Imperial College London

Coursework 1

Imperial College London

Autumn Term

70028 - Reinforcement Learning

Author:
Nestoras Neofytou (CID: 01350804)

Lecturer: Dr A. Aldo Faisal

Date: November 10, 2020

Question 1: Understanding of MDPs

a)

My personalised trace of states and rewards, generated using my CID (01350804), is:

$$\tau = s_3\ 1\ s_1\ 3\ s_3\ 1\ s_2\ 0\ s_2\ 0\ s_2\ 0\ s_2\ 0$$

b)

The Markov Decision Process (MDP) obtained from trace τ is is drawn below:

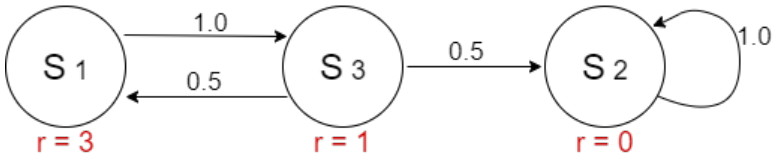

Figure 0.1: Graphical representation of the MDP from τ. The probability of a transition occurring from origin to destination is shown on each arrow.The rewards (r) gained from each state are stated in red.

As shown in the MDP diagram obtained from a single trace, the state S_2 is an absorbing state. This means that when an agent reaches that state the process is always terminated. This representation is only an approximation of the MDP, using only a single trace, thus it is greatly inaccurate. The transition probabilities P_{ij} were estimated using the proportion of occurrence in the sampled trace τ. The $\hat{P}_{ij}$ estimator is the Maximum Likelihood Estimator (MLE) and the probability of each transition from state i to state j is estimated using:

$$\hat{P}_{ij} = \frac{n_{ij}}{\sum_{ij}^{N} n_{ij}},\ where\ \begin{matrix} n_{ij} = \ number\ of\ transitions\ from\ i\ to\ j \\ N = \ number\ of\ states \end{matrix} \tag{1}$$

MLE estimators are unbiased, thus for an infinite number of traces the diagram will converge to the actual representation of the MDP studied.

c)

Using the MDP obtained in part b, we express its Transition ($\mathbf{T}$) and Reward ($\mathbf{R}$) matrix. The notation used arranges the state space S as $S = [s_1, s_2, s_3]$.

$$\mathbf{T} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 0 \end{bmatrix}\ \ and\ \ \mathbf{R} = \begin{bmatrix} 3 \\ 1 \\ 0 \end{bmatrix}$$

The unknown process is modeled as a MDP, with a terminal state with zero reward. Using the $\mathbf{T}$ and $\mathbf{R}$ calculated in part b, we use the Bellman equation given in Equation (2), to find the state value from the expected reward. Since there only a few steps before reaching the terminal state s_2 with zero reward, we use the high discount rate of $\gamma = 1$. The discount factor determines how "far-sighted" the state values are, by reducing the effect of states reached further in time.

$$\begin{gathered} \boldsymbol{v} = \mathbf{R} + \gamma \mathbf{T} \boldsymbol{v} \\ \boldsymbol{v} = (\mathbf{I} - \gamma \mathbf{T})^{-1} \mathbf{R}\ \ \rightarrow R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \end{gathered} \tag{2}$$

For a single trace and using $\gamma = 1$, $\hat{V}(s(t=0)) = 1 + 3 + 1 = 5$

Question 2: Understanding of GridWorlds

a)

Depending on my CID (01350804) the used GridWorld has parameters $\mathbf{p = 0.7}$ $\gamma = \mathbf{0.2}$. The absorbing states are s_3 and s_{11} with +10 and -100 rewards respectively.

b) Dynamic Programming

The Dynamic Programming approach used to obtain the optimal state Values and Policy was the Policy Iteration Algorithm. In the initialisation of the algorithm all the state values were set to zero, and the initial policy as "go north" everywhere. Policy Iteration is based on an alteration between evaluating the current policy and then improving it until it stabilises. The policy is stable when the difference of the updated state Value from the old one is under a pre-set threshold. In this case the selected threshold was 0.0001. Using these parameters the obtained Optimal state Values and Policy is shown below:

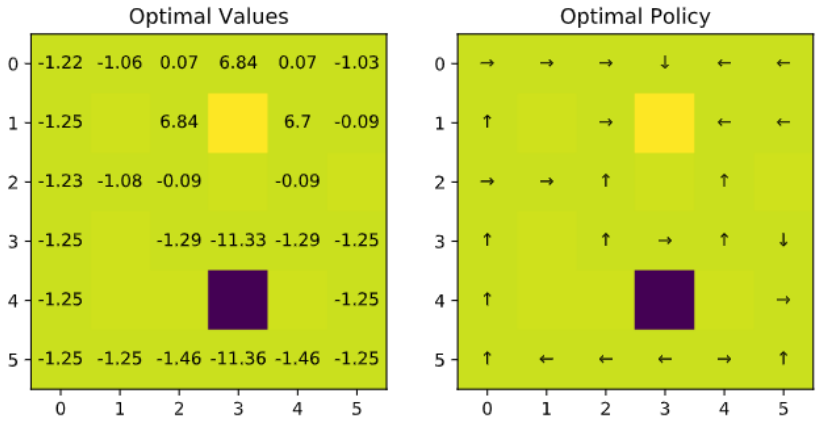

Figure 0.2: Optimal state Values (left) and Policy (right) as computed using Dynamic Programming algorithm Policy Iteration.

The same Algorithm using different parameters. The optimal values and Policies were obtained using Policy iteration using $p = [0.10, 0.25, 0.40, 0.55]$ for $\gamma = [0.3, 0.7]$ and shown in Figures 0.2 and 0.3.

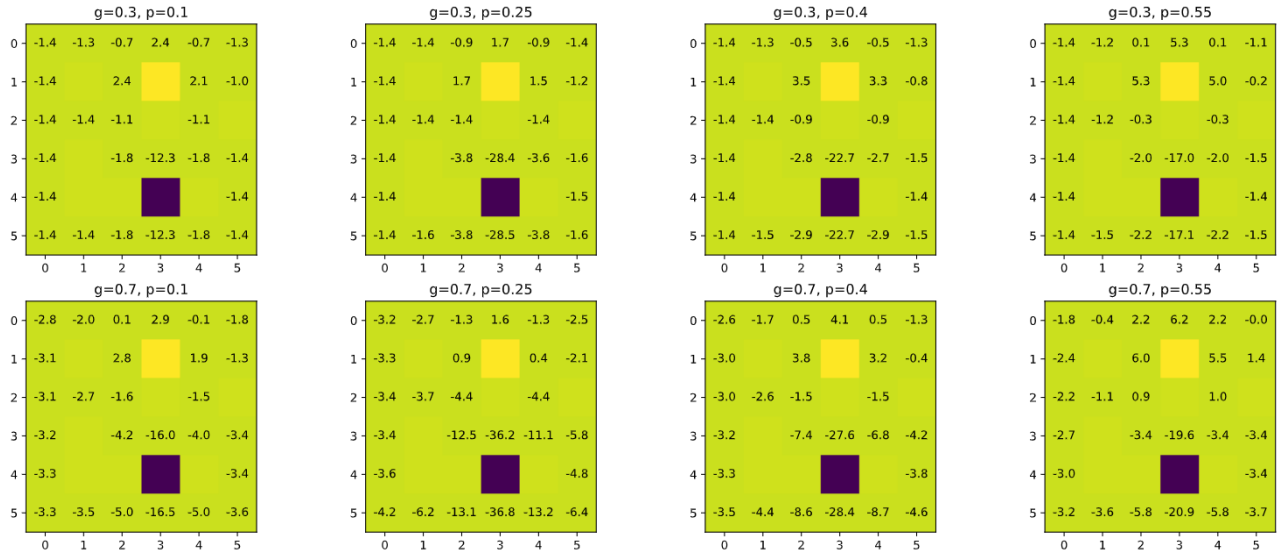

Figure 0.3: Optimal state Values as computed using Dynamic Programming algorithm Policy Iteration.

It can be observed that as the value of p increases, the value of states closer to the terminating states, are shifted towards the reward received upon reaching that state. This is true, since as p increases the more desired transition is more often successful.

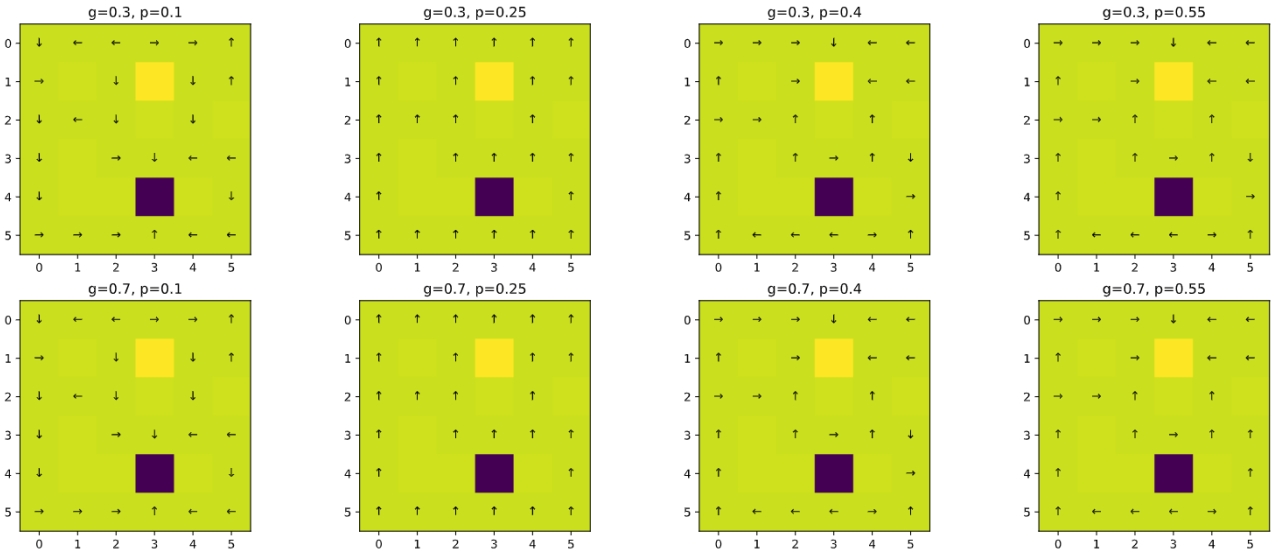

Figure 0.4: Optimal Policy as computed using Dynamic Programming algorithm Policy Iteration.

From the drawings of the optimal Policy, at p=0.25 there is equal chance of the agent to move to any direction, despite its decision. Thus the always "go north" policy set at initialisation remains unchanged. For p = 0.1, the agent decides to move away from the desired terminal state with +10 reward and decides to step into the terminal state with -100 reward, which should be avoided. Since p is under 0.25, the chosen action is the action with the least probability to be executed, thus the agent is making the right choice.

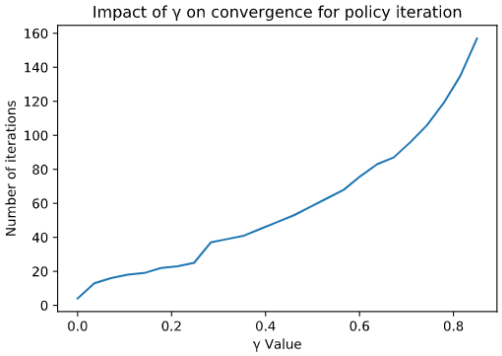

Figure 0.5: Effect of γ on number of required iterations.

In addition, for increasing γ, states located further from the terminal states correct their direction. This is because for larger γ, the value of a state is affected by values located further away. This makes the state to "see further" along a path and improve its decision in order to move to higher value areas. However, increasing γ, comes at a cost of computational power, since state values take longer to converge. The effect of gamma is shown in figure 0.5, where the number of iterations required increases with γ.

c) Monte Carlo RL

In Dynamic Programming, both the transition and reward matrix are required. However, such information is not clearly determined or available in realistic conditions. Thus the development of model-free methods arose. The method which we follow is the Monte Carlo (MC) on-policy with two different approaches. MC methods are based on developing a policy using trails performed on the examined environment. On each episode a trace τ is generated which has to reach a terminal state. The return (R_t) is defined as the total discounted reward. The value function is defined as the expected return, therefore as more returns are observed, the average should converge to the expected value. In MC Batch Learning, the policy is only updated only at the end of each batch. As a result all episodes within generate traces using the same policy.

$$R_t = r_{t+1} + _{t+1} + ... + \gamma^{T-1} r_T \qquad V^{\pi}(s) = E[R_t | S_t = s] \tag{3}$$

The first MC approach used is the MC batch Iteration, and the obtained optimal Values and Policy are reported in Figure 0.6.

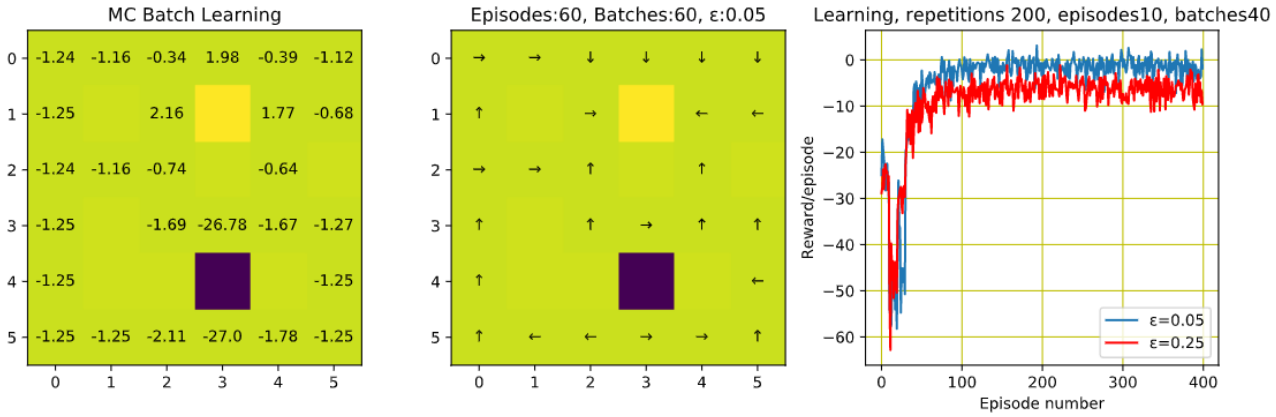

Figure 0.6: Optimal Values (left) and Policy (middle) as computed using Monte Carlo Batch Learning algorithm.(right) Collected reward progression over episodes, averaged with 200 process repetitions

The Policy used in initialisation, was the uniform policy over the whole state-action space. In addition the ϵ-greedy policy was introduced, which introduces a probability of which the agent will not chose the optimal action, to encourage further exploration. The ϵ-greedy policy is given as below:

$$\text{with probability } \epsilon : \text{choose an action at random} \tag{4}$$

$$\text{with probability } 1-\epsilon : a = \arg\max_a Q(s,a) \tag{5}$$

As observed in Figure 0.6(right), the increase of the exploration parameter ϵ improves the episodes to convergence, however it introduces a bias on the learning curve. The values computed using this algorithm are based on traces beginning from random initial states. To minimise the random effects, the process was repeated for 200 times and the average reward collected at episode is plotted.

The second MC method is the Monte Carlo Iterative Learning. The corresponding optimal Values and Policy are shown below:

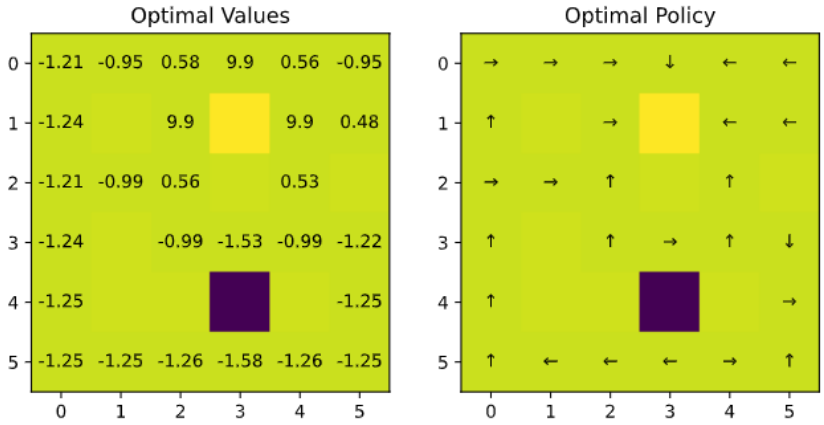

Figure 0.7: Optimal state Values and optimal Policy as computed using MC Iterative Learning with $\alpha = 0.2$ and $\epsilon = 0.1$ (*decaying*).

In MC Iterative Learning, on every iteration all the State action value estimations ($\hat{Q}$) are improved using the equation:

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s_t,a_t) + \alpha[R - \hat{Q}(s_t,a_t)] \tag{6}$$

where α is the learning rate. After the improvement of $\hat{Q}$, the policy is recalculated and is used to generate the next trace. Equation (6) is based on the calculation of a running mean. Thus $\alpha = 1/N$ where N is the number of the last N state visits considered. A larger learning rate converges faster, however, it introduces a grater variance around the value of convergence.

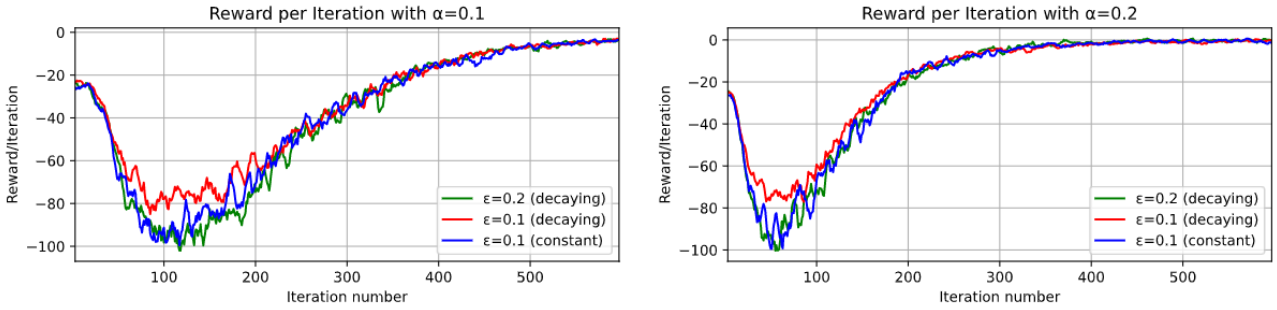

Figure 0.8: Collected rewards over iterations averaged from 300 repetitions for different values op α, and e. Lines where smoothed out using a moving average with period of 5 iterations, for easier interpretation.

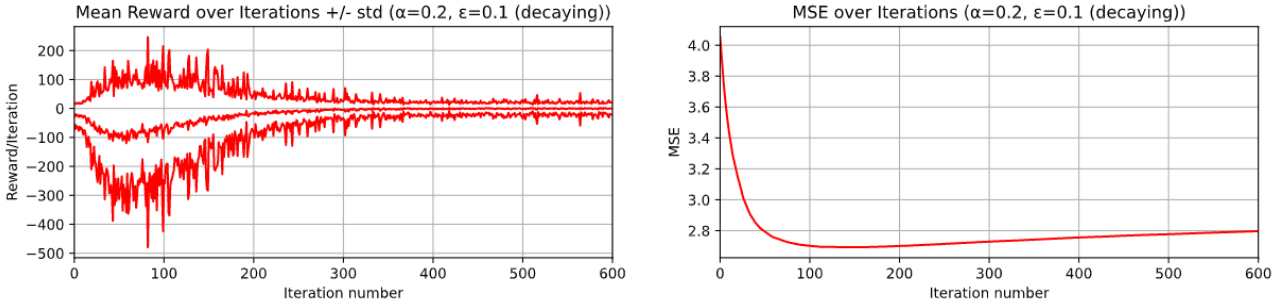

Figure 0.9: (left)Mean collected rewards ± Standard deviation, over Iterations from 300 repetitions.(right)Root Mean Squared Error over Iterations compared to optimal values obtained using Dynamic Programming in part b

As shown in figures above, the MC Iterative Learning algorithm approaches asymptotically the optimal Collected Reward value. In addition, its variance decreases significantly for over 300 iterations and minimises the RMS of the computed state Values.

d) Temporal Difference RL

The temporal Difference (TD) Learning is also a model-free method, which learns directly from experience. TD also learns from incomplete episodes by bootstrapping. The equation which updates its state action values (Q) after each iteration is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma Q(s',a') - Q(s,a)] \tag{7}$$

The implemented on-policy TD algorithm is SARSA. the policy was initialised as an 'always north' policy and the parameters used where for the learning rate $a = 0.2$ and exploration parameter $\epsilon = 0.1$. The obtained optimal Values and Policy are shown below:

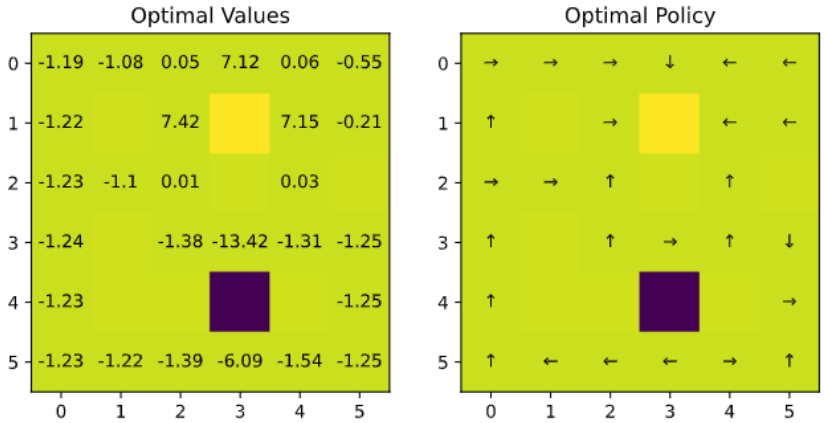

Figure 0.10: Optimal state Values and optimal Policy as computed using TD Learning with $\alpha = 0.2$ and $\epsilon = 0.1$ (*decaying*).

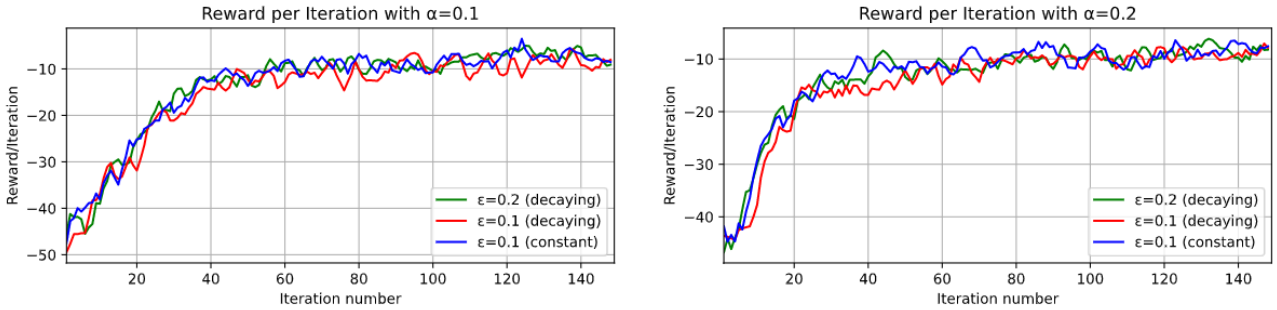

Figure 0.11: Collected rewards over iterations averaged from 300 repetitions for different values op α, and e. Lines where smoothed out using a moving average with period of 3 iterations, for easier interpretation.

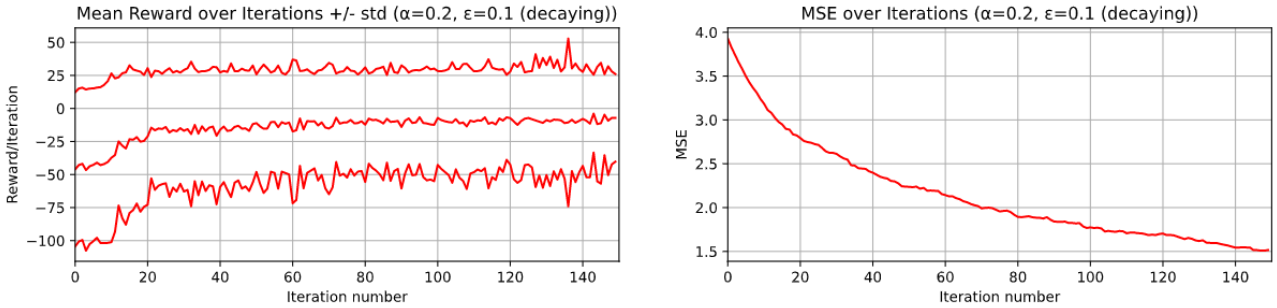

Figure 0.12: (left)Mean collected rewards ± Standard deviation, over Iterations from 300 repetitions. (right)Root Mean Squared Error over Iterations compared to optimal values obtained using Dynamic Programming in part b

Temporal Difference Learning , SARSA, converges at only 60 iterations for $a = 0.1$ and even earlier for the higher $a = 0.2$ learning rate as shown in Figure 0.11. Both the convergence of the mean and RMSE to zero in Figure 0.12, justify its unbiased nature. The SARSA algorithm to converges to the optimal action-value ($Q(s, a)$) under the following conditions:

1. GLIE sequence of policies$\pi^k(a, s)$
   - All state-action pairs are explored infinite amount of times
   - The policy converges on a greedy policy
2. Robbins-Monroe sequence of step-sizes a_t
   - $\sum_{t=1}^{\infty} a_t = \infty$
   - $\sum_{t=1}^{\infty} a_t^2 < \infty$

e) Comparison of Learners

When a evaluating estimators Root Mean Square Error is calculated, and its evolution over iterations is plotted in Figures 0.8 and 0.12 for MC Iterative and SARSA algorithms respectively. The MSE of an estimator is defined in the equation below, where it is also proven to depend on both the variance and bias of the examined estimator.

$$\begin{aligned} MSE_{\hat{V}} &= E[\hat{V} - V] = Var[\hat{V}] + E[(\hat{V} - V)^2] \\ &= Var[\hat{V}] + (Bias\ of\ \hat{V})^2 \end{aligned} \tag{8}$$

In our case $\hat{V}$ was the mean of all states in GridWorld, and it was compared to the mean of all Gridworld states computed using Dynamic Programming. Dynamic Programming was selected to calculate V, as it uses the Transition and reward matrix of the MDP, i.e it has a complete knowledge of the environment.

The first factor affecting the MSE of $\hat{V}$, is the variance of the estimator. However, both RMSE curves in figures 0.8 and 0.12 where averaged over 300 repetitions. As the variance of any estimator is proportional to 1/repetitions, we assume that the effect of $\text{Var}[\hat{V}]$ on MSE is negligible. As a result the MSE curves converge to the bias of the estimated $\hat{V}$.

A set of exploration factors (ϵ) was used with two different learning rates (α) for both MC Iterative and SARSA algorithms. The progression of the average agent's collected reward over iterations/episodes is shown in Figures 0.8 and 0.11 respectively. In order to improve convergence, ϵ was reduced after each iteration in an exponential manner as shown below, in order to satisfy the GLIE condition.

$$\epsilon_k = \epsilon_{initial}(e^{-k/400}) \tag{9}$$

As a result, ϵ was reduced exponentially by 63% after 400 iterations/episodes. The convergence on greedy policy reduces the bias of algorithms which results to the Mean collected reward to converge to zero.

Since TD methods exploit the Markov Property of process, they converge to the optimal value faster and have a smaller variance. This is demonstrated by the large gap between mean+std and mean-std in Figure 0.9(left). Despite narrowing down the variance is still large compared to the SARSA estimation, since the non zero RMSE in Figure 0.9(right) even after 600 episodes.

In conclusion there is a variance-bias trade off between the MC and TD methods. MC iteration has a larger variance and zero bias, and TD has a non zero bias and smaller variance. However with the introduction of the decaying ϵ, the convergence is improved and its RMSE approaches zero. Combined with the much faster convergence, the SARSA algorith is the better estimator to use.

Appendix

Dynamic Programming

```
    def policy_iteration(self, discount=gamma_cid, threshold = 0.0001):
        ## Slide 139 of the lecture notes for pseudocode ##

        # Transition and reward matrices, both are 3d tensors, c.f. ...
            internal state
        T = self.get_transition_matrix()
        R = self.get_reward_matrix()

        # Initialisation
        policy = np.zeros((self.state_size, self.action_size)) # Vector ...
            of 0
        policy[:,0] = 1 # Initialise policy to choose action 1 systematically
        epochs = 0
        policy_stable = False # Condition to stop the main loop

        while not(policy_stable):

            # Policy evaluation
            V, epochs_eval = self.policy_evaluation(policy, threshold, ...
                discount)
            epochs += epochs_eval # Increment epoch

            # Set the boolean to True, it will be set to False later if ...
                the policy prove unstable
            policy_stable = True
```

```
22
23              # Policy iteration
24              for state_idx in range(policy.shape[0]):
25
26                  # If not an absorbing state
27                  if not(self.absorbing[0,state_idx]):
28
29                      # Store the old action
30                      old_action = np.argmax(policy[state_idx,:])
31
32                      # Compute Q value
33                      Q = np.zeros(4) # Initialise with value 0
34                      for state_idx_prime in range(policy.shape[0]):
35                          for action in range(4):
36                              Q[action] += ...
                                    T[state_idx_prime][state_idx][action] * ...
                                    (R[state_idx_prime][state_idx][action] + ...
                                    discount * V[state_idx_prime])
37
38                      # Compute corresponding policy
39                      new_policy = np.zeros(4)
40                      new_policy[np.argmax(Q)] = 1  # The action that ...
                           maximises the Q value gets probability 1
41                      policy[state_idx] = new_policy
42
43                      # Check if the policy has converged
44                      if old_action != np.argmax(policy[state_idx]):
45                          policy_stable = False
46
47          return V, policy, epochs
```

MC Methods
- Batch Learning

```
1       def MC_batch(self, episodes, batches=1, epsilon=0, gamma=gamma_cid):
2           #initialisation
3           Q = np.zeros((self.state_size,self.action_size))
4           Q[:,0]=1
5           policy = self.apply_e_greedy_Q(Q,epsilon=epsilon)
6           #End initialisation
7           #######
8           all_Returns = []
9           total_reward = []
10          #For each batch
11          for batch in range(batches):
12              Returns = [[None for a in range(grid.action_size)] for s in ...
                   range(self.state_size)]
13
14              for ep in range(episodes):
15                  G=0
16                  trace = self.generate_trace(policy) #new trace from new ...
                       starting loc every episode
17                  trace_reward = self.calculate_trace_reward(trace)
18                  total_reward.append(trace_reward)
19                  seen_pairs = np.ones((self.state_size, self.action_size))
20                  for step in trace[::-1]:
21                      G = gamma * G + float(step[2])
22
23                      action_idx = self.action_names.index(step[1])
```

```
                    if seen_pairs[step[0],action_idx]>0:
                        Returns[step[0]][action_idx] = G
                        seen_pairs[step[0],action_idx] = 0
            #batch_Returns, trace_reward = self.MC_evaluation(policy = ...
               greedy_applied_PolicyQ, episodes = episodes, discount = gamma)
            all_Returns.append(Returns.copy())

            for s in range(self.state_size):
                for a in range(self.action_size):
                    state_action_pair_values = []
                    for batch_idx in range(len(all_Returns)):
                        if all_Returns[batch_idx][s][a] != None:
                            state_action_pair_values.append(all_Returns[batch_idx][s][a
                    if len(state_action_pair_values) != 0:
                        Q[s][a] = ...
                           sum(state_action_pair_values)/len(state_action_pair_values)

            policy = self.apply_e_greedy_Q(Q,epsilon)

        V = np.average(Q, axis=1)

        return V, Q, policy, total_reward
```

- Iterative Learning

```
    def MC_Iterative(self, iterations, alpha, epsilon=0, ...
       update_epsilon=False, gamma=gamma_cid):
        #initialisation
        initial_epsilon = epsilon
        Q = np.zeros((self.state_size,self.action_size))
        Q[:,0]=1
        greedy_applied_PolicyQ = self.apply_e_greedy_Q(Q,epsilon=epsilon)
        #End initialisation
        total_reward = []
        Vs = []
        #for i in tqdm(range(iterations)):
        for i in range(iterations):
            R = 0
            trace = self.generate_trace(greedy_applied_PolicyQ)
            total_reward.append(self.calculate_trace_reward(trace))

            seen_pairs = np.ones((self.state_size, self.action_size))
            for step in trace[::-1]:
                action_idx = self.action_names.index(step[1])
                if seen_pairs[step[0],action_idx]>0:
                    R = gamma*R + float(step[2])
                    Q[step[0]][action_idx] = Q[step[0]][action_idx] + ...
                       alpha*(R - Q[step[0]][action_idx])
                    seen_pairs[step[0],action_idx] = 0
            if update_epsilon:
                epsilon = initial_epsilon*(math.exp(-i/400))
            greedy_applied_PolicyQ = self.apply_e_greedy_Q(Q, epsilon)
            V = self.Q2V(Q, epsilon)
            Vs.append(V)

        return Vs, Q, greedy_applied_PolicyQ, np.asarray(total_reward, ...
           dtype=np.float32)
```

TD Learning - SARSA

```
1         def SARSA(self, iterations, alpha, epsilon=0, ...
              update_epsilon=True, gamma=gamma_cid):
2             R = self.get_reward_matrix()
3             locations, neighbours, absorbing = self.get_topology()
4             initial_epsilon = epsilon
5
6             Q = np.random.rand(self.state_size, self.action_size)
7             for loc in self.absorbing_locs:
8                 Q[self.loc_to_state(loc,self.locs)]=0
9
10            total_rewards = []
11            Vs = []
12            for i in range(iterations):
13                self.init_state = self.get_random_loc()
14                state_index = self.loc_to_state(self.starting_loc, self.locs)
15
16                total_reward = 0
17                policy = self.apply_e_greedy_Q(Q, epsilon=epsilon)
18                action_index = random.choices([0,1,2,3], ...
                      policy[state_index])[0]
19                if update_epsilon:
20                    epsilon = initial_epsilon*(math.exp(-i/400))
21                while True:
22                    if state_index in absorbing[0].nonzero()[0]:
23                        total_rewards.append(total_reward)
24                        break
25
26                    effect_probability = ...
                          self.action_randomizing_transition_matrix[action_index]
27                    effect = random.choices([0,1,2,3], effect_probability)[0]
28
29                    destination_index = int(neighbours[state_index][effect])
30
31                    reward = R[destination_index][state_index][effect]
32                    total_reward = total_reward + reward
33
34                    policy = self.apply_e_greedy_Q(Q, epsilon=epsilon)
35                    action_index_prime = random.choices([0,1,2,3], ...
                          policy[destination_index])[0]
36
37                    Q[state_index][action_index] = ...
                          Q[state_index][action_index] + alpha * (reward + ...
                          gamma * Q[destination_index][action_index_prime] - ...
                          Q[state_index][action_index])
38                    V = self.Q2V(Q, epsilon)
39                    state_index = destination_index
40                    action_index= action_index_prime
41
42                Vs.append(V)
43
44            return Vs, Q, policy, np.asarray(total_rewards, dtype=np.float32)
```

Support Functions

```
1     def apply_e_greedy_Q(self, Q, epsilon):
2         _, neighbours, _ = self.get_topology()
```

```
3           new_policy = np.full((self.state_size, ...
                self.action_size),epsilon/self.action_size)
4           immediate_reward=np.full(self.state_size, self.default_reward)
5
6           for i in range(len(self.absorbing_locs)):
7               immediate_reward[self.loc_to_state(self.absorbing_locs[i],self.locs)]...
8               =self.special_rewards[i]
9
10          for state_idx in range(self.state_size):
11              immediate_rewards = [immediate_reward[int(neighbour)] for ...
                    neighbour in neighbours[state_idx]]
12              value_of_neighbours = ...
                    list([Q[state_idx][j]+immediate_rewards[j] for j in ...
                    range(len(immediate_rewards))])
13              new_policy[state_idx, np.argmax(value_of_neighbours)] = 1 - ...
                    epsilon + epsilon/self.action_size
14          return new_policy
15
16      def calculate_trace_reward(self,trace):
17          np_trace = np.array(trace)
18          rewards = np.asarray(np_trace[:,2], dtype=np.float32)
19          total = rewards.sum()
20          return total
21
22      def Q2V(self, Q, epsilon=0):
23          Vs=[]
24          for s in range(self.state_size):
25              V = (1-epsilon)*max(Q[s]) ...
                    +(epsilon*np.sum(Q[s]))/self.action_size
26              Vs.append(V)
27          return np.array(Vs)
28
29      def get_random_loc(self):
30          return random.choice(list(filter(
31              lambda x: x not in self.obstacle_locs and x not in ...
                    self.absorbing_locs,
32              [(i,j) for i in range(6) for j in range(6)])))
33
34      def generate_trace(self, policy):
35          locations, neighbours, absorbing = self.get_topology()
36          R = self.get_reward_matrix()
37          # get random starting location
38          self.starting_loc = self.get_random_loc()
39
40          state_index = self.loc_to_state(self.starting_loc, self.locs)
41          trace = []
42          while True:
43              if state_index in absorbing[0].nonzero()[0]:
44                  return trace
45
46              action = random.choices([0,1,2,3], policy[state_index])
47              effect_probability = ...
                    self.action_randomizing_transition_matrix[action][0]
48              effect = random.choices([0,1,2,3], effect_probability)[0]
49
50              destination_index = int(neighbours[state_index][effect])
51              trace.append((state_index, ["N", "E", "S", "W"][effect], ...
                    R[destination_index][state_index][effect]))
52              state_index = destination_index
```