Reinforcement Learning in Mouse Maze Experiment

Neeve Kadosh
Rowan University

December 22, 2019

1 Abstract

Reinforcement Learning (RL) is a subcategory of Machine Learning (ML) used in training robots and achieving superhuman game play performance. RL is used with an agent in an unknown environment and will learn overtime to achieve some unknown goal. The agent is only aware of the actions that it can do; opening a door for example is a type of action. In the environment, as the agent moves around, the location or state in the environment is updated and a reward (positive or negative) for each action conducted is provided to the agent.

2 Introduction

Python will be used to simulate the experiment. A mouse will be used as the agent, a maze will be used as the environment, varying electric shocks will be used as negative rewards, and cheese will be used as the ultimate positive reward for the mouse. The objective is to place a mouse in a maze and for the mouse to learn the optimal path to take to find the cheese at the end of the maze in the least number of moves possible by utilizing Q-Learning.

3 Background: Q-Learning

Q-learning is an advanced form of RL and will be used to teach the mouse by following the following figure:

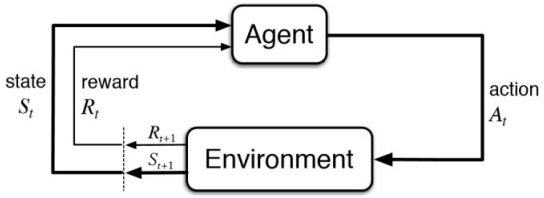

Figure 1: RL Block Diagram

Figure 1 represents the block diagram for applied RL. The agent will conduct an action in an environment. From that action, the agent will obtain a new state and a new reward. From Q-learning, the q-table will be calculated from these results. The q-table, or quality table, is a matrix of the number of states by the number of actions. A state is considered any space or cell in the maze the mouse can enter. The actions the mouse can do are up, right, down, and left. Therefore, the q-table is of shape number of states by number of actions. The q-table determines the quality of each action in each state. To determine the best action to take in each state, the maximum argument of the 4 actions in that state is used to find the new action for the agent to take. The following formula is used to solve for the q-table which will be represented by Q:

$$\boxed{Q[s, A] = Q[s, A] + \alpha * (R + \gamma * max(Q[s', :]) - Q[s, A])} \quad (1)$$

The following table is a description explaining the variables of the formula.

Variable	Description
α	learning rate
γ	reward discount factor
s	old state
s'	new state
A	action
R	reward

In the code, the learning rate is denoted by lr, the reward discount factor is denoted by gamma, the action is denoted by action, the reward is denoted by reward, the new state is denoted by the last variable appended to the state history list, and the old state is the second to last variable appended to the state history list.

4 Approach

The mouse only knows 4 actions it can conduct: up, right, down, and left. The environment that will be discussed is the maze itself. There are 3 level difficulties, easy, medium, and hard. The easy maze difficulty is a 5x5 maze, the medium maze difficulty is a 10x10 maze, and the hard maze difficulty is a 20x20 maze. The mouse will

always start at the top left corner of the maze and the cheese will always be in the bottom right hand corner of the maze.

For debugging purposes, rather than use object oriented method of writing code and parsing through each function with inputs, a maze class was created with each variable attached to that variable. 16 functions were created and the maze class variable is the input to each of those functions. For example, if the move history wanted to be called, $maze.move\_history$ would be typed and would output a list with the number of moves it took to find the cheese over each game played. Once the mouse finds the minimum number of moves, the algorithm terminates because the mouse found the cheese in the least number of moves.

4.1 Architecture

The method of implementing this experiment will be from scratch. This is without the use of keras-rl, tensorforce, or any other RL toolboxes in python. Initially, the objective was to use the toolboxes, but along the way, many complications arose in attempt to use them. Such issues were that keras-rl toolbox, the input had to be in the form of gym-ai environment and rewards. Tensorforce had similar issues that the functions and code had to be formatted in a particular way which was not realized until after all the code was written; because the realization was made after most of the code was written, all the code would have had to have been rewritten. In terms of comparison with other built RL Maze experiments, none of the projects found were implemented from scratch making this experiment uniquely different from the rest. By implementing this from scratch, this allows for more research which allowed increased understanding of the concepts in q-learning.

The implementation is done with one core algorithm with multiple functions, for example: reward function, act function, show environment function, update state function, etc... All the variables are initialized in a class and called into the main algorithm.

4.2 Environments

The following figures 2, 3, & 4 are the 3 varying difficulty maze environments.

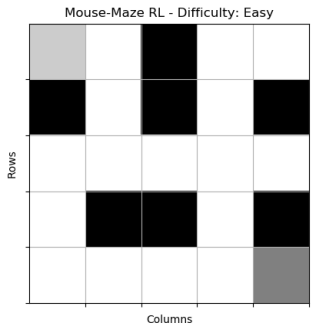

Figure 2: Easy 5x5 Maze Environment

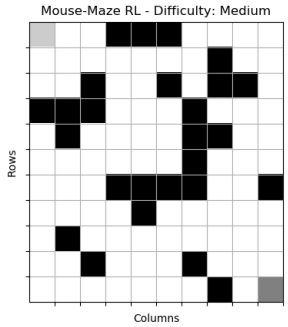

Figure 3: Medium 10x10 Maze Environment

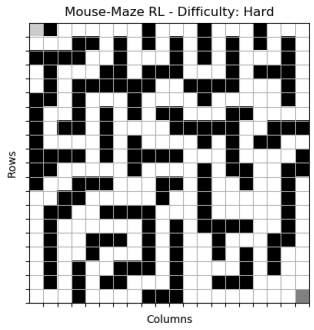

Figure 4: Hard 20x20 Maze Environment

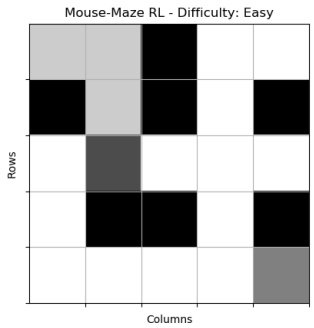

Figure 5: Example: Visualizing 3 Actions in Easy Environment

Figure 5 represents the easy maze having conducted 3 actions. In all the mazes, the black squares are invalid locations or walls, the border is also a wall. The white squares are free states the mouse can go into. The 3 light gray squares are the visited states the mouse was at. The 4^{th} square at the end of the light gray visited squares is a darker gray square which is the mouse. The cheese is the medium gray square at the bottom right corner of the maze.

4.3 State

A state is any position the mouse can move into. A wall or boundary is an invalid state. The number of states are calculated by taking the number of squares in the maze minus the number of invalid squares in the maze.

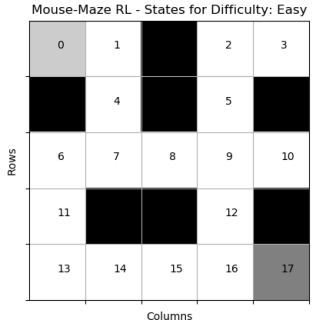

Figure 6: Easy Maze - State Matrix

Figure 6 is a representation of the states in the easy maze. The mouse starts in state 0 in the maze and can either go up, right, down, or left. The easy maze has 18 states for the mouse to explore or exploit. The medium maze has 84 free states and the hard

maze has 238 states for the mouse to explore or exploit.

Also, the state space does not limit the action space. The mouse does not have knowledge that there is a wall next to it, but rather learns based off the action it conducts in the state not to make that action because of the negative reward. In any state the mouse is in, the mouse can conduct any action even if the action is invalid. When the mouse is a state and conducts an invalid move the mouse remains in the same state it was just at not moving at all despite making an action.

4.4 Reward

The rewards are obtained after every action. The rewards are given on a numerical scale and are highlighted in the following table.

Description	Reward
Mouse finds the cheese	+1.00
Mouse moves into wall or boundary	-0.75
Mouse moves into free cell	-0.05
Mouse moves into a visited cell	-0.30
Mouse moves into state was just at	-2.00

4.5 Exploration vs. Exploitation

Both exploration and exploitation are a means for the agent to conduct actions. If the mouse is exploring, this means the mouse is conducting random actions. After conducting random actions, the q-table is constantly updated with rewards and states after each action. If the mouse is exploiting, this means the mouse is using the maximum argument from the q-table to determine the optimal action to take in each state.

To determine when the mouse should explore and when the mouse should exploit is based off a variable epsilon, ϵ. The ϵ is a dynamic decimal value between 0 and 1 and will change after each game. If ϵ is 20% or 0.20, this means the mouse will explore the maze 20% of the time and will exploit the maze 1-ϵ or 80% of the time.

5 Algorithm

5.1 Input

Maze environment

5.2 Q-Table Formula

$$Q[s, A] = Q[s, A] + \alpha * (R + \gamma * max(Q[s', :]) - Q[s, A]) \quad (2)$$

5.3 Main Loop

While optimal_path == False:

1. While winner == False:
   (a) Reset maze environment
   (b) if random.uniform(0,1) $< \epsilon$:
      i. Explore maze
         A. Conduct random action
         B. Obtain new reward and new state
         C. Calculate Q[s, A]
         D. Update maze environment
   (c) else:
      i. Exploit maze
         A. Calculate Q[s,A]
         B. A = max argument(Q[s, :])
         C. Conduct action
         D. Obtain new reward and new state
         E. Update maze environment
   (d) if game == 'Winner':
      i. End game because agent found the cheese
      ii. winner = True
2. if number moves $>$ move limit:
   (a) Increase ϵ & γ
3. else:
   (a) Decrease and normalize ϵ & γ
4. if number moves $<$ move stopping condition:
   (a) Agent found optimal path
   (b) optimal_path = True

6 Results

6.1 Easy Maze

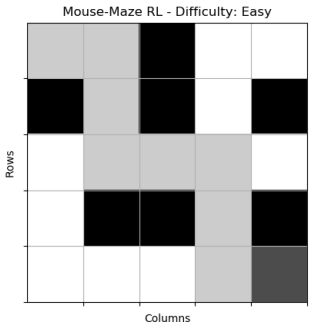

Figure 7: Easy Maze - Optimal Path with Minimum Number of Moves

Figure 7 represents the easy 5x5 maze environment. The agent took both the optimal path and the minimum number of moves in this maze. Minimum number of actions possible are 9 moves and the agent acted 9 times. To show a representation of the q-table for where the optimal results are drawn from are highlighted in the table below.

States	North	East	South	West	Optimal Action
0	-3.213	-2.469	-4.339	-4.409	East
1	-2.148	-1.969	-1.945	-2.000	South
2	-3.791	0.457	-0.342	-2.269	South
3	-1.257	-2.482	-1.257	-0.358	West
4	-2.000	-1.002	-0.357	-0.885	South
5	-1.519	-1.520	-0.253	-2.964	South
6	-3.573	-2.192	0.352	-0.977	East
7	-1.015	-0.992	-0.209	-0.711	East
8	-1.954	-0.645	-0.495	-0.994	East
9	-2.000	-0.987	-0.247	-2.267	South
10	-1.768	-0.698	0.328	-0.131	West
11	-0.514	-1.496	-3.399	-3.400	North
12	-1.000	-0.559	-2.013	-2.360	South
13	-2.179	-0.978	-0.524	-1.000	East
14	-0.377	-0.104	-0.667	-2.548	South
15	-2.482	-1.524	-0.359	-2.483	South
16	-1.805	1.000	-1.273	-0.4382	East
17	N/A	N/A	N/A	N/A	Cheese

This q-table comprises of the results of what the agent learned from the q-learning algorithm with the rewards and parameters set. Notice in the table, the optimal action

column is the maximum argument of the row for each state. The q-table is represented on the maze below via directional arrows to highlight the optimal action the mouse learned in each state.

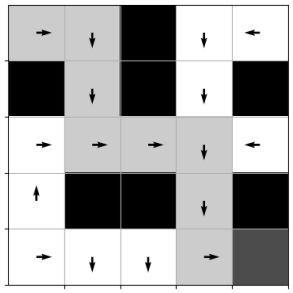

Figure 8: Easy Maze - Q-Table Optimal Action with Directional Arrows

Figure 8 represents the q-table's optimal actions to take in each state of the easy maze. This is the optimal action space the agent learned in each state.

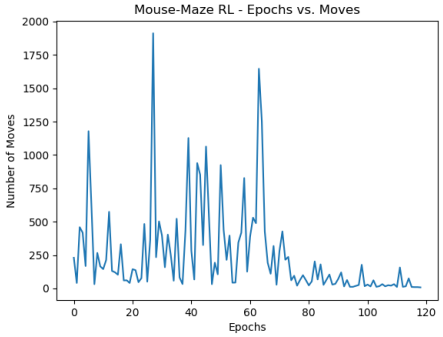

Figure 9: Easy Maze - Moves per Game Graph

Figure 9 represents the results of the easy maze tested. The y axis is the number of moves or actions and the x axis represents the number of epochs or games played. After 80 games, the mouse consistently performed in comparison to the first 80 games. After 120 games the mouse effectively found the optimal path in the least number of moves possible to find the cheese in 9 moves. Overall, the mouse effectively learned because the number of moves converged and consistently took the mouse few moves to find the cheese.

6.2 Medium Maze

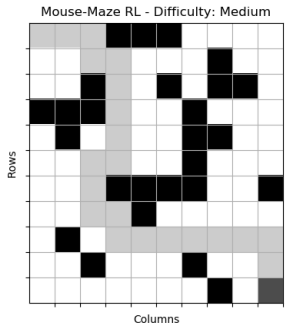

Figure 10: Medium Maze - Optimal Path with Minimum Number of Moves

Figure 10 represents the optimal path with the minimum number of moves for the medium maze difficulty. The mouse found the optimal path in 25 moves or actions which is the least possible number of actions.

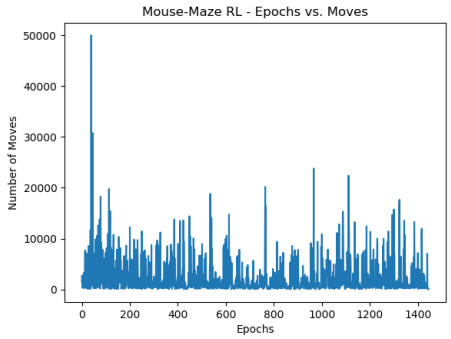

Figure 11: Medium Maze - Moves per Game Graph

Figure 11 highlights the number of games played and the number of moves it took for the mouse to find the cheese in each game. Initially, the mouse has a higher ϵ and therefore explores more in the beginning. The majority of the results per game indicate the mouse was not able to learn effectively and had under 2000 moves each game despite eventually reaching a minimum number of moves after 1400 games played.

6.3 Hard Maze

Some key factors to note about this maze again is the 20x20 dimension size and the 238 free states the mouse can move into. The minimum possible moves the mouse can accomplish in this maze to reach the cheese is 51 moves. The minimum number of moves the mouse took to optimize performance was consistently around 200 moves. Eventually, the mouse was able to get 142 moves which was the best for a long time. The hard maze was especially challenging to not only optimize, but for the mouse to learn because of the constraints that came with the harder maze. The main constraint was time, it took in comparison to the other mazes several hours to run the game once to find the best path. When trying to see if the maze would ever find the minimum number of moves of 51, a stopping condition was set and that if the mouse found the cheese in 51 moves the game would quit and the results would be shown. This test was conducted overnight and ran for 12 hours and after almost 30,000 games, the mouse never found the cheese in 51 moves. The least number of moves was 77 moves.

One of the factors that were decided to optimize this specific test was to set the move limit to a very small number. During these tests described above that sat overnight, the move limit was 250,000. The new move limit created to optimize performance is only 1000 moves. The results are as follows:

The mouse played 83,583 games with a move stopping condition of 1000 and produced figure 12. This is the result for the q-learning algorithm applied to the hard maze after the number of games played and a move stopping condition of 1000.

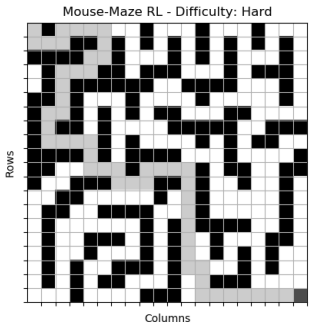

Figure 12: Hard Maze - Optimal Path with Minimum Number of Moves

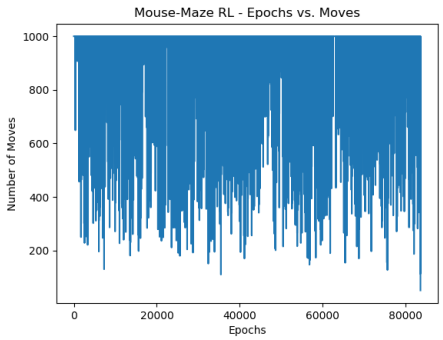

Figure 13: Hard Maze - Moves per Game Graph

Figure 13 represents the number of moves it took the mouse to find the cheese over the number of games the mouse was tested on. Before the stopping condition was added, the mouse for some instances of testing would take 1 million to 2 million to sometimes 10 million moves to find the cheese per game because the algorithm was not optimized. Figure 13 highlights the immense losses to the game with a move stopping condition which allowed for better optimization and increased performance of the mouse. The results from this figure are challenging to extract convergence, but indicate the mouse did successful learn the optimal path to take and the minimum move count to find the cheese at the same time. The significance of the mouse success is that the mouse did not take any unnecessary moves to find the cheese. The mouse had several different possible paths that it could have taken to find the cheese as well. Up to this experiment, the mouse constantly stuck to the top of the maze traveling across the right then going down which takes 53 moves to reach the cheese. Achieving the optimal path and finding the cheese is crucial in highlighting the impact of the q-learning algorithm.

6.4 Alternative Mazes

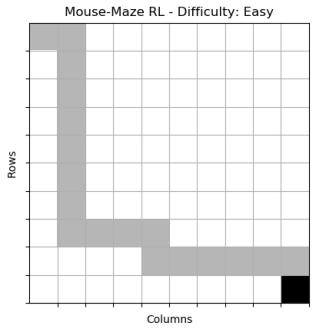

Figure 14: Alternative Easy Maze - Optimal Path with Minimum Number of Moves

This maze with 0 walls aside from boundaries took the agent 18 moves to find the cheese. There are multiple paths the mouse can take in this case, but it optimized performance and found the cheese.

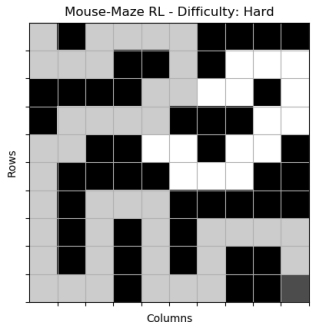

Figure 15: Alternative Hard Maze - Optimal Path with Minimum Number of Moves

Initially, this was the hard difficulty maze because of the complexity despite the smaller 10x10 size. The testing for this came before optimizing parameters and optimizing rewards for the mouse. When testing, the mouse was not extremely successful because it took the mouse 172 moves to find the cheese and did not efficiently learn. In comparison with figure 15 and 12, the larger 20x20 maze performed better after optimization than the smaller 10x10 maze.

7 Discussion

The mouse successfully found the optimal path in the least number of actions possible to find the cheese on all 3 maze difficulties. By utilizing rewards and varying parameters, the following sections will discuss the optimization process of the model.

7.1 How the Algorithm was Optimized

The algorithm was optimized by varying the following parameters:

- Rewards
- Gamma
- Learning rate
- Epsilon
- Move stopping condition

To begin, the setup process was very time consuming to create the environments and ensure the mouse was moving and acting throughout the maze. Once everything worked properly in that aspect as it did, the optimization process was allowed to begin. Optimization to begin was conducted on the medium difficulty maze as that was the first maze created.

7.1.1 Reward Optimization

The rewards system is the most crucial aspect of reinforcement learning and can cause the agent learn perfectly or learn nothing. To begin, there was only one negative reward which was for the mouse trying to move into walls or maze boundaries. The only positive rewards was the cheese. The mouse did not effectively learn to find the cheese but learned that running into walls is not the best idea. Next, the mouse travels throughout the entire maze and at some point while moving to find the cheese ends up in states it was already at. These are called visited positions which became a negative reward in the reward function. This incentives the mouse to not go into states it was already at so it makes more progress and covers newer routes it had not discovered previously. By adding the visited positions to negative rewards, the mouse performed much better but not to the desired success.

The third optimized reward added for the mouse was to give it a small negative reward after each action to incentive the mouse to find the cheese in the least number of moves possible. This enhanced the mouse's performance throughout experimentation. The final reward added was added based off the behavior of the mouse. The behavior that was observed was the mouse was getting stuck in between states and was learning to go back and forth between those states. The mouse would essentially get stuck moving between two states for thousands, if not millions, of moves because it learned that that was the best action to take in those two states. From this behavior,

a large negative reward was given to the mouse for this behavior. The results from this negative reward caused the behavior to be rarely observed following that. This final reward showed the most promise and increased the mouse's performance the tremendously in terms of time, games played, and move convergence.

7.1.2 Gamma Optimization

While gamma may not have as large an impact as the reward, it is still a multiplying factor in the q-learning formula used. This variable was fixed for the duration of the test and never varied once the program was tested. As gamma increased to above 1.2, the performance of the agent decreased. As gamma was varied underneath 0.5, the performance once again decreased. The best consistent performance for gamma was 0.9 and this was obtained through trial and error and recommendations from online resources [3].

7.1.3 Learning Rate Optimization

The learning rate was initially fixed and was tested with trial and error. The optimal learning rate obtained was at 1.2. A decision was made to make the learning rate change after each game based off the success of the agent. The move stopping condition was used to update the learning rate. If the agent went above 50,000 actions, then the learning would increase, otherwise it would decrease and normalize to 1.2. This worked effectively in the algorithm. The starting learning rate varied, but the best starting learning rate appeared to be 3 when testing the algorithm.

7.1.4 Epsilon Optimization

The epsilon variable to differentiate between exploration and exploitation was updated with the learning rate. Meaning as the agent approached the move stopping condition, it would mean to increase epsilon and explore more than exploit more. The epsilon value began at 90% and decreased to 20% over the success of the agent. These values to begin and end were optimal, other parameters to begin and start were tested with 70% and 10% or 80% and 15%, but it appeared the 90% begin and 20% stop worked effectively.

7.1.5 Move Stopping Condition Optimization

The difficulty of the maze plays a tremendous impact on the mouse to perform effectively. As difficulty increases, the complexity increases and therefore each maze needed a set optimal method of succeeding. This includes optimizing for the learning rate and epsilon, as well as the move limit. For the all 3 mazes, the optimal learning rate was 1.2. For all 3 mazes, the optimal starting epsilon was 0.9. For the easy maze, the stopping epsilon was 0.15. For the medium maze, the stopping epsilon was 0.20. For the hard maze, the stopping epsilon was 0.25. The easy maze stopping condition was 10,000 moves, the medium maze was 50,000 moves. Unlike the other 2 mazes,

the hard maze stopping condition was 1,000 moves. Varying stopping conditions were tested and experimented with, but these stopping conditions worked best.

7.2 Broader Impact

The broader impact on public safety, welfare, and the environment were not impacted by RL in game play applications. Ethics in RL with game play were impacted.

7.2.1 Ethical Issues

The following is an example of a scientific experiment. A real mouse is trying to find cheese somewhere in a maze, where the person conducting the experiment is testing the memory skills of the mouse. If the mouse moves into a position it was already in, the mouse gets a small electric shock. If the mouse runs into a wall, the mouse gets a larger electric shock. If the mouse goes into a position that it was just in and not making any progress, the mouse gets an immense electric shock nearly killing it. Finally, the mouse finds the cheese and is extremely happy. The mouse is in the maze, makes a wrong move, and goes into a dead end. The mouse learned it will have to go back to where it was previously meaning it will get shocked again. For this mouse, people sympathize as this an example of animal cruelty. But, the mouse is simply an agent in a python simulation in a simulated maze with simulated rewards and punishments [2].

This example and this experiment is used to exemplify the effect of the reward system. Deciding what rewards to give to the agent that is trying to learn is unethical. The rewards choice may immoral for the average user. What the user decides to teach the agent has potential to be unethical as well. In addition, the way of which the agent goes about learning may be a non traditional method and may be unethical too.

7.3 Engineering Standards

The main standard followed for this design project were IEEE standards. Other standards followed for the research was the NSPE code of ethics. In these cases, to avoid plagiarism and apply credit to where credit is due by referencing their work in the references. Next, to follow standards including PEP 8 standards. By following python standards, this will increase success in industry if code is to ever get published.

7.4 Design Constraints

- Time to train the agent took from seconds to minutes to hours.
- Code implemented from scratch without keras-rl, tensorforce, or any other RL toolboxes.
- Hyper parameter tuning
- Computer speed to run the model

8 Conclusion

RL is growing field and will play an integral in the technology of the future. To highlight a simple task of teaching a mouse to solve a maze indicates that understanding of the q-learning model. With this knowledge, allows one to apply techniques to intricate everyday problems and enhance the future. After conducting immense research, undergoing trial and error, and optimizing the solution, the simple mouse-maze experiment was successfully implemented. Despite the entire experiment conducted from scratch, the result allowed for a much deeper understanding of the raw concepts behind RL. Perhaps, if the experiment utilized keras-rl, tensorforce, or any other RL toolbox, there is possibility the mouse would have performed better, learned faster, and succeeded earlier on. The conducted mouse-maze experiment was successful on all three fronts: the easy 5x5 maze, the medium 10x10 maze, and the (very) hard 20x20 maze. Rewards play an immense impact in q-learning. After this experiment, conclusions hold the rewards as one of the most important factors in the entire algorithm. After modeling this experiment and varying several parameters, the algorithm was successfully optimized; the mouse found the cheese by taking both the optimal path and the least number of actions possible.

9 Python Code

9.1 Mouse Maze RL Algorithm Python Code

```
import numpy as np
import matplotlib.pyplot as plt
from Maze_Class import Maze_Class
import random
from prettytable import PrettyTable
import time
from scipy.optimize import curve_fit
plt.close('all')

# Assign maze to all variables in Maze Class
maze = Maze_Class()

# Visualization of the maze
def show(maze):
    plt.figure()
    plt.grid('on')
    ax = plt.gca()
    ax.set_xticks(np.arange(0.5, maze.n_rows, 1))
    ax.set_yticks(np.arange(0.5, maze.n_cols, 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
```

```
        plt.title('Mouse-Maze RL - Difficulty: ' + maze.difficulty)
        plt.xlabel('Columns')
        plt.ylabel('Rows')
        plt.imshow(maze.env, cmap='gray')

        plt.show()

    # Plot epochs vs. number of moves
    def plot_results(maze):
        fig = plt.figure()
        ax = fig.add_subplot(111)
        plt.plot(np.arange(0, maze.n_epochs), maze.move_history)
        plt.title('Mouse-Maze RL - Epochs vs. Moves')
        ax.set_xlabel('Epochs')
        ax.set_ylabel('Number of Moves')

        plt.show()

    # Show on path, the direction of what the agent learned from #q-table
    def arrow(maze):
        direction = []
        arrow_actions = {
                0: (0, 1),      # North
                1: (1, 0),      # East
                2: (0, -1),     # South
                3: (-1, 0)}     # Wests
        for idx in range(len(maze.q)):
            action = np.argmax(maze.q[idx, :])
            direction.append(arrow_actions[action])

        plt.figure()
        ax = plt.gca()
        ax.set_xticks(np.arange(0.5, maze.n_rows, 1))
        ax.set_yticks(np.arange(0.5, maze.n_cols, 1))
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        plt.imshow(maze.env, cmap='gray')
        state = 0
        for i in range(maze.n_rows):
            for j in range(maze.n_cols):
                if maze.state_tracker[j, i] != -1:
                    x_direct = direction[state][0]
                    y_direct = direction[state][1]
                    ax.quiver(i, j, x_direct, y_direct)
                    state += 1
        plt.show()
```

```
# Plot maze with state number over each state
def plot_state_matrix(maze):
    plt.figure()
    plt.grid('on')
    ax = plt.gca()
    ax.set_xticks(np.arange(0.5, maze.n_rows, 1))
    ax.set_yticks(np.arange(0.5, maze.n_cols, 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    plt.title('Mouse-Maze RL - States for Difficulty: ' + maze.difficulty)
    plt.xlabel('Columns')
    plt.ylabel('Rows')
    reset(maze)
    plt.imshow(maze.env, cmap='gray')

    state = 0
    for i in range(maze.n_rows):
        for j in range(maze.n_cols):
            if maze.state_tracker[i, j] != -1:
                ax.annotate(str(state), xy=(j, i),xytext=(j, i))
                state += 1
    plt.show()

# Plot on plot_results function the regression trend line of
#number of moves vs. number of games played
# 3rd degree function to model regression line, line of best
#fit will vary
def func(x, a, b, c, d):
    return a*(x**3) + b*(x**2) + c*x + d
def convergence(maze):
    # Games played list
    games_played = np.arange(0, maze.n_epochs)
    # Call results plot move history vs games played
    plot_results(maze)
    # Fit regression line
    popt, pcov = curve_fit(func, games_played, maze.move_history)
    # Plot regression line
    plt.plot(games_played, func(games_played, *popt), 'r-',
    label='fit: a=%5.3f, b=%5.3f, c=%5.3f, d=%5.3f' % tuple(popt))

    plt.show()


# Completes an action, checks action validity,
#and gets new reward and state
```

```
def act(maze, action):
    status_check = 'invalid'
    # Moving the mouse
    new_state = np.add(maze.mouse_pos, maze.action_dict[action])

    # Check if move is possible
    new_move = (new_state[0], new_state[1])
    # Call valid move function
    status_check = valid_move(maze, new_move)

    if status_check == 'valid':
        # Update visited position history
        maze.prev_pos = maze.mouse_pos
        # Updates visited and doesn't include if been there x many times
        if maze.prev_pos not in maze.visited_history:
            maze.visited_history.append(maze.prev_pos)

        # Assign new state to mouse position
        maze.mouse_pos = (new_move)

        # Update action history
        maze.action_history.append(action)

        # Update maze environment mark
        maze.env[maze.mouse_pos] = maze.mouse_mark
        maze.env[maze.prev_pos] = maze.visited_mark

        # Update maze current state
        get_state(maze)

    # Update number of moves
    maze.n_moves += 1

    # Obtain reward or penalty
    maze.reward = reward(maze, status_check)
    maze.reward_history.append(maze.reward)

# Rewards
def reward(maze, status_check):
    # Finding the cheese!
    if maze.mouse_pos == maze.cheese_pos:
        reward = 1
        maze.reward_tracker[maze.mouse_pos] =
        maze.reward_tracker[maze.mouse_pos] + reward
        return reward
```

```
        # If agent was just in that state, penalize for bad move,
        #learn to make different moves
        if len(maze.state_history) > 3:
            if maze.state_history[len(maze.state_history)-3] ==
            maze.state_history[-1]:
                reward = -2
                maze.reward_tracker[maze.mouse_pos] =
                maze.reward_tracker[maze.mouse_pos] + reward
                return reward

        # Visting a previous position
        if maze.mouse_pos in maze.visited_history:
            reward = -.3
            maze.reward_tracker[maze.mouse_pos] =
            maze.reward_tracker[maze.mouse_pos] + reward
            return reward

        # Invalid move penalty
        if status_check == 'invalid':
            reward = -1
        # Move into a free cell reward
        else:
            reward = -0.5

        maze.reward_tracker[maze.mouse_pos] =
        maze.reward_tracker[maze.mouse_pos] + reward

        return reward

    # Did the mouse win or lose
    def game_status(maze):
        if maze.mouse_pos == maze.cheese_pos:
            return 'winner'
        else:
            return 'loser'

    # Reset the environment to initial parameters
    def reset(maze):
        maze.action_history = []
        maze.visited_history = []
        maze.mouse_pos = (0, 0)
        maze.prev_pos = (0, 0)
        maze.state = 0
        maze.state_history = []
        maze.state_history.append(maze.state)
        maze.env = np.copy(maze.reset_env)
```

```
        maze.env[maze.mouse_pos] = maze.mouse_mark
        maze.env[maze.cheese_pos] = maze.cheese_mark
        maze.env[maze.prev_pos] = maze.visited_mark
        maze.n_moves = 0
        maze.n_invalid_moves = 0
        maze.reward_history = []
        maze.table = PrettyTable()
        maze.table.field_names = ['Moves', 'Action',
        'State', 'Reward']

    # Check if action is possible
    def valid_move(maze, new_move):
        if new_move == maze.cheese_pos:
            return 'valid'
        elif (new_move[1] == -1) | (new_move[0] == -1):
            maze.n_invalid_moves += 1
            return 'invalid'
        elif (new_move[0] == maze.n_rows) | (new_move[1] ==
        maze.n_cols):
            maze.n_invalid_moves += 1
            return 'invalid'
        elif maze.valid_location[new_move] == True:
            return 'valid'
        else:
            maze.n_invalid_moves += 1
            return 'invalid'

    # Retrieve state of the mouse
    def get_state(maze):
        maze.state = int(maze.state_tracker[maze.mouse_pos[0],
        maze.mouse_pos[1]])
        maze.state_history.append(int(maze.state))

    # Randomly roam around the maze
    def explore(maze):
        maze.action = random.randint(0, 3)

        act(maze, maze.action)
        if maze.n_moves > 2:
            maze.q[maze.state_history[len(maze.state_history)-2],
            maze.action] =
            maze.q[maze.state_history[len(maze.state_history)-2],
            maze.action] + maze.lr * (maze.reward + maze.gamma *
            np.max(maze.q[maze.state, :]) -
            maze.q[maze.state_history[len(maze.state_history)-2],
            maze.action])
```

```
        # Obtain important maze information
        maze_info(maze)

    # Utilize rewards to determine optimal next move
    def exploit(maze):
        #Q[state, action] = Q[state, action] +
        #                   lr * (reward + gamma * np.max(Q[new_state, :])
        #                  - Q[state, action])
        if maze.n_moves > 2:
            maze.q[maze.state_history[len(maze.state_history)-2],
            maze.action] =
                maze.q[maze.state_history[len(maze.state_history)-2],
                maze.action] + maze.lr *
                (maze.reward + maze.gamma *
                np.max(maze.q[maze.state, :])
                - maze.q[maze.state_history[len(maze.state_history)-2],
                maze.action])

        # Obtain optimal action
        maze.action = np.argmax(maze.q[maze.state, :])

        # Conduct the action
        act(maze, maze.action)

        # Update table
        maze_info(maze)

    # Print important information about the maze
    def maze_info(maze):
        maze.table.add_row([maze.n_moves, maze.action_term[maze.action],
                            maze.state, maze.reward])

    # Maze move stopping condition
    def game_difficulty_stopping_condition(maze):
        if maze.difficulty == 'Easy':
            if maze.n_moves <= 9:
                maze.move_stopping_condition = True
        if maze.difficulty == 'Medium':
            if maze.n_moves <= 25:
                maze.move_stopping_condition = True
        if maze.difficulty == 'Hard':
            if maze.n_moves <= 51:
                maze.move_stopping_condition = True

    # Increase epsilon and lr if surpassed move limits
    def game_difficulty_update_epsilon_lr(maze):
```

```
        # Easy difficulty
        if maze.difficulty == 'Easy':
            # Increase epsilon if exceeds certain number of moves
            if maze.n_moves == 1000:
                maze.epsilon += .025
                maze.lr += .075

            # Cutoff number of moves as a loss and continue next episode
            if maze.n_moves == 10000:
                maze.epsilon += .1
                maze.lr += .075
                maze.winner = True
                maze.n_losses += 1

        # Medium difficulty
        if maze.difficulty == 'Medium':
            # Increase epsilon if exceeds certain number of moves
            if maze.n_moves == 10000:
                maze.epsilon += .025
                maze.lr += .075

            # Cutoff number of moves as a loss and continue next episode
            if maze.n_moves == 50000:
                maze.epsilon += .1
                maze.lr += .075
                maze.winner = True
                maze.n_losses += 1

        # Hard difficulty
        if maze.difficulty == 'Hard':
           # Increase epsilon if exceeds certain number of moves
            #if maze.n_moves == 50000:
                #maze.epsilon += .025
                #maze.lr += .075

            # Cutoff number of moves as a loss and continue next episode
            if maze.n_moves == 1000:
                #maze.epsilon += .1
                #maze.lr += .075
                maze.winner = True
                maze.n_losses += 1

    # Normalize epsilon and learning rate
    def normalize_epsilon_lr(maze):
        if maze.difficulty == 'Easy':
            if maze.epsilon > .15:
```

```
            maze.epsilon -= .025
        elif maze.difficulty == 'Medium':
            if maze.epsilon > .20:
                maze.epsilon -= .025
        elif maze.difficulty == 'Hard':
            if maze.epsilon > .25:
                maze.epsilon -= .025
        # Normalize lr
        if maze.lr > 1.2:
            maze.lr -= .1

# Main algorithm that loops learning process and calls other functions
def run_game(maze):
    # Begin timer
    start_time = time.time()

    print('Starting up the game')

    # Stoping condiion checker
    maze.move_stopping_condition = False

    while maze.move_stopping_condition == False:
        # Reset condition for new game
        reset(maze)
        # Game winning condition
        maze.winner = False
        while maze.winner == False:
            # Randomly explores the maze
            if random.uniform(0, 1) < maze.epsilon:
                explore(maze)
                maze.explore_moves += 1
            # Utilize q-table to determine optimal move
            else:
                exploit(maze)
                maze.exploit_moves += 1

            # Check if game is a win, else continue
            if game_status(maze) == 'winner':
                maze.winner = True

            # Update epsilon and lr based off maze difficulty
            game_difficulty_update_epsilon_lr(maze)

        # Normalize epsilon and lr based off maze difficulty
        #after each epoch
        normalize_epsilon_lr(maze)
```

```
            # Store epsilon change over number of epochs
            maze.epsilon_history.append(maze.epsilon)
            # Store move history over number of epochs
            maze.move_history.append(maze.n_moves)

            # Checks stopping condition based off maze difficulty
            game_difficulty_stopping_condition(maze)

            # Print result every epoch (game)
            print('Epoch:', maze.n_epochs+1)
            print('Moves:', maze.n_moves)
            print('Epsilon {:0.4f}'.format(maze.epsilon))
            print('Learning Rate {:0.1f}'.format(maze.lr))
            print('------------------\n')
            maze.n_epochs += 1

        # Games complete, plot the results
        # Epochs (Games) vs. Moves
        plot_results(maze)

        # Time elapsed from initial run
        elapsed_time = time.time() - start_time

        # Print results after final optimal minimum moves achieved
        print('Elapsed time: {:.3f} seconds'.format(elapsed_time))
        print('Total epochs:', maze.n_epochs)
        print('Total wins:', maze.n_wins)
        print('Total losses:', maze.n_losses)
        print('Optimal move count:', maze.n_moves)


# Run the game
if __name__ == "__main__":
    run_game(maze)
    show(maze)
    arrow(maze)
    convergence(maze)
```

9.2 Maze Class Python Code

```
import numpy as np
from prettytable import PrettyTable

class Maze_Class():
```

```
    def __init__(self):

        # Select maze difficulty
        # Options: 'Easy', 'Medium', 'Hard'
        self.difficulty = 'Hard'

        # Maze environment setup
        #
        # Hard maze environment 20x20
        if self.difficulty == 'Hard':
            self.env = np.array([
                [ 1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  0.,
1.,  1.,  1.,  0.,  1.,  1.,  1.,  0.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  0.,  0.,  1.,  0.,  1.,  0.,
1.,  0.,  1.,  0.,  1.,  0.,  1.,  0.,  1.,  0.,  1.],
                [ 0.,  0.,  0.,  0.,  1.,  1.,  0.,  1.,  1.,
1.,  0.,  1.,  0.,  1.,  0.,  1.,  1.,  1.,  0.,  1.],
                [ 1.,  0.,  1.,  1.,  1.,  0.,  0.,  1.,  0.,
0.,  0.,  1.,  1.,  1.,  0.,  1.,  0.,  0.,  0.,  1.],
                [ 1.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,
1.,  1.,  0.,  0.,  0.,  0.,  1.,  1.,  1.,  0.,  1.],
                [ 0.,  0.,  1.,  0.,  1.,  1.,  1.,  0.,  1.,
1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  0.,  1.],
                [ 0.,  1.,  1.,  0.,  1.,  0.,  1.,  0.,  1.,
0.,  0.,  1.,  1.,  1.,  0.,  0.,  1.,  1.,  1.,  1.],
                [ 0.,  1.,  0.,  0.,  1.,  0.,  1.,  1.,  1.,
1.,  0.,  0.,  0.,  0.,  0.,  1.,  1.,  0.,  0.,  0.],
                [ 0.,  1.,  1.,  1.,  1.,  0.,  1.,  0.,  1.,
1.,  1.,  1.,  0.,  1.,  0.,  1.,  0.,  0.,  1.,  1.],
                [ 0.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,  0.,
0.,  0.,  1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  0.],
                [ 0.,  0.,  1.,  1.,  1.,  1.,  1.,  0.,  1.,
1.,  1.,  1.,  0.,  1.,  0.,  0.,  1.,  1.,  0.,  0.],
                [ 0.,  1.,  1.,  0.,  0.,  0.,  1.,  1.,  1.,
0.,  0.,  1.,  0.,  1.,  1.,  0.,  1.,  1.,  0.,  1.],
                [ 1.,  1.,  0.,  0.,  1.,  1.,  1.,  1.,  1.,
0.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  0.,  1.],
                [ 1.,  0.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,
1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  0.,  1.],
                [ 1.,  0.,  1.,  1.,  0.,  0.,  0.,  1.,  0.,
1.,  0.,  1.,  1.,  0.,  1.,  1.,  1.,  0.,  0.,  1.],
                [ 1.,  0.,  1.,  1.,  0.,  1.,  1.,  1.,  0.,
1.,  0.,  1.,  1.,  0.,  1.,  0.,  1.,  0.,  1.,  1.],
```

```
                [ 1.,  0.,  1.,  0.,  1.,  1.,  0.,  0.,  0.,
1.,  0.,  1.,  1.,  1.,  1.,  0.,  1.,  0.,  1.,  1.],
                [ 1.,  0.,  1.,  0.,  1.,  0.,  0.,  1.,  1.,
1.,  0.,  1.,  1.,  0.,  0.,  0.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  0.,
0.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]
            ])
        # Medium maze environment 10x10
        elif self.difficulty == 'Medium':
            self.env = np.array([
                [ 1.,  1.,  1.,  0.,  0.,  0.,  1.,  1.,  1.,
1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  0.,  1.,
1.],
                [ 1.,  1.,  0.,  1.,  1.,  0.,  1.,  0.,  0.,
1.],
                [ 0.,  0.,  0.,  1.,  1.,  1.,  0.,  1.,  1.,
1.],
                [ 1.,  0.,  1.,  1.,  1.,  1.,  0.,  0.,  1.,
1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  0.,  1.,  1.,
1.],
                [ 1.,  1.,  1.,  0.,  0.,  0.,  0.,  1.,  1.,
0.],
                [ 1.,  1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,
1.],
                [ 1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
1.],
                [ 1.,  1.,  0.,  1.,  1.,  1.,  0.,  1.,  1.,
1.],
                [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  0.,  1.,
1.]
            ])
        # Easy maze environment 5x5
        elif self.difficulty == 'Easy':
            self.env = np.array([
                [ 1.,  1.,  0.,  1.,  1.],
                [ 0.,  1.,  0.,  1.,  0.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  0.,  0.,  1.,  0.],
                [ 1.,  1.,  1.,  1.,  1.]
            ])

        # Number of rows an columns
        self.n_rows, self.n_cols = np.shape(self.env)
```

```
# Reset environment
self.reset_env = np.copy(self.env)

# Number of states
self.n_states = np.sum(self.env == 1)

# State tracking matrix for q-learning
self.state_tracker = np.copy(self.env)
self.state_counter = 0
for i in range(self.n_rows):
    for j in range(self.n_cols):
        # Valid state
        if self.state_tracker[i, j] == 1:
            self.state_tracker[i,j] = self.state_counter
            self.state_counter += 1
        else:
            # Hit a wall
            self.state_tracker[i, j] = -1

# Current state in state matrix and history
self.state = 0
self.state_history = []
# Begin at state 0
self.state_history.append(self.state)

# Number of free cells
self.free_cells = self.n_states

# Number of actions
self.n_actions = 4
# Actions: N, S, E, W
self.action_dict = {
    0: (-1, 0),     # North
    1: (0, 1),      # East
    2: (1, 0),      # South
    3: (0, -1)}     # West
# Action in words
self.action_term = {
    0: 'Up',
    1: 'Right',
    2: 'Down',
    3: 'Left'}

# Total elements
self.size = self.env.size
```

```
        # Visualize grey scale mark
        # 0 - black:invalid, 1 - white:valid
        self.visited_mark = 0.8
        self.mouse_mark = 0.3
        self.cheese_mark = 0.5

        # Position of cheese
        self.cheese_pos = (self.n_rows-1, self.n_cols-1)

        # Mouse position initialized at start
        self.mouse_pos = (0, 0)
        self.prev_pos  = (0, 0)

        # Visited history
        self.visited_history = []

        # Action integer from 0-3
        self.action = -1
        #Track action history for debug
        self.action_history = []

        # Initializing maze with above positions in environment
        self.env[self.mouse_pos] = self.mouse_mark
        self.env[self.cheese_pos] = self.cheese_mark
        self.env[self.prev_pos] = self.visited_mark

        # For resetting environment
        self.reset_env[self.mouse_pos] = self.mouse_mark
        self.reset_env[self.cheese_pos] = self.cheese_mark
        self.reset_env[self.prev_pos] = self.visited_mark

        # Checking validity for validity function
        self.valid_location = self.reset_env == 1
        # Because there are markers on those positions, manually change
        #validity
        self.valid_location[self.mouse_pos] = True
        self.valid_location[self.cheese_pos] = True

        # Number of invalid moves
        self.n_invalid_moves = 0

        # Rewards
        self.reward = 0
        # Track reward history for debugging
        self.reward_history = []
        self.reward_tracker = np.zeros((np.shape(self.env)))
```

```
        # Move counter
        self.n_moves = 0
        self.move_history = []

        # Track exploration vs. exploitation moves for debugging
        self.explore_moves = 0
        self.exploit_moves = 0

        # Learning rate
        self.lr = 3.00

        # Initialize q-matrix (q-quality)
        self.q = np.zeros((self.n_states, self.n_actions))

        # Discount balance in future and immediate rewards
        self.gamma = 0.90

        # Percent separation to explore or exploit
        self.epsilon = 0.85
        self.epsilon_history = []

        # Visualizing important data for debugging purposes
        self.table = PrettyTable()
        self.table.field_names = ['Moves', 'Action', 'State', 'Reward']

        # Move stopping condition
        self.move_stopping_condition = False

        # Game condition
        self.winner = False

        # Number of wins
        self.n_wins = 0
        self.n_losses = 0

        # Number of epochs
        self.n_epochs = 0
```

References

[1] A. Joy, “Pros and Cons of Reinforcement Learning,” Pythonista Planet, March 31 2019.

[2] E. Santana, “Deep Reinforcement Learning for Maze Solving,” np., nd.

[3] S. Gite, “Practical Reinforcement Learning – 02 Getting Started with Q-Learning,” Towards Data Science, April 4 2017.