j)
        we = (i,j-1)

        # If the neighbour is a valid location, accept it, otherwise, stay put
        if(direction == 'nr' and self.is_location(nr)):
            return nr
        elif(direction == 'ea' and self.is_location(ea)):
            return ea
        elif(direction == 'so' and self.is_location(so)):
            return so
        elif(direction == 'we' and self.is_location(we)):
            return we
        else:
            #default is to return to the same location
            return loc

############################################


    ####### Methods for Dynammic Programming #########

    def policy_evaluation(self, policy, threshold, discount, V = 0):

        # Make sure delta is bigger than the threshold to start with
        delta= 2*threshold

        #Get the reward and transition matrices
    R = self.get_reward_matrix()
    T = self.get_transition_matrix()

    # The value is initialised at 0
    if V == 0:
        V = np.zeros(policy.shape[0])
    # Make a deep copy of the value array to hold the update during the evaluation
    Vnew = np.copy(V)

    # While the Value has not yet converged do:
    while delta>threshold:
        for state_idx in range(policy.shape[0]):
            # If it is one of the absorbing states, ignore
            if(self.absorbing[0,state_idx]):
                continue

            # Accumulator variable for the Value of a state
            tmpV = 0
            for action_idx in range(policy.shape[1]):
                # Accumulator variable for the State-Action Value
                tmpQ = 0
                for state_idx_prime in range(policy.shape[0]):
                    tmpQ = tmpQ + T[state_idx_prime,state_idx,action_idx] *
                         (R[state_idx_prime,state_idx, action_idx] + discount * V[state_idx_prime])

                tmpV += policy[state_idx,action_idx] * tmpQ

            # Update the value of the state
            Vnew[state_idx] = tmpV

        # After updating the values of all states, update the delta
        delta = max(abs(Vnew-V))
        # and save the new value into the old
        V=np.copy(Vnew)

    return V


### policy iteration algorithm that computes our optimal policy as well as optimal value function
    using dynamic programming
def policy_iteration_algorithm(self,policy, threshold, discount):
    #get current optimal value function for initial policy
    V = self.policy_evaluation(policy, threshold, discount)

    #Get the reward and transition matrices
    R = self.get_reward_matrix()
    T = self.get_transition_matrix()

    policy_stable = False

    while not policy_stable:

        #make deep copy of policy for later comparison
        b = np.copy(policy)
        for state_idx in range(policy.shape[0]):

            # If it is one of the absorbing states, ignore
            if(self.absorbing[0,state_idx]):
                continue

            #set Q(s,a) for s with index state_idx
            values_per_action = np.zeros(self.action_size)
            for action_idx in range(self.action_size):
                # Accumulator variable for the State-Action Value
                tmpQ = 0
                for state_idx_prime in range(self.state_size):
```

```
                tmpQ = tmpQ + T[state_idx_prime,state_idx,action_idx] *
                    (R[state_idx_prime,state_idx, action_idx] + discount * V[state_idx_prime])

            #fill Q(s,a) for s with index state_idx
            values_per_action[action_idx] = tmpQ
        #update greedy policy
        policy[state_idx] = np.zeros(self.action_size)
        policy[state_idx][np.argmax(values_per_action)] = 1

    #if policy does no longer change after iterations leave while loop
    if (b == policy).all():
        policy_stable = True
    #get optimal value function for optimal policy found before
    V = self.policy_evaluation(policy, threshold, discount)

return policy,V


def draw_deterministic_policy(self, Policy):
    # Draw a deterministic policy
    # The policy needs to be a np array of 22 values between 0 and 3 with
    # 0 -> N, 1->E, 2->S, 3->W
    plt.figure()

    plt.imshow(self.walls+self.rewarders +self.absorbers)
    #plt.hold('on')
    for state, action in enumerate(Policy):
        if(self.absorbing[0,state]):
            continue
        arrows = [r"$\uparrow$",r"$\rightarrow$", r"$\downarrow$", r"$\leftarrow$"]
        action_arrow = arrows[action]
        location = self.locs[state]
        plt.text(location[1], location[0], action_arrow, ha='center', va='center')

    plt.savefig('arrows_policy.png')
    plt.show()

def draw_values(self, V):
    plt.figure()

    plt.imshow(self.walls+self.rewarders +self.absorbers)
    #plt.hold('on')
    for state, value in enumerate(V):
        if(self.absorbing[0,state]):
            continue
        location = self.locs[state]
        plt.text(location[1], location[0], np.round_(value, 1), ha='center', va='center')

    plt.savefig('values.png')
    plt.show()
##########################


########## Helper methods for MC and SARSA####################

def rmse(self, actual, predicted):
    return np.sqrt(np.mean((predicted-actual)**2))

def generate_episode(self, policy, starting_state = None):
    #get random starting location and index of this location in our locs list or always start from
        same starting_point (possible because of epsilon greedy policy)
    if not starting_state:
        state = self.get_starting_loc()
    else:
        state = starting_state
    state_idx = self.loc_to_state(state,self.locs)
    #declare empty episode
    episode = np.empty((0,5))

    #Get the reward and transition matrices
    R = self.get_reward_matrix()
    T = self.get_transition_matrix()
    i = 0
    #loop forever until we hit a terminal state s_1 or s_11
    while state not in self.absorbing_locs:
        #choose action in currwent state randomly according to policy
        action_idx = np.random.choice([0,1,2,3], 1, p = policy[state_idx])

        #find out at which state you end up according to transition matrix (walls etc. already
            included)
        next_state_idx = np.random.choice(range(len(self.locs)), 1, p =
            np.concatenate(T[:,state_idx,action_idx], axis=0))[0]
        next_state_array = np.array([[i, state_idx, action_idx[0],
            R[next_state_idx,state_idx,action_idx][0], next_state_idx]])

        #append to episode the init state, action, reward collected and next state
        episode = np.append(episode,next_state_array, axis=0)

        #update new state
        state = self.locs[int(next_state_idx)]
        state_idx = self.loc_to_state(state,self.locs)
        i+=1

    return episode


########## MC algorithm ####################

def monte_carlo_iterative_optimisation(self, iterations, alpha, epsilon, discount = 1,
     optimal_value = None, starting_state = None, const_epsilon=False, epsilon_steps = False):

    #Get the reward and transition matrices
    R = self.get_reward_matrix()
    T = self.get_transition_matrix()

    #set initial policy to uniform, which is an epsilon greedy policy in the first iteration
    policy= np.zeros((grid.state_size, grid.action_size))
    policy.fill(0.25)

    #set total returns array to later evaluate the learning progress of agent
    total_returns = np.zeros(iterations)

    #check if we want to compute rmse
    if optimal_value:
        optimal_values = self.policy_iteration_algorithm(policy, discount)[1]
        rmse = np.zeros(iterations)
    else:
        rmse = 0

    #set initial Q to 0
    Q = np.zeros([policy.shape[0], self.action_size])


    for i in range(iterations):

        #generate episode with random starting state if starting_state = None
        episode = self.generate_episode(policy, starting_state)

        #get the total return for an episode discounted, flip because of backward discounted
        discounted_total_returns = np.zeros(episode.shape[0])
        for index,reward in enumerate(np.flip(episode[:,3])):
            discounted_total_returns[index] = reward*discount**index
        total_returns[i] = np.sum(discounted_total_returns)


        for state_idx in range(policy.shape[0]):
            #find starting index where state occurs for the first time and calculate the sum of
                returns from the found index onwards
            if state_idx in episode[:,1]:

                for action_idx in range(self.action_size):
                    #create mask
                    mask = ((episode[:,1]==state_idx) & (episode[:,2]==action_idx))
                    #check whether mask exists
                    if mask.any():
                        starting_index = int(episode[mask][0][0])

                        #discount returns of each state action pair
                        discounted_returns = np.zeros(episode[starting_index:].shape[0])
                        for idx,r in enumerate(episode[starting_index:,3]):
                            discounted_returns[idx] = r*discount**idx

                        #sum up over the successor returns
                        individual_return = np.sum(discounted_returns)

                        #update q iteratively
                        Q[state_idx][action_idx] +=
                             alpha*(individual_return-Q[state_idx][action_idx])

        #set up new epsilon greedy policy according to epsilon**i or epsilon or epsilon/(i+1)
        for policy_state in range(self.state_size):
            epsilon_decay = epsilon**i
            if const_epsilon:
                epsilon_decay = epsilon
            if epsilon_steps:
                epsilon_decay = epsilon/(i+1)
            policy[policy_state] = epsilon_decay/(self.action_size)
            policy[policy_state][np.argmax(Q[policy_state])] = 1-epsilon_decay +
                epsilon_decay/(self.action_size)


        # if optimal value True then calculate rmses
        if optimal_value:
            temp_V = np.array([np.dot(policy[state],Q[state]) for state in range(self.state_size)])
            rmse[i] = self.rmse(optimal_values, temp_V)

    #return values and not state action values
    V = np.array([np.dot(policy[state],Q[state]) for state in range(policy.shape[0])])
    return policy, V, total_returns, rmse, Q


### MC algorithm with decaying alpha, only for trying out ##########

#def mc_iterative_optimisation_decaying_alpha(self, iterations, alpha, epsilon, discount = 1,
    optimal_value = None, starting_state = None):
#
#    #Get the reward and transition matrices
#    R = self.get_reward_matrix()
#    T = self.get_transition_matrix()
#
#    #set initial policy to uniform, which is an epsilon greedy policy in the first iteration
#    policy= np.zeros((grid.state_size, grid.action_size))
#    policy.fill(0.25)
#
#    #set returns array to keep count and the total return(return of entire episode)
#    returns = np.zeros([policy.shape[0],4])
#    total_returns = np.zeros(iterations)
```

```
    #
    #    #check if we want to compute rmse
    #    if optimal_value:
    #        optimal_values = self.policy_iteration_algorithm(policy, discount)[1]
    #        rmse = np.zeros(iterations)
    #    else:
    #        rmse = 0
    #
    #    #set initial Q to 0
    #    Q = np.zeros([policy.shape[0], self.action_size])
    #
    #    for i in range(iterations):
    #        episode = self.generate_episode(policy)
    #        #get the total return for an episode discounted
    #        discounted_total_returns = np.zeros(episode.shape[0])
    #        for index,reward in enumerate(episode[:,3]):
    #            discounted_total_returns[index] = reward*discount**index
    #        total_returns[i] = np.sum(discounted_total_returns)
    #        #total_returns[i] = np.sum(episode[:,3])
    #        for state_idx in range(policy.shape[0]):
    #            #find starting index where state occurs for the first time and calculate the sum of
    returns from the found index onwards
    #            if state_idx in episode[:,1]:
    #
    #                for action_idx in range(self.action_size):
    #                    #create mask
    #                    mask = ((episode[:,1]==state_idx) & (episode[:,2]==action_idx))
    #                    #check whether mask exists
    #                    if mask.any():
    #                        starting_index = int(episode[mask][0][0])
    #
    #                        #discount returns of each stae action pair
    #                        discounted_returns = np.zeros(episode[starting_index:].shape[0])
    #                        for idx,r in enumerate(episode[starting_index:,3]):
    #                            discounted_returns[idx] = r*discount**idx
    #
    #                        #sum up over the successor returns
    #                        individual_return = np.sum(discounted_returns)
    #                        returns[state_idx][action_idx] += 1
    #
    #                        Q[state_idx][action_idx] +=
    alpha/(returns[state_idx][action_idx])*(individual_return-Q[state_idx][action_idx])
    #
    #                #set policy for that state to epsilon/4
    #        for policy_state in range(self.state_size):
    #            epsilon_decay = np.maximum(epsilon**i, 0.05)
    #            policy[policy_state] = epsilon_decay/(self.action_size)
    #            policy[policy_state][np.argmax(Q[policy_state])] = 1-epsilon_decay +
    epsilon_decay/(self.action_size)
    #                #policy[state_idx] = epsilon/(self.action_size*(i+2))
    #                #policy[state_idx][np.argmax(Q[state_idx])] = 1-epsilon/(i+2) +
    epsilon/(self.action_size*(i+2))
    #
    #        # if optimal value True then calculate rmses
    #        if optimal_value:
    #            temp_V = np.array([np.dot(policy[state],Q[state]) for state in range(self.state_size)])
    #            rmse[i] = self.rmse(optimal_values, temp_V)
    #
    #    #make policy deterministic
    #    for policy_state in range(self.state_size):
#
    #            policy[policy_state] = 0
    #            policy[policy_state][np.argmax(Q[policy_state])] = 1
    #
    #    V = np.array([np.dot(policy[state],Q[state]) for state in range(policy.shape[0])])
    #    return policy, V, total_returns, rmse, Q
```

```python
##### SARSA algorithm #################

def sarsa(self, iterations, alpha, epsilon, discount = 1, optimal_value = None, const_alpha =
    False, const_epsilon = False):

    #Get the reward and transition matrices
    R = self.get_reward_matrix()
    T = self.get_transition_matrix()

    #set initial policy to uniform, which is an epsilon greedy policy in the first iteration
    policy= np.zeros((grid.state_size, grid.action_size))
    policy.fill(0.25)

    #set total returns array for later comparison of algorithm progress over episodes trained
    total_returns = np.zeros(iterations)
    #set array which will correspond to our state_action counter N(s,a)
    state_action_counter = np.zeros([policy.shape[0],4])

    #check if we want to compute rmse
    if optimal_value:
        optimal_values = self.policy_iteration_algorithm(policy, discount)[1]
        rmse = np.zeros(iterations)
    else:
        rmse = 0

    #set initial Q to -1 since our step penalty is -1 and for terminal states set eqqual to 0
    Q = np.ones([self.state_size, self.action_size])*-1
    for terminal in self.absorbing_indizes:
        Q[terminal,:] = 0


    for i in range(iterations):

        #get random starting state
        state = self.get_starting_loc()
        state_idx = self.loc_to_state(state,self.locs)

        #choose action in starting state randomly according to policy
        action_idx = np.random.choice([0,1,2,3], 1, p = policy[state_idx])

        #increment to take into account discount for keeping track of the total_returns array
        episode_counter = 0

        #initialize array which will be used to store all rewards per episode to later calculate
            total backwards discounted return
        disc_return_array = np.empty(0)

        #loop forever until we hit a terminal state s_1 or s_11
        while state_idx not in self.absorbing_indizes:

            #observe next R and S'
            next_state_idx = np.random.choice(range(len(self.locs)), 1, p =
                np.concatenate(T[:,state_idx,action_idx], axis=0))[0]
            next_return = R[next_state_idx,state_idx,action_idx][0]
            #increment N(s,a)
            state_action_counter[state_idx][action_idx] += 1

            #append array with reward
            disc_return_array = np.append(disc_return_array, next_return)

            #choose next action
            next_action_idx = np.random.choice([0,1,2,3], 1, p = policy[state_idx])
            #check if next state is absorbing state and assign new Qs
            if int(self.absorbing[0,next_state_idx]) == 1:
                Q[state_idx][action_idx] +=
                    alpha/(state_action_counter[state_idx][action_idx])*(next_return-Q[state_idx][action_idx])
            else:
                Q[state_idx][action_idx] +=
                    alpha/(state_action_counter[state_idx][action_idx])*(next_return+discount*Q[next_state_idx][

            #if alpha given is constant then do this
            if const_alpha == True:
                if int(self.absorbing[0,next_state_idx]) == 1:
                    Q[state_idx][action_idx] += alpha*(next_return-Q[state_idx][action_idx])
                else:
                    Q[state_idx][action_idx] +=
                        alpha*(next_return+discount*Q[next_state_idx][next_action_idx]-Q[state_idx][action_idx])

            #update policy according to decaying epsilon equal to epsilon/(i+1)
            policy[state_idx] = epsilon/(self.action_size*(i+1))
            policy[state_idx][np.argmax(Q[state_idx])] = 1-epsilon/(i+1) +
                epsilon/(self.action_size*(i+1))

            #if constant epsilon given, update policy accordingly
            if const_epsilon == True:
                policy[state_idx] = epsilon/(self.action_size)
                policy[state_idx][np.argmax(Q[state_idx])] = 1-epsilon + epsilon/(self.action_size)

            #iterate
            state_idx = next_state_idx
            action_idx = next_action_idx
            episode_counter += 1

            #backwards discounting and save in total_returns array
            if state_idx in self.absorbing_indizes:
                for j,return_value in enumerate(np.flip(disc_return_array)):
                    total_returns[i] += return_value * discount ** j

        # if optimal value True then calculate rmses
        if optimal_value:
            temp_V = np.array([np.dot(policy[state],Q[state]) for state in range(self.state_size)])
            rmse[i] = self.rmse(optimal_values, temp_V)

    #make policy deterministic
    for policy_state in range(self.state_size):

            policy[policy_state] = 0
            policy[policy_state][np.argmax(Q[policy_state])] = 1

    #calculate V from our almost deterministic policy and Q
    V = np.array([np.dot(policy[state],Q[state]) for state in range(self.state_size)])
    return policy, V, total_returns, rmse, Q


####### means over sufficient number of algorithm runs - relevant for plotting learning curves
    #########

def mc_mean_total_return(self,iterations, sufficient_number,alpha,epsilon, discount = 1,
    optimal_value = None, starting_state = None, const_epsilon=False, epsilon_steps = False):
    total_returns = np.zeros([iterations,sufficient_number])
    total_rmse = np.zeros([iterations,sufficient_number])

    for i in range(sufficient_number):
        total_returns[:,i], total_rmse[:,i] = self.monte_carlo_iterative_optimisation(iterations,
            alpha, epsilon, discount, optimal_value, starting_state, const_epsilon,
            epsilon_steps)[2:4]
    return np.mean(total_returns, axis = 1), np.std(total_returns, axis = 1), np.mean(total_rmse,
        axis = 1)


def sarsa_mean_total_return(self,iterations, sufficient_number,alpha,epsilon, discount = 1,
    optimal_value = None, const_alpha= False, const_epsilon=False):
    total_returns = np.zeros([iterations,sufficient_number])
    total_rmse = np.zeros([iterations,sufficient_number])

    for i in range(sufficient_number):
        total_returns[:,i], total_rmse[:,i] = self.sarsa(iterations, alpha, epsilon,
            discount,optimal_value, const_alpha, const_epsilon)[2:4]

    return np.mean(total_returns, axis = 1), np.std(total_returns, axis = 1), np.mean(total_rmse,
        axis = 1)

#def mc_mean_total_return_decaying_alpha(self,iterations, sufficient_number,alpha,epsilon,
    discount = 1, optimal_value = None, starting_state = None):
#    total_returns = np.zeros([iterations,sufficient_number])
#    for i in range(sufficient_number):
#        total_returns[:,i] = self.mc_iterative_optimisation_decaying_alpha(iterations, alpha,
    epsilon, discount, optimal_value, starting_state)[2]
#
#    return np.mean(total_returns, axis = 1), np.std(total_returns, axis = 1)
```

```
################ run DP and plot - Question 2a), figure 2 ###################
discount = 0.35
threshold = 0.001
grid = GridWorld()

#build uniform policy with 0.25 for each direction
uniform_policy= np.zeros((grid.state_size, grid.action_size))
uniform_policy.fill(0.25)

optimal_policy, optimal_value_func = grid.policy_iteration_algorithm(uniform_policy, threshold,
    discount)

#create an array with the indeices of all the policies that are 1
optimal_arrows = np.zeros(grid.state_size).astype(int)
optimal_arrows = np.argmax(optimal_policy, axis=1)
#draw arrow grid
grid.draw_deterministic_policy(optimal_arrows)

#draw value grid
grid.draw_values(optimal_value_func)


################ generate an episode - For testing ###################
discount = 0.35
threshold = 0.001
grid = GridWorld()
```

```
#initiate first policy to uniform
uniform_policy= np.zeros((grid.state_size, grid.action_size))
uniform_policy.fill(0.25)


episode = grid.generate_episode(uniform_policy)
#np.flip(episode)
np.flip(episode[:,3])


################ MC with constant alpha; calls monte_carlo_iterative_optimisation - Question 2c),
    figure 3 ###################
discount = 0.35
iterations = 40000
alpha = 0.002
epsilon = 0.9999
grid = GridWorld()

mc_policy, mc_V, returns, rmse, Q = grid.monte_carlo_iterative_optimisation(iterations, alpha,
    epsilon,discount, optimal_value = None, const_epsilon = False, epsilon_steps = False)

#create an array with the indeices of all the policies that are 1
optimal_arrows = np.zeros(grid.state_size).astype(int)
optimal_arrows = np.argmax(mc_policy, axis=1)
#draw arrow grid
grid.draw_deterministic_policy(optimal_arrows)

#draw value grid
grid.draw_values(mc_V)


################ MC with constant alpha; calls monte_carlo_iterative_optimisation - Question 2c),
    figure 3 ###################
discount = 0.35
iterations = 600
alpha = 0.025
epsilon = 0.994
grid = GridWorld()

mc_policy, mc_V, returns, rmse, Q = grid.monte_carlo_iterative_optimisation(iterations, alpha,
    epsilon,discount, optimal_value = None, const_epsilon = False, epsilon_steps = False)

#create an array with the indeices of all the policies that are 1
optimal_arrows = np.zeros(grid.state_size).astype(int)
optimal_arrows = np.argmax(mc_policy, axis=1)
#draw arrow grid
grid.draw_deterministic_policy(optimal_arrows)

#draw value grid
grid.draw_values(mc_V)


################ plot learning curve over sufficient number of times - Question 2c), figure 4 b) and
    c) - for a) just use sufficient_number = 1 ###################
discount = 0.35
iterations =600
sufficient_number = 250
alpha = 0.025
epsilon = 0.994
grid = GridWorld()


returns, std, _ = grid.mc_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount)


plt.figure()
```

```python
plt.plot(np.linspace(0,iterations,iterations), returns, color='r')
#plt.fill_between(np.linspace(0,iterations,iterations), returns - std, returns + std)
plt.xlabel('Episodes')
plt.ylabel('Total discounted return')
plt.savefig('returns_mc_once.png')
plt.show()


plt.figure()
plt.plot(np.linspace(0,iterations,iterations), returns, color='r')
plt.fill_between(np.linspace(0,iterations,iterations), returns - std, returns + std)
plt.xlabel('Episodes')
plt.ylabel('Total discounted return')
plt.savefig('std.png')
plt.show()


######## draw mc alpha epsilon comparison - Question 2c), figure 5
def draw_varying_alpha_epsilon_mc(iterations, sufficient_number, discount):

    fig, ax = plt.subplots(2, 3, sharex='col')

    alpha = 0.4
    epsilon = 0.994
    returns, _, _ = grid.mc_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount)
    ax[0,0].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[0,0].set_title('alpha: {}, epsilon: {}^i'.format(alpha, epsilon))

    alpha = 0.001
    epsilon = 0.994
    returns, _, _ = grid.mc_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount)
    ax[0,1].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[0,1].set_title('alpha: {}, epsilon: {}^i'.format(alpha, epsilon))


    alpha = 0.001
    epsilon = 1
    returns, _, _ = grid.mc_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount,
        optimal_value = None, starting_state = None, const_epsilon=False, epsilon_steps=True)
    ax[0,2].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[0,2].set_title('alpha: {}, epsilon: {}/i'.format(alpha, epsilon))

    alpha = 0.025
    epsilon = 1
    returns, _, _ = grid.mc_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount,
        optimal_value = None, starting_state = None, const_epsilon=False, epsilon_steps=True)
    ax[1,0].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[1,0].set_title('alpha: {}, epsilon: {}/i'.format(alpha, epsilon))

    alpha = 0.025
    epsilon = 0.001
    returns, _, _ = grid.mc_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount,
        optimal_value = None, starting_state = None, const_epsilon=True)
    ax[1,1].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[1,1].set_title('alpha: {}, epsilon: {}'.format(alpha, epsilon))

    alpha = 0.025
    epsilon = 0.99
    returns, _, _ = grid.mc_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount,
        optimal_value = None, starting_state = None, const_epsilon=True)
    ax[1,2].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[1,2].set_title('alpha: {}, epsilon: {}'.format(alpha, epsilon))

    fig.set_figheight(8)
    fig.set_figwidth(15)
    plt.savefig('comparison.png')
    plt.show()


grid = GridWorld()
iterations = 600
sufficient_number = 250
draw_varying_alpha_epsilon_mc(iterations,sufficient_number,0.35)


################ TD with decaying alpha; calls sarsa - Question 2d), figure 6 ###################
discount = 0.35
iterations = 40000
alpha = 1 # alpha is 1/n(s,a)
epsilon = 1 #epsilon is 1/i
grid = GridWorld()

td_policy, td_V, returns, rmse, Q = grid.sarsa(iterations, alpha, epsilon, discount, optimal_value =
    None, const_alpha = False)

#create an array with the indeices of all the policies that are 1
optimal_arrows = np.zeros(grid.state_size).astype(int)
optimal_arrows = np.argmax(td_policy, axis=1)
#draw arrow grid
grid.draw_deterministic_policy(optimal_arrows)

#draw value grid
grid.draw_values(td_V)


################ plot convergence over sufficient number of times - Question 2d), figure 7 b) and c)
    - for a) just use sufficient_number = 1 ###################
discount = 0.35
iterations =200
sufficient_number = 250
alpha = 1
epsilon = 1
grid = GridWorld()


returns, std, _ = grid.sarsa_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount,
    optimal_value = None, const_alpha = False, const_epsilon = False)


plt.figure()
plt.plot(np.linspace(0,iterations,iterations), returns, color='r')
plt.xlabel('Episodes')
plt.ylabel('Total discounted return')
plt.savefig('returns_td.png')
plt.show()


plt.figure()
plt.plot(np.linspace(0,iterations,iterations), returns, color='r')
plt.fill_between(np.linspace(0,iterations,iterations), returns - std, returns + std)
plt.xlabel('Episodes')
plt.ylabel('Total discounted return')
plt.savefig('std_td.png')
plt.show()


######## draw sarsa alpha epsilon comparison - Question 2d), figure 8 ######
def draw_varying_alpha_epsilon_sarsa(iterations, sufficient_number, discount):

    fig, ax = plt.subplots(1, 4, sharex='col')

    alpha = 0.1
    epsilon = 1
    returns, _, _ = grid.sarsa_mean_total_return(iterations, sufficient_number,alpha,epsilon,
        discount)
    ax[0].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[0].set_title('decaying alpha: {}/N(s,a), decaying epsilon: {}/i'.format(alpha, epsilon))

    alpha = 0.4
    epsilon = 1
    returns, _, _ = grid.sarsa_mean_total_return(iterations, sufficient_number,alpha,epsilon,
        discount, optimal_value = None, const_alpha = True, const_epsilon = False)
    ax[1].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[1].set_title('constant alpha: {}, decaying epsilon: {}/i'.format(alpha, epsilon))

    alpha = 1
    epsilon = 0.05
    returns, _, _ = grid.sarsa_mean_total_return(iterations, sufficient_number,alpha,epsilon,
        discount, optimal_value = None, const_alpha = False, const_epsilon = True)
    ax[2].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[2].set_title('decaying alpha: {}/N(s,a), constant epsilon: {}'.format(alpha, epsilon))

    alpha = 1
    epsilon = 0.95
    returns, _, _ = grid.sarsa_mean_total_return(iterations, sufficient_number,alpha,epsilon,
        discount, optimal_value = None, const_alpha = False, const_epsilon = True)
    ax[3].plot(np.linspace(0,iterations,iterations), returns, color='r')
    ax[3].set_title('decaying alpha: {}/N(s,a), constant epsilon: {}'.format(alpha, epsilon))


    fig.set_figheight(4)
    fig.set_figwidth(23)
    plt.savefig('comparison_td.png')
    plt.show()


grid = GridWorld()
iterations = 200
sufficient_number = 250
draw_varying_alpha_epsilon_sarsa(iterations,sufficient_number,0.35)


################ plot estimation error against episodes - Question 2e), figure 9 a)
    ###################
discount = 0.35
grid = GridWorld()
iterations = 40000

#parameters for MC
epsilon = 0.9999
alpha = 0.002
rmse = grid.monte_carlo_iterative_optimisation(iterations, alpha, epsilon, discount, optimal_value =
    True)[3]


#parameters for SARSA
alpha = 1
epsilon = 1
rmse_sarsa = grid.sarsa(iterations, alpha, epsilon, discount, optimal_value = True)[3]

plt.figure()
plt.plot(np.linspace(0,iterations,iterations), rmse, color='r', label='MC')
plt.plot(np.linspace(0,iterations,iterations), rmse_sarsa, color='b', label='SARSA')
plt.ylabel('RMSE')
plt.xlabel('Episodes')
plt.legend()
plt.savefig('RMSE_MCandTD')
plt.show()
```

```
################ plot estimation error against episodes for multiple parameters - Question 2e),
    figure 9 b) ###################
discount = 0.35
iterations = 1000
epsilon = 0.035
grid = GridWorld()
plt.figure()


alpha = 0.03
rmse = grid.sarsa(iterations, alpha, epsilon, discount, optimal_value = True, const_alpha = True,
    const_epsilon=True)[3]
plt.plot(np.linspace(0,iterations,iterations), rmse, label='SARSA - alpha {}'.format(alpha))

alpha = 0.01
rmse = grid.sarsa(iterations, alpha, epsilon, discount, optimal_value = True, const_alpha = True,
    const_epsilon=True)[3]
plt.plot(np.linspace(0,iterations,iterations), rmse, label='SARSA - alpha {}'.format(alpha))


alpha = 0.03
rmse = grid.monte_carlo_iterative_optimisation(iterations, alpha, epsilon, discount, optimal_value =
    True, starting_state = None, const_epsilon = True)[3]
plt.plot(np.linspace(0,iterations,iterations), rmse, label='MC - alpha {}'.format(alpha))


alpha = 0.01
rmse = grid.monte_carlo_iterative_optimisation(iterations, alpha, epsilon, discount, optimal_value =
    True, starting_state = None, const_epsilon = True)[3]
plt.plot(np.linspace(0,iterations,iterations), rmse, label='MC - alpha {}'.format(alpha))


plt.ylabel('RMSE')
plt.xlabel('Episodes')
plt.legend()
plt.savefig('RMSE_comparison')
plt.show()


################ plot estimation error against rewards SARSA once- Question 2e), figure 10 a)
    ###################
discount = 0.35
iterations = 600
alpha = 1
epsilon = 1
grid = GridWorld()


_, _, returns, rmse, Q = grid.sarsa(iterations, alpha, epsilon, discount, optimal_value = True)

plt.figure()
plt.scatter(returns, rmse, s = 20)
plt.show()




###### plot averaged RMSE over averaged Total returns SARSA- Question 2e), figure 10 b)
discount = 0.35
iterations =600
sufficient_number = 100
alpha = 1
epsilon = 1
```

```
grid = GridWorld()


returns, _, rmse = grid.sarsa_mean_total_return(iterations, sufficient_number,alpha,epsilon,
    discount, optimal_value = True, const_alpha = False, const_epsilon = False)


plt.figure()
plt.scatter(returns, rmse, s=3)
#plt.fill_between(np.linspace(0,iterations,iterations), returns - std, returns + std)
plt.xlabel('Averaged backwards discounted returns')
plt.ylabel('Averaged RMSE')
plt.savefig('what.png')
plt.show()


###### plot averaged RMSE over averaged Total returns MC- Question 2e), figure 10 c)
discount = 0.35
iterations =600 #300
sufficient_number = 100 #
alpha = 0.025
epsilon = 0.994
grid = GridWorld()


returns, _, rmse = grid.mc_mean_total_return(iterations, sufficient_number,alpha,epsilon, discount,
    optimal_value = True)


plt.figure()
plt.scatter(returns, rmse, s=3)
#plt.fill_between(np.linspace(0,iterations,iterations), returns - std, returns + std)
plt.xlabel('Averaged backwards discounted returns')
plt.ylabel('Averaged RMSE')
plt.savefig('what.png')
plt.show()
```