Assignment 5b - Understanding Reinforcement Learning through Q-Learning

Tanushree Kumar | tanushree.kumar@outlook.com

DATA 640 - Fall 2024

Professor Steve Knode

University of Maryland Global Campus

Due: October 29, 2024

Introduction

This assignment aims to explain and build a solid understanding of the basic concepts of Q-Learning and how it works. The information and Python code used in this assignment are from an article by Sayak Paul (Paul, 2019) on an introduction to the area of Q-Learning.

Q-Learning is a type of reinforcement learning (RL) where an agent learns optimal actions in an environment by maximizing cumulative rewards through trial and error. In RL, an agent interacts with an environment through trial and error, receiving rewards based on its actions. Q-Learning specifically focuses on finding the "Q-values" for action-state pairs, which estimate the expected future rewards. These Q-values help the agent make better decisions over time by balancing exploration, which is trying new actions, and exploitation, which is using known high-reward actions. It’s widely used in applications like games, robotics, and autonomous systems. An excellent use case example of Q-Learning is the article by Sayak Paul where the problem is based on building autonomous robots for a guitar-building factory.

Overview of the Q-Learning Code

This Q-Learning code is designed to find the optimal route between two locations in a simple environment represented as a graph. The environment is defined by nine locations from L1 to L), each corresponding to a state and connected to others with specific reward values. The rewards matrix indicates potential transitions and their associated rewards. Q-Learning works by iterating over 1,000 episodes to update the Q-values for each state-action pair. For each episode, a random starting location is chosen, and the algorithm calculates the temporal difference based on the Bellman equation to adjust the Q-values. Using a balance of exploration and exploitation, the model refines the Q-values to converge toward an optimal policy for moving from one location to another.

Only the numpy library is imported for handling matrices and numerical operations. There are two key parameters for Q-Learning: gamma and alpha. Gamma is the discount factor that controls how much future rewards are valued relative to immediate rewards. Alpha is the learning rate that controls how much new information overrides previous knowledge in Q-table updates. The states and actions are defined next, which represent the nine locations in the environment. The 'location_to_state' dictionary maps location names to numerical indices, with each location index corresponding to a possible action, allowing movement between locations based on connectivity. The 'rewards' matrix defines the reward values for moving from one location (row) to another (column). Values of 1 indicate a reward for a possible action and a value of 0 for an impossible action. The 'location_to_state' dictionary reverses the mapping, allowing the algorithm to retrieve location names from state indices.

The 'get_optimal_route' function is the core of this code. This function determines the best route from a given starting point to the desired end location based on the learned Q-values. In other words, it takes in 'start_location' and 'end_location' and uses Q-Learning to find the optimal path. A modified rewards matrix called 'rewards_new' is created where reaching the ending state (goal) is prioritized by setting its self-transition reward to 999.

The Q table is initialized as a 9x9 matrix of zeros, with each cell representing a state-action pair. For each of the 1,000 episodes, a random 'current_state' is chosen and playable actions from the current state are identified. Only actions leading to non-zero rewards are considered as these represent feasible paths. An action from the list of playable actions is randomly chosen, leading to the 'next_state'. The temporal difference is computed by comparing the new state's reward with the Q-value for the chosen action, which is adjusted by gamma. Then, the Q-table is updated using the temporal difference and the learning rate, alpha.

Once the Q-Learning process is complete, the optimal route is generated by starting from ‘start_location’ by selecting actions based on the highest Q-values in each state until reaching the ‘end_location’. This approach to Q-Learning allows the agent to learn an optimal policy based on repeated interactions with the environment, effectively navigating the "maze" to find the shortest route.

Calling ‘get_optimal_route’ from location L9 to L1 prints the optimal route, learned by the Q-Learning algorithm, demonstrating how the agent has mapped out an efficient path through exploration and Q-value updates. The result is printed as ‘get_optimal_route('L9', 'L1')’, which displays the optimal path learned by the agent through self-improvement.

Assignment Questions

Question 3

Successfully running the code resulted in running the command print(get_optimal_route ('L9', 'L1')) to get the output of ['L9', 'L8', 'L5', 'L2', 'L1']. This is due to the output representing the optimal route determined by the Q-Learning algorithm for moving from location L9 to L1. This path was chosen because it maximizes cumulative rewards based on the learned Q-values, which reflect the agent’s experience through iterative exploration of possible moves. Each step in this path corresponds to a transition to a neighboring location that either has a direct reward or leads closer to the end goal, balancing immediate and future rewards due to the discount factor gamma. By following the highest Q-values at each location, the agent identifies the shortest path in terms of rewards rather than physical distance, resulting in an efficient route. Looking at Figure 1 in Appendix A, it can be seen how the agent would understand which would be the most efficient path. Starting from L9, the next step would be to go L8 since it is the only option. From L8, the agent has three options to choose from: L9, L7, and L5. Running through 1,000 iterations trained the agent to learn which action-state pair has the highest value, in this case, L5. As a result, the agent moves from L9 to L8 to L5 to L2 to L1.

Question 5

To further understand how Q-Learning works, the code was run with different levels of hyperparameters gamma and alpha. Here, gamma is the discount factor and alpha is the learning rate. As mentioned previously, gamma is the discount factor that controls how much future rewards are valued relative to immediate rewards; and alpha is the learning rate that controls how much new information overrides previous knowledge in Q-table updates. Having the ideal values is crucial as higher values could lead to divergence and lower values could lead to slower learning.

To see how different values would affect the learning process, three variations were tested. The first variation had both values be 0.05; the second variation had the adjusted parameters to 0.05 for gamma, and reverted to the original value for alpha; and the third variation was vice-versa of the second variation. This can be visualized in the table below:

Variation	Value of Gamma	Value of Alpha
Default	0.9	0.75
1	0.05	0.05
2	0.05	0.75
3	0.9	0.05

Table 1. Table of values for gamma and alpha.

Figures 2, 3, and 4 in Appendix A show the output of the Q-value array for each variation respectively. Figure 6 shows the Q-value array for the default values of gamma and alpha for comparison.

The Q-value arrays clearly show the difference in changing the parameter values. When both, gamma and alpha, have a low value of 0.05, convergence can be observed as all of the Q-values are significantly lower in comparison to the default Q-value array. The first two values in the array are bigger compared to the rest of the array where the majority of the values are close to 0 and 1. When both gamma and alpha are set to low values, the Q-learning updates are cautious and rely less on future rewards due to the low discount factor (gamma), while also taking only small steps in each update due to the low learning rate (alpha). This results in significantly smaller Q-values across the array, indicating minimal accumulation of reward feedback. Although the first two values are relatively higher, most values are close to 0 and 1. This configuration demands more episodes, increasing computational time and energy, to approach convergence.

When gamma = 0.05 and alpha = 0.75, the first 2 values in the array, 1051.57 and 3.67, are also bigger in comparison to the rest of the array where the majority of the values are close to 0 and 1. Having a lower gamma value places a strong emphasis on immediate rewards rather than long-term ones. Here, the model prioritized immediate gains, causing large Q-values, such as 1051.57 and 3.67, where immediate rewards are high. This contrasts with most other Q-values remaining near 0 or 1, as the model largely disregards cumulative rewards from future states, focusing instead on high-reward steps right away. This approach is useful when immediate rewards are highly valuable but limits the model's exploration and optimization for longer paths.

When gamma = 0.9 and alpha = 0.05, the first 2 values in the array, 2705.61 and 913.50, are also bigger in comparison to the rest of the array where some of the values are 0 but some values range from 10 to 1000. Here, the model strongly considers future rewards due to the high discount factor (gamma), and it updates Q-values more conservatively because of the low learning rate (alpha). Consequently, Q-values like 2705.61 and 913.50 emerge as high values, indicating states with significant future reward potential, while other values range broadly from 10 to 1000 or even 0. The distribution shows that the agent values long-term rewards more consistently across states, as it accumulates knowledge from multiple episodes gradually, favoring broader exploration.

Question 6

To see how many times the while loop in the ‘get_optimal_route’ function gets executed when the print(get_optimal_route('L9', 'L1')) command is run, a step counter was added before the while loop and the return statement was edited to return the number of iterations the while loop takes. The while loop is iterated for 4 times, which makes sense and matches the amount of moves the agent has to make from L9 to L1. This outcome aligns with the route layout, showing that each iteration represents one move through the Q-learning optimized path, ending once the agent reaches L1. This confirms that the loop iterates precisely as needed to find the optimal path. Figure 7 in Appendix A shows the modified code and step count output.

Question 7

The Q-Learning process for loop in the ‘get_optimal_route’ function performs 1000 iterations. To assess if these many iterations were necessary, two different values were tested, one with 200 iterations and another with 50 iterations. This was done by changing the value in the for loop parameter. Figures 8, 9, and 10 in Appendix A below show the Q-values for 1000, 200, and 50 iterations respectively.

It can be seen that the number of iterations in training directly impacts the quality of the learned Q-values. For example, after 1000 iterations, the Q-values are high and consistent across certain states, with clear values for optimal paths. At 200 iterations, values decrease slightly but still show a recognizable optimal path. However, with only 50 iterations, the Q-values are notably lower, with less-defined patterns, leading to a suboptimal learned policy. Fewer iterations limit the algorithm's opportunity to explore and reinforce high-value paths, resulting in less accurate Q-value estimations for decision-making.

Running fewer iterations reduces training time but can lead to an under-trained model with lower-quality Q-values, impacting the accuracy of the agent's actions in finding the optimal route. Increasing iterations, as seen in the 1000 iteration example, allows the Q-values to stabilize, reinforcing the paths that lead to successful outcomes and improving the policy's overall performance.

Question 8

Initially, the path of ‘print(get_optimal_route('L9', 'L1'))’ was run. This time, the reverse path of ‘print(get_optimal_route('L1', 'L9'))’ was run to assess any differences. However, this was not possible as the code kept iterating without returning. Upon close inspection, it was found that in the rewards matrix, a value of ‘1’ was missing from the L2 row. The missing ‘1’ caused the agent to think that going from L2 to L5 was not allowed, hence why the code didn’t return an output. This was fixed by modifying the L2 row from [1,0,1,0,0,0,0,0,0] to [1,0,1,0,1,0,0,0,0]. This fix allowed the agent to move L2 to L5 since this was now an allowable action.

Figure 11 shows the fixed code and the output for going from L1 to L9. The while loop iterated 4 times which makes sense since it is the same path, only reversed. The Q-value array for the forward and reverse paths also had similar values.

Question 9

A tenth state of L10 was added, where it is only possible to go from L10 to L9 and vice versa. This can be visualized in Figure 12 in Appendix A. The code was modified to add L10 by adding to the ‘location_to_state’ dictionary; adding to the actions; adding to the rewards matrix; changing the dimensions of the Q array; changing the parameters for the ‘current_state’; and adding two new print statements for L10 to L1 and for L10 to L4. All these modifications in the code can be seen in the Code Appendix.

The results of running ‘get_optimal_route('L10', 'L1')’ and ‘get_optimal_route('L10', 'L4')’ yield the optimal routes [L10, L9, L8, L5, L2, L1] of 5 steps and [L10, L9, L8, L7, L4] of 4 steps respectively. This outcome aligns with the previously set expectations. The Q-learning algorithm correctly identifies efficient paths to each target location by following the highest Q-values. The high Q-values along these routes indicate that these paths were reinforced more strongly, guiding the agent toward optimal navigation from L10 to the specified destinations.

Conclusions

This Q-learning assignment provided hands-on experience with reinforcement learning principles, especially how an agent learns to navigate a graph-based environment through trial and error, refining choices with updated Q-values. Adjusting parameters like gamma and alpha demonstrated the balance between immediate and long-term rewards and the importance of a well-trained Q-table for optimal routing. Testing with altered hyperparameters and episode counts emphasized the trade-offs in learning efficiency and solution quality.

Adding a new location of L10 also demonstrated Q-learning’s adaptability in expanding environments, while debugging helped highlight how missing or incorrect reward connections directly impact route discovery. By experimenting with different configurations and troubleshooting issues, it became clear that careful consideration of hyperparameters, reward matrices, and episode counts is essential for building a robust model.

References

Banoula, M. (2023, November 7). *What is Q-Learning: Everything you Need to Know | Simplilearn*. Simplilearn.com. https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-Q-Learning

Paul, S. (2019, May 15). *An introduction to Q-Learning: Reinforcement Learning*. FloydHub Blog. https://floydhub.ghost.io/an-introduction-to-Q-Learning-reinforcement-learning/

Sayak Paul. (2022). *FloydHub-Q-Learning-Blog/Q-Learning using numpy.ipynb at master · sayakpaul/FloydHub-Q-Learning-Blog*. GitHub. https://github.com/sayakpaul/FloydHub-Q-Learning-Blog/blob/master/Q-Learning%20using%20numpy.ipynb

Appendix A

All figures and visualizations mentioned in the report can be seen below. The Python script used to generate the algorithm and the output is attached as a separate file and in the Code Appendix.

Figure 1

Figure of the sample environment showing the nine different positions. (Sayak Paul, 2022)

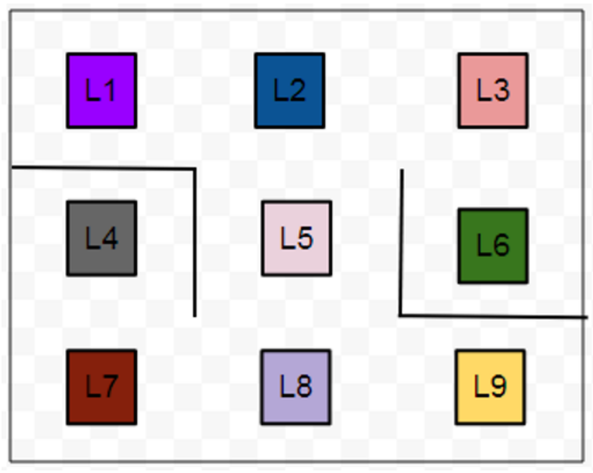
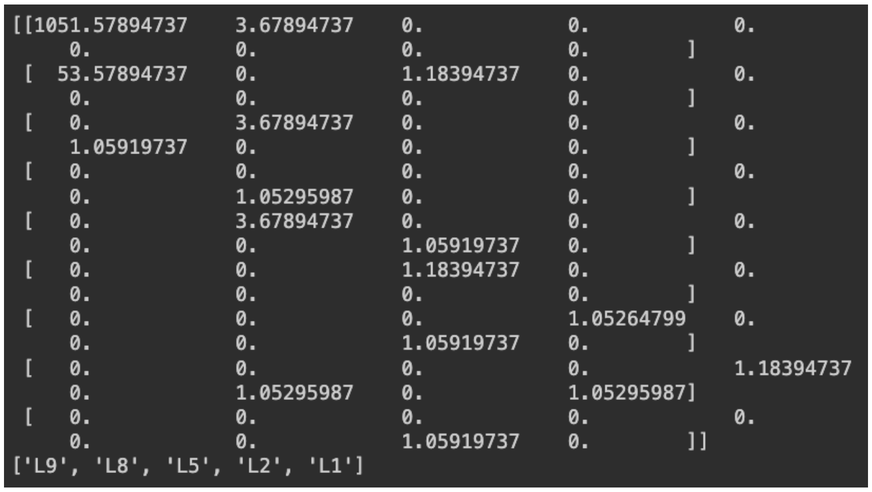


Figure 2

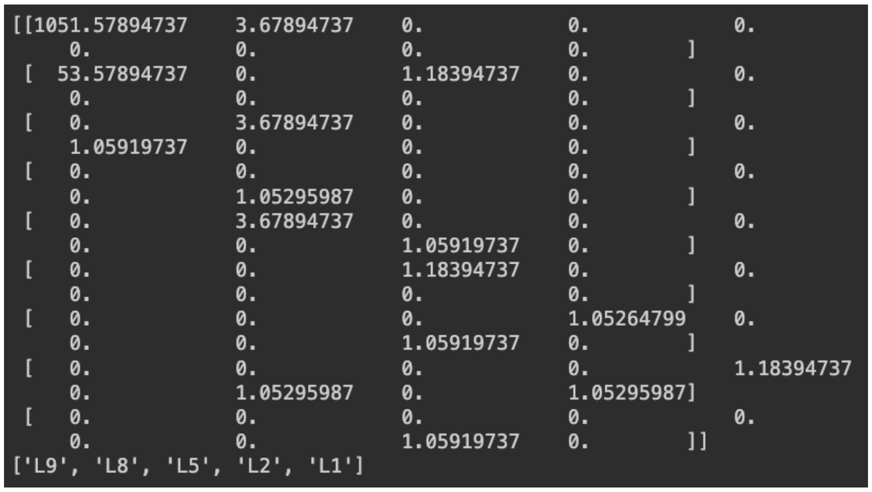
Figure of the Q-value array for gamma = 0.05 and alpha = 0.05.



```
[[1051.57894737    3.67894737    0.            0.            0.
     0.            0.            0.            0.        ]
 [  53.57894737    0.            1.18394737    0.            0.
     0.            0.            0.            0.        ]
 [   0.            3.67894737    0.            0.            0.
     1.05919737    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.            1.05295987    0.            0.        ]
 [   0.            3.67894737    0.            0.            0.
     0.            0.            1.05919737    0.        ]
 [   0.            0.            1.18394737    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.            1.05264799    0.
     0.            0.            1.05919737    0.        ]
 [   0.            0.            0.            0.            1.18394737
     0.            1.05295987    0.            1.05295987]
 [   0.            0.            0.            0.            0.
     0.            0.            1.05919737    0.        ]]
['L9', 'L8', 'L5', 'L2', 'L1']
```

Figure 3

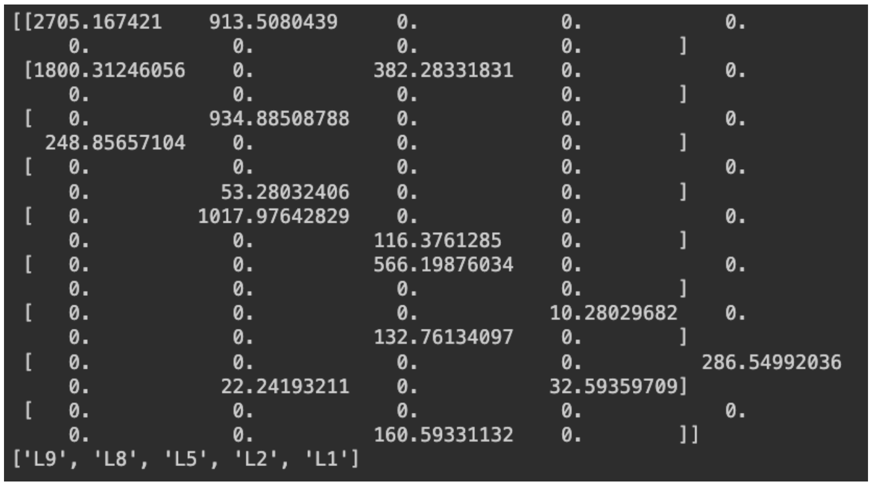
Figure of the Q-value array for gamma = 0.05 and alpha = 0.75.



```
[[1051.57894737    3.67894737    0.            0.            0.
     0.            0.            0.            0.        ]
 [  53.57894737    0.            1.18394737    0.            0.
     0.            0.            0.            0.        ]
 [   0.            3.67894737    0.            0.            0.
     1.05919737    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.            1.05295987    0.            0.        ]
 [   0.            3.67894737    0.            0.            0.
     0.            0.            1.05919737    0.        ]
 [   0.            0.            1.18394737    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.            1.05264799    0.
     0.            0.            1.05919737    0.        ]
 [   0.            0.            0.            0.            1.18394737
     0.            1.05295987    0.            1.05295987]
 [   0.            0.            0.            0.            0.
     0.            0.            1.05919737    0.        ]]
['L9', 'L8', 'L5', 'L2', 'L1']
```

Figure 4

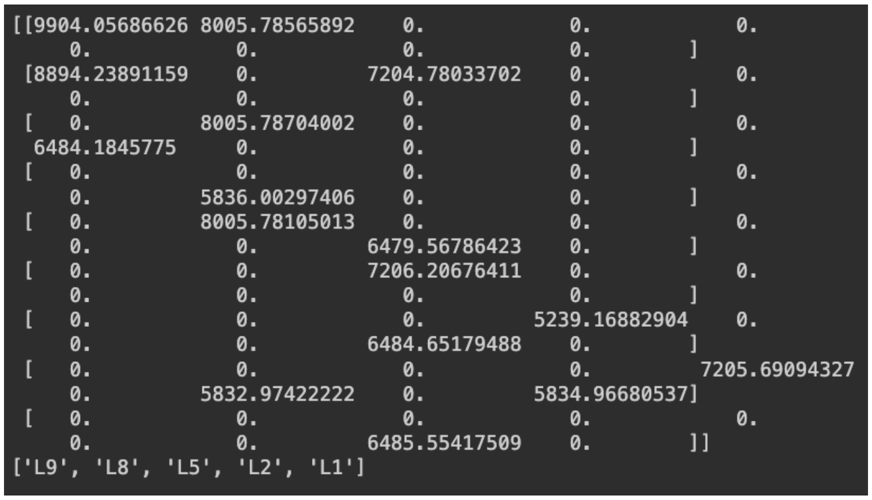
Figure of the Q-value array for gamma = 0.9 and alpha = 0.05.



```
[[2705.167421    913.5080439     0.            0.            0.
     0.            0.            0.            0.        ]
 [1800.31246056    0.          382.28331831    0.            0.
     0.            0.            0.            0.        ]
 [   0.          934.88508788    0.            0.            0.
   248.85657104    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.           53.28032406    0.            0.        ]
 [   0.         1017.97642829    0.            0.            0.
     0.            0.          116.3761285     0.        ]
 [   0.            0.          566.19876034    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.           10.28029682    0.
     0.            0.          132.76134097    0.        ]
 [   0.            0.            0.            0.          286.54992036
     0.           22.24193211    0.           32.59359709]
 [   0.            0.            0.            0.            0.
     0.            0.          160.59331132    0.        ]]
['L9', 'L8', 'L5', 'L2', 'L1']
```

Figure 6

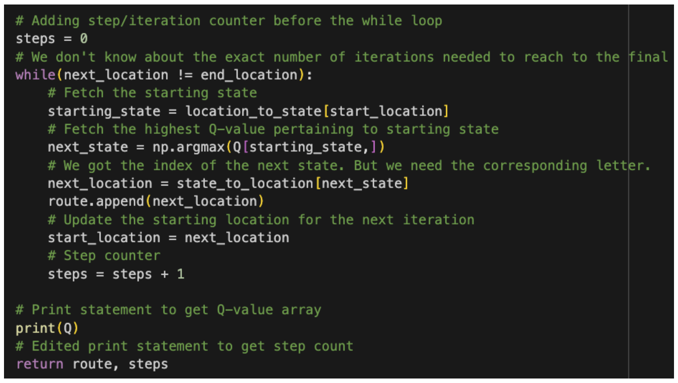
Figure of the Q-value array for gamma = 0.9 and alpha = 0.75.



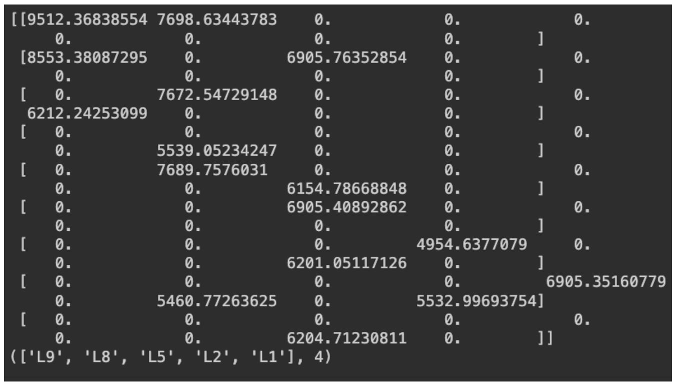
```
[[9904.05686626 8005.78565892    0.            0.            0.
     0.            0.            0.            0.        ]
 [8894.23891159    0.         7204.78033702    0.            0.
     0.            0.            0.            0.        ]
 [   0.         8005.78704002    0.            0.            0.
  6484.1845775     0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.         5836.00297406    0.            0.        ]
 [   0.         8005.78105013    0.            0.            0.
     0.            0.         6479.56786423    0.        ]
 [   0.            0.         7206.20676411    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.         5239.16882904    0.
     0.            0.         6484.65179488    0.        ]
 [   0.            0.            0.            0.         7205.69094327
     0.         5832.97422222    0.         5834.96680537]
 [   0.            0.            0.            0.            0.
     0.            0.         6485.55417509    0.        ]]
['L9', 'L8', 'L5', 'L2', 'L1']
```

Figure 7

Figure of the code showing the modifications and figure of the array and iteration count for question 6.



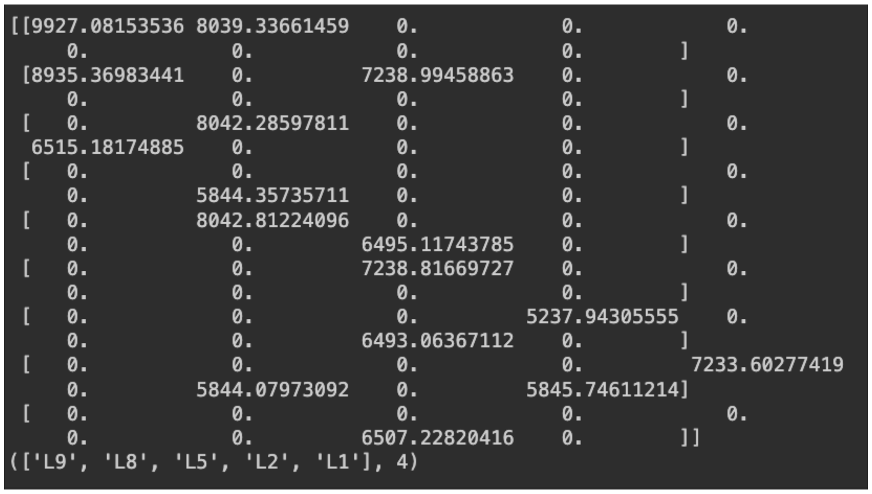
```python
# Adding step/iteration counter before the while loop
steps = 0
# We don't know about the exact number of iterations needed to reach to the final
while(next_location != end_location):
    # Fetch the starting state
    starting_state = location_to_state[start_location]
    # Fetch the highest Q-value pertaining to starting state
    next_state = np.argmax(Q[starting_state,])
    # We got the index of the next state. But we need the corresponding letter.
    next_location = state_to_location[next_state]
    route.append(next_location)
    # Update the starting location for the next iteration
    start_location = next_location
    # Step counter
    steps = steps + 1

# Print statement to get Q-value array
print(Q)
# Edited print statement to get step count
return route, steps
```



```
[[9512.36838554 7698.63443783    0.            0.            0.
     0.            0.            0.            0.        ]
 [8553.38087295    0.         6905.76352854    0.            0.
     0.            0.            0.            0.        ]
 [   0.         7672.54729148    0.            0.            0.
  6212.24253099    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.         5539.05234247    0.            0.        ]
 [   0.         7689.7576031     0.            0.            0.
     0.            0.         6154.78668848    0.        ]
 [   0.            0.         6905.40892862    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.         4954.6377079     0.
     0.            0.         6201.05117126    0.        ]
 [   0.            0.            0.            0.         6905.35160779
     0.         5460.77263625    0.         5532.99693754]
 [   0.            0.            0.            0.            0.
     0.            0.         6204.71230811    0.        ]]
(['L9', 'L8', 'L5', 'L2', 'L1'], 4)
```

Figure 8

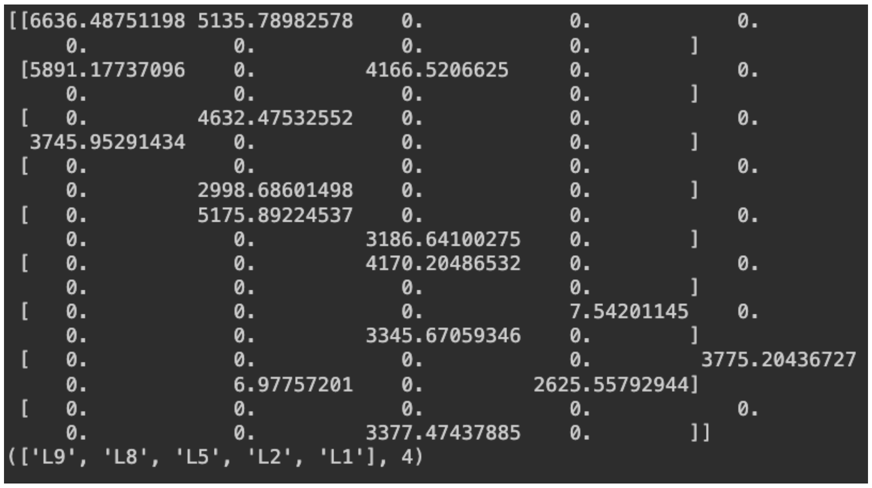
Figure of the Q-value array for 1000 iterations.



```
[[9927.08153536 8039.33661459    0.            0.            0.
     0.            0.            0.            0.        ]
 [8935.36983441    0.         7238.99458863    0.            0.
     0.            0.            0.            0.        ]
 [   0.         8042.28597811    0.            0.            0.
  6515.18174885    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.         5844.35735711    0.            0.        ]
 [   0.         8042.81224096    0.            0.            0.
     0.            0.         6495.11743785    0.        ]
 [   0.            0.         7238.81669727    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.         5237.94305555    0.
     0.            0.         6493.06367112    0.        ]
 [   0.            0.            0.            0.         7233.60277419
     0.         5844.07973092    0.         5845.74611214]
 [   0.            0.            0.            0.            0.
     0.            0.         6507.22820416    0.        ]]
(['L9', 'L8', 'L5', 'L2', 'L1'], 4)
```

Figure 9

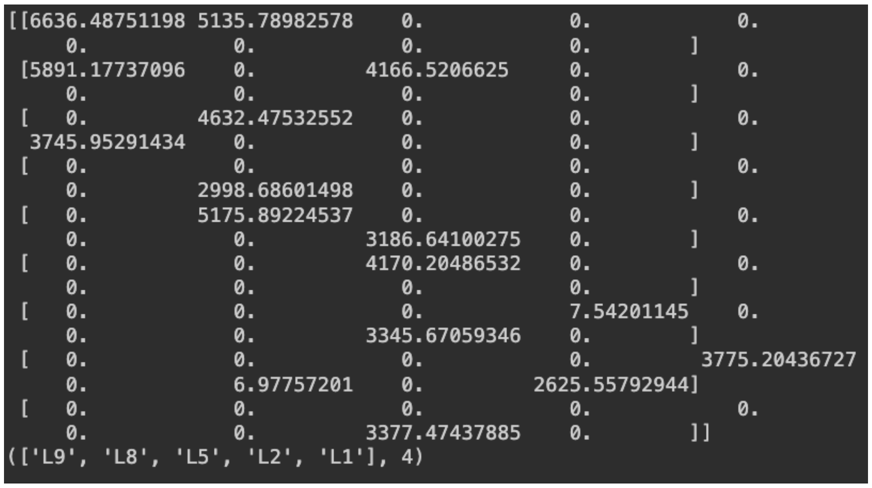
Figure of the Q-value array for 200 iterations.



```
[[6636.48751198 5135.78982578    0.            0.            0.
     0.            0.            0.            0.        ]
 [5891.17737096    0.         4166.5206625     0.            0.
     0.            0.            0.            0.        ]
 [   0.         4632.47532552    0.            0.            0.
  3745.95291434    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.         2998.68601498    0.            0.        ]
 [   0.         5175.89224537    0.            0.            0.
     0.            0.         3186.64100275    0.        ]
 [   0.            0.         4170.20486532    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.            7.54201145    0.
     0.            0.         3345.67059346    0.        ]
 [   0.            0.            0.            0.         3775.20436727
     0.            6.97757201    0.         2625.55792944]
 [   0.            0.            0.            0.            0.
     0.            0.         3377.47437885    0.        ]]
(['L9', 'L8', 'L5', 'L2', 'L1'], 4)
```

Figure 10

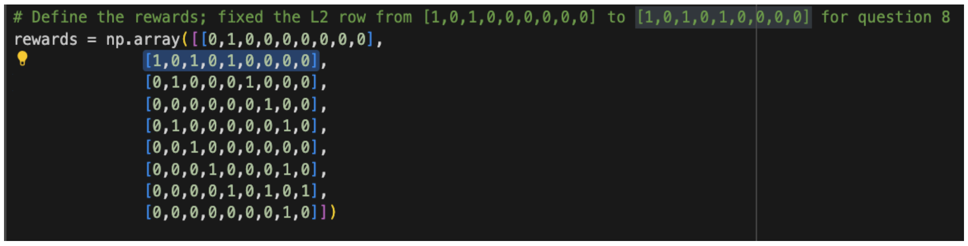
Figure of the Q-value array for 50 iterations.



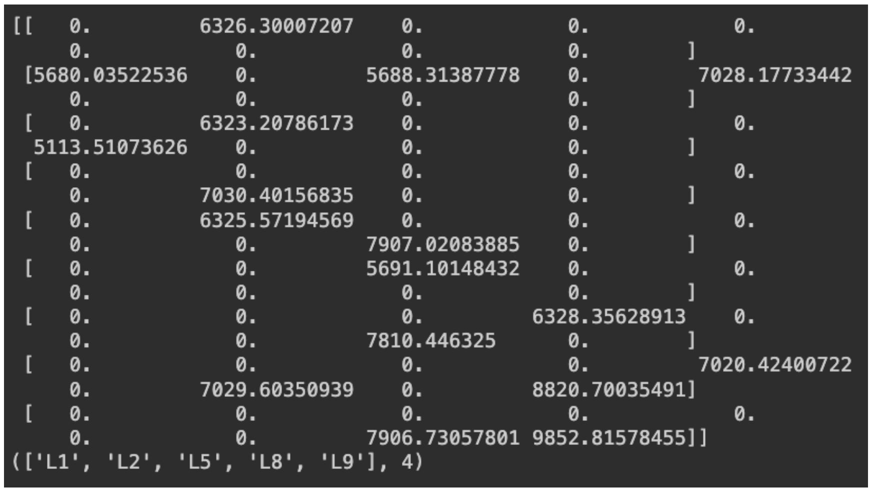
```
[[6636.48751198 5135.78982578    0.            0.            0.
     0.            0.            0.            0.        ]
 [5891.17737096    0.         4166.5206625     0.            0.
     0.            0.            0.            0.        ]
 [   0.         4632.47532552    0.            0.            0.
  3745.95291434    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.         2998.68601498    0.            0.        ]
 [   0.         5175.89224537    0.            0.            0.
     0.            0.         3186.64100275    0.        ]
 [   0.            0.         4170.20486532    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.            7.54201145    0.
     0.            0.         3345.67059346    0.        ]
 [   0.            0.            0.            0.         3775.20436727
     0.            6.97757201    0.         2625.55792944]
 [   0.            0.            0.            0.            0.
     0.            0.         3377.47437885    0.        ]]
(['L9', 'L8', 'L5', 'L2', 'L1'], 4)
```

Figure 11

Figures of the fixed rewards matrix and the output for L1 to L9.



```
# Define the rewards; fixed the L2 row from [1,0,1,0,0,0,0,0,0] to [1,0,1,0,1,0,0,0,0] for question 8
rewards = np.array([[0,1,0,0,0,0,0,0,0],
              [1,0,1,0,1,0,0,0,0],
              [0,1,0,0,0,1,0,0,0],
              [0,0,0,0,0,0,1,0,0],
              [0,1,0,0,0,0,0,1,0],
              [0,0,1,0,0,0,0,0,0],
              [0,0,0,1,0,0,0,1,0],
              [0,0,0,0,1,0,1,0,1],
              [0,0,0,0,0,0,0,1,0]])
```



```
[[   0.         6326.30007207    0.            0.            0.
     0.            0.            0.            0.        ]
 [5680.03522536    0.         5688.31387778    0.         7028.17733442
     0.            0.            0.            0.        ]
 [   0.         6323.20786173    0.            0.            0.
  5113.51073626    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.         7030.40156835    0.            0.        ]
 [   0.         6325.57194569    0.            0.            0.
     0.            0.         7907.02083885    0.        ]
 [   0.            0.         5691.10148432    0.            0.
     0.            0.            0.            0.        ]
 [   0.            0.            0.         6328.35628913    0.
     0.            0.         7810.446325      0.        ]
 [   0.            0.            0.            0.         7020.42400722
     0.         7029.60350939    0.         8820.70035491]
 [   0.            0.            0.            0.            0.
     0.            0.         7906.73057801 9852.81578455]]
(['L1', 'L2', 'L5', 'L8', 'L9'], 4)
```

Figure 12

Figure of the sample environment with the new tenth state, L10.

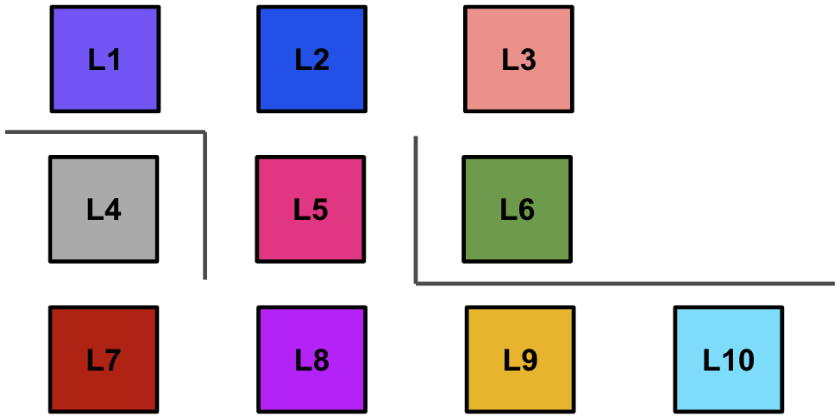
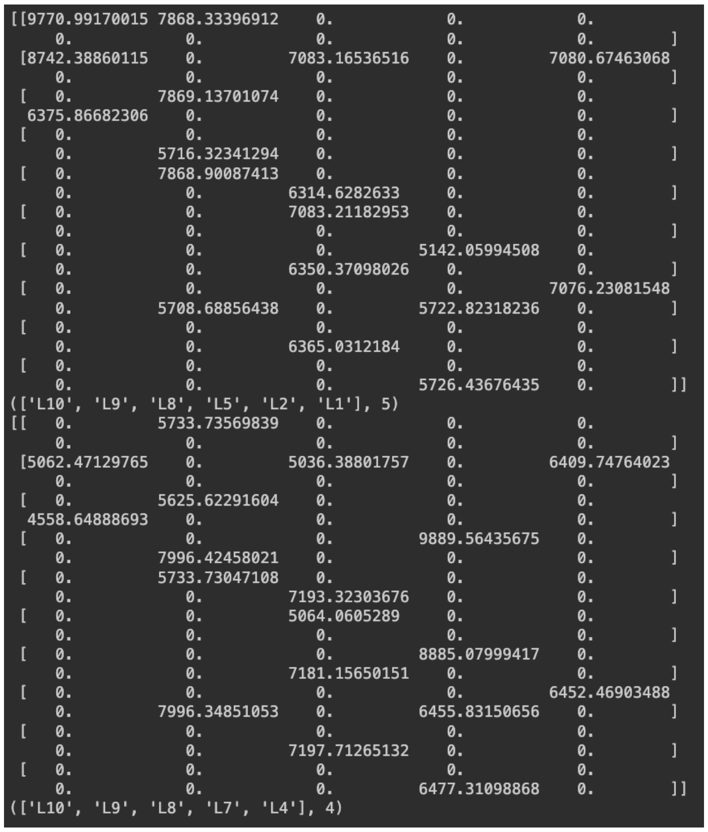


Figure 13

Figure of the Q-value array and output for the paths L10 to L1 (first output) and L10 to L4 (second output).



```
[[9770.99170015 7868.33396912    0.            0.            0.
     0.            0.            0.            0.            0.        ]
 [8742.38860115    0.         7083.16536516    0.         7080.67463068
     0.            0.            0.            0.            0.        ]
 [   0.         7869.13701074    0.            0.            0.
  6375.86682306    0.            0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.         5716.32341294    0.            0.            0.        ]
 [   0.         7868.90087413    0.            0.            0.
     0.            0.         6314.6282633     0.            0.        ]
 [   0.            0.         7083.21182953    0.            0.
     0.            0.            0.            0.            0.        ]
 [   0.            0.            0.         5142.05994508    0.
     0.            0.         6350.37098026    0.            0.        ]
 [   0.            0.            0.            0.         7076.23081548
     0.         5708.68856438    0.         5722.82318236    0.        ]
 [   0.            0.            0.            0.            0.
     0.            0.         6365.0312184     0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.            0.            0.         5726.43676435    0.        ]]
(['L10', 'L9', 'L8', 'L5', 'L2', 'L1'], 5)
[[   0.         5733.73569839    0.            0.            0.
     0.            0.            0.            0.            0.        ]
 [5062.47129765    0.         5036.38801757    0.         6409.74764023
     0.            0.            0.            0.            0.        ]
 [   0.         5625.62291604    0.            0.            0.
  4558.64888693    0.            0.            0.            0.        ]
 [   0.            0.            0.         9889.56435675    0.
     0.         7996.42458021    0.            0.            0.        ]
 [   0.         5733.73047108    0.            0.            0.
     0.            0.         7193.32303676    0.            0.        ]
 [   0.            0.         5064.0605289     0.            0.
     0.            0.            0.            0.            0.        ]
 [   0.            0.            0.         8885.07999417    0.
     0.            0.         7181.15650151    0.            0.        ]
 [   0.            0.            0.            0.         6452.46903488
     0.         7996.34851053    0.         6455.83150656    0.        ]
 [   0.            0.            0.            0.            0.
     0.            0.         7197.71265132    0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.            0.            0.         6477.31098868    0.        ]]
(['L10', 'L9', 'L8', 'L7', 'L4'], 4)
```

Code Appendix

The Python script used to generate the algorithm and the output can be seen below and is also attached as a separate file.

```
# Modified code

# Import only numpy
import numpy as np
# Initialize parameters
gamma = 0.9 # Discount factor
alpha = 0.75 # Learning rate

# Adjusted parameters to 0.05 for both for question 5
# gamma = 0.05 # Discount factor.
# alpha = 0.05 # Learning rate.

# # Adjusted parameters to 0.05 for gamma and reverted to
original value for alpha for question 5
# gamma = 0.05 # Discount factor.
# alpha = 0.75 # Learning rate.

# # Adjusted parameters to 0.05 for alpha and reverted to
original value for gamma for question 5
# gamma = 0.9 # Discount factor.
# alpha = 0.05 # Learning rate.

# Define the states; added the tenth state, L10 for question 9
location_to_state = {
   'L1' : 0,
   'L2' : 1,
   'L3' : 2,
   'L4' : 3,
   'L5' : 4,
   'L6' : 5,
    'L7' : 6,
    'L8' : 7,
    'L9' : 8,
    'L10' : 9
}
# Define the actions; added the 10th action for question 9
actions = [0,1,2,3,4,5,6,7,8,9]
# Define the rewards; fixed the L2 row from [1,0,1,0,0,0,0,0,0]
to [1,0,1,0,1,0,0,0,0] for question 8; added L10 to the matrix
for question 9
rewards = np.array([[0,1,0,0,0,0,0,0,0,0],
                    [1,0,1,0,1,0,0,0,0,0],
                    [0,1,0,0,0,1,0,0,0,0],
                    [0,0,0,0,0,0,1,0,0,0],
                    [0,1,0,0,0,0,0,1,0,0],
                    [0,0,1,0,0,0,0,0,0,0],
                    [0,0,0,1,0,0,0,1,0,0],
                    [0,0,0,0,1,0,1,0,1,0],
                    [0,0,0,0,0,0,0,1,0,1],
                    [0,0,0,0,0,0,0,0,1,0]])

# Maps indices to locations
state_to_location = dict((state,location) for location,state in
location_to_state.items())
def get_optimal_route(start_location,end_location):
    # Copy the rewards matrix to new Matrix
    rewards_new = np.copy(rewards)
    # Get the ending state corresponding to the ending location
as given
    ending_state = location_to_state[end_location]
    # With the above information automatically set the priority
of the given ending state to the highest one
    rewards_new[ending_state,ending_state] = 999
    # -----------Q-Learning algorithm-----------
    # Initializing Q-Values; changed array size to 10x10 for
question 9
    Q = np.array(np.zeros([10,10]))
    # Q-Learning process; changed the parameter value to 200 and
50 for question 7
    for i in range(1000):
        # Pick up a state randomly
        current_state = np.random.randint(0,10) # Python excludes
the upper bound; changed the parameter value for question 9
        # For traversing through the neighbor locations in the
maze
        playable_actions = []
        # Iterate through the new rewards matrix and get the
actions > 0
        for j in range(9):
            if rewards_new[current_state,j] > 0:
                playable_actions.append(j)
        # Pick an action randomly from the list of playable
actions leading us to the next state
        next_state = np.random.choice(playable_actions)
        # Compute the temporal difference
        # The action here exactly refers to going to the next
state
        TD = rewards_new[current_state,next_state] + gamma *
Q[next_state, np.argmax(Q[next_state,])] -
Q[current_state,next_state]
        # Update the Q-Value using the Bellman equation
        Q[current_state,next_state] += alpha * TD
    # Initialize the optimal route with the starting location
    route = [start_location]
    # We do not know about the next location yet, so initialize
with the value of starting location
    next_location = start_location
```

```
   # Adding step/iteration counter before the while loop for
question 6
   steps = 0
   # We don't know about the exact number of iterations needed
to reach to the final location hence while loop will be a good
choice for iteratiing
   while(next_location != end_location):
       # Fetch the starting state
       starting_state = location_to_state[start_location]
       # Fetch the highest Q-value pertaining to starting state
       next_state = np.argmax(Q[starting_state,])
       # We got the index of the next state. But we need the
corresponding letter.
       next_location = state_to_location[next_state]
       route.append(next_location)
       # Update the starting location for the next iteration
       start_location = next_location
       # Step counter
       steps = steps + 1

   # Print statement to get Q-value array for question 5
   print(Q)
   # Edited print statement to get step count for question 6
   return route, steps
 # print(get_optimal_route('L9', 'L1'))

# Reverse path for question 8
# print(get_optimal_route('L1', 'L9'))

# Testing paths from L10 for question 9
print(get_optimal_route('L10', 'L1'))
print(get_optimal_route('L10', 'L4'))
```