

Title Page

Assignment 5b: Reinforcement Learning

Melissa Hunfalvay

Email: Melissa.Hunfalvay@gmail.com

Data 640 9040

Spring 2022

Professor Steve Knode

Date: March 29th, 2022

Introduction

The purpose of this assignment is to train a robot to navigate an environment (a guitar building factory) such that the robot moves through the factory in the most efficient manner to collect the required parts for building the guitar. To achieve this goal requires that the robot learn the factory layout (environment), the rooms or locations within the environment (states, $n = 9$) and how to navigate them (actions).

Not all the locations within the factory are of equal importance. Body woods are of the highest priority location within the factory. Furthermore, the robot can start from any location (state) on the factory floor. Finally, there are some locations that cannot be directly located as there are obstacles preventing access from various angles (states). Hence, the specific problem statement is defined as training the robots so they can find learn the shortest route from any starting point (state) within the factory floor (environment) to any other location in the factory on their own.

Question 4: Q-Learning Code Described

Original code is credited to Sayak Paul (2019). The Q-learning code steps are outlined in Table 1.

Table 1: Q-Learning Code Explained

Step	Purpose	Description	Sample Code
1	Imports	Import numpy	import numpy as np
2	Initialize key parameters		
a	α	Learning rate	alpha = 0.9
b	γ	Discount Factor	gamma = 0.75
c	States	Define the states (rooms) on the factory floor from L1 to L9, call them 1 to 9 for Python code	'L1': 0,
d	Rewards	Map out the rewards within the environment by assigning 1 to states where we want the robot to move and 0 to locations that we do not want the robot to move (i.e. where they obtain no reward).	0,1,0,0,0,0,0,0,0
e	Backward mapping	Inverse the map from the states back to the original locations	state_to_location = dict((state,location) for location,state in location_to_state.items())
f	Actions	Define the actions or moves the robots can take from room to room within the environment (factory).	actions = [0,1,2,3,4,5,6,7,8]
g	Optimal Route	Create a function to define the optimal route for the robot to travel from any starting to ending points (arguments)	def get_optimal_route(start_location,end_location):
3	Q-Learning Algorithm		
a	Initialize the Q-values	Clear the board and make all the values zero to begin	Q = np.array(np.zeros([9,9]))
b	Learn Robot - Learn	1000 tries for the robot to learn the environment	for i in range(1000):
c	Choose a starting state	Randomly pick from 1 of the 9 states the starting point for exploration	current_state = np.random.randint(0,9)
d	Find the rewards	Iterate through the environment with the reward matrix to get to the states that are directly reachable from the starting points	playable_actions = []
e	Append	Find actions >0 and update the matrix	playable_actions.append(i)
f	Choose a starting action	Randomly choose which direction to move from the current state	next_state = np.random.choice(playable_actions)
g	Temporal equation	Compute the temporal difference before 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]
h	Update the Q-Value	Using the Bellman equation: $Q_t(s,a) = Q_{t-1}(s,a) + \alpha TD_t(s,a)$	Q[current_state,next_state] += alpha * TD
4	Optimal route function		
a	Initialize the Route	Find the starting point from which you want the robot to move	route = [start_location]
b	Set the next location	As we don't know how the robot will travel we need to set the next location to also be the starting location	next_location = start_location
c	While loop	While the next location is not equal to the ending location ...then, keep iterating.	while(next_location != end_location):
d	Start location	Find the starting state on the matrix	starting_state = location_to_state[start_location]
e	Q-Value	Find the highest Q-value from that starting state	next_state = np.argmax(Q[starting_state,])
f	Letter	Find the corresponding letter to the next state	next_location = state_to_location[next_state]
g	Append route	Update the route	route.append(next_location)
h	Append start location	Update the start location to the most current state	start_location = next_location
4	Print	Show me, step-by-step how the robot traveled through the matrix	['L9', 'L8', 'L5', 'L2', 'L1']

Question 3: Explain the Optimal Route

Figure 1 shows the print command and output from the base code.

```
print(get_optimal_route('L9', 'L1'))  
['L9', 'L8', 'L5', 'L2', 'L1']
```

Figure 1: Optimal Route Output from Base Code

The `get_optimal_route` command had the starting position as location L9 on the environmental map and L1 as the desired location. The route the robot took from the starting position (L9) to the desired end position (L1) is shown in Figure 2 via the red arrows. This is the most efficient path for the robot to take given that there is a barrier around L6 and L4. This result shows the robot did learn how to effectively travel from L9 to L1.

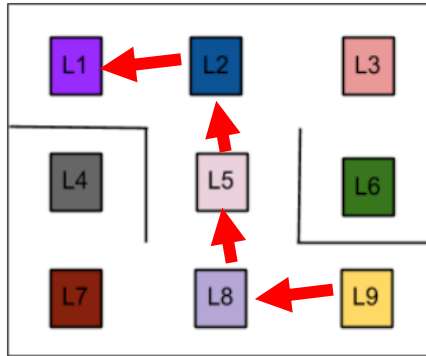


Figure 2: Route taken by the Robot from L9 to L1

Question 5: Alpha (α) & Gamma (γ) Parameters

The Alpha parameter refers to learning rate, which controls how quickly the robot adapts to the environment. A low alpha represents a small magnitude of steps for the robot to take. If too small the processing time and, in turn, the learning time can be very long. A large alpha will result in the robot learning quickly however, the tradeoff may be sub-optimal steps (or weighted values in the matrix).

The Gamma parameter refers to the discount rate, which quantifies how far the robot is from the desired reward. If the gamma is low (0.1) the robot is far from the destination, a high gamma (0.9) indicates the robot is close to the destination. Both Alpha and Gamma values range from 0-1.0. Results for various alpha and gamma parameters are shown in table 2.

Table 2: Alpha and Gamma Hyperparameter Levels and Results from L9 to L1

Gamma	Alpha	Result of Route L9 to L1	Factory Floor/Gamma States												
0.9	0.75	L9, L8, L5, L2, L1. As described above this set of parameters got the desired result moving effectively from L9 to L1 in a most efficient manner.	<table><tr><td>L1</td><td>1.0</td><td>0.9</td></tr><tr><td>1.0</td><td>0.9</td><td>0.81</td></tr><tr><td>0.9</td><td>0.81</td><td>0.729</td></tr><tr><td>0.81</td><td>0.729</td><td>L9 0.66</td></tr></table>	L1	1.0	0.9	1.0	0.9	0.81	0.9	0.81	0.729	0.81	0.729	L9 0.66
L1	1.0	0.9													
1.0	0.9	0.81													
0.9	0.81	0.729													
0.81	0.729	L9 0.66													
0.05	0.05	Very low alpha rate (small magnitude of changes) and low gamma rate which guides the robot toward the destination, resulted in long processing times. Ultimately, resulting in system crash and no results were able to be produced for this option.	<table><tr><td>L1</td><td>1.0</td><td>0.95</td></tr><tr><td>1.0</td><td>0.95</td><td>0.9</td></tr><tr><td>0.95</td><td>0.9</td><td>0.85</td></tr><tr><td>0.9</td><td>0.85</td><td>L9 0.8</td></tr></table>	L1	1.0	0.95	1.0	0.95	0.9	0.95	0.9	0.85	0.9	0.85	L9 0.8
L1	1.0	0.95													
1.0	0.95	0.9													
0.95	0.9	0.85													
0.9	0.85	L9 0.8													
1.0	1.0	Very high alpha resulted in the robot not having any guidance from the starting position. Coupled with high gamma rates resulted in no guidance from the starting position towards the destination. This too resulted in a system crash and no results were able to be produced for this option.	<table><tr><td>L1</td><td>1.0</td><td>0</td></tr><tr><td>1.0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>L9 0</td></tr></table>	L1	1.0	0	1.0	0	0	0	0	0	0	0	L9 0
L1	1.0	0													
1.0	0	0													
0	0	0													
0	0	L9 0													
0.75	0.60	L9, L8, L5, L2, L1. High-to-moderate gamma and moderate alpha allowed the robot to take moderate steps towards the goal <i>and</i> be guided by meaningful changes in gamma rates to reinforce the reward states.	<table><tr><td>L1</td><td>1.0</td><td>0.75</td></tr><tr><td>1.0</td><td>0.75</td><td>0.50</td></tr><tr><td>0.75</td><td>0.50</td><td>0.25</td></tr><tr><td>0.50</td><td>0.25</td><td>L9 0</td></tr></table>	L1	1.0	0.75	1.0	0.75	0.50	0.75	0.50	0.25	0.50	0.25	L9 0
L1	1.0	0.75													
1.0	0.75	0.50													
0.75	0.50	0.25													
0.50	0.25	L9 0													

Question 6: While Loop Explained

When you run the `print(get_optimal_route('L9', 'L1'))` how many times does the while loop in the `get_optimal_route()` get executed? 4 times. Why is it that number? Because: 1. There are four moves (actions) taken by the robot from L9 to L8 to L5 to L2 to L1. 2. The Q-Values get higher as the robot makes each move, which directs the robot to the most efficient path which has 4 moves. 3. The end state of `end_location` is not reached until the robot moves to L1. At L1 the while loop ends, as `next_location == end_location`. The while condition is not satisfied and therefore, the while loop is exited.

Question 7: Number of Iterations to find the Optimal Loop

Original code uses 1000 iterations, however, only 105 iterations are needed to obtain a successful result (see Table 3).

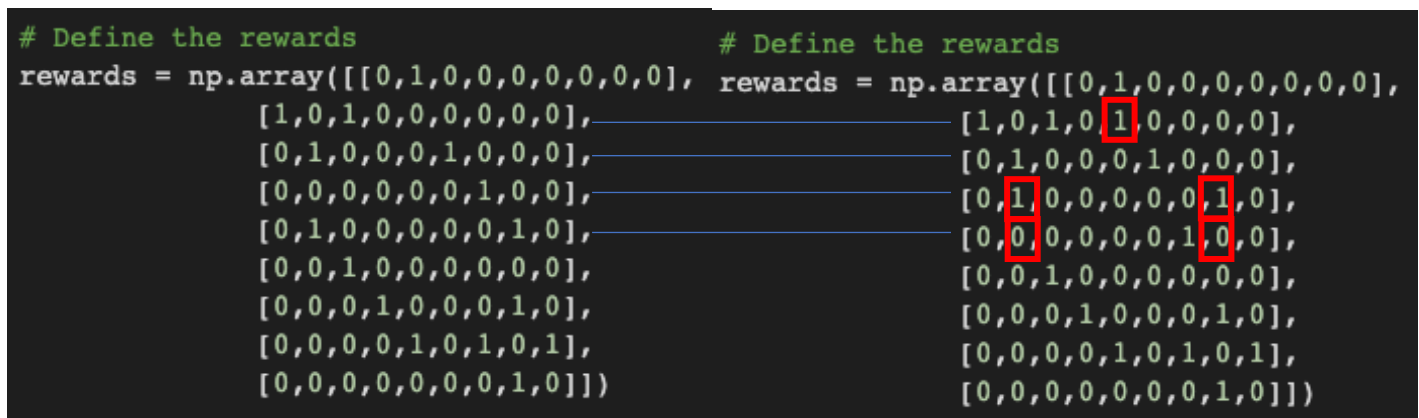
Table 3: Varying the Number of Iterations

# Iterations	Result and Explanation (Route L9 to L1)
1000	Results = L9, L8, L5, L2, L1. After 1000 iterations the code return the correct result.
50, 100	Results = no return. 50 or 100 iterations are not enough for the robot to know how to move through the environment. Therefore, no result is returned, and the system keeps running an endless loop.
105, 110, 115, 125, 150, 200	Results = L9, L8, L5, L2, L1. 200 iterations gives the robot enough learning iterations to understand the environment and the states within in order to form a correct and efficient path

Question 8: Reverse Path

When running the reverse path from L1 to L9 the code keeps iterating without returning a result. This is because we are asking the robot to go to a destination location that is not desired (i.e. L9). The gamma states are lower the further from L1 and therefore the robot does not want to go from a higher gamma state to a lower one. Gamma at L2 is 1.0 which is the highest reward state. Gamma at L5 is 0.9 and therefore the robot will not want to make that move and will remain “stuck” at L2.

To enable the robot to navigate from L2 to L5 the rewards need to be adjusted (see Figure 3). Figure 3.a shows the original matrix and reward structure. Figure 3.b. shows an adjustment to the gamma reward values to enable the robot to follow the higher reward states and therefore reverse the path and become “unstuck” from L2 to L5 (see Figure 4, print output).



3.a: Original Matrix

3.b: Adjusted matrix

Figure 3: Show adjustment of the reward states to encourage the robot to take actions from L2 to L5

```
print(get_optimal_route('L1', 'L9'))  
['L1', 'L2', 'L5', 'L7', 'L8', 'L9']
```

Figure 4: Print command for adjusted matrix from Figure 3.b.

Question 9: Adding a 10th State

Table 4 shows the adjusted environment (matrix) for a 10th state and Table 5 shows the adjusted discount factors.

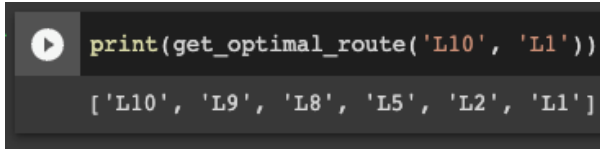
Table 4: L10 with the rewards table.

	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10
L1	0	1	0	0	0	0	0	0	0	0
L2	1	0	1	0	1	0	0	0	0	0
L3	0	1	0	0	0	1	0	0	0	0
L4	0	0	0	0	0	0	1	0	0	0
L5	0	1	0	0	0	0	0	1	0	0
L6	0	0	1	0	0	999	0	0	0	0
L7	0	0	0	1	0	0	0	1	0	0
L8	0	0	0	0	1	0	1	0	1	0
L9	0	0	0	0	0	0	0	1	0	1
L10	0	0	0	0	0	0	0	0	1	0

Table 5: Discount factor of 0.9

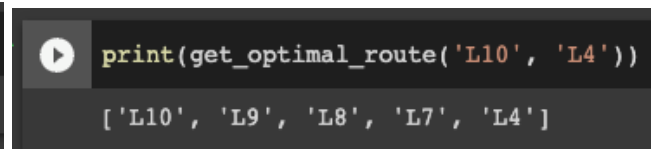
L1	1.0	0.9	0.81
1.0	0.9	0.81	0.729
0.9	0.81	0.729	0.66
0.81	0.729	0.66	L10

The Appendix shows the code changes enabling the robot to move from L10 to L1, and L10 to L4 also see Figure 5. Results in both cases are expected as the robot moves from one space to a predefined end location, the reward values increase e.g., Table 5 L9 reward is 0.66, L8 reward is 0.729. As the robot had 1000 trials to learn the environment and is given the desired end point (L1) it can successfully and efficiently navigate the factory floor in both examples (L10 to L1 and L10 to L4).



```
print(get_optimal_route('L10', 'L1'))
['L10', 'L9', 'L8', 'L5', 'L2', 'L1']
```

Figure 5.a: L10 to L1 route



```
print(get_optimal_route('L10', 'L4'))
['L10', 'L9', 'L8', 'L7', 'L4']
```

Figure 5.b: L10 to L4 route

Conclusions & Takeaways

In conclusion, RL was used to train a robot (agent) to successfully navigate (take actions), to (forward), and from (reverse), various rooms (spaces) within a factory (environment). The robot did this correctly and efficiently therefore helping the guitar luthiers convey the necessary parts needed to craft a guitar.

References

Paul, S. (2019). An introduction to Q-learning: Reinforcement learning. *Floydhub*, retrieved on March 21st, 2022, from: <https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/>

Appendix

Appendix A: Code for Question 9 with 10 States

```
# 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],
                    [1,0,1,0,0,0,0,0,0,0],
                    [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())

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

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

# 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

# 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

return route
print(get_optimal_route('L10', 'L4'))

```