

In the name of God



University of Tehran  
College of Engineering  
Faculty of ECE



# Intelligent Systems

## HW6

**Reza Jahani**

**810198377**

Fall 1401

## Table of Contents

<b>Model Based Reinforcement Learning – Analytical</b>	<b><i>3</i></b>
<b>Model Based Reinforcement Learning – Implementation</b>	<b><i>8</i></b>
<b>Model Free Reinforcement Learning – Implementation</b>	<b><i>12</i></b>
<b>Additional Section</b>	<b><i>21</i></b>

## Question 1) Model Based Reinforcement Learning – Analytical

In this problem it is aimed to solve a Model Based RL problem employing policy iteration algorithm. This algorithm consists of 2 steps. Model Evaluation & Model Improvement. Concerning the former, a random policy is initiated as  $\pi$ . For each state, value is calculated using the bellman equation presented in eq. 1.1. This task is done until convergence. In the latter previous policy is altered in a way to improve the model. This task is conducted by changing the actions in state (What policy actually means) in a way which makes sense towards the higher values.

The environment and number of each state alongside the initial values of states is presented in figure 1.1.

$$V(s) = \sum_a \pi(a, s) \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')] \quad (eq. 1.1)$$

0 <sup>3</sup>	0 <sup>5</sup>	0 <sup>8</sup>	3
0 <sup>2</sup>	0 <sup>4</sup>	0 <sup>7</sup>	-2
0 <sup>1</sup>		0 <sup>6</sup>	0 <sup>9</sup>

Figure 1.1: Environment

Green values in each state represent the value of the state which are initialized with zero first except terminal points. Number of state is determined by the blue color and the reward for all states even the terminal points are zero (This is the given data by the question.).

Here is what we'll do further.

- Initialize state values, random policy  
# Model Evaluation
- Calculate  $V(s)$  for all  $s \in S$ . (eq. 1.1)
- Iterate till convergence of the  $V(s)$  values  
# Model Improvement
- $\pi' \rightarrow \pi$

Since these iterations are so monopolizing,  $V(s)$  calculation is only repeated once (In general it should be repeated till convergence). Now we perform overall iterations of policy iteration.










			3
			-2
			

Figure 1.2: Initial Policy

$$p(\text{main action}) = 0.6, p'(\text{side action}) = 0.2, \gamma = 0.2$$

▪ Iteration 1

➤ Policy Evaluation (Iteration 1)

for  $s = 1, 2, \dots, 6 \rightarrow R_{ss'}^a = 0, V(s') = 0 \rightarrow V(s) = 0$

$$s = 7; V(s) = p[r_{s_6} + \gamma \times V(s_6)] + p'[r_{s_4} + \gamma \times V(s_4)] + p'[r_{\text{terminal } 2} + \gamma \times V(s_{\text{terminal}})] = \\ = 0.6[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times -2] = -0.108$$

Values for the states are calculated as above. We continue with this approach.

$$s = 8; V(s) = 0.6[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times 3] = 0.162$$

$$s = 9; V(s) = 0.6[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times 0] + 0.2[-2 + 0.2 \times 0] = -0.108$$

0 <sup>3</sup>	0 <sup>5</sup>	0.162 <sup>8</sup>	3
0 <sup>2</sup>	0 <sup>4</sup>	-0.108 <sup>7</sup>	-2
0 <sup>1</sup>		0 <sup>6</sup>	-0.108 <sup>9</sup>

Figure 1.3: Values after 1 iteration

➤ Policy Improvement

→	→	→	3
←	←	↑	-2
→		↑	←

Figure 1.4:  $\pi'$  (Better Policy based on current values)

$$\pi = \pi'$$

▪ Iteration 2

➤ Policy Evaluation (Iteration 1)

for  $s = 1, 2, 3, 4$ ;  $R_{ss'}^a = 0, V(s') = 0 \rightarrow V(s) = 0$

$s = 5$ ;  $V(s) = 0.6[0 + 0.2 \times 0.162] + 0.2[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times 0] = 0.217$

$s = 6$ ;  $V(s) = 0.6[0 + 0.2 \times -0.108] + 0.2[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times -0.108] = -0.02$

$s = 7$ ;  $V(s) = 0.6[0 + 0.2 \times 0.162] + 0.2[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times -2] = -0.086$

$s = 8$ ;  $V(s) = 0.6[0 + 0.2 \times 3] + 0.2[0 + 0.2 \times 0.162] + 0.2[0 + 0.2 \times -0.086] = 0.4$

$s = 9$ ;  $V(s) = 0.6[0 + 0.2 \times -0.02] + 0.2[0 + 0.2 \times -2] + 0.2[0 + 0.2 \times -0.108] = -0.11$

0 <sup>3</sup>	0.217 <sup>5</sup>	0.4 <sup>8</sup>	3
0 <sup>2</sup>	0 <sup>4</sup>	-0.086 <sup>7</sup>	-2
0 <sup>1</sup>		-0.02 <sup>6</sup>	-0.11 <sup>9</sup>

Figure 1.5: Values after 2 iterations

➤ Policy Improvement

→	→	→	3
←	↑	↑	-2
→		↑	←

Figure 1.6:  $\pi'$   
 $\pi = \pi'$

▪ Iteration 3

➤ Policy Evaluation (Iteration 1)

for  $s = 1, 2$ ;  $R_{ss'}^a = 0, V(s') = 0 \rightarrow V(s) = 0$

$s = 3$ ;  $V(s) = 0.6[0 + 0.2 \times 0.217] + 0.2[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times 0] = 0.03$

$s = 4$ ;  $V(s) = 0.6[0 + 0.2 \times 0.217] + 0.2[0 + 0.2 \times 0] + 0.2[0 + 0.2 \times -0.086] = 0.024$

$s = 5$ ;  $V(s) = 0.6[0 + 0.2 \times 0.4] + 0.2[0 + 0.2 \times 0.024] + 0.2[0 + 0.2 \times 0.217] = 0.0667$

$s = 6$ ;  $V(s) = 0.6[0 + 0.2 \times -0.086] + 0.2[0 + 0.2 \times -0.02] + 0.2[0 + 0.2 \times -0.11] = -0.01$

$s = 7$ ;  $V(s) = 0.6[0 + 0.2 \times 0.4] + 0.2[0 + 0.2 \times 0.24] + 0.2[0 + 0.2 \times -2] = -0.041$

$s = 8$ ;  $V(s) = 0.6[0 + 0.2 \times 3] + 0.2[0 + 0.2 \times 0.4] + 0.2[0 + 0.2 \times -0.041] = 0.42$

$s = 9$ ;  $V(s) = 0.6[0 + 0.2 \times -0.01] + 0.2[0 + 0.2 \times -2] + 0.2[0 + 0.2 \times -0.11] = -0.12$

3 0.03	5 0.067	8 0.42	3
2 0	4 0.024	7 -0.041	-2
1 0		6 -0.01	9 -0.12

Figure 1.7: Values after 3 iterations

➤ Policy Improvement

→	→	→	3
↑	↑	↑	-2
↑		↑	←

Figure 1.8:  $\pi'$   
 $\pi = \pi'$

After 3 iterations of policy iteration algorithm, final optimal policy can be observed in figure 1.10.

3 0.03	5 0.067	8 0.42	3
2 0	4 0.024	7 -0.041	-2
1 0		6 -0.01	9 -0.12

Figure 1.9: Final State Values after 3 iterations of policy iteration

→	→	→	3
↑	↑	↑	-2
↑		↑	←

Figure 1.10: Optimal  $\pi(s, a)$  (Policy)

## Question 2) Model Based Reinforcement Learning – Implementation

In this section we are aiming to solve the gambler problem using the value iteration algorithm to find the optimal policy. In this problem, the agent wants to bet in every set on heads of a coin which probability of appearing head is  $p_H$ . The agent keeps betting until whether full success is achieved by winning 100\$ or to lose all the money. Below the parameters of this problem involving the states, rewards, actions, and the policy is defined.

State of the agent depends on value of possessed money. Since the agent starts with a certain amount of money and tries to win 100\$ maximum, last state is 99. As defined in the question, reward is +1 when the agent wins the betting and 0 in other cases. Agent also wants to bet a certain amount in every state and the minimum value of betting can be 0 and maximum value for bet is the amount of money that agent owns with condition that he won't bet more than 50 since he wants to win 100\$. Because if the agent is in state 51 and bets 51 and also wins, the hypothesis of the question will be denied. Obviously, the policy defines how much to bet when the agent has a certain amount of money.

- States:  $s \in \{1, 2, 3, 4, \dots, 99\}$
- Rewards:  $r \in \{0, +1\}$ 
  - +1: Winning
  - 0: Else
- Actions:  $a \in \{0, 1, 2, 3, 4, \dots, \min(100 - s, s)\}$
- Policy:  $\pi(s, a) = x$ 
  - Bet  $x$ , when in state  $s$

After training the agent in the environment the policy is obtained for this model and the results can be observed as a plot of amount of bet per money that our agent has in every situation.

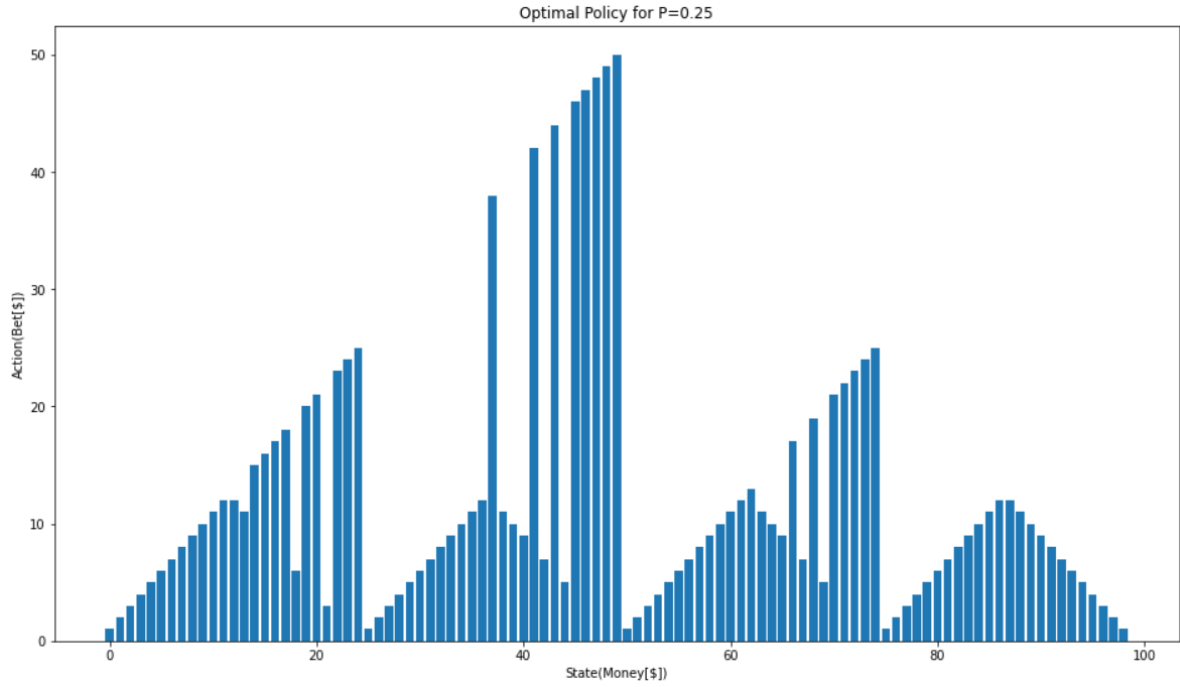


Figure 2.1: Optimal Policy for  $p_H = 0.25$



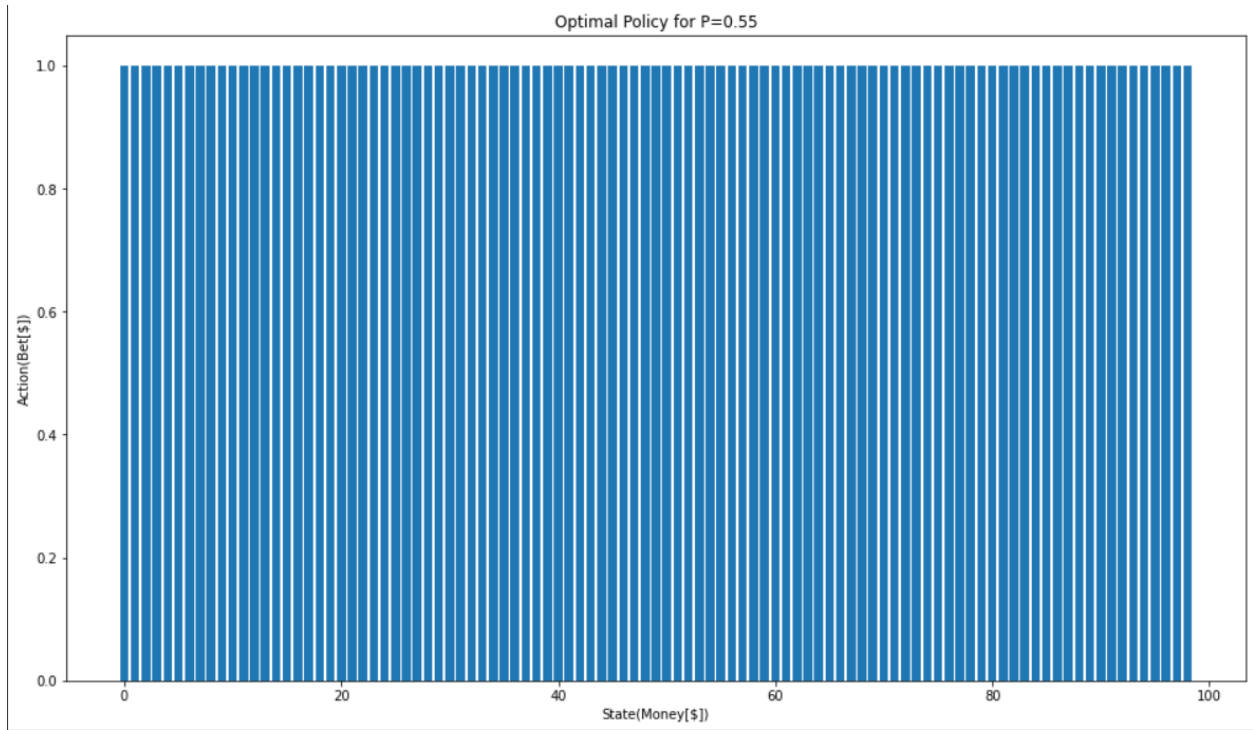


Figure 2.2: Optimal Policy for  $p_H = 0.55$

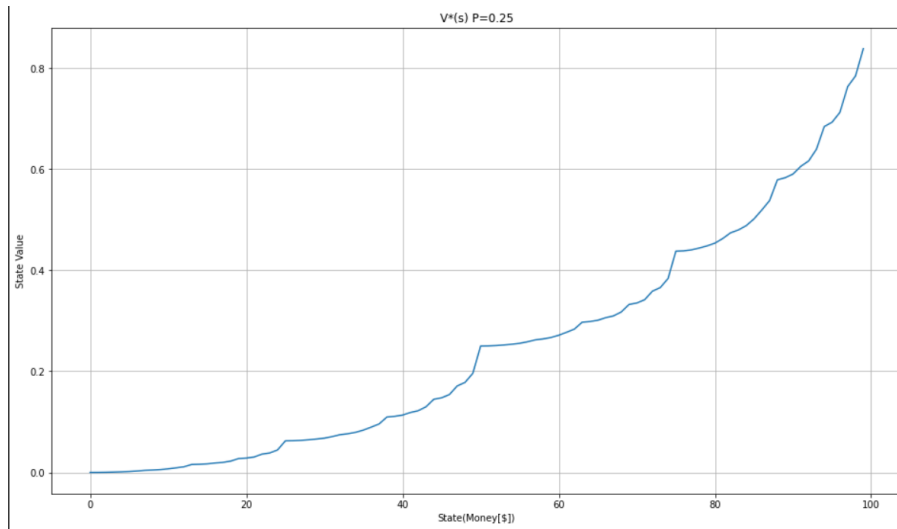


Figure 2.3: State Values for  $p_H = 0.25$

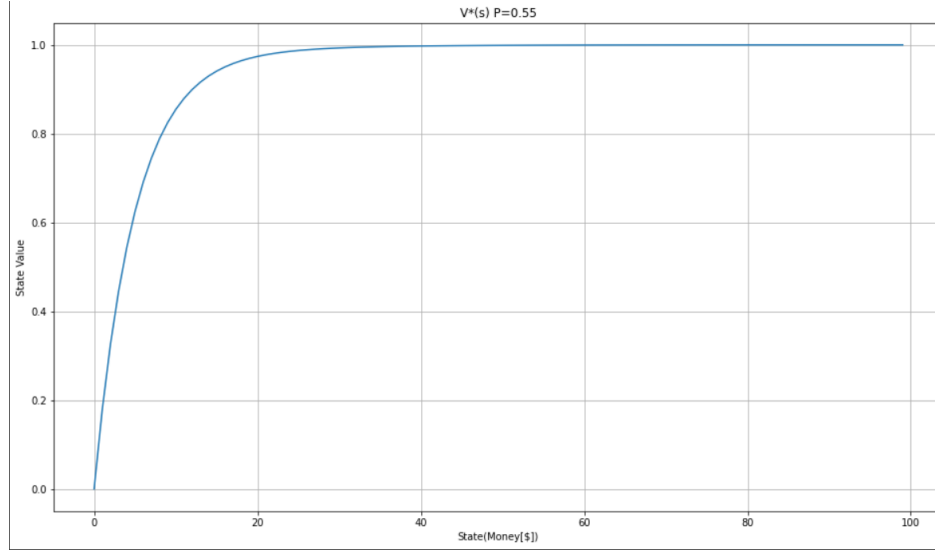


Figure 2.4: State Values for  $p_H = 0.55$

### Analysis of the Results:

As a short explanation about the difference between these policies when these two values of probabilities as 0.25 and 0.55 is presented below.

When  $p_H = 0.55$  the probability is somehow uniform and since we don't know what will happen next, best choice is to invest 1 unit of the money in each state. On the other hand, when  $p_H = 0.25$ , the probability is not uniform and depending on how much money the agent has, it is better to sometimes risk and bet more money to gain higher stake.

In addition, about the state values plot it is obvious that in first case where  $p_H = 0.25$ , as the agent reaches higher states and have more money, it is more valuable since more risk is possible. On the contrary, when  $p_H = 0.55$  and somehow uniform distribution is in practice, values get saturated from a certain state to the end and all of them has the same value since the agent has a certain money and has to bet 1\$ each time.

### Implementation Algorithm:

In this part short explanations about the process of implementation and the algorithm is presented. As explained in the beginning of the question, we have defined states, actions, rewards and other necessary parameters for solving the problem.

We start with those parameters again.

- States:  $s \in \{1, 2, 3, 4, \dots, 99\}$
- Rewards:  $r \in \{0, +1\}$ 
  - +1: Winning
  - 0: Else
- Actions:  $a \in \{0, 1, 2, 3, 4, \dots, \min(100 - s, s)\}$
- Policy:  $\pi(s, a) = x$ 
  - Bet  $x$ , when in state  $s$

- *Initialize  $V(s)$  for all  $s$  in states*
  - *For  $s$  in states*
- *Calculate  $Q(s, a)$  for all possible actions in state  $s_i$*
- *Keep the maximum  $Q(s, a)$  as the value of that state ( $V(s)$ )*
- *This process is continued till convergence of  $V(s)$  values.*
  - *Get the optimal Policy*
  - *For  $s$  in states*
- *Calculate  $Q(s, a)$*
- $\pi(s, a) = \operatorname{argmax}_a Q(s, a)$

### Value Iteration Algorithm

Above, the algorithm employed for value iteration approach is explained. Below in figures the modular code can be observed.

```
global theta
global discount_factor
theta = 1e-4
discount_factor = 1

# This function is employed to calculate the maximum value for a state and optimal action
def Q_star(state, values, rewards, p):
    actions = range(1, min(state, 100-state)+1)
    action_values = []
    for i in range(len(actions)):
        action_val = p*(rewards[state+actions[i]]+discount_factor*values[state+actions[i]]) + (1-p)*(rewards[state-actions[i]]+discount_factor*values[state-actions[i]])
        action_values.append(action_val)
    V = np.max(action_values)
    a = np.argmax(action_values) + 1
    return V, a

# This function is used to calculate the optimal value for states
def State_Values(p):
    rewards = np.zeros(101)
    rewards[100] = 1
    values = np.zeros(101)
    states = np.arange(1, 100)

    while True:
        delta = 0
        for i in range(len(states)):
            V, _ = Q_star(states[i], values, rewards, p)
            delta = max(delta, np.abs(values[i+1]-V))
            values[i+1] = V
        if delta < theta:
            break
    return values
```

Figure 2.3

As observed in figure 2.3, theta which is used for stopping condition and discount factor are initialized as global variables for further uses.

### Q\_star Function:

First function gets state and state values and rewards and  $p_H$  as the input and calculates  $Q(s, a)$  for all actions using bellman equation. As already explained actions are limited depending on which state the agent is in. Then in a for loop the Q is calculated for each action. After this process, the maximum value is stored as the value of the state ( $V(s)$ ) and the argument which makes this parameter maximum will be the optimal action so far. Outputs of this function are  $V(s), a^*$ .

$s$ : current state (current money of agent)  
 $a$ : action (bet amount)  
 $s'$ : next state (money after this round of betting)  
 Coin is tossed  $\rightarrow \begin{cases} p_H \rightarrow \text{agent wins} \rightarrow s' = s + a \\ 1 - p_H \rightarrow \text{agent loses} \rightarrow s' = s - a \end{cases}$

$$Q(s, a) = \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')]$$

#### State\_Value Function:

This function is used to calculate the final and optimum values for each state. This function only gets  $p_H$  as the input and employs the previous function to calculate values of each state based on the previously explained method (Checking all possible action and keeping the maximum obtained value). The delta term is used to stop the solving whenever the equations were converging.

```
# This function uses the values of the states to find the optimal policy
def Policy_Finder(values,p):
    rewards = np.zeros(101)
    rewards[100] = 1
    Optimal_Actions = []
    states = range(1,100)
    for i in range(len(states)):
        _,a = Q_star(states[i],values,rewards,p)
        Optimal_Actions.append(a)
    return Optimal_Actions

# General module for value iteration algorithm on this problem
def Module(p):
    values = State_Values(p)
    policy = Policy_Finder(values,p)
    return policy,values
```

Figure 2.4

#### Policy\_Finder Function:

This function gets the optimal values for states and using that, optimal action for each state is predicted. It passes on each state and using the Q\_star function and getting the optimum action based on previous explanations.

#### Module Function:

This is the top level function which organizes the complete algorithm. First state values are computed and then the best policy is figured.

## Question 3) Model Free Reinforcement Learning – Implementation

### Section 1) Solving with Q-Learning – Random Based

In this section it is aimed to begin at a random point, then generate random action in an infinite loop. This loop continues until the taxi succeeds in picking up the fare, and dropping at the specified destination. In this process, the rewards are accumulated and the termination point for the progress determined with the flag variable of *done*.



```
[21] import gym

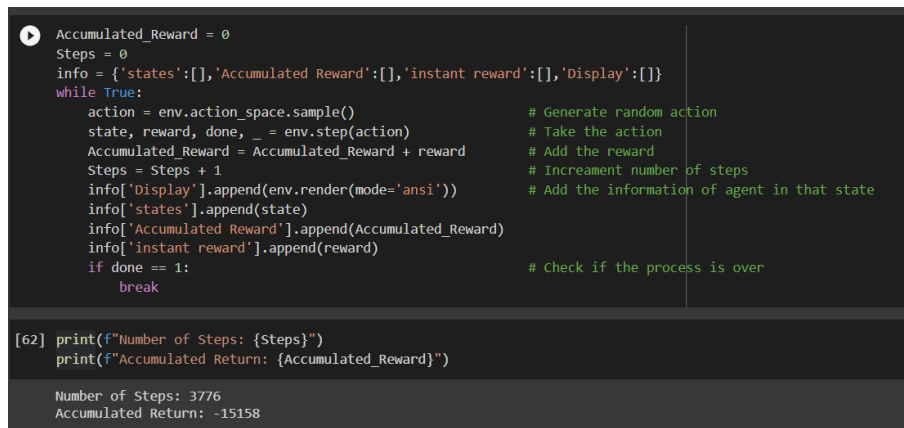
env = gym.make("Taxi-v3").env

env.reset()
print(env.render(mode='ansi'))
```

```
+-----+
|R: | : :G| |
| : | : : |
| : | : : |
| : | : : |
|Y| : |B: |
+-----+
```

Figure 3.1.1: Initial Environment

As observed above in figure 3.1.1, gym library is imported and the environment of the game is generated, and after the reset method, the game field is pictured.



```
Accumulated_Reward = 0
Steps = 0
info = {'states':[], 'Accumulated_Reward':[], 'instant_reward':[], 'Display':[]}
while True:
    action = env.action_space.sample() # Generate random action
    state, reward, done, _ = env.step(action) # Take the action
    Accumulated_Reward = Accumulated_Reward + reward # Add the reward
    Steps = Steps + 1 # Increment number of steps
    info['Display'].append(env.render(mode='ansi')) # Add the information of agent in that state
    info['states'].append(state)
    info['Accumulated_Reward'].append(Accumulated_Reward)
    info['instant_reward'].append(reward)
    if done == 1: # Check if the process is over
        break

[62] print(f"Number of Steps: {Steps}")
    print(f"Accumulated Return: {Accumulated_Reward}")
```

```
Number of Steps: 3776
Accumulated Return: -15158
```

Figure 3.1.2: Random Progress of the Taxi

In figure 3.1.2, the overall progress of the random based game is presented.

**Accumulated Reward** is initiated with zero and is employed for presenting the general reward obtained throughout the game. **Steps** is initiated with zero and represents the number of this random steps necessary for accomplishment of the task. **Info** dictionary is initiated to save the render in each step for further display.

The loop begins with choosing a random action among all possible action for the agent. Storing the **reward** obtained by the action and the **done** flag to check whether the task is over or not, it is added to the **Accumulated Reward**. **Step** number is incremented by 1 and current info is added to the dictionary. In the end, using the **done** flag, it is checked whether to stop the loop or not.

As observed in figure 3.1.2, after the random process of taking a random action and storing further states, number of steps taken to accomplish the task is presented alongside the overall reward achieved by the agent throughout the game.

- ❖ The agent has taken 3776 steps.
- ❖ The agent has achieved -15158 reward which means the agent has been punished for the actions by the unit of 15158.

```
for i in range(Steps):
    clear_output(wait=True)
    print(f"Step {i+1}")
    print(info['Display'][i])
    print(f"State: {info['states'][i]}")
    print(f"Instant Reward: {info['instant reward'][i]}")
    print(f"Accumulated Reward: {info['Accumulated Reward'][i]}")
    sleep(0.5)
```

Figure 3.1.3: Code employed for display

In figure 3.1.3, the code which is used for displaying the game environment and state and the instant reward achieved in addition to the accumulated reward or punishment term is printed in each step. In overall, the graphical movement of the agent (taxi) can be seen alongside other information of the agent.

Some random pictures of this graphical movement of the agent is presented in figure 3.1.4.



Figure 3.1.4

After implementing only in one episode, we do this trial in more episodes to observe the number of necessary steps to finish the task and the accumulated rewards in total.

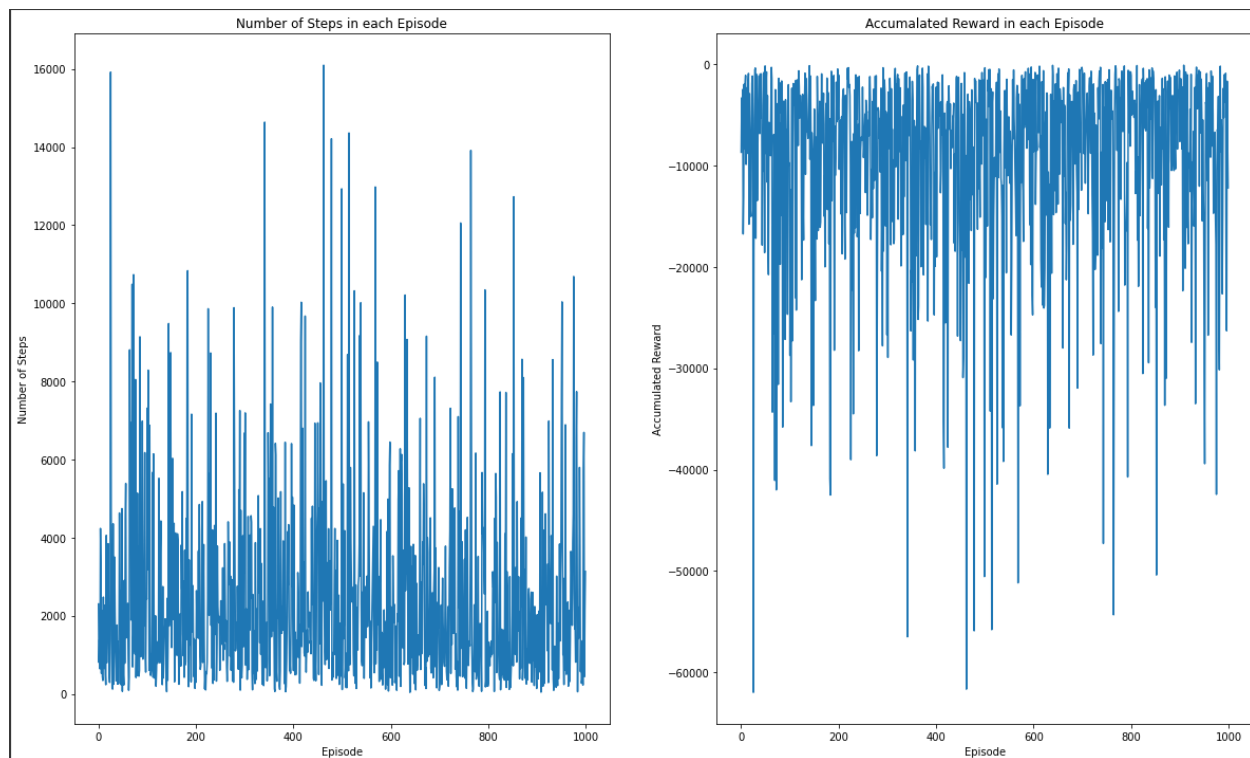


Figure 3.1.5: Number of Steps & Reward in Episodes

As it can be observed above in figure 3.1.5, since the whole program is based on random actions all the time, number of necessary steps to accomplish the task varies in different episodes and accordingly, the reward is different.

## Section 2) Solving with Q-Learning – Intelligent Progress

In this section we aim to solve the problem using Q-Learning method. Before we observe the code and explain the algorithm employing the code, we go through the theoretical algorithm.

- Initialize the Q Table employed for choosing the action in each state.
  - Iterate on episodes
- Explore the environment by taking the actions and update the Q Table values using below equation.
- $Q(s, a) = Q(s, a) + \alpha \left( R_{ss'}^a + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$
- *Iterate until convergence*

### Q-Learning Algorithm

```
[81] env = gym.make("Taxi-v3").env

[86] print("Action Space {}".format(env.action_space))
     print("State Space {}".format(env.observation_space))

     Action Space Discrete(6)
     State Space Discrete(500)

[159] num_action = 6
      num_state = 500
      num_episodes = 2000
      epsilon = 0.1
      alpha = 0.1
      discount_factor = 0.8

[160] Q_Table = np.zeros((num_state, num_action))
```

Figure 3.2.1: Initial Steps

In figure 3.2.1, the environment is generated and the Q Table is initialized with correct shapes as are (number of state, actions). Then number of episodes and epsilon and alpha and discount factor are initialized as hyper parameters.



```

Num_Steps = []
Reward_total = []
for i in range(num_episodes):
    env.reset()
    ps = env.s
    step = 0
    Reward_ = 0
    while True:
        if np.random.uniform(0,1) < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(Q_Table[ps,:])

        ns, reward, done, _ = env.step(action)
        Reward_ = Reward_ + reward
        step = step + 1

        if done==1:
            break

        Q_Table[ps,action] = Q_Table[ps,action] + alpha*(reward + discount_factor*np.max(Q_Table[ns,:]) - Q_Table[ps,action])
        ps = ns

    Num_Steps.append(step)
    Reward_total.append(Reward_)

```

Figure 3.2.2: Q Learning Algorithm Implemented

Number of steps and Reward total is initiated as an empty list to append the step number and reward in each episode. After resetting the environment and getting the first state the agent enters the infinite loop. Employing the epsilon greedy algorithm, we take the action according to the probability which will be explained later how. Then the agent takes the action and it gains a reward and transmits to next state. Obtained reward is added to the total reward of the agent in that episode and the number of step is incremented. Applying the equation presented in Q-Learning algorithm, the Q-Table is updated and this loop goes on until the end of the episode which will be accomplishing the task of dropping the fare at the specified location.

Below we explain the concept behind epsilon greedy approach.  
In each state for taking the action, the agent has 2 options.

$$action(s) = \begin{cases} prob. = \epsilon & argmax_a \{Q(s, a)\} \\ prob. = 1 - \epsilon & Random\ action \end{cases}$$

The aim behind this approach is to let the agent observe all the states and possible situation at first since the Q table is not optimal from the beginning.

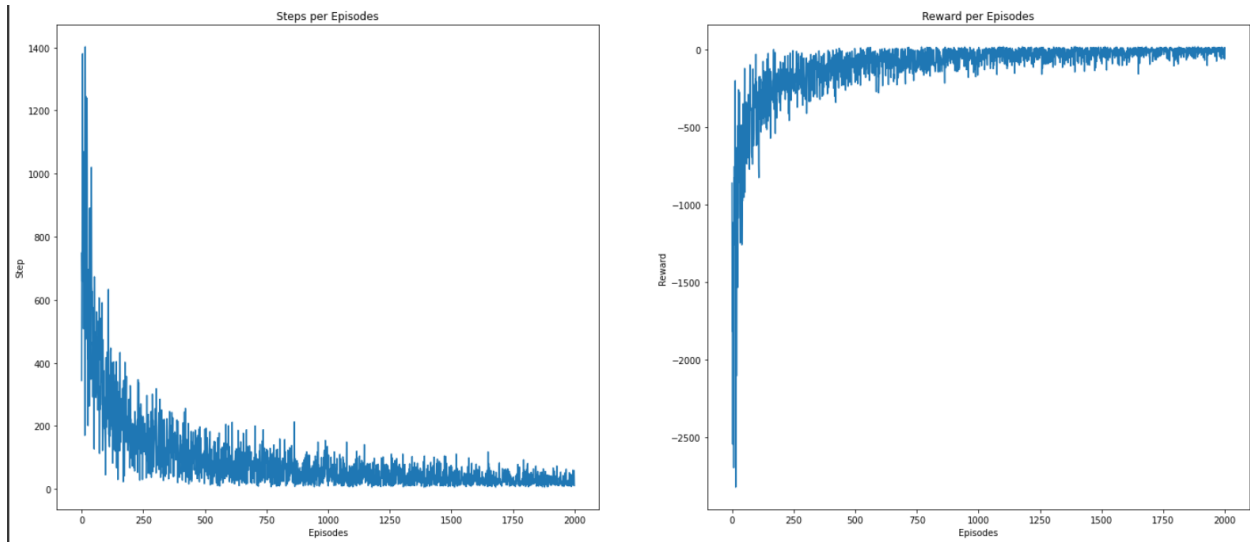


Figure 3.2.3: Steps and Rewards per Episodes

As it can be observed after some number of epochs, the agent has learned the environment and will know how to interact with the environment to maximize its long term reward and to accomplish its task. As the number of episodes go up, number of necessary steps decrease since the agent learns more and the reward increases.

Some more points to mention about the implementation and hyper parameter tuning:

- $\epsilon$  &  $\alpha$  can change by a rate during the training process to optimize the learning more.
- $\epsilon$  &  $\alpha$  are the hyper parameters of the problem so by doing greedy search and looking for the best  $\epsilon$  &  $\alpha$ , convergence and optimality of the Q Table can improve which leads to better results in general.

Also there are other points to mention if we wanted to change the rewards of the state:

- It is possible to gain a better convergence result by changing the rewards in different states. It will be explained as below:
- By changing or in other words, increasing the rewards of locations near fares and destinations, the edges, the agent would learn that getting near those points is better since it would be capable of picking up the fare sooner and dropping the fare earlier and the task would be done better.

## Testing Section:

For further checking, a test case is presented to prove the quality of learning of the agent. So the environment is reset and we can observe the movement of the agent to see what it will do in one episode.

```
[10] env.reset()
      print(env.render(mode='ansi'))

+-----+
|R: | : :G|
| : | : :|
| : :Y: :|
| | : | :|
| | : | :|
|Y| : |B:|
+-----+

test_step = 0
test_reward = 0
state = env.s
info = {'states':[], 'Accumulated Reward':[], 'instant reward':[], 'Display':[]}
while True:
    action = np.argmax(Q_Table[state,:])
    state, reward, done, _ = env.step(action)
    test_step = test_step + 1
    test_reward = test_reward + reward
    info['Display'].append(env.render(mode='ansi')) # Add the information of agent in that state
    info['states'].append(state)
    info['Accumulated Reward'].append(test_reward)
    info['instant reward'].append(reward)
    if done==1:
        break

[14] print(f"Number of steps: {test_step}")
      print(f"Accuired reward by the agent: {test_reward}")

Number of steps: 10
Accuired reward by the agent: 11
```

Figure 3.2.4: Testing the Agent

As observed in figure 3.2.4, the first state can be seen that the taxi is located in the middle of the field while a fare is waiting for the taxi in the above left corner and the destination is located at below left corner.

As already explained in previous sections, step and reward is initialized to keep track of the agent performance. First state of the agent is obtained and a dictionary is employed for saving the information of each step. Entering the infinite loop, the action is chosen depending on the current state based on the optimal computed Q Table. After the action is taken, state changes, reward is acquired, and done flag is used to check the termination condition. Information of each step is stored and other parameters is updated. In the end we can observe that the agent succeeded in finishing the task in 10 steps while obtaining an overall reward of 11.

This fact completely illustrates the intelligence of this approach for solving the problem versus the random approach used in previous section.

To observe the graphical movement of the taxi and taken actions in each step, all the actions alongside the state and acquired reward is presented in figure 3.2.5. In addition, in the notebook, a simple code is provided to see the agent taking the actions in a row to actually observe the movement of the taxi.

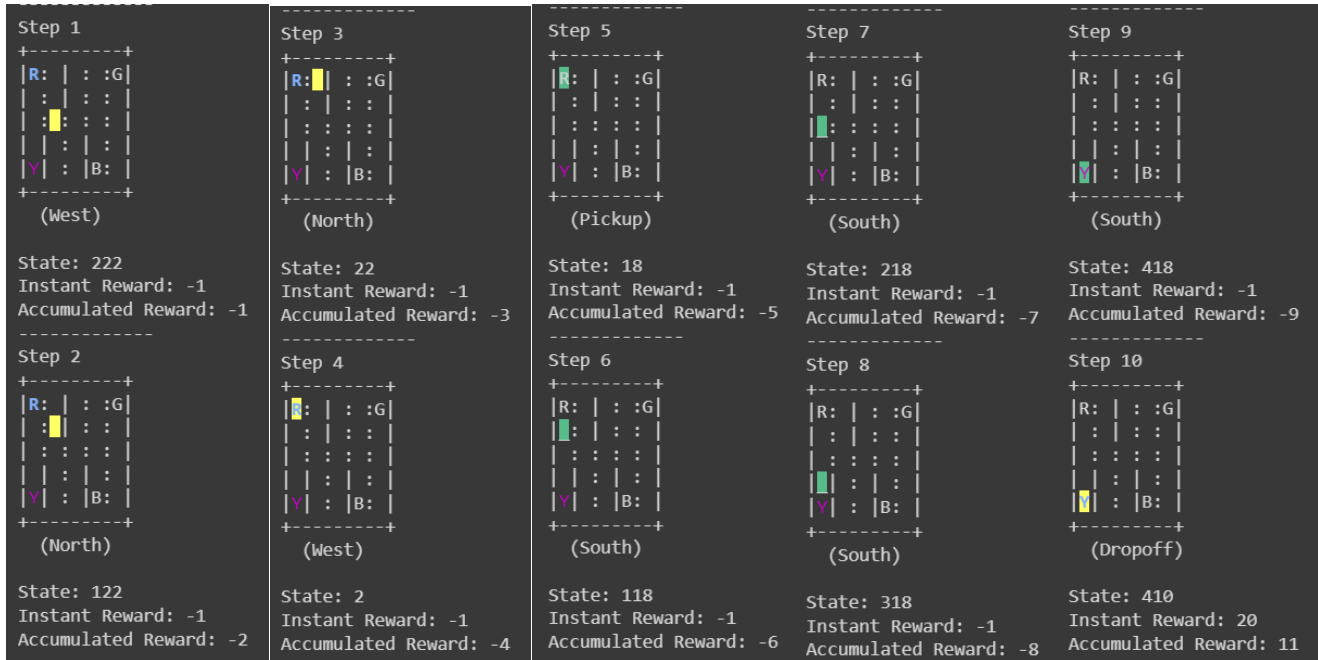


Figure 3.2.5: Agent Performance in Test Case

As it is observed above and based on other explanations, employing Q-Learning, the agent will learn optimally how to act in the environment.

As a comparison to the first approach which did not involve Q-Learning, in this approach the agent learns to not only maximize the long term reward, but also take fewer steps. Below in a bullet list, significant improvements are provided.

- In Q-Learning approach, the agent gets more and better rewards. In other words, the agent reward would be even positive while in the random based steps approach, the reward was negative and the agent were kept being punished.
- In Q-Learning approach, agent takes fewer steps to accomplish the task than the random based actions approach.

## Additional Section

- All of the implementations for this HW are in the google colab file named *HW6\_810198377.ipynb*.
- All questions and their sections and parts are separated with the related headings and titles.