Reinforcement Learning

Project 3

Maurico Gomez Macedo

Michael Bimal

Introduction

Reinforcement Learning is a crucial field in artificial intelligence, concentrating on how agents can optimize cumulative rewards through specific actions within an environment. Temporal Difference (TD) learning methods like Sarsa and Q-learning are especially notable for their capability to learn despite having incomplete knowledge of the environment. We explore a grid world scenario, a widely utilized setup in RL studies. This scenario features a finite, two-dimensional grid where an agent must navigate from an initial position to a terminal state, while avoiding designated penalty areas.

Sarsa: a well-known reinforcement learning algorithm, is employed to optimize a policy that maximizes expected cumulative rewards for an agent engaging with its environment. It is an on-policy learning method, which implies it evaluates the policy it actively adheres to, maintaining a Q-function Q(s, a) that forecasts expected cumulative rewards. This algorithm frequently utilizes an episilon-greedy policy to effectively balance the needs of exploration and exploitation.

Q-learning: Another prominent algorithm in reinforcement learning and operates as an off-policy Temporal Difference (TD) learning method. This means it determines the value of the optimal policy without depending on the agent's current actions, even if the agent is following a different policy at the time. Similar to other methods, Q-learning employs an epsilon-greedy policy for choosing actions, which helps maintain a balance between exploring new possibilities and exploiting known strategies.

The Gradient Monte Carlo Method: It is arguably the most widely used algorithm in reinforcement learning, estimating the value of a state by averaging the returns from numerous episodes originating from that state. This approach depends on complete episodes to determine the value function and does not necessitate a model of the environment.

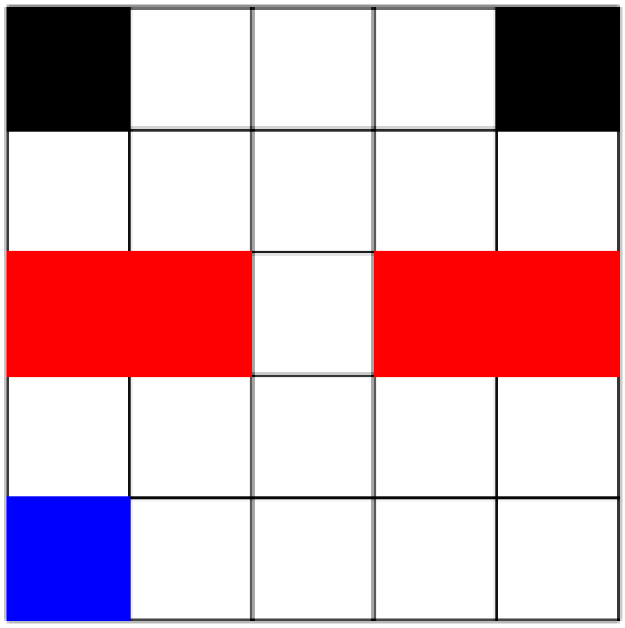
Semi-Gradient TD (0): A temporal difference learning approach that updates the value function using the difference between the current state's estimated value and the value of the next state, weighted by a linear function approximation. This method combines the aspects of Monte Carlo and dynamic programming.

Objective: This project investigates the application of two fundamental reinforcement learning algorithms, Sarsa and Q-learning, within a grid world framework.

Grid world: The agent starts at the blue square and moves to a neighboring state with equal probability. If the agent moves to a red state, it receives a reward of -20 and goes back to the start, i.e., the blue square. A move between any two other states receives a reward of -1. A move that attempts to move outside of the grid receives a reward of -1. The black squares serve as a terminal states.

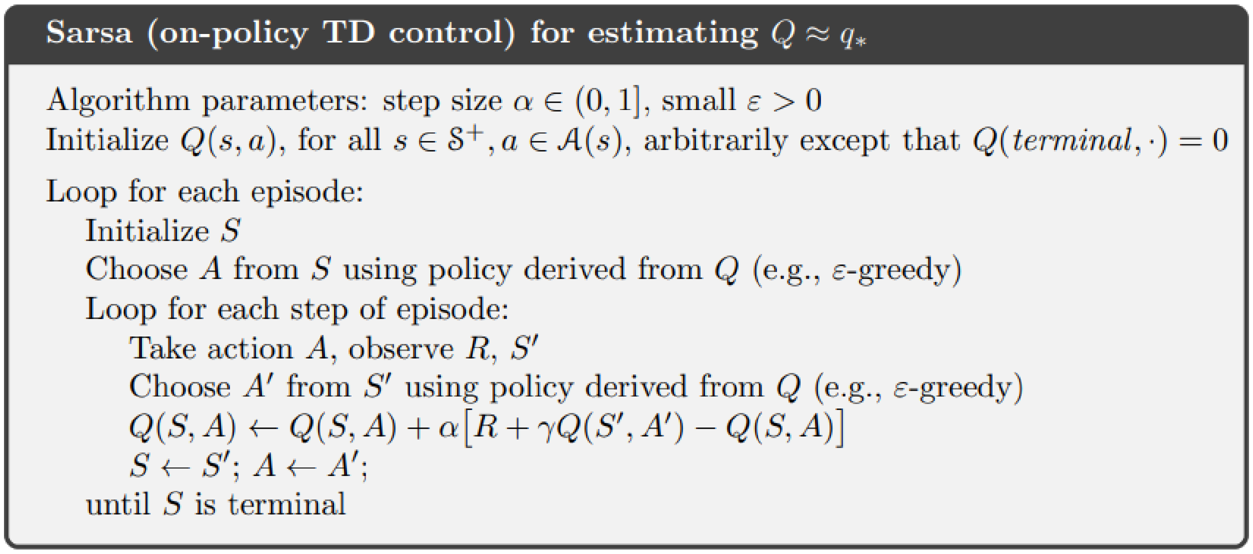
Part 1

Consider the following grid world problem



The agent starts at the blue square and moves to a neighbouring state with equal probability. If the agent moves to a red state, it receives a reward of -20 and goes back to the start, i.e., the blue square. A move between any two other states receives a reward of -1. A move that attempts to move outside of the grid receives a reward of -1. The black squares serve as a terminal states. Intuitively, you can see how the goal here is to pass through the opening in the red "wall" and get to one of the black squares and hence terminate the episode. Use the Sarsa and Q-learning algorithms to learn the optimal policy for this task. Plot a trajectory of an agent utlizing the policy learned by each of the methods. Are they different or similar? Why or why not? You may assume to use ϵ-greedy action selection for this task. How does the sum of rewards over an episode behaves for each of these two methods.

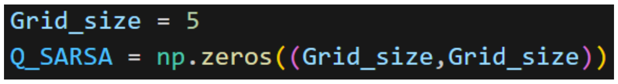
SARSA



```
Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

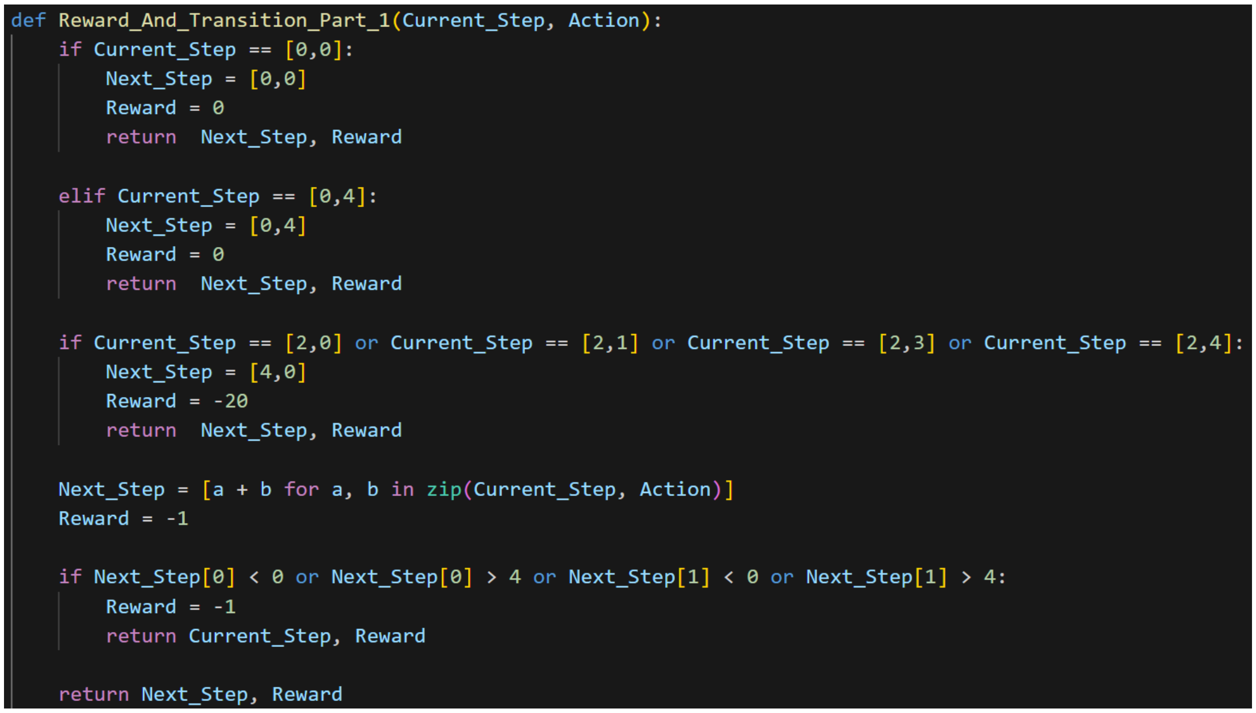
Code explanation

First, we created a matrix of 5 x 5 and a starting value of 0. We are going to update the value function. The Learning rate α is 0.10 and the discount factor γ is 0.95.



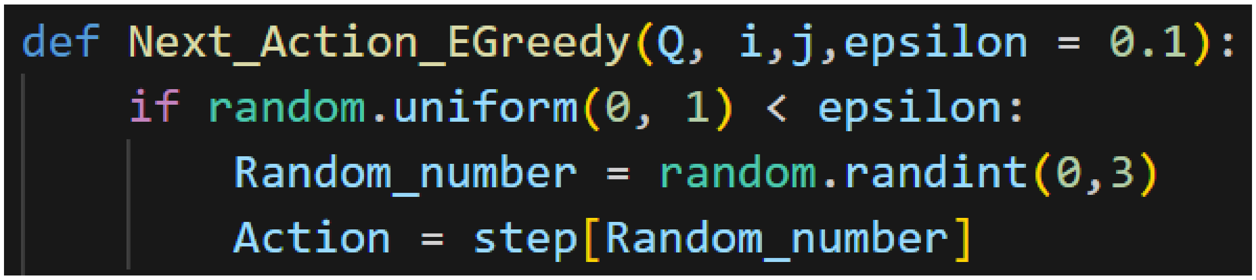
```
Grid_size = 5
Q_SARSA = np.zeros((Grid_size,Grid_size))
```

With the function Reward_And_Transition_Part_1, we will get the reward for each action, and we are going to get the next state. Here, we verify if the agent is in a terminal state, out of boundaries, or in a red or blue cell.



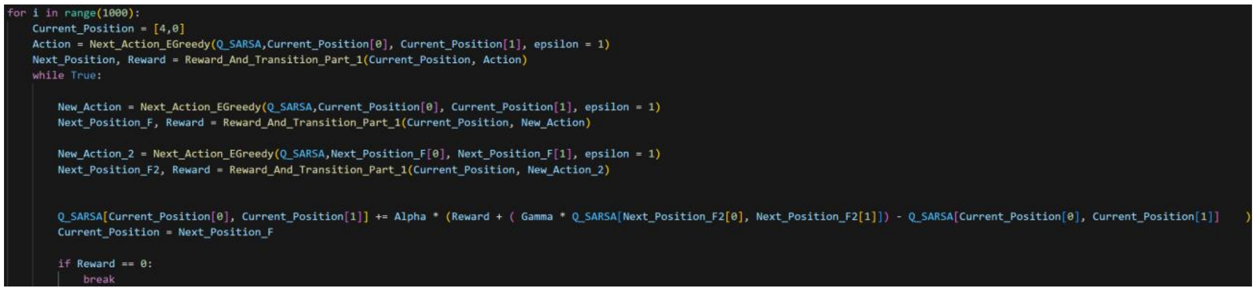
```
def Reward_And_Transition_Part_1(Current_Step, Action):
    if Current_Step == [0,0]:
        Next_Step = [0,0]
        Reward = 0
        return  Next_Step, Reward

    elif Current_Step == [0,4]:
        Next_Step = [0,4]
        Reward = 0
        return  Next_Step, Reward

    if Current_Step == [2,0] or Current_Step == [2,1] or Current_Step == [2,3] or Current_Step == [2,4]:
        Next_Step = [4,0]
        Reward = -20
        return  Next_Step, Reward

    Next_Step = [a + b for a, b in zip(Current_Step, Action)]
    Reward = -1

    if Next_Step[0] < 0 or Next_Step[0] > 4 or Next_Step[1] < 0 or Next_Step[1] > 4:
        Reward = -1
        return Current_Step, Reward

    return Next_Step, Reward
```

The function Next_Action_EGreedy will take the following action, verifying that the agent is not outside boundaries. For the SARSA algorithm, the agent takes random actions, so we only have to set a 1 in the epsilon value, and it will take a random action.



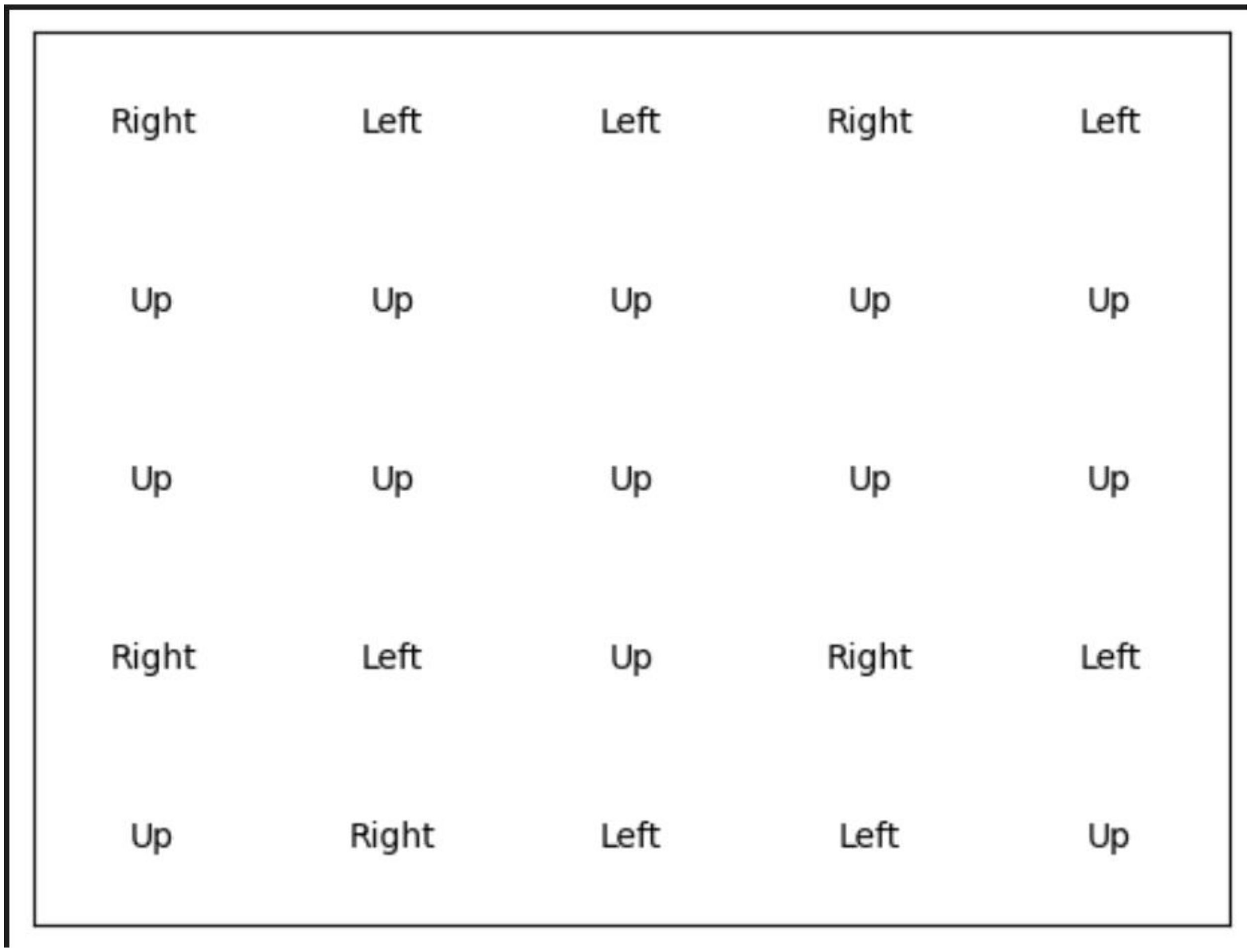
```
def Next_Action_EGreedy(Q, i,j,epsilon = 0.1):
    if random.uniform(0, 1) < epsilon:
        Random_number = random.randint(0,3)
        Action = step[Random_number]
```

We create the loop to update the values for Q_SARSA, set the starting point (4,0), take the first action, the next position, and the Reward, and take the action and the future position as the pseudocode says. For each episode, the agent will move until the Reward equals 0, which means that it has reached a final state.



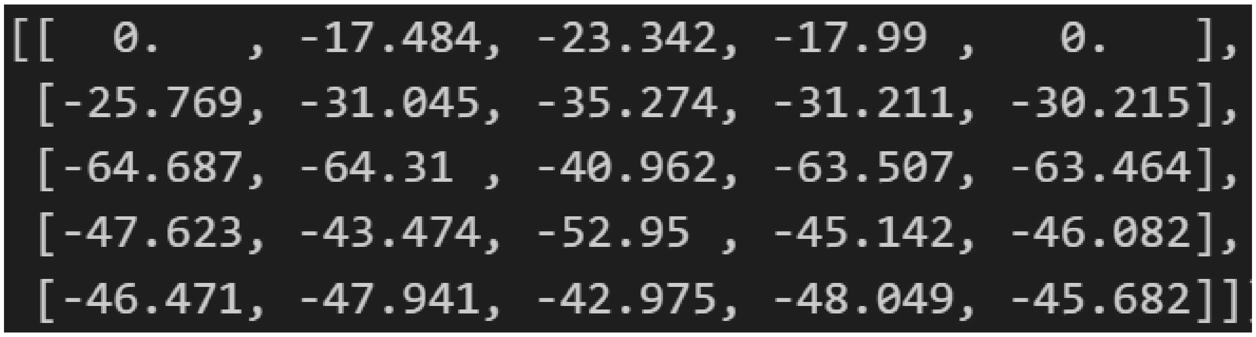
```python
for i in range(1000):
    Current_Position = [4,0]
    Action = Next_Action_EGreedy(Q_SARSA,Current_Position[0], Current_Position[1], epsilon = 1)
    Next_Position, Reward = Reward_And_Transition_Part_1(Current_Position, Action)
    while True:

        New_Action = Next_Action_EGreedy(Q_SARSA,Current_Position[0], Current_Position[1], epsilon = 1)
        Next_Position_F, Reward = Reward_And_Transition_Part_1(Current_Position, New_Action)

        New_Action_2 = Next_Action_EGreedy(Q_SARSA,Next_Position_F[0], Next_Position_F[1], epsilon = 1)
        Next_Position_F2, Reward = Reward_And_Transition_Part_1(Current_Position, New_Action_2)


        Q_SARSA[Current_Position[0], Current_Position[1]] += Alpha * (Reward + ( Gamma * Q_SARSA[Next_Position_F2[0], Next_Position_F2[1]]) - Q_SARSA[Current_Position[0], Current_Position[1]]    )
        Current_Position = Next_Position_F

        if Reward == 0:
            break
```

The following matrix shows the final policy the agent has gotten.

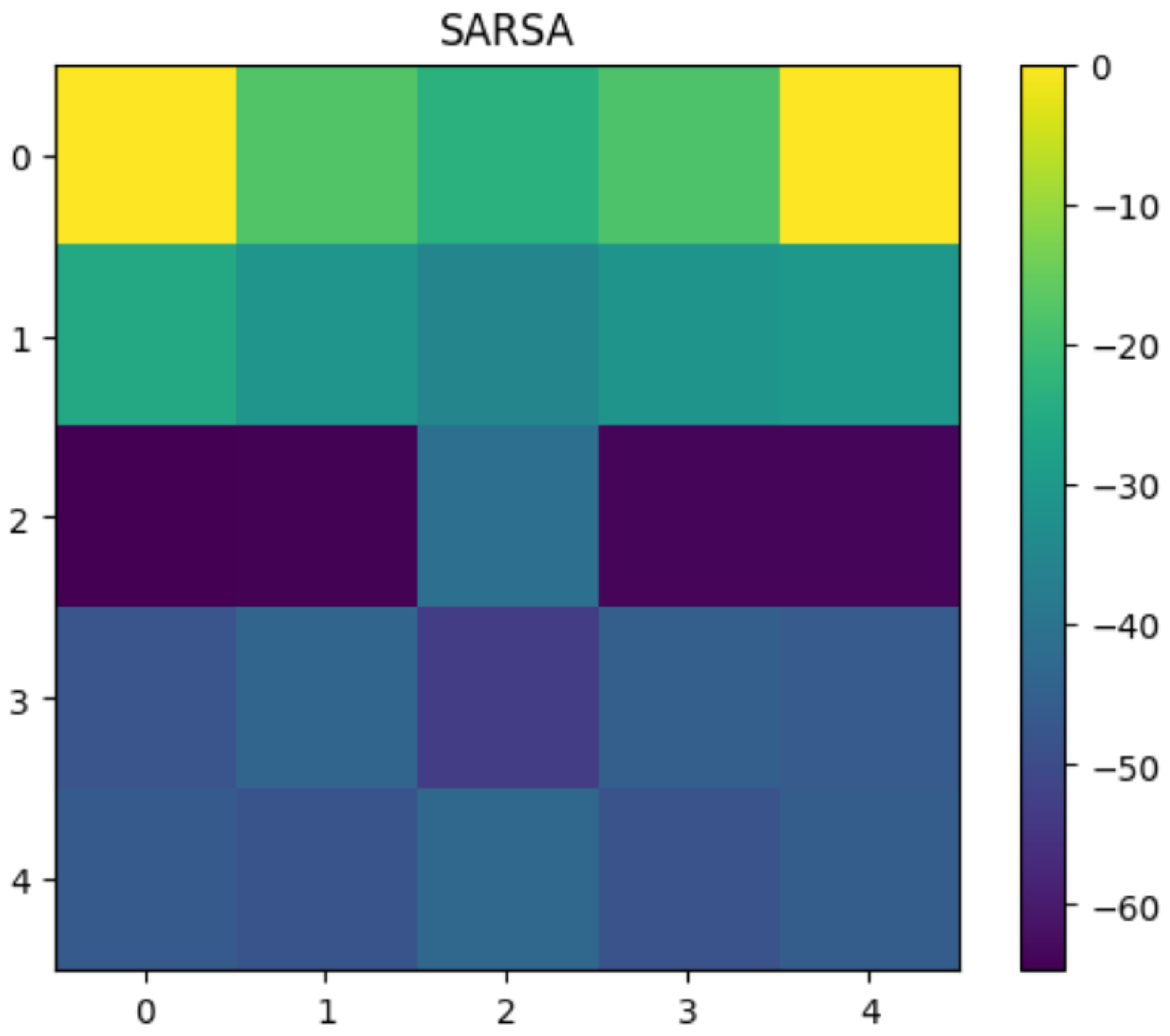


Right	Left	Left	Right	Left
Up	Up	Up	Up	Up
Up	Up	Up	Up	Up
Right	Left	Up	Right	Left
Up	Right	Left	Left	Up

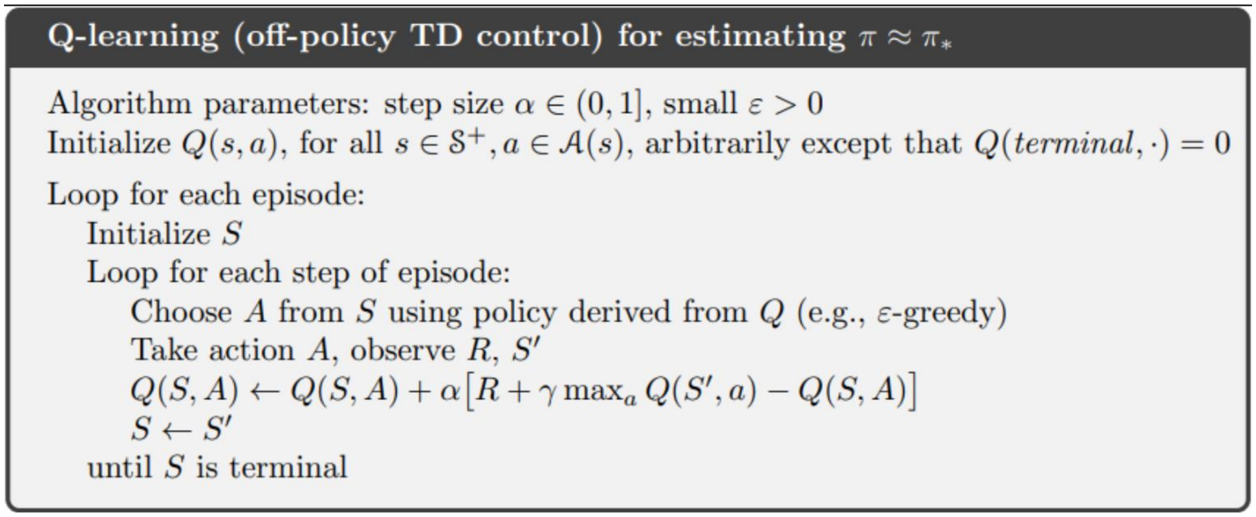
Here are the values obtained for the value function and a heatmap for better visualization.



```
[[  0.   , -17.484, -23.342, -17.99 ,   0.   ],
 [-25.769, -31.045, -35.274, -31.211, -30.215],
 [-64.687, -64.31 , -40.962, -63.507, -63.464],
 [-47.623, -43.474, -52.95 , -45.142, -46.082],
 [-46.471, -47.941, -42.975, -48.049, -45.682]]
```



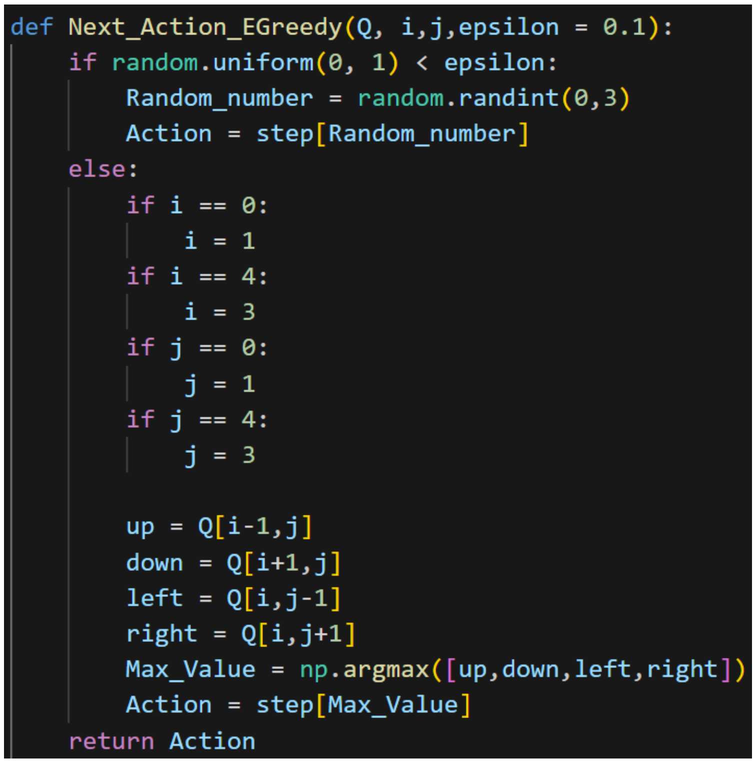
Q-Learning



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Code explanation

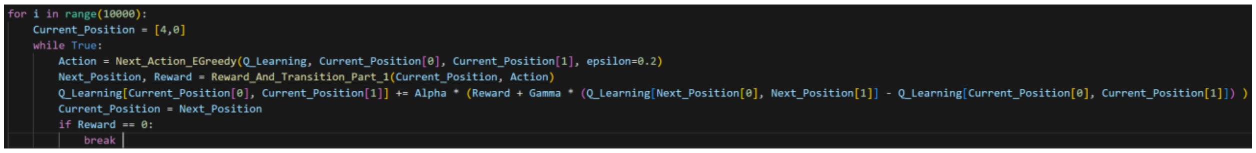
The Q-Learning algorithm is mostly the same as the SARSA algorithm, with the difference that the Q value for the next state should be the highest. We use the same functions as the SARSA algorithm.



```python
def Next_Action_EGreedy(Q, i,j,epsilon = 0.1):
    if random.uniform(0, 1) < epsilon:
        Random_number = random.randint(0,3)
        Action = step[Random_number]
    else:
        if i == 0:
            i = 1
        if i == 4:
            i = 3
        if j == 0:
            j = 1
        if j == 4:
            j = 3

        up = Q[i-1,j]
        down = Q[i+1,j]
        left = Q[i,j-1]
        right = Q[i,j+1]
        Max_Value = np.argmax([up,down,left,right])
        Action = step[Max_Value]
    return Action
```

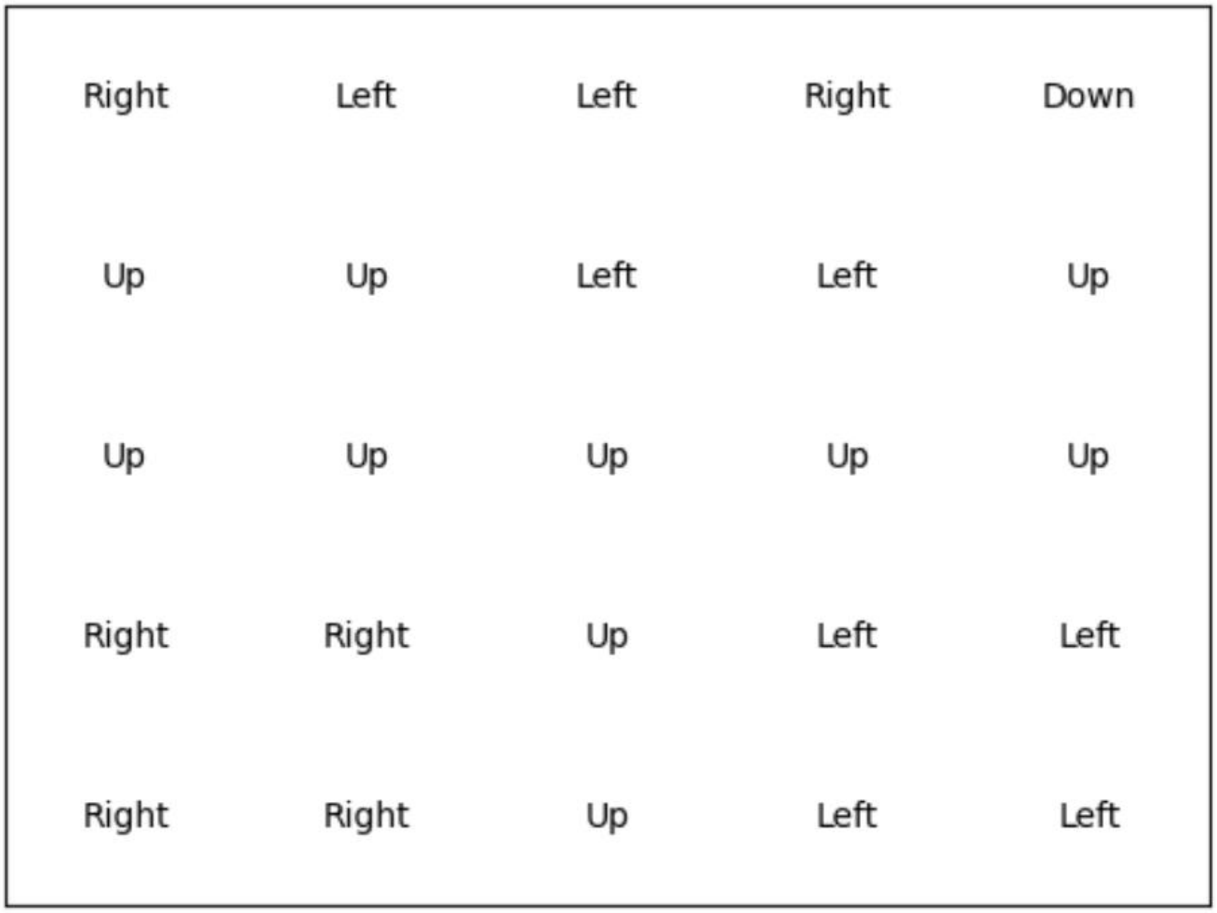
We use the function np.argmax to get the highest value index and take that one.

The agent starts in the state (4,0) and starts taking an action and taking the next state based on the max value; we set an epsilon of 0.10.



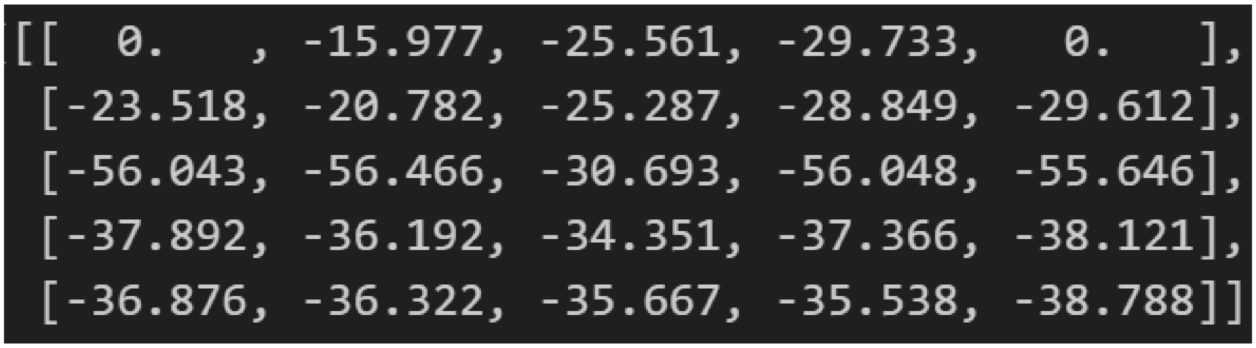
```
for i in range(10000):
    Current_Position = [4,0]
    while True:
        Action = Next_Action_EGreedy(Q_Learning, Current_Position[0], Current_Position[1], epsilon=0.2)
        Next_Position, Reward = Reward_And_Transition_Part_1(Current_Position, Action)
        Q_Learning[Current_Position[0], Current_Position[1]] += Alpha * (Reward + Gamma * (Q_Learning[Next_Position[0], Next_Position[1]] - Q_Learning[Current_Position[0], Current_Position[1]]) )
        Current_Position = Next_Position
        if Reward == 0:
            break
```

The following matrix shows the final policy the agent has gotten.

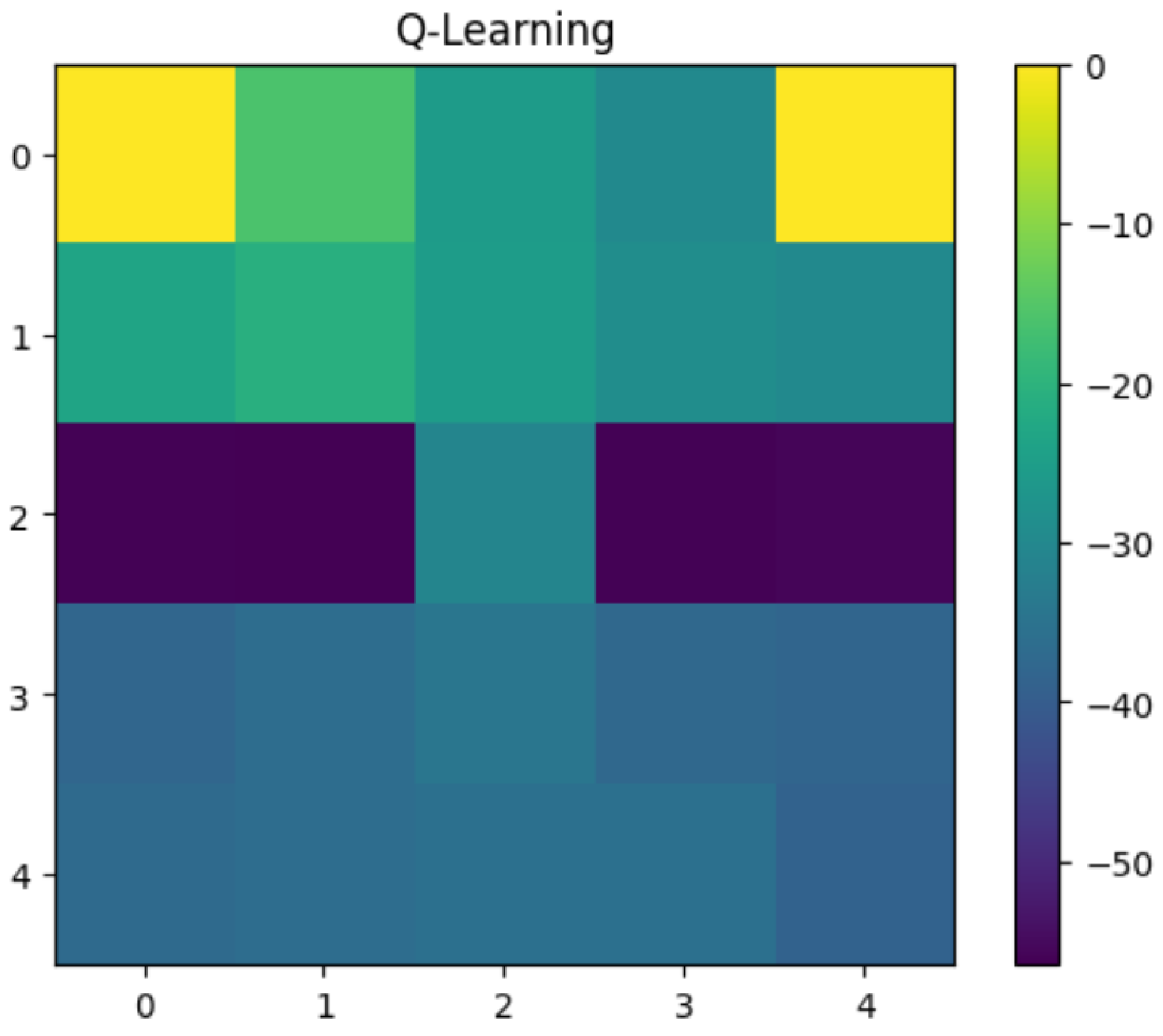


Right	Left	Left	Right	Down
Up	Up	Left	Left	Up
Up	Up	Up	Up	Up
Right	Right	Up	Left	Left
Right	Right	Up	Left	Left

Here are the values obtained for the value function and a heatmap for better visualization.



```
[[  0.   , -15.977, -25.561, -29.733,   0.   ],
 [-23.518, -20.782, -25.287, -28.849, -29.612],
 [-56.043, -56.466, -30.693, -56.048, -55.646],
 [-37.892, -36.192, -34.351, -37.366, -38.121],
 [-36.876, -36.322, -35.667, -35.538, -38.788]]
```



In Sarsa, this sum of rewards over episodes tends to be more variable because it relies on the on-policy nature of learning; its Q-values are updated according to what actions were actually done following whichever policy is in force. It entails a delayed convergence of the sum of rewards to a high level in the cases when an exploration phase by the agent can let it collect less Reward more consistently. In contrast, Q-learning usually gives a faster and more stable reward increase due to its off-policy nature. Since it looks at the optimum future rewards regardless of the current policy, it could more rapidly accumulate rewards to attain a very stable level of performance.

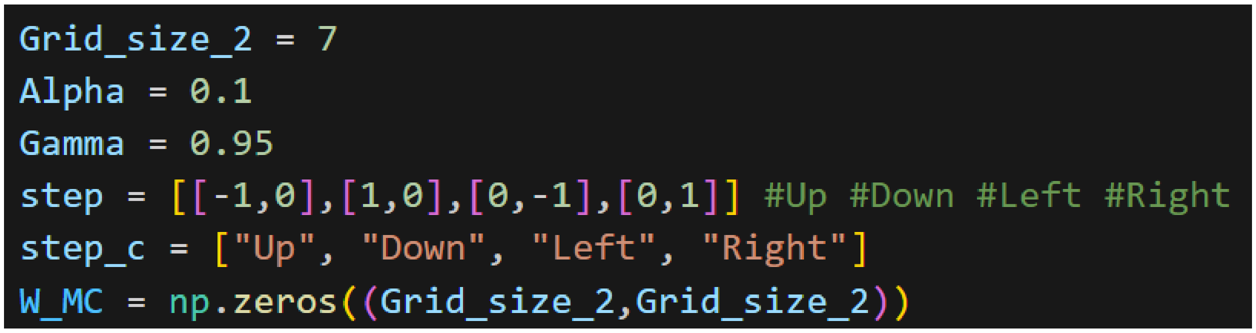
Part 2

Consider a scenario where we have a random walk on a 7×7 grid. That is, we are equally likely to move up, down, left, or right. Suppose that we start the random walk at the precise center of the grid.

We assume that the lower left and upper right corners are terminal states, with, respectively, rewards of -1 and 1. Rewards for transitions between two states are 0, if an attempt to transition outside the wall is made, the agent stays in the same spot and receives a reward of 0. Compute the value function for this "random walk" policy using (1) gradient Monte Carlo method and (2) the semi-gradient TD(0) method with an affine function approximation. How does it compare to the exact value function?

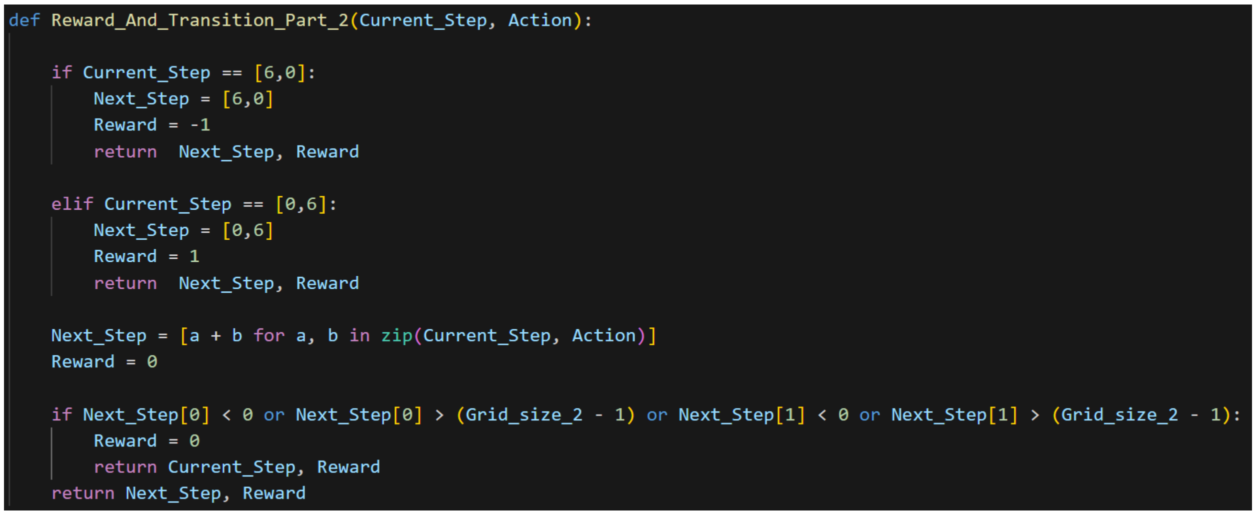
Gradient Monte Carlo

First, we create the W function, a 7 x 7 size matrix where we will update the values; we set an alpha value of 0.10 and Gamma 0f 0.95.



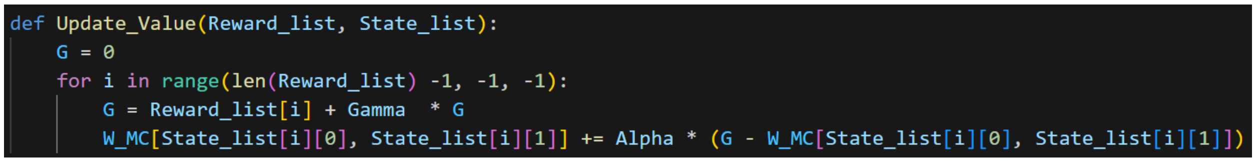
```python
Grid_size_2 = 7
Alpha = 0.1
Gamma = 0.95
step = [[-1,0],[1,0],[0,-1],[0,1]] #Up #Down #Left #Right
step_c = ["Up", "Down", "Left", "Right"]
W_MC = np.zeros((Grid_size_2,Grid_size_2))
```

With the function Reward_And_Transition_Part_2, we will get the Reward for each action, and we are going to get the next state. Here, we verify if the agent is in a terminal state.



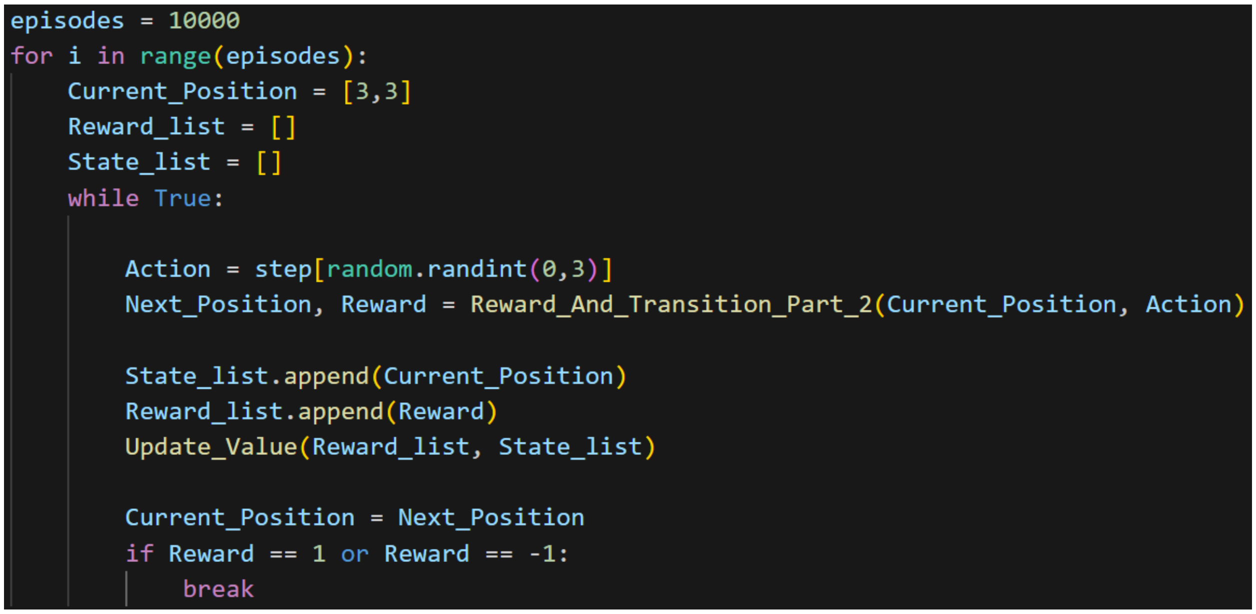
```python
def Reward_And_Transition_Part_2(Current_Step, Action):

    if Current_Step == [6,0]:
        Next_Step = [6,0]
        Reward = -1
        return  Next_Step, Reward

    elif Current_Step == [0,6]:
        Next_Step = [0,6]
        Reward = 1
        return  Next_Step, Reward

    Next_Step = [a + b for a, b in zip(Current_Step, Action)]
    Reward = 0

    if Next_Step[0] < 0 or Next_Step[0] > (Grid_size_2 - 1) or Next_Step[1] < 0 or Next_Step[1] > (Grid_size_2 - 1):
        Reward = 0
        return Current_Step, Reward
    return Next_Step, Reward
```

We created a function to upload the W function according to the formula provided by the pseudocode.



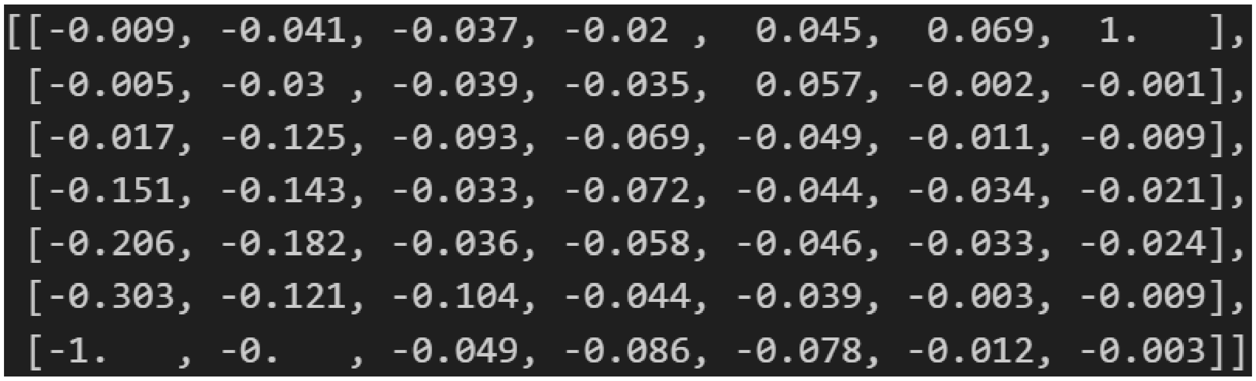
```python
def Update_Value(Reward_list, State_list):
    G = 0
    for i in range(len(Reward_list) -1, -1, -1):
        G = Reward_list[i] + Gamma  * G
        W_MC[State_list[i][0], State_list[i][1]] += Alpha * (G - W_MC[State_list[i][0], State_list[i][1]])
```

In this case, we did 10,000 episodes and updated 10,000 (multiplied by the number of states of each episode) times the W function according to the route the agent followed in each episode.

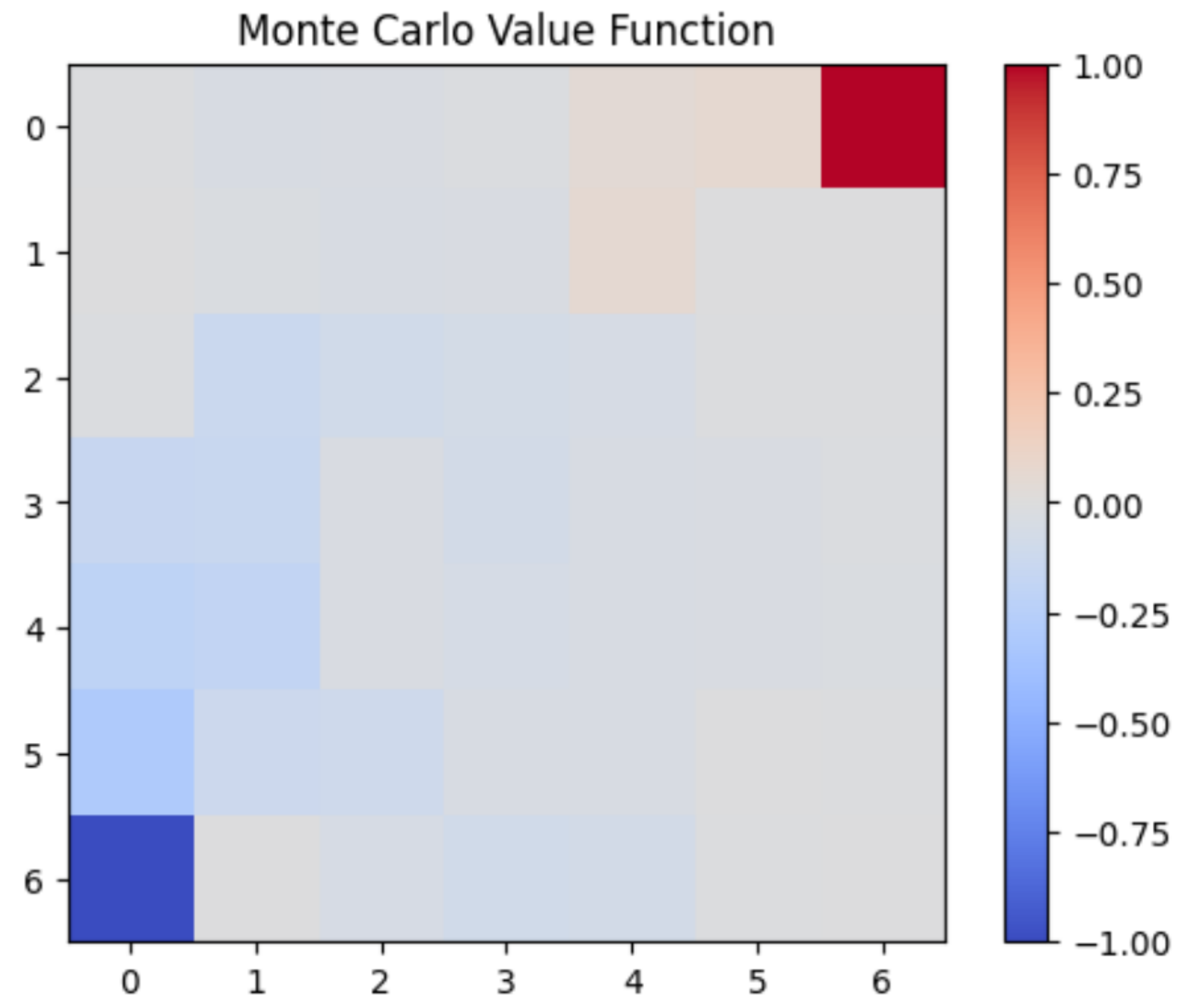


```python
episodes = 10000
for i in range(episodes):
    Current_Position = [3,3]
    Reward_list = []
    State_list = []
    while True:

        Action = step[random.randint(0,3)]
        Next_Position, Reward = Reward_And_Transition_Part_2(Current_Position, Action)

        State_list.append(Current_Position)
        Reward_list.append(Reward)
        Update_Value(Reward_list, State_list)

        Current_Position = Next_Position
        if Reward == 1 or Reward == -1:
            break
```

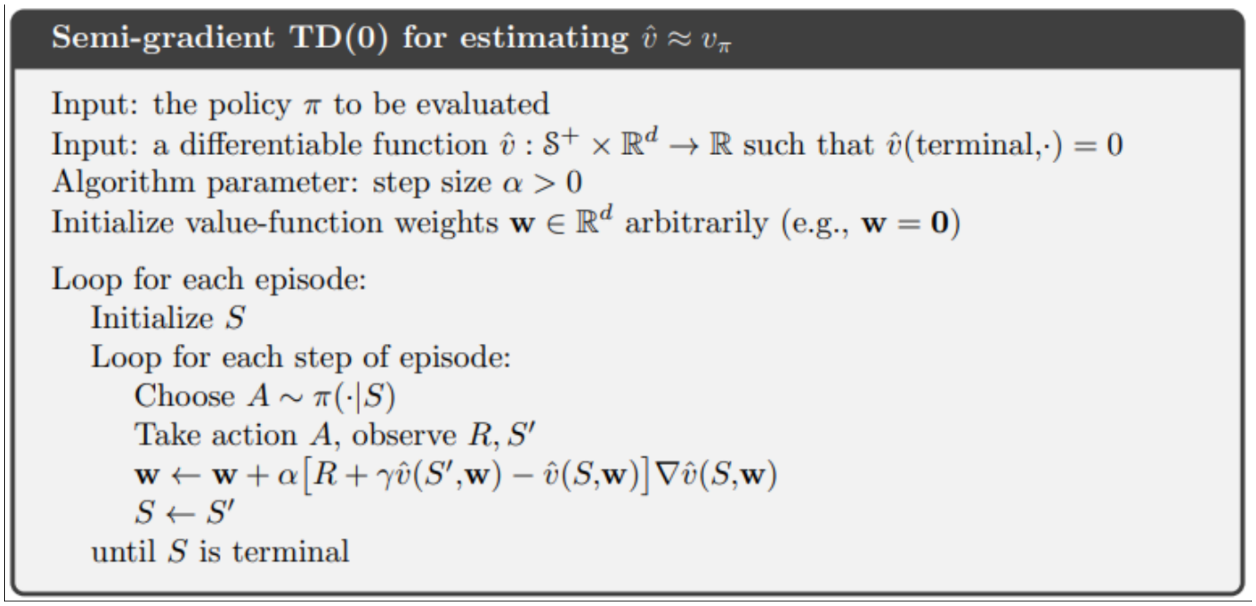
Here are the values obtained for the value function and a heatmap for better visualization.



```
[[-0.009, -0.041, -0.037, -0.02 ,  0.045,  0.069,  1.   ],
 [-0.005, -0.03 , -0.039, -0.035,  0.057, -0.002, -0.001],
 [-0.017, -0.125, -0.093, -0.069, -0.049, -0.011, -0.009],
 [-0.151, -0.143, -0.033, -0.072, -0.044, -0.034, -0.021],
 [-0.206, -0.182, -0.036, -0.058, -0.046, -0.033, -0.024],
 [-0.303, -0.121, -0.104, -0.044, -0.039, -0.003, -0.009],
 [-1.   , -0.   , -0.049, -0.086, -0.078, -0.012, -0.003]]
```



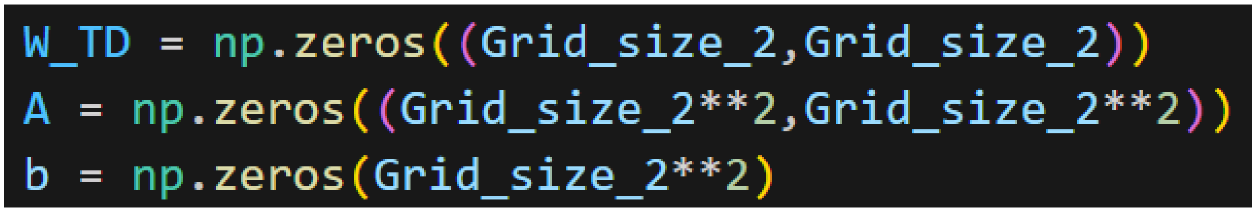
Semi-gradient TD(0)



```
Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A \sim \pi(\cdot|S)$
        Take action $A$, observe $R, S'$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
        $S \leftarrow S'$
    until $S$ is terminal
```

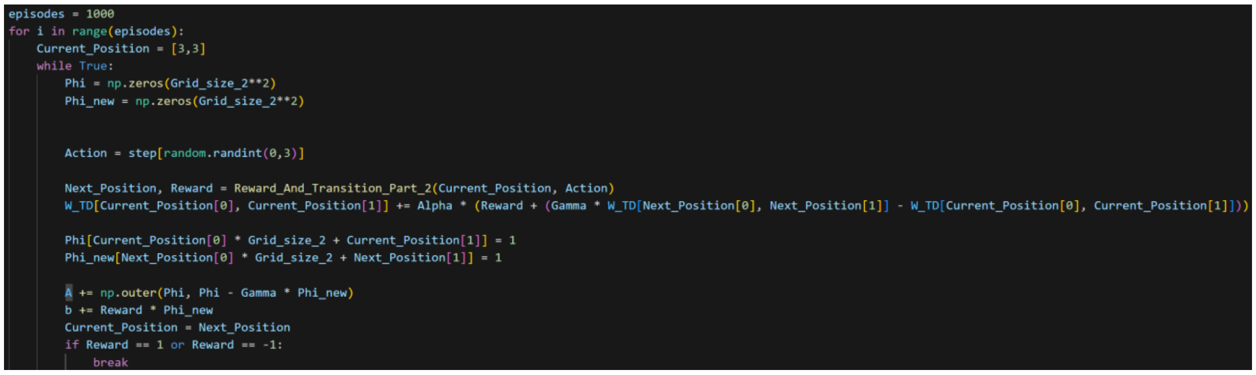
We got the W function in 2 different ways: the first one is iteratively based on the pseudocode, and the second one uses linear methods.

First, we created a matrix of size 7 x 7 for the iterative method, a second matrix of size 49 x 49, and a vector of length 49 for the linear method.



```
W_TD = np.zeros((Grid_size_2,Grid_size_2))
A = np.zeros((Grid_size_2**2,Grid_size_2**2))
b = np.zeros(Grid_size_2**2)
```

The following loop with 1000 episodes creates both W functions. We will explain both of them.

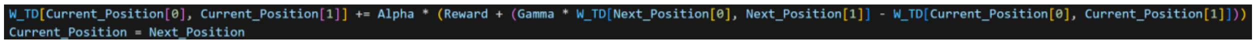


```
episodes = 1000
for i in range(episodes):
    Current_Position = [3,3]
    while True:
        Phi = np.zeros(Grid_size_2**2)
        Phi_new = np.zeros(Grid_size_2**2)


        Action = step[random.randint(0,3)]

        Next_Position, Reward = Reward_And_Transition_Part_2(Current_Position, Action)
        W_TD[Current_Position[0], Current_Position[1]] += Alpha * (Reward + (Gamma * W_TD[Next_Position[0], Next_Position[1]] - W_TD[Current_Position[0], Current_Position[1]]))

        Phi[Current_Position[0] * Grid_size_2 + Current_Position[1]] = 1
        Phi_new[Next_Position[0] * Grid_size_2 + Next_Position[1]] = 1

        A += np.outer(Phi, Phi - Gamma * Phi_new)
        b += Reward * Phi_new
        Current_Position = Next_Position
        if Reward == 1 or Reward == -1:
            break
```

Iterative

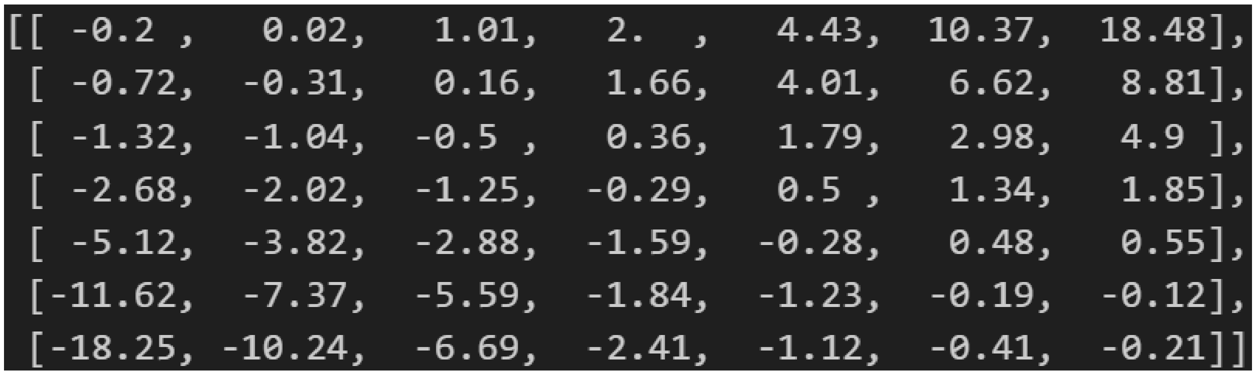
The iterative method follows the pseudocode provided previously and everything is done in the following lines of code.

```
$\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
$S \leftarrow S'$
```

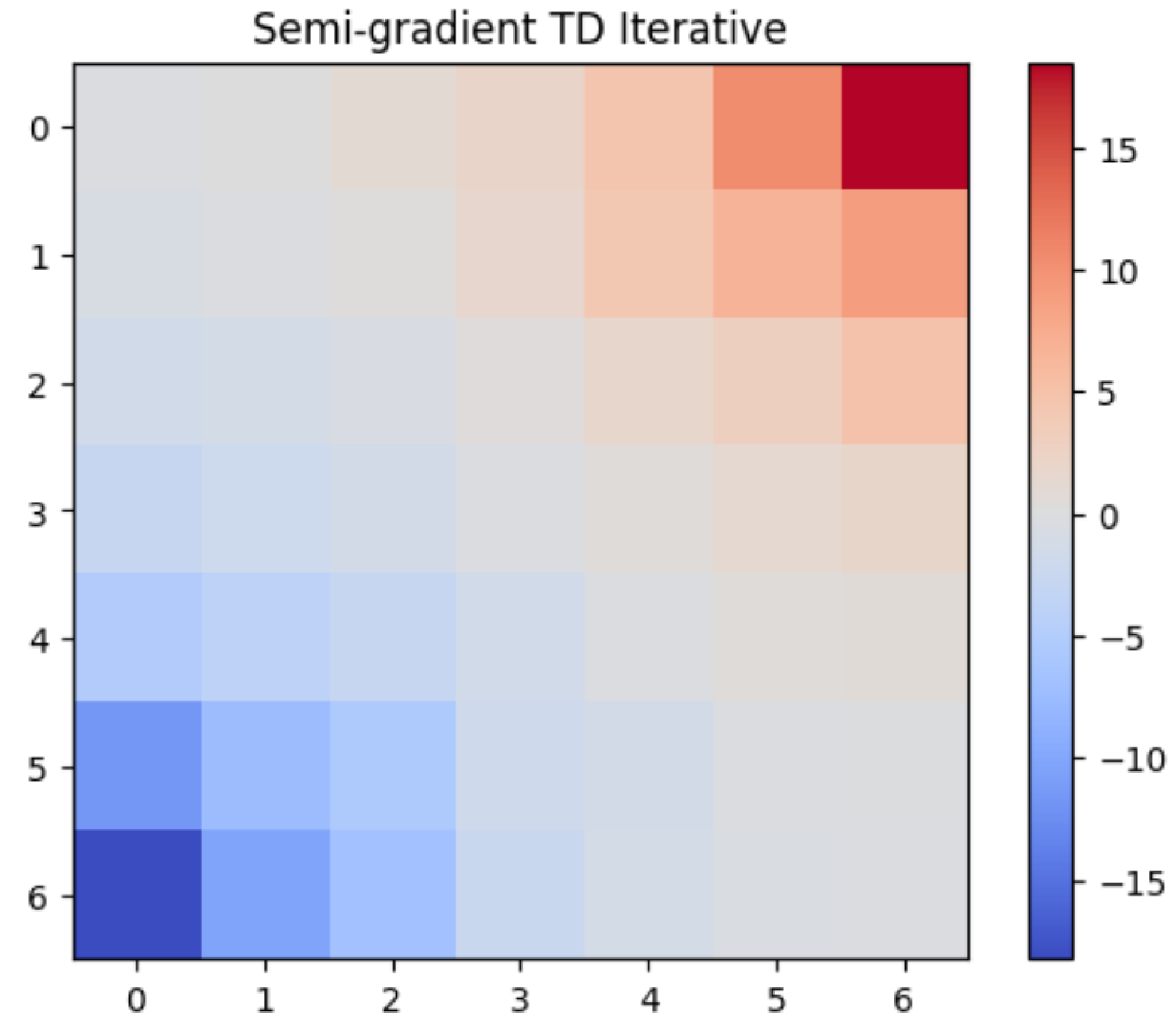


```
W_TD[Current_Position[0], Current_Position[1]] += Alpha * (Reward + (Gamma * W_TD[Next_Position[0], Next_Position[1]] - W_TD[Current_Position[0], Current_Position[1]]))
Current_Position = Next_Position
```

Result of W function



```
[[ -0.2 ,   0.02,   1.01,   2.  ,   4.43,  10.37,  18.48],
 [ -0.72,  -0.31,   0.16,   1.66,   4.01,   6.62,   8.81],
 [ -1.32,  -1.04,  -0.5 ,   0.36,   1.79,   2.98,   4.9 ],
 [ -2.68,  -2.02,  -1.25,  -0.29,   0.5 ,   1.34,   1.85],
 [ -5.12,  -3.82,  -2.88,  -1.59,  -0.28,   0.48,   0.55],
 [-11.62,  -7.37,  -5.59,  -1.84,  -1.23,  -0.19,  -0.12],
 [-18.25, -10.24,  -6.69,  -2.41,  -1.12,  -0.41,  -0.21]]
```



Linear Method (Least Squares)

For this method, we followed the formulas for A and b learned in the class.

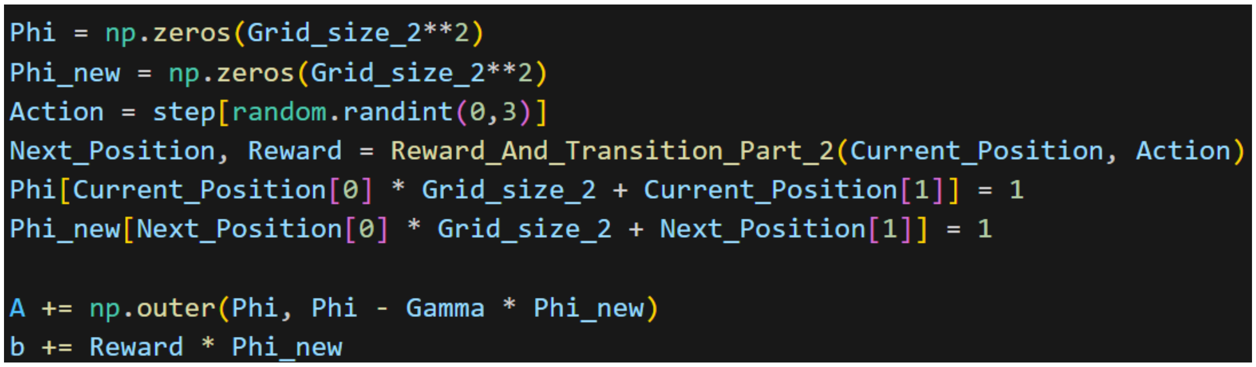
$$\widehat{A}_t = \sum_{k=0}^{t-1} x_k \, (x_k - \gamma x_{k+1})^T + I$$

$$\hat{b}_t = \sum_{k=0}^{t-1} R_{k+1} x_k$$

$$W_t = \widehat{A}_t^{-1} \hat{b}_t$$

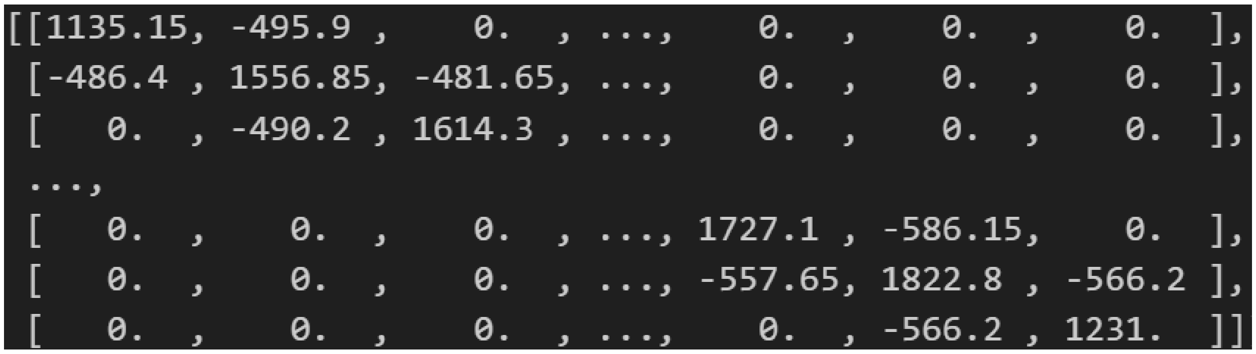
In this case we use x_k as a one hot coding where the agent has passed.

“A” must be a matrix of size 49 x 49 and “b” a vector of length 49.



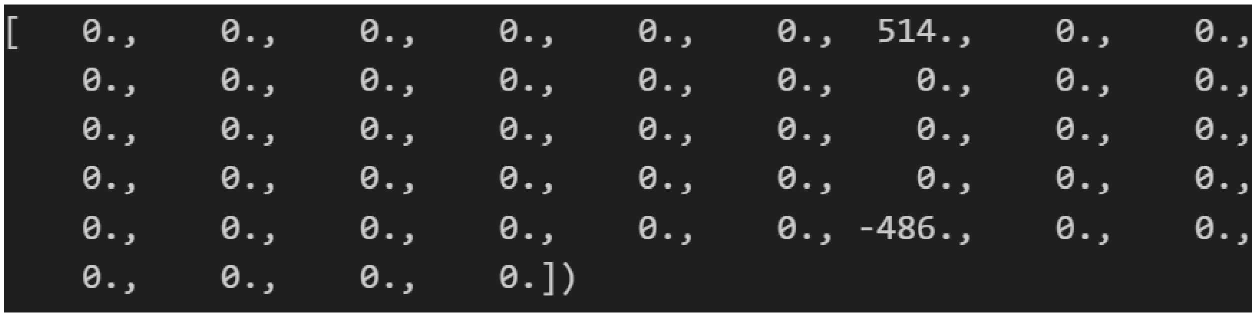
```
Phi = np.zeros(Grid_size_2**2)
Phi_new = np.zeros(Grid_size_2**2)
Action = step[random.randint(0,3)]
Next_Position, Reward = Reward_And_Transition_Part_2(Current_Position, Action)
Phi[Current_Position[0] * Grid_size_2 + Current_Position[1]] = 1
Phi_new[Next_Position[0] * Grid_size_2 + Next_Position[1]] = 1

A += np.outer(Phi, Phi - Gamma * Phi_new)
b += Reward * Phi_new
```

Result of A



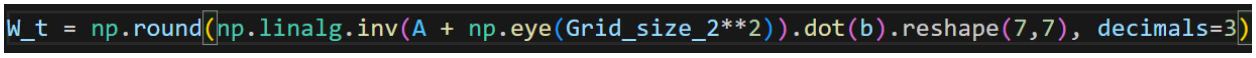
```
[[1135.15, -495.9 ,    0.  , ...,    0.  ,    0.  ,    0.  ],
 [-486.4 , 1556.85, -481.65, ...,    0.  ,    0.  ,    0.  ],
 [   0.  , -490.2 , 1614.3 , ...,    0.  ,    0.  ,    0.  ],
 ...,
 [   0.  ,    0.  ,    0.  , ..., 1727.1 , -586.15,    0.  ],
 [   0.  ,    0.  ,    0.  , ..., -557.65, 1822.8 , -566.2 ],
 [   0.  ,    0.  ,    0.  , ...,    0.  , -566.2 , 1231.  ]]
```

Result of b

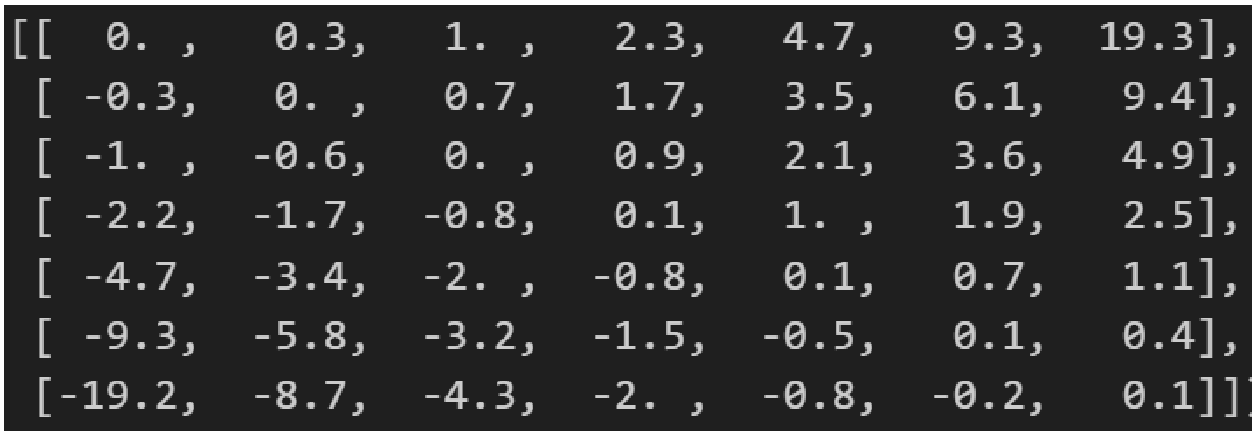


```
[   0.,    0.,    0.,    0.,    0.,    0.,  514.,    0.,    0.,
    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
    0.,    0.,    0.,    0.,    0.,    0., -486.,    0.,    0.,
    0.,    0.,    0.,    0.])
```

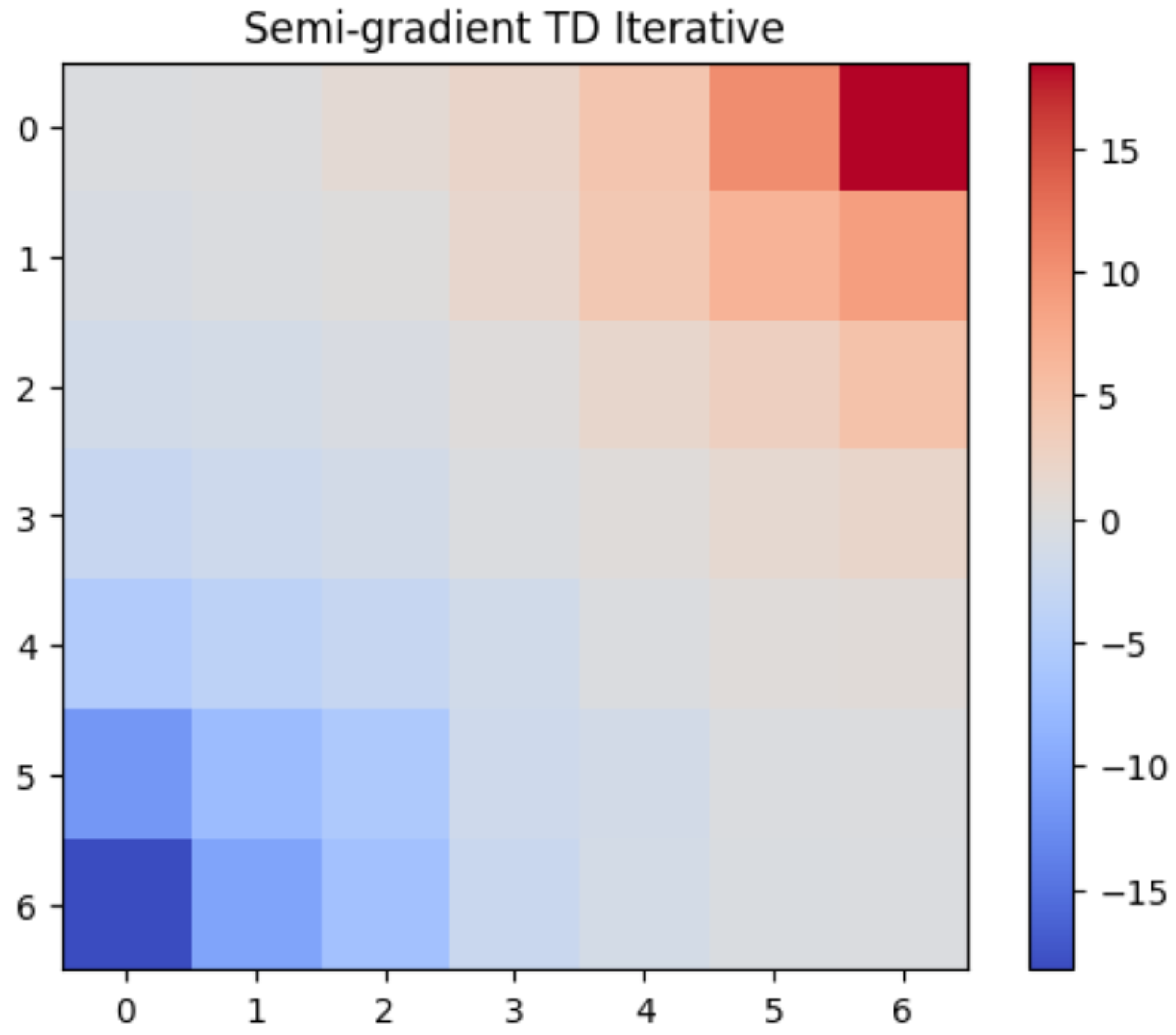
We add the identity matrix to A and solve the linear equation.



```
W_t = np.round(np.linalg.inv(A + np.eye(Grid_size_2**2)).dot(b).reshape(7,7), decimals=3)
```



```
[[  0. ,   0.3,   1. ,   2.3,   4.7,   9.3,  19.3],
 [ -0.3,   0. ,   0.7,   1.7,   3.5,   6.1,   9.4],
 [ -1. ,  -0.6,   0. ,   0.9,   2.1,   3.6,   4.9],
 [ -2.2,  -1.7,  -0.8,   0.1,   1. ,   1.9,   2.5],
 [ -4.7,  -3.4,  -2. ,  -0.8,   0.1,   0.7,   1.1],
 [ -9.3,  -5.8,  -3.2,  -1.5,  -0.5,   0.1,   0.4],
 [-19.2,  -8.7,  -4.3,  -2. ,  -0.8,  -0.2,   0.1]]
```



Both W functions have similar values

The Gradient Monte Carlo method arrives at the value function by averaging returns from several episodes to converge very slowly to the actual value function over many episodes. An interesting aspect is that cells are more affected by their neighbors. SemiGradient TD(0) applies one-step temporal difference learning with function approximation. At the same time, its convergence may be faster than Monte Carlo; it depends on the choice of features and the learning rate.