Reinforcement Learning

Roy Phelps
Royphelps1@gmail.com

Table of Contents

Introduction

The task at hand is to design and implement a robotic system to optimize the efficiency and productivity of a guitar-building factory. These autonomous robots will serve as invaluable assistants to the skilled guitar luthiers, streamlining the production process by autonomously transporting essential guitar components from various locations within the factory warehouse to craftsmen. These components encompass a range of vital guitar parts, including polished wood fretboards, guitar bodies, pickups, and more.

The primary challenge lies in developing a routing system that enables the robots to navigate efficiently through the warehouse, minimizing travel time and ensuring that the luthiers have access to the parts they require promptly. A diagram of the problem is found in the appendix here *Figure 1*. The luthiers have emphasized that the polished wood for guitar bodies, stored in location L6, is of utmost importance. The autonomous robots will act as agents within the factory warehouse environment, serving as a bridge between raw materials and the skilled artisans crafting high-quality guitars.

Overview of the Q-Learning Code

The code in this analysis implements the Q-Learning algorithm to find the optimal route in a maze-like environment. The maze is represented as a grid of sates, where each state corresponds to a location in the environment. The states are defined in the **location_to_state** dictionary, and the possible actions are represented as indices in the **actions** list. The rewards for moving from one location to another are stored in the **rewards** matrix, where positive values indicate favorable moves. The Q-Learning algorithm aims to learn the optimal policy for navigating this maze by iteratively updating the Q-values, which represent the expected cumulative rewards for taking a specific action in a particular state. The algorithm uses a discount factor called, **gamma**, and a learning rate called, **alpha**, to balance exploration and exploitation.

The **get_optimal_route** function takes a location starting from the beginning and an ending location as input and used Q-Learning to degerming the optimal route from the starting location to the ending location, from **L9** to **L1**. It first modifies the **rewards** matrix to prioritize reaching the ending location by setting a high reward for the ending state. Then, it initialized the Q-values and runs the Q-Learning process for a fixed number of iterations. During this process, it explores the maze, updating the Q-values based on the temporal difference and the Bellman equation, $Q[s,a] = R(s) + \gamma \sum[P(s'|s,a) * \sum[\pi(a'|s') * Q(s',a')]]$. Once the Q-values have been learned, the function constructs the optimal route by following the path with the highest Q-values from the starting location to the ending location and returns this route as a list of locations.

Results Explanation

The Q-learning algorithm starts with the initializing of the Q-value matrix, which is a 9x9 matrix since there are 9 states. During the Q-Learning process, the algorithm iterates for 1000 episodes which in each episode, it randomly selects a current state and explores its neighboring states. Exploration is guided by the rewards and Q-values. The algorithm tends to select actions with higher rewards or Q-values. It then calculates the temporal difference (TD) using the Bellman equation and updates the Q-values accordingly. After training the Q-values, the algorithm starts at **L9** to the ending location **L1**. At each step, it chooses the next location by selecting the neighboring location with the highest Q-value and repeats this process until it reaches **L1**. So, the results are **L9, L8, L5, L2,** and **L1** as seen in *Figure 2* in the appendix.

Hyperparameter Tuning

The hyperparameters **gamma** and **alpha** in the Q-Learning algorithm control the learning process and the trade-off between exploration and exploitation. The **gamma** parameter is the discount factor that determines the importance of future rewards in the agent's decision-making process. A higher **gamma** value makes the agent prioritize long-term rewards. When the **gamma** value is set to 0.05, the agent gives relatively less importance to future rewards.

The **alpha** is the learning rate which controls how much the agent updates its Q-values based on new information. A higher **alpha** makes the agent adapt more quickly to new rewards while a lower value makes it update its Q-values more slowly.

Adjusting both hyperparameters to 0.05 each makes the learning process slower, and the agent might require more iterations to converge to an optimal policy. Running the code with these hyperparameters, results in a runtime of around the same time and with same the optimal route. The output is in *Figure 3*.

The While Loop

To find out how many times the while loop in the **get_optimal_route()** makes, adding a step counter within the while loop is performed, making sure that the hyperparameters are set back to **gamma = 0.9** and **alpha = 0.75.** The steps it took to the Q-Learning algorithm to start at location **L9** and ending at **L1** is **4**, *Figure 4*. After running the code, the Q-Learning loop backtracks from the **end_location** to the **start_location** by selecting the action with the highest Q-value at each state until it reaches the **start_location**. The number of steps it takes to reach **L1** from **L9** is 4.

Iterations

To see if changing the number of iterations changes the optimal path, the iterations are adjusted to 50. However, the output and number of steps remains the same, *Figure 5*. Another adjustment to the iterations to 200, yields the same results. So, trying an even lower number of iterations, 20, things start to become very different, *Figure 6*. From the figure, the optimal path is **L9** to **L8** to **L1** with 2 steps. With fewer iterations, the Q-values may not have fully converged to their true values, and the algorithm may not have explored all possible stat-action pairs adequately. As a result, the calculated optimal route may not be truly optimal (Lambert, 2020).

Reverse Path

Reversing the path from the starting location of **L9** to the ending **L1**, to the starting location of **L1** to the ending location of **L9**, the algorithm iterates without returning because there is no direct path from **L2** to **L5** in the rewards matrix. The rewards matrix indicates that you can only move from **L2** to **L1** and **L2** to **L3** with a reward of 1 each. There is no direct connection between **L2** and **L5**, so it keeps iterating in the while loop, trying to find a path that does not exist.

To make this case work, modification of the rewards matrix to include a path from **L2** to **L5** with a non-zero reward. This will give the Q-Learning algorithm a path to learn and find the optimal route. The adjustment of the rewards matrix can be seen here, *Figure 7*. To better visualize the change in one number in the matrix, *Figure 8* shows the comparison as a table.

State Addition

To add some possible complexity and accuracy to the algorithm, the code is modified with the addition of a tenth state, seen here in the diagram with **L10** added *Figure 9*, and the associated rewards matrix that demonstrate the effectiveness of the Q-Learning algorithm in finding the optimal routes within the environment. To complete this change, the full code is in the *Code Appendix*.

When the program is run, the request for the **optimal_route** from **L10** to **L1** is placed. The algorithm correctly identified the sequence of states, considering the reward structure, which involves transitioning from **L10** to **L9** and then from **L9** to **L1** with the most accumulative rewards. Similarly, when we run the **optimal_route** from **L10** to **L4,** the algorithm recognized the direct route from **L10** to **L4**, where there is a reward for transitioning between these two states. The results are consistent with the expiations and underscore the Q-Learning algorithm's capacity to determine the most efficient path based on the rewards and transitions defined in the environment. The results are in *Figure 10*.

Conclusions and Takeaways

In conclusion, the task of designing and implementing a robotic system to enhance the efficiency of a guitar-building factory is a challenging one, with a key focus on developing a routing system for autonomous robots to navigate the factory efficiently. The Q-Learning algorithm, as demonstrated in this analysis, offers a powerful approach for finding optimal routes in maze-like environments. The key takeaways include the importance of hyperparameters, such as **gamma** and **alpha**, which control the balance between exploration and exploitation and can significantly affect the learning process. Also, the number of iterations can impact the accuracy of the optimal route, with fewer iterations potentially leading to suboptimal results. In cases where a direct path does not exist in the rewards matrix, modifications may be required to enable the process to find a solution. Lastly, expanding the environment by introducing additional states, like the inclusion of **L10**, shows the Q-Learning algorithm's effectiveness in identifying optimal routes based on the rewards.

References

Lambert, N. (2020, April 8). *Fundamental Iterative Methods of Reinforcement Learning*. Retrieved from Towards Data Science: https://towardsdatascience.com/fundamental-iterative-methods-of-reinforcement-learning-df8ff078652a

Appendix

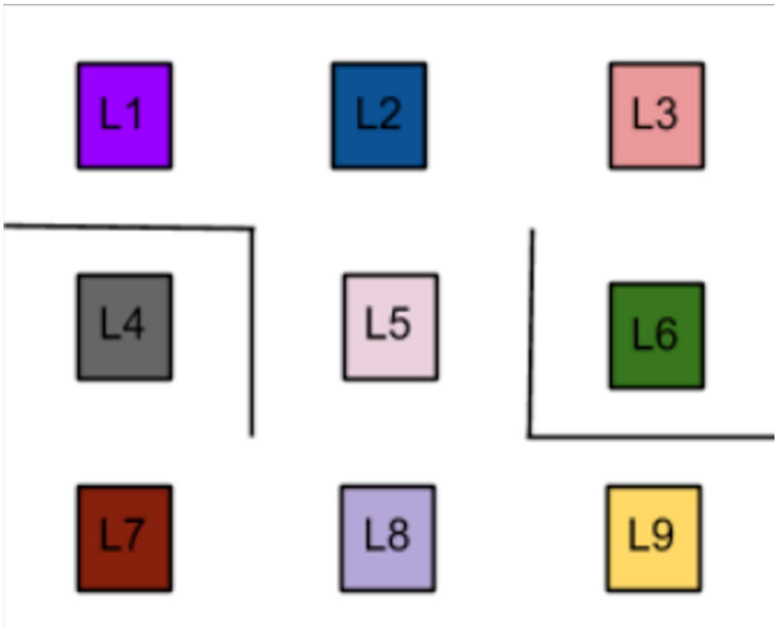

Figure 1: Problem Diagram

```
['L9', 'L8', 'L5', 'L2', 'L1']
```

Figure 2: Question 3

```
['L9', 'L8', 'L5', 'L2', 'L1']
```

Figure 3: Question 4

```
(['L9', 'L8', 'L5', 'L2', 'L1'], 4)
```

Figure 4: Question 5

```
(['L9', 'L8', 'L5', 'L2', 'L1'], 4)
```

Figure 5: Question 6

```
(['L9', 'L8', 'L1'], 2)
```

Figure 6: Question 6. 20 Iterations

```python
# Define the rewards
rewards = np.array([[0,1,0,0,0,0,0,0,0], # L1
              [1,0,1,0,1,0,0,0,0],  # L2 and added a reward from L2 -> L5
              [0,1,0,0,0,1,0,0,0],  # L3
              [0,0,0,0,0,0,1,0,0],  # L4
              [0,1,0,0,0,0,0,1,0],  # L5
              [0,0,1,0,0,0,0,0,0],  # L6
              [0,0,0,1,0,0,0,1,0],  # L7
              [0,0,0,0,1,0,1,0,1],  # L8
              [0,0,0,0,0,0,0,1,0]]) # L9
```

Figure 7: Rewards Matrix Adjustment

Rewards Matrix

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

Adjusted Rewards Matrix

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

Figure 8: Change in the Matrix

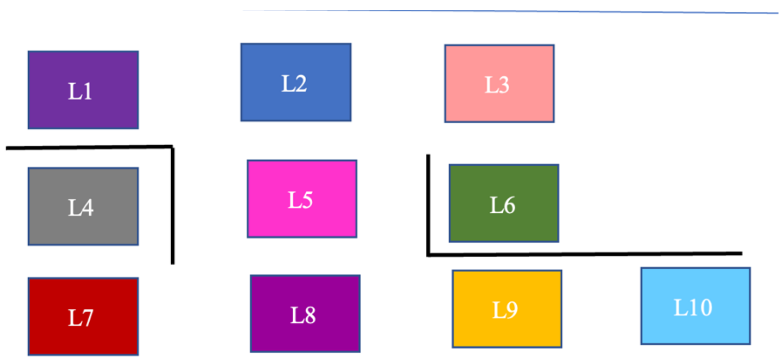

Figure 9: Question 9 Diagram

```
(['L10', 'L9', 'L8', 'L5', 'L2', 'L1'], 5)
(['L10', 'L9', 'L8', 'L7', 'L4'], 4)
```
Figure 10: Question 9

Code Appendix

```
# ~~Roy Phelps~~
# 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 # Added L10 as the tenth state
}

# Define the actions
actions = [0,1,2,3,4,5,6,7,8,9] # Added action for L10

# Define the rewards
rewards = np.array([[0,1,0,0,0,0,0,0,0,0], # L1
        [1,0,1,0,1,0,0,0,0,0],  # L2 and added a reward from L2 -> L5
        [0,1,0,0,0,1,0,0,0,0],  # L3
        [0,0,0,0,0,0,1,0,0,0],  # L4
        [0,1,0,0,0,0,0,1,0,0],  # L5
        [0,0,1,0,0,0,0,0,0,0],  # L6
        [0,0,0,1,0,0,0,1,0,0],  # L7
        [0,0,0,0,1,0,1,0,1,0],  # L8
        [0,0,0,0,0,0,0,1,0,0],  # L9
        [0,0,0,0,0,0,0,0,1,0]]) # L10

# 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]))  # Updated the size for new state L10

    # Q-Learning process
    for i in range(1000):

        # Pick up a state randomly with L10 added
        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):  # Updated range to include L10
            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

    # Execution count times
    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

        # Count the steps
        steps = steps + 1

    return route, steps

print(get_optimal_route('L10', 'L1'))
print(get_optimal_route('L10', 'L4'))
```