A Report for Reinforcement Learning Coursework: Dynamic Maze Solving Project

Ziyi Guo, MSc Data Science, zg2u21@sonton.ac.uk

Department of Electronics and Computer Science, University of Southampton

Abstract

Reinforcement Learning has been developing rapidly in recent years, which provides new solutions in many fields based on AI techniques. Maze puzzle is a traditional problem that can be solved with RL algorithms by training the agent to travel in the maze wisely. In this paper, a new dynamic maze problem is proposed and the Deep Q-learning Network is explored to give possible solutions. The work can be found in *https://github.com/EzrealGUO/COMP6247-Dynamic-Maze-Solver.*

1 Introduction

As an advanced machine learning technology, reinforcement learning architecture is built based on the movement of a specifically defined ***agent***, which sets off ***actions*** through a series of ***states*** in the ***environment***, acquiring timely ***rewards*** (penalties) from the feedback system and finally reaching the final goal state. In the process of reaching the final state, the ***agent*** is probably failing again and again to accumulate experiences for the solution, but could reach the optimal destination at the end of its journey after abundant attempts.

In this project, the main objective is implementing a reinforcement learning based algorithm to solve a dynamic maze problem with a scale of square 199 (Fig 1) where a predefined ***agent*** needs to travel from the top left corner in the maze to the bottom right corner in an environment consisting of accessible and prohibited cells, namely paths and walls, and randomly generated fires in the accessible cells making the paths unavailable. With abundant efficient training, the ***agent*** should be able to avoid all the walls and fires in its journey and reach the destination located in (199,199) with a minimum traversal time.

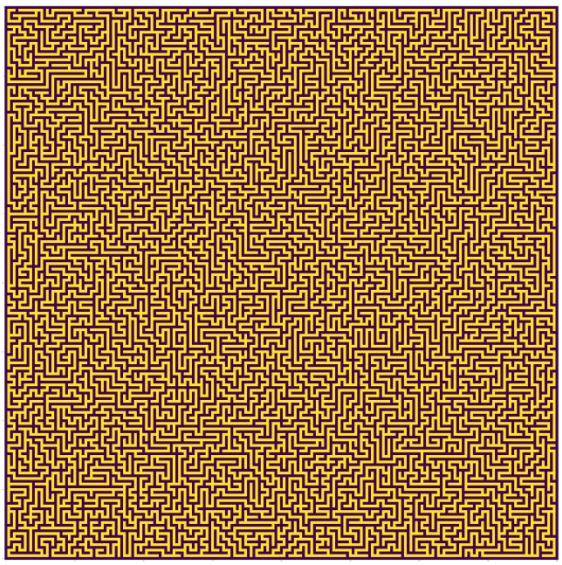

Fig 1. Target Maze in Depth Map

Basically, this paper promotes the maze-solver work based on the Q-learning theories combined with deep neural networks to train the ***agent***. In order to maximize the reproducibility and the interpretability of the proposed algorithms, the experiments are carried out in stages of pre-experiment (Chapter 2) and formal experiment (Chapter 3), following the process of proposing problem solution from simple to complex and undertaking training cost from low to high. According to the results of abundant experiments, the Deep Q-learning algorithm illustrated in this project is effective in solving the dynamic maze problem as well as related puzzles.

2 Related Work and Pre-experiment

In the pre-experiment, basic Deep Q-learning architecture is implemented based on ***Keras*** of the deep learning library ***Tensorflow*** in *Python*, tested on a static maze with no generated fires of small scale and evaluated based on the initial performances.

2.1 Experimental Environment Set-up

Above all, necessary experimental environment and basic theoretical architecture are built for the pre-experiment on the initial design of Deep Q-learning algorithm. The small-scale static maze of square 10 is generated as below.

```
maze = np.array([
    [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
    [ 0,  1,  1,  1,  1,  0,  0,  0,  1,  0],
    [ 0,  1,  0,  0,  1,  1,  1,  1,  1,  0],
    [ 0,  1,  0,  0,  0,  1,  0,  0,  1,  0],
    [ 0,  1,  0,  1,  0,  1,  0,  0,  1,  0],
    [ 0,  1,  0,  1,  0,  1,  0,  0,  0,  0],
    [ 0,  1,  1,  1,  0,  1,  1,  1,  1,  0],
    [ 0,  1,  0,  0,  0,  1,  0,  0,  0,  0],
    [ 0,  1,  0,  0,  0,  1,  1,  1,  1,  0],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
])
```

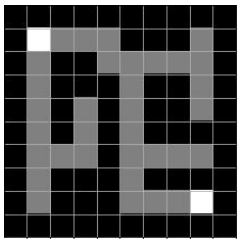

Fig 2. Initial Static Maze

From the observation of the initial static maze, it is indicated that the agent needs to travel from (1,1) to (8,8) without entering the locations of walls. This can be regarded as the elements of ***Markov Decision Process*** (***MDP***), where the environment consists of paths (white blocks) and walls (grey blocks) and the ***agent*** aims to reach the destination located in (8,8). The agent action is depend on the current state only.

In the proposed model, it is firstly necessary to define the ***action*** that the agent could take, and related program encodes the agent action of going *left*, *up*, *right* and *down* respectively as number arrays of [*0*,*1*,*2*,*3*], which actually indicates the *transformation* of ***state***. It is worth mentioning that when implementing the dynamic maze later, the action of *wait* is going to be encoded as well. Secondly, as the core design of the Q-learning algorithm, it is essential to judge the ***actions*** with the ***reward*** score, which quantifies the agent goal to acquire largest reward:

$$s' = T(s,a)$$
$$r = R(s,a)$$

Under the circumstance above, the ***reward*** is given as follows:

- Return ***+10*** for reaching the destination, in order to positively award the arrival greatly and only;
- Return ***-10*** for entering the regions of wall, in order to stop the illegal action with severe punishment;
- Return ***-0.01*** for *every single* action, in order to encourage the agent to seek the shortest routine;
- Return ***-0.1*** for revisiting a experienced block, in order to prevent the agent from struggling around.

```
def update_state(self, action):
    nrows, ncols = self.maze.shape
    nrow, ncol, nmode = self.state
    agent_row, agent_col, mode = self.state
    if self.maze[agent_row, agent_col] > 0:
        self.visited.add((agent_row, agent_col))
    valid_actions = self.valid_actions()
    if not valid_actions:
        nmode = 'blocked'
    elif action in valid_actions:
        nmode = 'valid'
        if action == LEFT:
            ncol -= 1
        elif action == UP:
            nrow -= 1
        elif action == RIGHT:
            ncol += 1
        elif action == DOWN:
            nrow += 1
    else:
        mode = 'invalid'
    self.state = (nrow, ncol, nmode)
```

```
def get_reward(self):
    agent_row, agent_col, mode = self.state
    nrows, ncols = self.maze.shape
    if agent_row == 8 and agent_col == 8:
        return 10.0
    if mode == 'blocked':
        return self.min_reward - 1
    if (agent_row, agent_col) in self.visited:
        return -0.1
    if mode == 'invalid':
        return -10
    if mode == 'valid':
        return -0.01
```

Based on the amount of ***reward*** given by every action, there should be a threshold to judge the reasonableness of the agent action. Similar to the misleading convergence of deep network training with an optimizer due to being stuck in a local minimum, the *convergence* of agent training in a far-away block from destination can be interpreted as being stuck in the wrong location. Numerically, a threshold here is set to make the judgement on whether the agent is on the right routine, which is defined as the maze size (*10*10*) multiplied with the value of

every single action reward (-0.01) and indicates the definite lose of way when receiving a even lower value of summing over all the rewards than it. Moreover, the actions that could hit the walls (including boundaries) are eliminated in the program while such actions would still be severely punished once taking place.

```
def valid_actions(self, cell=None):
    if cell is None:
        row, col, mode = self.state
    else:
        row, col = cell
    actions = [0, 1, 2, 3]
    nrows, ncols = self.maze.shape
    if row == 1:
        actions.remove(1)
    elif row>1 and self.maze[row-1,col] == 0:
        actions.remove(1)
    if row == nrows-2:
        actions.remove(3)
    elif row<nrows-2 and self.maze[row+1,col] == 0:
        actions.remove(3)
    if col == 1:
        actions.remove(0)
    elif col>1 and self.maze[row,col-1] == 0:
        actions.remove(0)
    if col == ncols-2:
        actions.remove(2)
    elif col<ncols-2 and self.maze[row,col+1] == 0:
        actions.remove(2)
    return actions
```

```
def reset(self, agent):
    self.agent = agent
    self.maze = np.copy(self.maze)
    nrows, ncols = self.maze.shape
    row, col = agent
    self.maze[row, col] = agent_mark
    self.state = (row, col, 'start')
    self.threshold = -0.01 * self.maze.shape[0]**2
    self.reward_sum = 0
    self.visited = set()

def game_status(self):
    if self.reward_sum < self.threshold:
        return 'lose'
    agent_row, agent_col, mode = self.state
    nrows, ncols = self.maze.shape
    if agent_row == 8 and agent_col == 8:
        return 'win'
    return 'not_over'
```

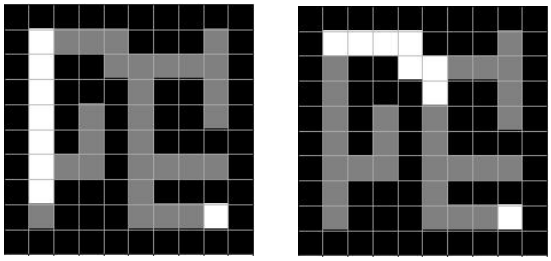

Fig 3. Test Samples of Action Implementation

According to the definitions and tests above, it is now able to build the training architecture simulating the travel with the Deep Q-learning Maze class, the starting point of the agent for training the a deep neural network to predict actions based on current state and the environment.

2.2 Experimental Training for Action Policy

Illustrating the previous coding for the basic set-ups on environment, agent action and reward, it is now aimed to develop the Q-learning strategy, namely ***policy π***, for the agent to automatically explore the maze and carry out reasonable actions facing every possible state after abundant attempts of travelling, which indicates the ***policy π*** is basically the ***action*** as a function of ***state***:

$$action = \pi(s)$$

Definitely to begin without any policy, totally random policy is applied in the training, which may cause quite many errors and travel failures, but the ***reward*** function will accumulate experience for the agent to get more advanced by numerically judging the current policy:

$$A = R(s_1, a_1) + R(s_2, a_2) + ... + R(s_n, a_n)$$

Once having the policy π, actions for travelling in the maze are output sequentially with the *state sequence*. The main point at present becomes the definition of the policy initialization. In this project based on Q-learning method, the *policy* ***π*** is illustrated by the Q-value function $Q(s,a)$ which simply describes the maximum reward with tuple sequence (s,a) mentioned above; also, the Bellman's Equation taking advantage of the recursive property of the function $Q(s,a)$ helps to make approximation as:

$$\pi(s) = \arg\max_{i=0,1,...,n-1} Q(s, a_i)$$

$$Q(s,a) = R(s,a) + \max_{i=0,1,...,n-1} Q(s', a_i), (s' = T(s,a))$$

With the prior definition above, a deep neural network is built in this project to take the surrounding information of current location as input and output the vector of Q-value predictions approximating $Q(s,a)$ value for all actions, which actually defines the policy corresponding to each action after abundant training. Normal training for deep neural network works with large datasets, which is impractical in this case to firstly generate a great number of state and Q-value information, while in stead this project uses the neural network to sample training data from

```
    def travel(model, DQNmaze, agent_start):
        DQNmaze.reset(agent_start)
        envstate = DQNmaze.observe()
        while True:
            prev_envstate = envstate
            q = model.predict(prev_envstate)
            action = np.argmax(q[0])
            envstate, reward, game_status = DQNmaze.act(action)
            if game_status == 'win':
                return True
            elif game_status == 'lose':
                return False

    def get_data(self, data_size=10):
        env_size = self.memory[0][0].shape[1]
        mem_size = len(self.memory)
        data_size = min(mem_size, data_size)
        inputs = np.zeros((data_size, env_size))
        targets = np.zeros((data_size, self.num_actions))
        for i, j in enumerate(np.random.choice(range(mem_size), data_size, replace=False)):
            envstate, action, reward, envstate_next, game_over = self.memory[j]
            inputs[i] = envstate
            targets[i] = self.predict(envstate)
            Q_sa = np.max(self.predict(envstate_next))
            if game_over:
                targets[i, action] = reward
            else:
                targets[i, action] = reward + self.gamma * Q_sa
        return inputs, targets
```

travel simulation with the objective function of the Bellman's Equation. After each Travel (Epoch), the network will output and store the environment, action and reward information as new training data. A great number of attempts are meaningful for the architecture and weights initialization of neural network weights to approximate optimum. Moreover, hyperparameters ε to promote stochastic actions separated from policy and γ to optimize Bellman's Equation is also required for greater exploration. previous experiences to acquire possibly better policy.

```
    def build_model(maze, lr=0.001):
        model = Sequential()
        model.add(Dense(maze.size, input_shape=(maze.size,)))
        model.add(PReLU())
        model.add(Dense(maze.size))
        model.add(PReLU())
        model.add(Dense(num_actions))
        model.compile(optimizer='adam', loss='mse')
        return model

    def qtrain(model, maze, **opt):
        epsilon = 0.1
        n_epoch = opt.get('n_epoch', 15000)
        limit = opt.get('limit', 1000)
        data_size = opt.get('data_size', 50)
        weights_file = opt.get('weights_file', "")
        name = opt.get('name', 'model')
        model.load_weights(weights_file)
        dqnmaze = DQNmaze(maze)
        experience = Experience(model, limit=limit)
        for epoch in range(n_epoch):
            loss = 0.0
            agent = （1,1）
            game_over = False
            envstate = dqnmaze.observe()
            n_episodes = 0
            episode = [prev_envstate, action, reward, envstate, game_over]
            experience.remember(episode)
            n_episodes += 1
            # Train neural network model
            inputs, targets = experience.get_data(data_size=data_size)
            h = model.fit(
                    inputs,
                    targets,
                    epochs=8,
                    batch_size=16,
                    verbose=0,
                )
            loss = model.evaluate(inputs, targets, verbose=0)
        # Save
        h5file = name + ".h5"
        json_file = name + ".json"
        model.save_weights(h5file, overwrite=True)
        with open(json_file, "w") as outfile:
            json.dump(model.to_json(), outfile)

        return 0
```

2.3 Pre-experiment Results

As a result, the training process converges in a place where the agent can 100% arrive at the destination always with the possibly greatest reward, indicating the success of training the available action policy for agent based on the neural network model and the weights, which takes an amount of 756 iterations and around 1443 minutes in all and an average of 116 seconds per epoch. In more detail, the algorithm takes 20 iterations to find a possible but maybe not best solution in a first time, 313 iterations to find the weights that could possibly lead to the solution at 50%, 754 iterations to find the weights that definitely can figure out the best solution, and the last two iterations to make the validation.

Based on the trained model, the agent is tested again to travel through the maze. It can be observed that the agent easily finds the best solution to travel from (1,1) to (8,8) with a minimum amount of actions of 14 and a maximum reward of 9.86, avoiding all the detours and dead ends on its way.

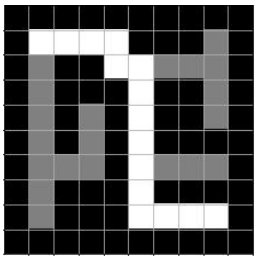

Fig 4. Visualized Routine of Agent Travel

It can be concluded from the pre-experiment that the Deep Q-learning Network is an effective architecture to handle maze solving problem. Despite of the unexpected high training cost observed of the neural network, it shows considerable accuracy of predicting the best action even with a baseline model as well as the potential capacity to deal with more complex problems due to the expandable neural network and provides reliable base for the following work.

3 Formal Experiments

Illustrating the previous work, formal experiments are carried out based on the expansion of Deep Q-learning architecture to handle the original maze problem. Great changes are made to the model as in the original problem, it is necessary to deal with dynamic input and output in both the static maze of large scale and the dynamic maze.

3.1 Static Maze Solver

In the pre-experiment, it has been shown that the Deep Q-learning Network works well with small static maze problem. Facing the huge maze with a size of square 201, however, the circumstance could become much more complex from a series of aspects:

- **The Dramatic Increase of Training Cost.** It is easy to see that with the increase of maze scale, finding only one possible solution can be already hard enough. Suppose that there is no barriers in the way, the agent will still have to take 396 (199*2) actions to move to the destination which has been 30 times of the previous solution while the actual situation is much more complex. Apparently, the training will be expensive.
- **The Change of Environment Definition and Input.** In the pre-experiment, the algorithm takes the whole maze as input and trains the agent to travel wisely. However, it is impractical for a huge maze due to not only the great cost to transform the whole maze into input vector but also the weak relationship between the specific actions and the entire maze (matrix sparsity). It is not allowed to do so in the pre-defined problem as well that the only information got should be the nearest eight locations around a current state. The input as well as the architecture has to be altered for the neural network.
- **The Obligation of Early Stop.** For the maze with large scale, the training is more likely to be stuck due to the overlong action sequence of solution that makes it hard to get positive feedback. The constant negative feedback can get the agent to be struggling around a dead end. A more strict extent of threshold setting is obliged under this circumstance.

According to the analysis above, great changes are made to the model architecture to make it fit the new circumstance. The details are as below.

Firstly, alter the reward function to fit the new maze. This is to ensure the punishment is received to optimize the agent action but the success of travel will give the agent positive feedback.

```
    # Update Reward based on Pre-defined Rules
    def get_reward(self):
        agent_row, agent_col, mode = self.state
        nrows, ncols = self.maze.shape
        if agent_row == 199 and agent_col == 199:
            return 50
        if mode == 'blocked':
            return self.min_reward - 1
        if (agent_row, agent_col) in self.visited:
            return -0.1
        if mode == 'invalid':
            return -50
        if mode == 'valid':
            return -0.01
```

Secondly, as the core difference of obtaining the surrounding information only, corresponding changes need to be applied in the environmental observation and action validation function in the model architecture, which ensures the realization of the predefined problem but the constant updating of information of the whole maze as well.

```
    # Record Reflections to Action
    def act(self, action):
        self.update_state(action)
        reward = self.get_reward()
        self.reward_sum += reward
        status = self.travel_status()
        envstate = self.observe()
        return envstate, reward, status
    # Input Environment Information as Vector
    def observe(self):
        canvas = self.draw_env()
        envstate = canvas.reshape((1, -1))
        return envstate
    # Environment Encodes
    def draw_env(self):
        row, col, valid = self.state
        canvas[row, col] = agent_mark
        m = self.state[0]
        n = self.state[1]
        environment = get_information(m,n)
        canvas = np.zeros((3,3))
        for i in range(3):
            for j in range(3):
                canvas[i][j] = environment[i][j][0]
                self.maze[row+i-1][col+j-1] = environment
                [i][j][0]
        return canvas
```

```
    # Validation for Actions:
    def valid_actions(self, cell=None):
        if cell is None:
            row, col, mode = self.state
        else:
            row, col = cell

        actions = [0, 1, 2, 3]

        nrows, ncols = self.maze.shape

        if row == 1:
            actions.remove(1)
        elif row>1 and self.maze[row-1,col] == 0:
            actions.remove(1)
        if row == nrows-2:
            actions.remove(3)
        elif row<nrows-2 and self.maze[row+1,col] == 0:
            actions.remove(3)
        if col == 1:
            actions.remove(0)
        elif col>1 and self.maze[row,col-1] == 0:
            actions.remove(0)
        if col == ncols-2:
            actions.remove(2)
        elif col<ncols-2 and self.maze[row,col+1] == 0:
            actions.remove(2)

        return actions
```

Thirdly, the change of the input actually gives the thoroughly new tasks to the model training, which is to learning a mapping from the current surrounding environment to the prediction of the following action. Despite of the little variation of the model architecture, it should be aware that the agent has different origin to obtain the policy.

```
    # Inputs are Outputs of the Previous Step
    def travel(model, DQNmaze, agent_start):
        DQNmaze.reset(agent_start)
        envstate = DQNmaze.observe()
        while True:
            prev_envstate = envstate
            # get next action
            q = model.predict(prev_envstate)
            action = np.argmax(q[0])
            #  action, get rewards and new state
            envstateupdate, reward = DQNmaze.act(action)
            if travel_status == 'win':
                return True
            elif travel_status == 'lose':
                return False
    # Deinfe Neural Network
    def build_model(lr=0.001):
        model = Sequential()
        model.add(Dense(9, input_shape=(9,)))
        model.add(PReLU())
        model.add(Dense(9))
        model.add(PReLU())
        model.add(Dense(num_actions))
        model.compile(optimizer='adam', loss='mse')
        return model
```

```
    # Build Training Architecture
    def qtrain(model, maze, **opt):
        for epoch in range(n_epoch):
            loss = 0.0
            envstate = dqnmaze.observe()
            while not travel_over:
                prev_envstate = envstate
                # Prediction of Next Action
                if np.random.rand() < epsilon:
                    action = random.choice(valid_actions)
                else:
                    action = np.argmax(experience.predict
                                       (prev_envstate))
                # Apply action, get reward and new state
                envstate, reward = dqnmaze.act(action)
                # Store Experience
                episode = [prev_envstate, action, reward
                           envstate, travel_over]
                experience.remember(episode)
                # Train Neural Network
                inputs, targets = experience.get_data()
                h = model.fit( inputs, targets, epochs=8
                               batch_size=16,verbose=0)
                loss = model.evaluate(inputs, targets)
```

It is worth mentioning that the training memory that defines the the upper bound amount of training data about experiences for each iteration is raised to 3000, which is aimed at ensuring the training data is able to cover all the possible scenarios happening in the journeys of the agents due to the complexity proportional to the maze scale.

3.2 Dynamic Maze Solver

To make an expansion on the Deep Q-learning model functioning, the architecture is explored further on the dynamic maze. In addition to the previous environment setting, a random generated fire is illustrated on the ways with no walls remaining for up to 2 time units and the change of state will possibly cause new fires or increase the remaining time of existing fire while the agent could choose to stay in the original position, which will change the environment as well. It is obvious that the circumstance become much more complex once again with new puzzles:

- **The Dramatic Increase of Training Cost (Again).** Despite the scale of maze is the same as before, finding only one possible solution can be even harder due to the randomly generated fire, which makes the action sequence much more longer to bypass or wait for the fire. Also, bypassing the fire could lead the agent to the wrong and easily stuck in dead ends. All these indicate tremendous training cost and iteration difficulty.
- **The Almost Non-existence of Convergence.** Similarly as training the agent to travel in the static maze, the algorithm aimed to compute the set of weights that fit the dynamic maze. However, because of the constant variation of the environment, it will be extremely hard for the algorithm to training a policy indicating the definite mapping function from state *s* and action *a* due to the limitation of model complexity. Also, it would be hard for the whole training process to define a ***maximum reward***, as the rewards of the optimal solution (minimum path) are still varying due to different actions in the uncertain environment, making the judgement on model convergence harder.

According to the analysis above, a number of changes are made to the model architecture.

Firstly, the new actions and the corresponding rewards is specially encoded and the ending threshold need to be more tolerate due to the much more actions in expectation. The reward definition is based on firstly the possibility of positive reward received for the detection of a good solution, secondly the more serious extent of punishment on *wait* action than *move* action but more minor than *revisit* action to encourage the agent to explore greater but avoiding dead ends with seemly no fire

```
    # Actions dictionary
    actions_list = {
        LEFT: 'left',
        UP: 'up',
        RIGHT: 'right',
        DOWN: 'down',
        WAIT: 'wait',
    }

    # Update State
    def update_state(self, action):
        nrows, ncols = self.maze.shape
        nrow, ncol, nmode = self.state
        agent_row, agent_col, mode = self.state
        if self.maze[agent_row, agent_col] > 0:
            self.visited.add((agent_row, agent_col))
        valid_actions = self.valid_actions()
        elif action in valid_actions:
            nmode = 'valid'
            if action == LEFT:
                ncol -= 1
            elif action == UP:
                nrow -= 1
            elif action == RIGHT:
                ncol += 1
            elif action == DOWN:
                nrow += 1
            elif action == WAIT:
                mode = 'stop'
                pass
        else:
            mode = 'invalid'
        self.state = (nrow, ncol, nmode)

    # Update Reward based on Pre-defined Rules
    def get_reward(self):
        agent_row, agent_col, mode = self.state
        nrows, ncols = self.maze.shape
        if agent_row == 199 and agent_col == 199:
            return 100
        if (agent_row, agent_col) in self.visited:
            return -1
        if mode == 'invalid':
            return -100
        if mode == 'valid':
            return -0.01
        if mode == 'stop':
            return -0.1
```

Secondly, the fire encoding need to be specifically considered as both environment information and training data. As environment information, it is hoped to be the same as the walls which is forbidden to going through; while as training data, it is possibly better to be differently encoded from the walls to be distinguishable for agents to decide their policy. Moreover, with the theoretically great difficulty of convergence, the exploration factor is raised to 0.2.

```
    def observe(self):                          # Input Environment Information as Vector
        canvas = self.draw_env()
        envstate = canvas.reshape((1, -1))
        return envstate
    def draw_env(self):                         # Environment Encodes
        row, col, valid = self.state
        environment = get_information(row, col)
        canvas = np.zeros((3,3))
        for i in range(3):
            for j in range(3):
                canvas[i][j] = environment[i][j][0]
                self.maze[row+i-1][col+j-1] = environment[i][j][0]
                if canvas[i][j] and environment [i][j][1]:
                    canvas[i][j] = 2                # Training Data as Input
                    self.maze[row+i-1][col+j-1] = 0 # Maze Information for Validation
        return canvas
```

3.3 Experiment Results and Analysis

Firstly, the training of agent in the static maze is proved to be hard as is analyzed, which takes 2853 iterations to find the first possible solution in a total training time of around 27 hours and 43 minutes and 4956 iterations to find the model weights with the winning rate of 29.90% in the amount time of 46 hours and 14 minutes. Due to the limitation of computing resource, the experiments are shut down at the point where the agent could act based on the model weights and find the optimal solution with a rate of 51.80%, with the observation that the winning rate is almost not increasing at all largely because of the limited hidden units of the neural network available for GPU to operate. It is worth mentioning that the above circumstance is the best training result among the abundant experiments, in which most agents are unable to find even a single solution and stuck in the dead ends after a great number of training, while a few number of agents could find a first solution and then the winning rate growing becomes a little more rapid for a while but just one agent could win in the travel with a rate beyond 30% and stops at about 50%.

Based on the trained model, the agent is programmed to set off again following the policy from the neural network. The test experiments are processed for 50 times, among which the agent failed to get to the destination for 9 times, travelled to the destination with a comparatively low reward for 27 times, and successfully travelled to the destination with the same maximum reward for 14 times. From the output file (Solution.txt) recording the travel experience with the maximum reward, it can be observed that the agent experiences 3584 states (including start and end) and takes 3583 actions in all, acquiring a total reward of 14.17. This can be proved to be the best routine that the model-based agent could take as a number of test experiments similarly indicates this best solution. Moreover, as the solution of a static maze, it can somehow be regarded as a unique solution particularly **for this model** because the best sequence of actions based on a invariable environment is unique. Even if there are multiple solutions of the routines, the best solution is still unique. However, the results can not be proved to be the best solution **for the maze** for the neural network has not been fully trained and converged to the ideal winning rate due to hardware factors and even if the network was fully trained, it can not be ensured that the agent would not make any mistake in its journey due to the environmental complexity and there is no another existed better solution as no evidence has shown the routine is unique. For more details, the the agent travel is visualized as below (Fig 5).

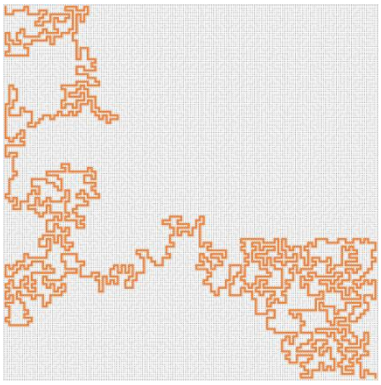

Fig 5. Visualization of Agent Travel

From the observation of the visualization, it can be inferred that the agent has not been stuck in any dead end or revisited any block in the previous routine. However, as is analyzed before, the solution can not be regarded as the best solution confidently due to the lack of training data and the result is not indicating there is no other better solutions of shorter routine. Generally however, the Deep Q-learning Network gives a possible solution for the agent to travel from (1,1) to (199,199) based on trained policy, which proves the architecture to be possibly not optimal and stable but successful on this static maze problem.

Secondly, the training of agent in the dynamic maze can be described as being almost meaningless, which takes 3189 iterations to find the first solution in a exaggeratedly long training time of 45 hours and 15 minutes and 4903 iterations to find the model with the winning rate of 10.00% in 63 hours in a best circumstance. In nearly all the experiments, the agent could not find the first solution even after significantly long time of training while the only one agent mentioned above travels further and is used to carry out the evaluation.

Based on the trained model, the agent is programmed to travel following the policy. With the prior knowledge that the model is going to perform poorly due to extremely difficult training, the main purpose of the test experiment is set to valid whether the model could output the solution, ignoring the algorithm efficiency and stability. This can be a trick but most researchers use the best results of their experiments as the model performance. As a result, in the hundreds of evaluation experiments, only 6 agents of all managed to arrive at the destination, which is much

lower in probability compared with the winning rate in model training, and the average amount of actions taken is around 35,500 while only the experiment with the maximum reward is recorded in the output file (Solution 2.txt). The output shows an intelligent agent coincidently travels from the starting point to the end experiencing 8731 states and taking 8732 actions in all, receiving a total reward of . This is apparently an efficient solution but can be proved to be weird based on the poorly trained model and interpreted as coincidence according to the following analysis:

- **The model is poorly trained and should not perform well.** It has been shown that the model takes significantly long time and difficulty to train and find a single solution. Also, the training is shut down at the point far away from convergence, indicating that the agent has not got clear strategy to travel efficiently in the dynamic maze. As a result, it is quite normal for the model-based agent to be stuck in the maze and with low probability to find a possible solution as is shown in the evaluation experiment, and is almost impossible to find the optimal solution with the series of most efficient actions. Potential probability of acquiring a good result may contribute to the uncertainty of model itself and the random exploration given by ε.
- **The model is theoretically never converging.** According to the introduction and analysis in the pre-experiment, it can be summarized that the aim of Deep Q-learning Network is to train a mapping from the state *s* and the environment to action *a* by optimizing value of Q-function. However, this is impractical in this circumstance as the mapping would be too complex for a neural network to represent as is shown in the figure below (Fig 6), in which the information and the correct action are really complex and misleading for agent to choose and learn due to the lack of knowledge about the whole maze layout, indicating there is no explicit mapping from the states to the actions. Moreover, as the core concept of Q-learning, it is basic to improve the Q-function value and thereby optimizing agent policy while the dynamic maze breaks this basic rule as the agent will largely receive varying reward even if travelling in the completely same routines due to the dynamic environment and the obligation of different times of waiting. The agent will be very much quite *'confused'* about the previous experiences on how to optimize Q-value on earth, especially in the long journey of constantly receiving negative rewards.

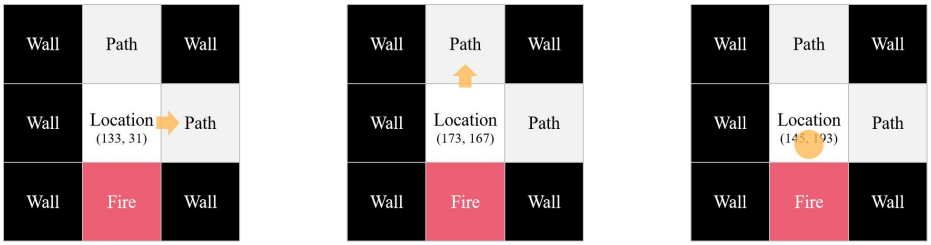

Fig 6. Uninterpretable Actions Observed in Similar Environments with Confusing Q-function Values

- **Visible difficulty is shown in model function expansion and reproduction.** Despite of the limitation of computing resource and the incompleted training of model, which is correlated to the poor performance, the Deep Q-learning Network might not be theoretically feasible to train the perfect agent for a dynamic maze puzzle according to the previous analysis and the model architecture is reproducible but the results could be quite different as is shown in the results of experiments that the lengths of action sequence in different tests have great variance. Therefore, the results may be irreproducible.

In all, the formal experiments implements the Deep Q-learning Network on the more challenging questions and acquired considerable results for further discussion. The applied algorithm is effective in training the agent to travel wisely in the static maze of large scale with abundant training and proper neural network architecture but is weak in training the agent to solve the dynamic maze problem with convincing stability.

4 Conclusion

In this paper, the Deep Q-learning Network is discussed and implemented to solve the traditional and new-style maze problems. The architecture implementation is all programmed in *Jupyter Notebook* with *Kernel* of *Python* 3, based on *Keras* in *Tensorflow* and the relevant environment needs to be configured to run the code, *eg*. loading *pip install tensorflow* and *pip install keras* in *Anaconda Prompt*. For further clarification and reproduction, the work in this paper can be found in *github* following *https://github.com/EzrealGUO/COMP6247-Dynamic-Maze-Solver*.

In conclusion, the Deep Q-learning Network shows great effectiveness in solving traditional static maze problems while the training processes with the objectives of small-scale and large-scale mazes all shows quite high training cost in time and computing resource. Despite of the different results concerning both accuracy and stability for the mazes in varying scales, the DQN model properties reflected and proved in the pre-experiment indicate that with

available computing resource and enough training, the model is able to converge to a optimal policy, based on which the agent can 100% travel in the maze following the best solution. Also, the change of input from the whole maze to the observation of the surrounding environment of the current state gives the more flexible model architecture and results in more detailed mapping from states to actions although the complexity of neural network need modification to fit the data.

However, the DQN performs poorly in the dynamic maze task not only because of the lack of complete agent training due to exaggeratedly high training cost, but conflicting with the dynamic environment on impractically optimizing the fluctuating Q-function value as well. As a result, the agent training is probably not converging even with enough computing resource while the output file indicates that the agent has somehow managed to optimize its policy and find a possible solution with uncountable attempts. It can be concluded that the DQN model is not an effective algorithm on the dynamic maze problem, which could to some limited extent optimize agent policy through the uncertain Q-value function, but is not likely to converge in a definite model and thus having low accuracy and stability of the general performance.

For further exploration on the dynamic maze problem, it is suggested that other advanced reinforcement learning based algorithms are worth implementing and testing on the very new puzzle according to the work in this paper. However, if attempts are still tried to be made on the DQN model to handle the dynamic maze puzzle, it is of interest to explore the CNN or RNN based model architecture to extract more meaningful information from the environment and the methodology of optimizing dynamic Q-function value, for example computing the estimation of the expectation mean and variance of all the possible maximum Q-function values. Anyway, there is still a long way in exploring the dynamic maze problem.

REFERENCE

[1] Clifton, Jesse, and Eric Laber. "Q-learning: theory and applications." Annual Review of Statistics and Its Application 7 (2020): 279-301.

[2] Osmanković, D., and Samim Konjicija. "Implementation of Q—Learning algorithm for solving maze problem." 2011 proceedings of the 34th international convention MIPRO. IEEE, 2011.

[3] Fan, Jianqing, et al. "A theoretical analysis of deep Q-learning." Learning for Dynamics and Control. PMLR, 2020.

[4] Sharma, Jivitesh, et al. "Deep q-learning with q-matrix transfer learning for novel fire evacuation environment." IEEE Transactions on Systems, Man, and Cybernetics: Systems 51.12 (2020): 7363-7381.