Data 640: Predictive Modeling Fall 2023

Assignment # 5b- Reinforcement Learning Main Assignment

Simon A Hochmuth

Email: shochmuth3@student.umgc.edu

Professor: Ed Herranz

Introduction:

This assignment is to analyze a Q-Learning model that has the goal of finding the shortest path in a warehouse based on a preset reward system. Where the agent will be made using reinforced learning, and it will output the fastest way to get from point A to B. The goal is to analyze, review, and change the supplied code to get a better understanding Q-Learning in practice.

Overview of the Q-Learning Code / Question 4:

The agent's job is to find the shortest path between each of the locations shown in figure 1 below. Sayak Paul described the map as a guitar warehouse with guitar parts at the various parts depicted L1 to L9. However, this could be anything from an Amazon warehouse to a helicopter factory, the theory will stay the same. The goal of the reinforcement learning model is

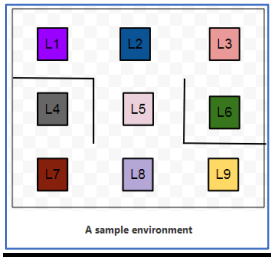

Figure 1: Map showing the positions of L1 to L9, with obstacles at L6 & L4

to help guide the agents, which in this case are warehouse robots, to a specific location in the warehouse. When looking at the map the black lines depict obstacles, likely walls. For this reason, robots positioned at L4, L7, L8, or L9 can only get to L1, L2, L3, or L6 by moving through the chokepoint at L5. These obstacles were created to create a prioritization at the L6 point. Lastly, the code does not allow for diagonal movement, for example if the robot was at L5 it would be unable to reach L3 without going to L2 first. The rewards within the code essentially describe the movement process previously discussed. Where L1 is in the top left can only move to L2, thus the reward is [0,1,0,0,0,0,0,0,0] where the 0 and 1 values are placed in the equivalent

locations for the points [L1,L2,L3,L4,L5,L6,L7,L8,L9]. For locations with more than one reward, or movement option, the code will choose the next move and calculate a score based on that. For instance, if a robot is at L5 and wants to move to L3, the agent would rate L2 higher than L8 because that is in the correct direction of movement. For longer paths this will continue to apply, where the value of 1 will be changed to increase or decrease the score so the robot will choose the correct direction by following the highest scored path. For example, if the robot was at L8, figure 3 shows the Q-values for the robot that would like to move to L1. Where a higher value

Figure 2: Q-values for a robot that would like to move towards L1 from L8.

of 7210 is placed in the L5 location because it moves the robot closer to the L1 position. Also, L9 and L7 would move the robot further from the location, which is why these two have lower values at approximately 5800. This pushes the agent to choose 7210, or to move to L5.

Question 3:

As previously discussed, the robot must move through L5 if it would like to get to the other side of the obstacles. For this question, the robot starts at the L9 point and wants to get to the L1 point. The robot will then look at the path and create scores for the paths that lead to L1 and provide higher scores for locations on the optimal path. In this case those points are L9, L8, L5, L2, and finally L1. Where the path can be seen in figure 3 below, and the Q-Values can be seen in figure 4 below. Notice how the rewards change based on the location of the robot, and the path that leads to the desired point. If you follow the rewards in figure 4, it will recreate the path shown in figure 3.

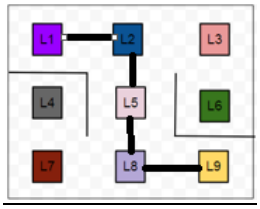

Figure 3: Shows the shortest path from L9 to L1.

```
    return route,Q
route,Q = get_optimal_route('L9', 'L1')
print(route)
print(Q)
```

```
['L9', 'L8', 'L5', 'L2', 'L1']
[[9904.06988513 8014.95338761    0.           0.           0.
     0.           0.           0.        ]
 [8910.78704698    0.         7168.36395012    0.           0.
     0.           0.           0.        ]
 [   0.         8009.77096068    0.           0.           0.
  6442.63489921    0.           0.           0.        ]
 [   0.           0.           0.           0.           0.
     0.         5815.92702196    0.           0.        ]
 [   0.         8017.95464737    0.           0.           0.
     0.           0.         6469.42870491    0.        ]
 [   0.           0.         7208.97500145    0.           0.
     0.           0.           0.        ]
 [   0.           0.           0.         5222.17906755    0.
     0.           0.         6461.16688463    0.        ]
 [   0.           0.           0.           0.        7210.60406894
     0.         5813.96324322    0.         5775.17849085]
 [   0.           0.           0.           0.           0.
     0.           0.         6472.63134484    0.        ]]
```

Figure 4: Shows the Q-Values for the path L9 to L1, with increased scores for the quickest path.

Question 5:

Now both the discount factor *gamma* and the learning rate *alpha* were each set to be 0.05, and the agent was told again to give the optimal route between L9 and L1. The agent was unable to return a path. This is because both gamma and alpha relate to how the agent learns, by dropping these values to 0.05, it makes so the agent does not learn enough through its training of 1000 iterations. A way to fix this is to either increase gamma or alpha or increase the number of iterations it learns through. For instance, iterations = 10000 results in the expected optimal route.

Question 6:

The number of times the while loop gets executed should equal the number of steps it takes to get from the start to the end of the desired path. For instance, one would expect the number of steps for the path [L1,L2] to be 1 because it is only one step between the two points, where [L9,L1] would need to step from [L9,L8],[L8,L5],[L5,L2], and finally [L2,L1] for a total of 4 steps. After adding a step counter into the code, the code was rerun to return the number of steps, which confirmed 4 steps, and is shown in figure 6 below.

```
            steps += 1

    return route,Q,steps
route,Q,steps = get_optimal_route('L9', 'L1')
print(route)
#print(Q)
print("The amount of times the while loop was run in the get_optimal_route(L9,L1) function was {}" .format(steps))
```

```
['L9', 'L8', 'L5', 'L2', 'L1']
The amount of times the while loop was run in the get_optimal_route(L9,L1) function was 4
```

Figure 6: Shows the Q-Values for the path L9 to L1, with gamma & alpha = 0.05.

Question 7:

Next the number of iterations during the Q-Learning process was changed from 1000 to 50 as shown in figure 7 below. This ended up in an agent that was unable to return a path. This is since it was unable to learn enough in the 50 iterations. By increasing this value to 200, it was then able to result in the correct optimal route, then it was slowly lowered around 150 was the minimum amount of iteration it needed before it failed to return. This shows less than 1000 iterations is possible, but since this has a low runtime it does not hurt to increase agent training.

Question 8:

The agent was now asked to get_optimal_route('L1','L9'), which would expect to return L1,L2,L5,L8, and finally L9. Which would be the reverse of what was seen in the previous examples. However, in this case the agent is unable to return a value, where the rewards show why in figure 8 below. Where the L1 array shows a rewards value in the L2 position, but in the L2 array shows a 0 for the L5 position, which in turn would mean that the agent would never be able to reach the L5 position, therefore never finding an optimal path. The result after changing the 0 to a 1 in the L5 position can be seen in figure 9 below.

```
# Define the rewards
rewards = np.array([[0,1,0,0,0,0,0,0,0],
                    [1,0,1,0,0,0,0,0,0],
                    [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]])
```

Figure 8: The initial rewards array for each position

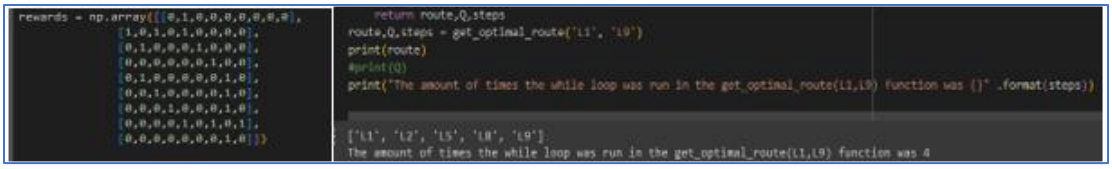

Figure 9: final rewards table and results.

Question 9:

Now an additional position has been created called L10, which will be to the right of the L9 position and only able to move through L9. The code to create that added this addition can be found in the code appendix below. After the code was implemented, the optimal route from L10 to L1 and L10 to L4 were found. The results can be found in figure 10 below, where L10 to L1 had 5 steps to reach the end of the path and L10 to L4 had 4 steps. Both these results are expected, because L10 to L1, should essentially mimic L9 to L1 with the addition of the L10 to L9 step, which it does. The L10 to L4 path should not go through L5 due to the obstacle and pass through L7. For that reason, both paths had results that were expected.

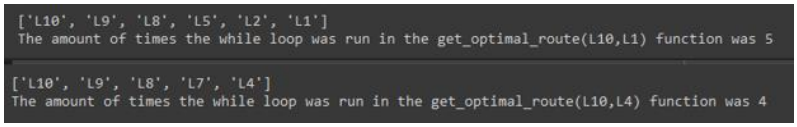

Figure 10: Shows results for routes L10 to L4 and L10 to L1

Conclusions and takeaways:

The agent was successful in finding optimal routes. However, it was limited by the values for the reward array, training iterations, and the training parameters. By lowering the training parameters gamma and alpha, the agent can learn slower even to a point that it cannot compute a route. The same situation occurs for a lowering subset of training iterations. If another location or point needs to be added, it must be added as a point, action, and have a reward array dedicated to it. Be sure to write the reward array to ensure movement can occur in the direction of interest. If all of values are correctly set up, then the agent will likely be accurate in providing the optimal route.

References:

Paul, S. (2023, February 7). *An introduction to Q-learning: Reinforcement learning*. FloydHub Blog. https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/

Code Appendix for Question 9

```
#source: https://github.com/sayakpaul/FloydHub-Q-Learning-
Blog/blob/master/Q-Learning%20using%20numpy.ipynb

# Only numpy
import numpy as np

# Initialize parameters
gamma = 0.9 # Discount factor
alpha = 0.75 # Learning rate

# Define the states
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
actions = [0,1,2,3,4,5,6,7,8,9]

# Define the rewards
rewards = np.array([
              [0,1,0,0,0,0,0,0,0,0], #1
              [1,0,1,0,0,0,0,0,0,0], #2
              [0,1,0,0,0,1,0,0,0,0], #3
              [0,0,0,0,0,0,1,0,0,0], #4
              [0,1,0,0,0,0,0,0,0,0], #5
              [0,0,1,0,0,0,0,1,0,0], #6
              [0,0,0,1,0,0,0,1,0,0], #7
              [0,0,0,0,1,0,1,0,1,0], #8
              [0,0,0,0,0,0,0,1,0,1], #9
              [0,0,0,0,0,0,0,0,1,0]  #10
              ])

# 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
    Q = np.array(np.zeros([10,10]))

    # Q-Learning process
    for i in range(1000):
        # Pick up a state randomly
        current_state = np.random.randint(0,10) # Python excludes the
upper bound
        # 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(10):
            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
    #print(Q)
    # 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
    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
        steps += 1

    return route,Q,steps
route,Q,steps = get_optimal_route('L10', 'L1')
print(route)
#print(Q)
print("The amount of times the while loop was run in the
get_optimal_route(L10,L1) function was {}" .format(steps))
```