Assignment 4

Mahdi Tabatabaei 400101515
Github Repository

Neuroscience of Learning, Memory, and Cognition

Dr. Aghajan

January 29, 2025

Contents

Maze Generation

In this part, we use following code to generate the maze:

```python
import random
import matplotlib.pyplot as plt
import numpy as np

mx = 20; my = 20 # width and height of the maze

maze = [[0 for x in range(mx)] for y in range(my)]
dx = [0, 1, 0, -1]; dy = [-1, 0, 1, 0] # 4 directions to move in the maze
color = [(0, 0, 0), (255, 255, 255)] # RGB colors of the maze

# start the maze from a random cell
cx = random.randint(0, mx - 1)
cy = random.randint(0, my - 1)
maze[cy][cx] = 1
stack = [(cx, cy, 0)] # stack element: (x, y, direction)

while len(stack) > 0:
    (cx, cy, cd) = stack[-1]
    # to prevent zigzags:
    # if changed direction in the last move then cannot change again
    if len(stack) > 2:
        if cd != stack[-2][2]: dirRange = [cd]
        else: dirRange = range(4)
    else: dirRange = range(4)

    # find a new cell to add
    nlst = [] # list of available neighbors
    for i in dirRange:
        nx = cx + dx[i]
        ny = cy + dy[i]
        if nx >= 0 and nx < mx and ny >= 0 and ny < my:
            if maze[ny][nx] == 0:
                ctr = 0 # of occupied neighbors must be 1
                for j in range(4):
                    ex = nx + dx[j]; ey = ny + dy[j]
                    if ex >= 0 and ex < mx and ey >= 0 and ey < my:
                        if maze[ey][ex] == 1: ctr += 1
                if ctr == 1: nlst.append(i)

    # if 1 or more neighbors available then randomly select one and move
    if len(nlst) > 0:
        ir = nlst[random.randint(0, len(nlst) - 1)]
        cx += dx[ir]; cy += dy[ir]; maze[cy][cx] = 1
        stack.append((cx, cy, ir))
    else: stack.pop()

maze = np.array(maze)
maze -= 1
maze = abs(maze)

maze[0][0] = 0
maze[mx-1][my-1] = 0

np.save('maze', np.array(maze))
```

Our 20×20 generated maze is:

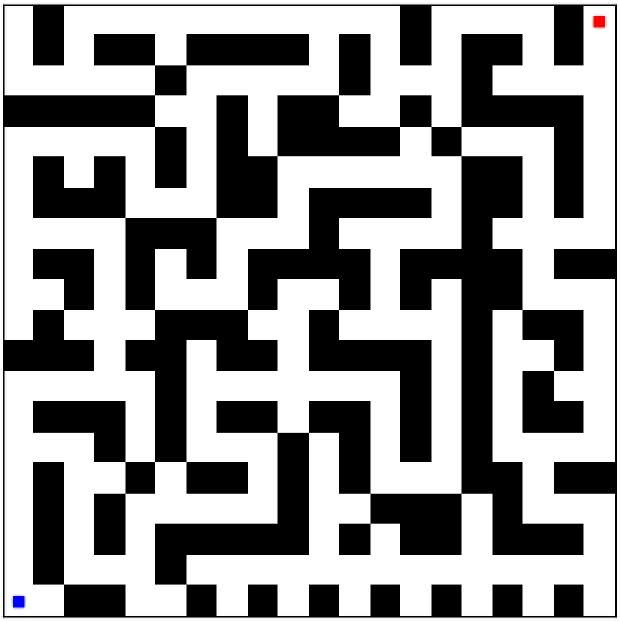

Figure 1: 20×20 Generated Maze

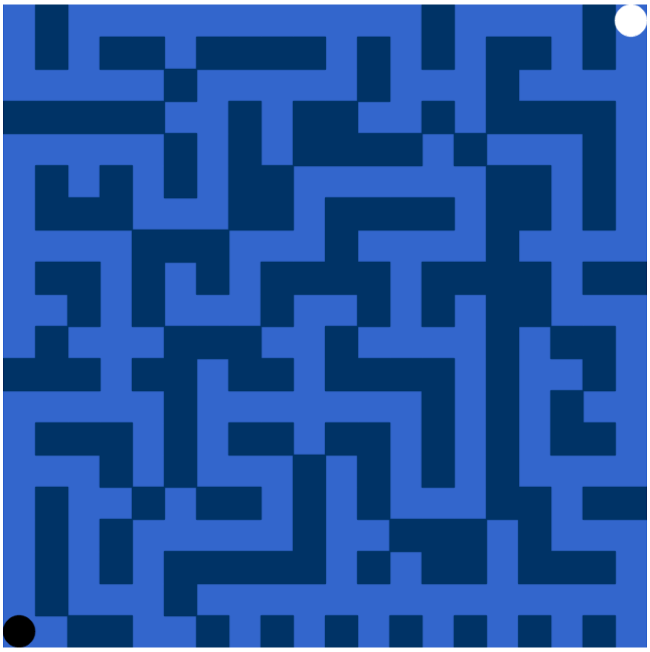

Figure 2: Other representation of 20×20 Generated Maze

Does the `Maze_Generator` guarantee that there is one path between the up-right and bottom-left cells?

The algorithm starts at a random cell and explores as far as possible along branches before backtracking. Each new cell is connected to the existing maze in a way that prevents loops and isolated sections. When adding a new cell, the code verifies that it has exactly one occupied neighbor (`ctr == 1`) and it prevents loops. We can also observe that the last 2 lines of the code guarantees that the mentioned 2 cells are white and by our explanations, we can discover that there is one path between these 2 cells.

Q-Learning

Q-learning is a popular model-free reinforcement learning algorithm used in machine learning and artificial intelligence applications. It falls under the category of temporal difference learning techniques, in which an agent picks up new information by observing results, interacting with the environment, and getting feedback in the form of rewards. Q-learning models engage in an iterative process where various components collaborate to train the model. This iterative procedure encompasses the agent exploring the environment and continuously updating the model based on this exploration. The key components of Q-learning include:

- Agents: Entities that operate within an environment, making decisions and taking actions.
- States: Variables that identify an agent's current position in the environment.
- Actions: Operations undertaken by the agent in specific states.
- Rewards: Positive or negative responses provided to the agent based on its actions.
- Episodes: Instances where an agent concludes its actions, marking the end of an episode.
- Q-values: Metrics used to evaluate actions at specific states.

In Q-learning I have implemented two classes: Environment and Agent.

Environment

The Environment class represents a maze environment where an agent navigates from a starting position to a goal position while avoiding walls. `is_valid_move` Ensures that a move is within maze boundaries and does not land on walls. `get_next_state` Computes the next state by adding the movement vector to the current position.

```python
class Environment:
    def __init__(self, maze_file):
        self.maze = np.load(maze_file)
        self.actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up
        self.start_state = (0, self.maze.shape[1] - 1)
        self.goal_state = (self.maze.shape[0] - 1, 0)

    def is_valid_move(self, x, y):
        return 0 <= x < self.shape[0] and 0 <= y < self.maze.shape[1] and self.maze[x, y] == 0

    def get_next_state(self, x, y, action):
        dx, dy = self.actions[action]
        next_x, next_y = x + dx, y + dy
        if self.is_valid_move(next_x, next_y):
            return next_x, next_y
        return x, y
```

Agent

The Agent class implements Q-learning. `alpha` is learning rate which shows how much the agent update q-values in each step. `gamma` is discount factor which shows how much future rewards matter compared to immediate rewards., and `epsilon` is exploration rate which shows the probability of taking a random action instead of the best-known action. `Q-table` stores the estimated rewards for each (state, action) pair with same shape as maze. `train` function runs Q-learning to update the Q-table based on rewards. By reaching the goal, reward is 1, by hitting the wall reward is -0.2, and by valid move reward is -0.01. We update the q-values by using the Bellman equation. By using `generate_solution_gif`, we create a gif of how our model is working. Besides, we us `show_policy` to observe the policy after running the algorithm. By using `_create_policy_gif`, we create the gif for policy evolution after each 100 episodes.

```python
class Agent:
    def __init__(self, environment, alpha=0.1, gamma=0.9, epsilon=0.1, num_episodes=5000):
        self.env = environment
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.num_episodes = num_episodes
        self.q_table = np.zeros((*self.env.maze.shape, len(self.env.actions)))

    def train(self, frame_interval=100, gif_name="policy_evolution.gif"):
        frame_filenames = []

        for episode in range(self.num_episodes):
            state = self.env.start_state
            episode_reward = 0
            while state != self.env.goal_state:
                x, y = state
                if random.uniform(0, 1) < self.epsilon:
                    action = random.randint(0, len(self.env.actions) - 1)  # Explore
                else:
                    action = np.argmax(self.q_table[x, y])  # Exploit
                next_x, next_y = self.env.get_next_state(x, y, action)

                if (next_x, next_y) == self.env.goal_state:
                    reward = 1
                elif (next_x, next_y) == (x, y):
                    reward = -0.2  # High penalty for hitting a wall
                else:
                    reward = -0.01  # Small penalty for valid move

                max_next_q = np.max(self.q_table[next_x, next_y])
                self.q_table[x, y, action] += self.alpha * (
                    reward + self.gamma * max_next_q - self.q_table[x, y, action]
                )

                state = (next_x, next_y)
                episode_reward += reward

            if episode % frame_interval == 0:
                filename = f"temp_policy_frame_{episode}.png"
                self._save_policy_frame(episode, filename)
                frame_filenames.append(filename)

            if episode % 1000 == 0:
```

```
                print(f"Episode {episode}, Epsilon: {self.epsilon:.3f}, Reward: {episode_reward}")

        self._create_policy_gif(frame_filenames, gif_name)

        # Cleanup temporary files
        for filename in frame_filenames:
            os.remove(filename)

    def generate_solution_gif(self, output_filename="solution_path.gif"):
        frames = []
        state = self.env.start_state
        solution_path = [state]
        while state != self.env.goal_state:
            x, y = state
            action = np.argmax(self.q_table[x, y])
            next_x, next_y = self.env.get_next_state(x, y, action)
            solution_path.append((next_x, next_y))
            state = (next_x, next_y)
            frame = np.copy(self.env.maze)
            for sx, sy in solution_path:
                frame[sx, sy] = 0.5  # Path is shown in gray
            frame[x, y] = 0.8  # Agent is shown in red
            fig, ax = plt.subplots(figsize=(5, 5))
            ax.imshow(frame, cmap="gray")
            ax.scatter(y, x, c="red", s=100)  # Highlight agent with a red dot
            ax.axis("off")
            plt.tight_layout()

            # Save frame to buffer
            buf = f"frame_{len(frames)}.png"
            plt.savefig(buf, dpi=100, bbox_inches='tight')
            frames.append(imageio.imread(buf))
            plt.close()
            os.remove(buf)  # Remove the intermediate PNG file

        # Ensure the last frame is included
        x, y = solution_path[-1]
        fig, ax = plt.subplots(figsize=(5, 5))
        ax.imshow(self.env.maze, cmap="gray")
        for sx, sy in solution_path:
            ax.scatter(sy, sx, c="gray", s=50)
        ax.scatter(y, x, c="red", s=100)  # Final position in red
        ax.axis("off")
        plt.tight_layout()
        buf = "final_frame.png"
        plt.savefig(buf, dpi=100, bbox_inches='tight')
        frames.append(imageio.imread(buf))
        plt.close()
        os.remove(buf)  # Remove the final PNG file

        # Save the GIF
        imageio.mimsave(output_filename, frames, fps=5)
        print(f"Q-learning completed. GIF saved as '{output_filename}'.")

    def show_policy(self):
        policy_arrows = {
            0: "\u2192",  # Right arrow
```

```
            1: "\u2193",  # Down arrow
            2: "\u2190",  # Left arrow
            3: "\u2191"   # Up arrow
        }

        maze_with_policy = np.copy(self.env.maze)
        fig, ax = plt.subplots(figsize=(5, 5))

        for i in range(maze_with_policy.shape[0]):
            for j in range(maze_with_policy.shape[1]):
                if maze_with_policy[i, j] == 0:  # Open cell
                    if (i, j) == (maze_with_policy.shape[0] - 1, 0):  # Bottom-left corner
                        ax.plot(j, i,'bs', markersize = 4)
                    else:
                        best_action = np.argmax(self.q_table[i, j])
                        arrow = policy_arrows[best_action]
                        ax.text(j, i, arrow, ha='center', va='center', color='black', fontsize=12)
        ax = plt.gca()
        ax.imshow(maze_with_policy, "Greys")
        plt.xticks([], [])
        plt.yticks([], [])

        plt.show()

    def _save_policy_frame(self, episode, filename):
        # Reuse the visualization logic from show_policy() but save to file
        policy_arrows = {
            0: "\u2192", 1: "\u2193", 2: "\u2190", 3: "\u2191"
        }
        maze = np.copy(self.env.maze)
        fig, ax = plt.subplots(figsize=(5, 5))

        for i in range(maze.shape[0]):
            for j in range(maze.shape[1]):
                if maze[i, j] == 0:  # Open cell
                    best_action = np.argmax(self.q_table[i, j])
                    arrow = policy_arrows[best_action]
                    ax.text(j, i, arrow, ha='center', va='center', color='black', fontsize=12)

        ax.imshow(maze, "Greys")
        plt.xticks([], [])
        plt.yticks([], [])
        plt.title(f"Episode: {episode}")
        plt.savefig(filename, bbox_inches='tight', dpi=100)
        plt.close()

    def _create_policy_gif(self, frame_files, output_filename, fps=2):
        # Compile frames into a GIF
        frames = [imageio.imread(filename) for filename in frame_files]
        imageio.mimsave(output_filename, frames, fps=fps)
        print(f"Policy evolution GIF saved as '{output_filename}'.")
```

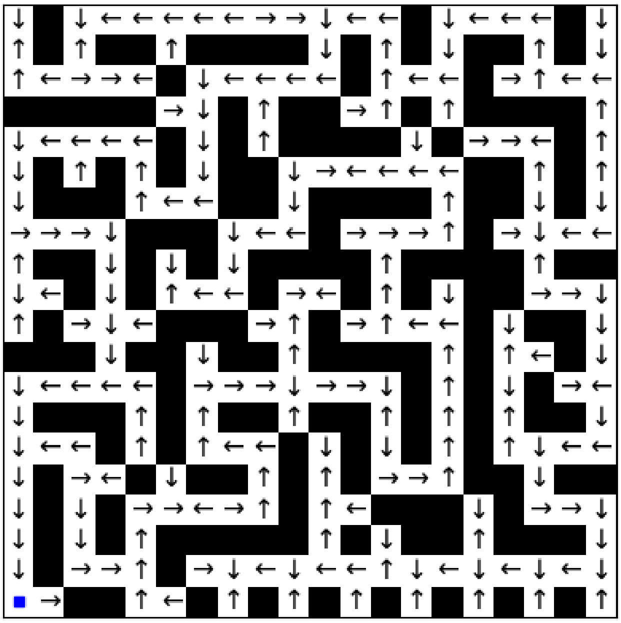

Figure 3: Policy Map Plot After Q-Learning Algorithm

Deep Q-Learning

Source of this explanations is this website. The Deep Q-Network (DQN) algorithm enhances the traditional Q-learning method by incorporating neural networks to approximate Q-values.

1. Initialize your Main and Target neural networks.
2. Choose an action using the Epsilon-Greedy Exploration Strategy.
3. Update your network weights using the Bellman Equation.

Initialize Your Target and Main Neural Networks
A core difference between Deep Q-Learning and Vanilla Q-Learning is the implementation of the Q-table. Instead of a Q-table, Deep Q-Learning uses a neural network to map input states to *(action, Q-value)* pairs. One important feature of Deep Q-Learning is the use of two neural networks: the **Main** and **Target** networks. These networks share the same architecture but have different weights. Every N steps, the weights of the Main network are copied to the Target network, stabilizing the learning process. In our implementation, we update the Target network every 100 steps.

Mapping States to (Action, Q-value) Pairs
The Main and Target neural networks map input states to *(action, Q-value)* pairs. Each output node represents an action and holds its respective Q-value as a floating-point number. Unlike probabilities, these values do not sum to 1.

Choosing an Action Using Epsilon-Greedy Strategy
In the Epsilon-Greedy Exploration strategy, the agent chooses a random action with probability ϵ and exploits the best-known action with probability $1-\epsilon$. The best-known action corresponds to the action with the highest predicted Q-value from the Main network.

Updating Network Weights Using the Bellman Equation

Once an action is selected, the agent performs it and updates the Main and Target networks according to the Bellman Equation. Deep Q-Learning employs Experience Replay to improve the learning process.

Experience Replay

Experience Replay is a mechanism that stores past experiences as tuples *(state, action, reward, next_state)*. This technique allows the agent to learn from past experiences in an offline manner, reducing correlations between consecutive experiences and stabilizing training.
Instead of updating the network after each step, Experience Replay trains the network using small batches every 4 steps. This method improves efficiency and accelerates Deep Q-Learning.

The Bellman Equation

Similar to Vanilla Q-Learning, Deep Q-Learning updates model weights using the Bellman Equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

The Temporal Difference (TD) target is computed using the Target network rather than the Main network. The Main network is then updated to fit these target Q-values, ensuring stable learning.

Here, we have `environment` and `agent` classes too.

Environment

```python
import numpy as np
import scipy.special as sp
import matplotlib.pyplot as plt
import copy

class MazeEnvironment:
    def __init__(self, maze, init_position, goal):
        x = len(maze)
        y = len(maze)

        self.boundary = np.asarray([x, y])
        self.init_position = init_position
        self.current_position = np.asarray(init_position)
        self.goal = goal
        self.maze = maze

        self.visited = set()
        self.visited.add(tuple(self.current_position))

        # initialize the empty cells and the euclidean distance from
        # the goal (removing the goal cell itself)
        self.allowed_states = np.asarray(np.where(self.maze == 0)).T.tolist()
        self.distances = np.sqrt(np.sum((np.array(self.allowed_states) -
                                        np.asarray(self.goal))**2,
                                        axis = 1))

        del(self.allowed_states[np.where(self.distances == 0)[0][0]])
        self.distances = np.delete(self.distances, np.where(self.distances == 0)[0][0])

```

```
        self.action_map = {0: [0, 1],
                            1: [0, -1],
                            2: [1, 0],
                            3: [-1, 0]}

        self.directions = {0: '→',
                            1: '←',
                            2: '↓ ',
                            3: '↑'}

        # the agent makes an action from the following:
        # 1 -> right, 2 -> left
        # 3 -> down, 4 -> up

    # introduce a reset policy, so that for high epsilon the initial
    # position is nearer to the goal (useful for large mazes)
    def reset_policy(self, eps, reg = 7):
        return sp.softmax(-self.distances/(reg*(1-eps**(2/reg)))**(reg/2)).squeeze()

    # reset the environment when the game is completed
    # with probability prand the reset is random, otherwise
    # the reset policy at the given epsilon is used
    def reset(self, epsilon, prand = 0):
        if np.random.rand() < prand:
            idx = np.random.choice(len(self.allowed_states))
        else:
            p = self.reset_policy(epsilon)
            idx = np.random.choice(len(self.allowed_states), p = p)

        self.current_position = np.asarray(self.allowed_states[idx])

        self.visited = set()
        self.visited.add(tuple(self.current_position))

        return self.state()


    def state_update(self, action):
        isgameon = True

        # each move costs -0.05
        reward = -0.05

        move = self.action_map[action]
        next_position = self.current_position + np.asarray(move)

        # if the goals has been reached, the reward is 1
        if (self.current_position == self.goal).all():
                reward = 1
                isgameon = False
                return [self.state(), reward, isgameon]

        # if the cell has been visited before, the reward is -0.2
        else:
            if tuple(self.current_position) in self.visited:
                reward = -0.2

```

```python
        # if the moves goes out of the maze or to a wall, the
        # reward is -1
        if self.is_state_valid(next_position):
            self.current_position = next_position
        else:
            reward = -1

        self.visited.add(tuple(self.current_position))
        return [self.state(), reward, isgameon]

    # return the state to be feeded to the network
    def state(self):
        state = copy.deepcopy(self.maze)
        state[tuple(self.current_position)] = 2
        return state


    def check_boundaries(self, position):
        out = len([num for num in position if num < 0])
        out += len([num for num in (self.boundary - np.asarray(position)) if num <= 0])
        return out > 0


    def check_walls(self, position):
        return self.maze[tuple(position)] == 1


    def is_state_valid(self, next_position):
        if self.check_boundaries(next_position):
            return False
        elif self.check_walls(next_position):
            return False
        return True


    def draw(self, filename):
        plt.figure()
        im = plt.imshow(self.maze, interpolation='none', aspect='equal', cmap='Greys');
        ax = plt.gca();

        plt.xticks([], [])
        plt.yticks([], [])

        ax.plot(self.goal[1], self.goal[0],
                'bs', markersize = 4)
        ax.plot(self.current_position[1], self.current_position[0],
                'rs', markersize = 4)
        plt.savefig(filename, dpi = 300, bbox_inches = 'tight')
        plt.show()
```

Agent

```python
import numpy as np
import scipy.special as sp
import matplotlib.pyplot as plt
import copy
```

```python
import torch
import torch.nn as nn
import torch.optim as optim
import collections

Transition = collections.namedtuple('Experience',
                                    field_names=['state', 'action',
                                                 'next_state', 'reward',
                                                 'is_game_on'])


class Agent:
    def __init__(self, maze, memory_buffer, use_softmax = True):
        self.env = maze
        self.buffer = memory_buffer # this is actually a reference
        self.num_act = 4
        self.use_softmax = use_softmax
        self.total_reward = 0
        self.min_reward = -self.env.maze.size
        self.isgameon = True


    def make_a_move(self, net, epsilon, device = 'cuda'):
        action = self.select_action(net, epsilon, device)
        current_state = self.env.state()
        next_state, reward, self.isgameon = self.env.state_update(action)
        self.total_reward += reward

        if self.total_reward < self.min_reward:
            self.isgameon = False
        if not self.isgameon:
            self.total_reward = 0

        transition = Transition(current_state, action,
                                next_state, reward,
                                self.isgameon)

        self.buffer.push(transition)


    def select_action(self, net, epsilon, device = 'cuda'):
        state = torch.Tensor(self.env.state()).to(device).view(1,-1)
        qvalues = net(state).cpu().detach().numpy().squeeze()

        # softmax sampling of the qvalues
        if self.use_softmax:
            p = sp.softmax(qvalues/epsilon).squeeze()
            p /= np.sum(p)
            action = np.random.choice(self.num_act, p = p)

        # else choose the best action with probability 1-epsilon
        # and with probability epsilon choose at random
        else:
            if np.random.random() < epsilon:
                action = np.random.randint(self.num_act, size=1)[0]
            else:
                action = np.argmax(qvalues, axis=0)
                action = int(action)
```

```python

        return action


    def plot_policy_map(self, net, filename, offset):
        net.eval()
        with torch.no_grad():
            fig, ax = plt.subplots()
            ax.imshow(self.env.maze, 'Greys')

            for free_cell in self.env.allowed_states:
                self.env.current_position = np.asarray(free_cell)
                qvalues = net(torch.Tensor(self.env.state()).view(1,-1).to('cuda'))
                action = int(torch.argmax(qvalues).detach().cpu().numpy())
                policy = self.env.directions[action]

                ax.text(free_cell[1]-offset[0], free_cell[0]-offset[1], policy)
            ax = plt.gca();

            plt.xticks([], [])
            plt.yticks([], [])

            ax.plot(self.env.goal[1], self.env.goal[0],
                    'bs', markersize = 4)
            plt.savefig(filename, dpi = 300, bbox_inches = 'tight')
            plt.show()
```

Implementation

```python
class ExperienceReplay:
    def __init__(self, capacity):
    self.capacity = capacity
    self.memory = collections.deque(maxlen=capacity)

    def __len__(self):
    return len(self.memory)

    def push(self, transition):
    self.memory.append(transition)

    def sample(self, batch_size, device = 'cuda'):
    indices = np.random.choice(len(self.memory), batch_size, replace = False)

    states, actions, next_states, rewards, isgameon = zip(*[self.memory[idx]
                                    for idx in indices])

    return torch.Tensor(states).type(torch.float).to(device), \
    torch.Tensor(actions).type(torch.long).to(device), \
    torch.Tensor(next_states).to(device), \
    torch.Tensor(rewards).to(device), torch.tensor(isgameon).to(device)

class fc_nn(nn.Module):
    def __init__(self, Ni, Nh1, Nh2, No = 4):
        super().__init__()

        self.fc1 = nn.Linear(Ni, Nh1)
```

```python
        self.fc2 = nn.Linear(Nh1, Nh2)
        self.fc3 = nn.Linear(Nh2, No)

        self.act = nn.ReLU()

    def forward(self, x, classification = False, additional_out=False):
        x = self.act(self.fc1(x))
        x = self.act(self.fc2(x))
        out = self.fc3(x)

        return out

class conv_nn(nn.Module):
    channels = [16, 32, 64]
    kernels = [3, 3, 3]
    strides = [1, 1, 1]
    in_channels = 1

    def __init__(self, rows, cols, n_act):
        super().__init__()
        self.rows = rows
        self.cols = cols

        self.conv = nn.Sequential(nn.Conv2d(in_channels = self.in_channels,
                                            out_channels = self.channels[0],
                                            kernel_size = self.kernels[0],
                                            stride = self.strides[0]),
                                  nn.ReLU(),
                                  nn.Conv2d(in_channels = self.channels[0],
                                            out_channels = self.channels[1],
                                            kernel_size = self.kernels[1],
                                            stride = self.strides[1]),
                                  nn.ReLU()
                                  )

        size_out_conv = self.get_conv_size(rows, cols)

        self.linear = nn.Sequential(nn.Linear(size_out_conv, rows*cols*2),
                                  nn.ReLU(),
                                  nn.Linear(rows*cols*2, int(rows*cols/2)),
                                  nn.ReLU(),
                                  nn.Linear(int(rows*cols/2), n_act),
                                  )

    def forward(self, x):
        x = x.view(len(x), self.in_channels, self.rows, self.cols)
        out_conv = self.conv(x).view(len(x),-1)
        out_lin = self.linear(out_conv)
        return out_lin

    def get_conv_size(self, x, y):
        out_conv = self.conv(torch.zeros(1,self.in_channels, x, y))
        return int(np.prod(out_conv.size()))

def Qloss(batch, net, gamma=0.99, device="cuda"):
    states, actions, next_states, rewards, _ = batch
    lbatch = len(states)
    state_action_values = net(states.view(lbatch,-1))
```

```
    state_action_values = state_action_values.gather(1, actions.unsqueeze(-1))
    state_action_values = state_action_values.squeeze(-1)

    next_state_values = net(next_states.view(lbatch, -1))
    next_state_values = next_state_values.max(1)[0]

    next_state_values = next_state_values.detach()
    expected_state_action_values = next_state_values * gamma + rewards

    return nn.MSELoss()(state_action_values, expected_state_action_values)
```

```
from environment import MazeEnvironment
from agent import Agent

maze = np.load('maze.npy')
initial_position = [0,len(maze)-1]
goal = [len(maze)-1, 0]
maze_env = MazeEnvironment(maze, initial_position, goal)

buffer_capacity = 10000
buffer_start_size = 1000
memory_buffer = ExperienceReplay(buffer_capacity)

agent = Agent(maze = maze_env,
              memory_buffer = memory_buffer,
              use_softmax = True
              )

net = fc_nn(maze.size, maze.size, maze.size, 4)
optimizer = optim.Adam(net.parameters(), lr=1e-4)
device = 'cuda'
net.to(device)
batch_size = 32
gamma = 0.9

num_epochs = 4000
cutoff = 3000
epsilon = np.exp(-np.arange(num_epochs)/(cutoff))
epsilon[epsilon > epsilon[100*int(num_epochs/cutoff)]] = epsilon[100*int(
num_epochs/cutoff)]

loss_log = []
best_loss = 1e5

running_loss = 0

for epoch in range(num_epochs):
    loss = 0
    counter = 0
    eps = epsilon[epoch]

    agent.isgameon = True
    _ = agent.env.reset(eps)

    while agent.isgameon:
        agent.make_a_move(net, eps)
        counter += 1

```

```
            if len(agent.buffer) < buffer_start_size:
                continue

            optimizer.zero_grad()
            batch = agent.buffer.sample(batch_size, device = device)
            loss_t = Qloss(batch, net, gamma = gamma, device = device)
            loss_t.backward()
            optimizer.step()

            loss += loss_t.item()

        if (agent.env.current_position == agent.env.goal).all():
            result = 'won'
        else:
            result = 'lost'

        if epoch%1000 == 0:
            agent.plot_policy_map(net, 'sol_epoch_'+str(epoch)+'.pdf', [0.35,-0.3])

        loss_log.append(loss)

        if (epoch > 2000):
            running_loss = np.mean(loss_log[-50:])
            if running_loss < best_loss:
                best_loss = running_loss
                torch.save(net.state_dict(), "best.torch")
                estop = epoch

        print('Epoch', epoch, '(number of moves ' + str(counter) + ')')
        print('Game', result)
        print('[' + '#'*(100-int(100*(1 - epoch/num_epochs))) +
              ' '*int(100*(1 - epoch/num_epochs)) + ']')
        print('\t Average loss: ' + f'{loss:.5f}')
        if (epoch > 2000):
            print('\t Best average loss of the last 50 epochs: ' + f'{best_loss:.5f}' + ', achieved at epoch', estop)
        clear_output(wait = True)

    torch.save(net.state_dict(), "net.torch")
    agent.plot_policy_map(net, 'solution.pdf', [0.35,-0.3])
```

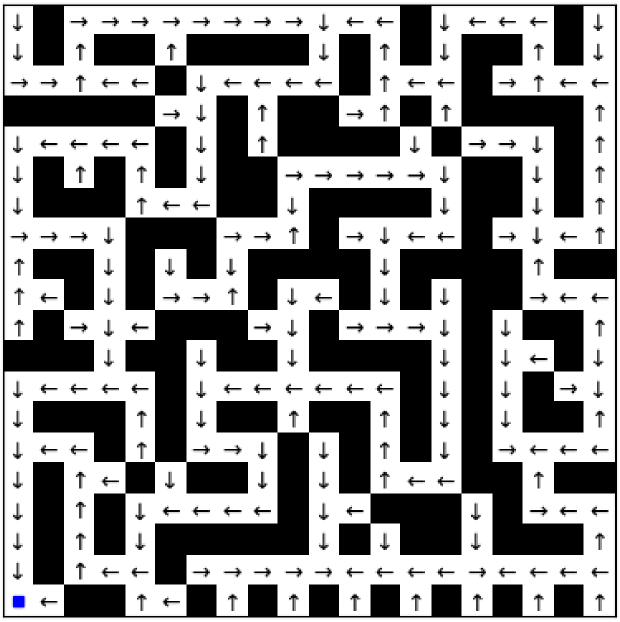

Figure 4: Policy Map Plot After 4000 Epochs of Training with Deep Q-Learning Algorithm

By comapring q-learning and deep q-learning, we understand that our deep method works better. But the training of the deep mwthod takes much time rather than the normal method. I have used 5000 episodes for q-learning and 4000 epoches for deep q-learning. Running of q-learning method takes 20 seconds and running of deep method takes 4 hours.

100 × 100 Maze

In this part, first, we create the maze.

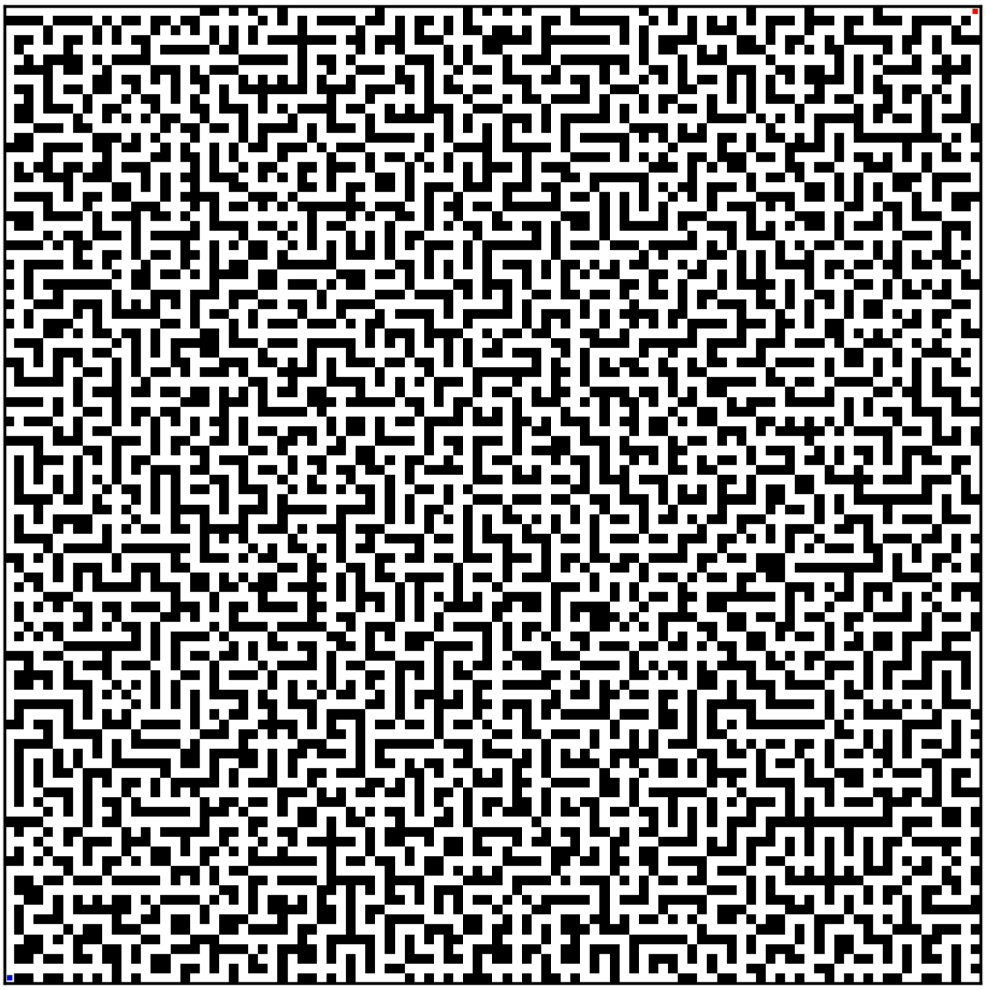

Figure 5: 100×100 Generated Maze

We use all scripts that we've used in previous parts. Now, We compare the results. It is to say that th running for a 100 × 100 maze takes more times in comparison with a 20 × 20 maze. for my computational limitations, the running of the q-learning method for 100 × 100 maze was for 500 episodes that could not learn well. the time for 1000 episodes was about 1 hours that it was not good to and because of that, I prefer to run the script for 500 episodes. The comparsion of q-learning and deep q-learning for 100 × 100 maze was not required. I know the time for running its could based on my gpu and colab's limitaions was not achievable. but the time and the computational cost for the larger maze would be more.

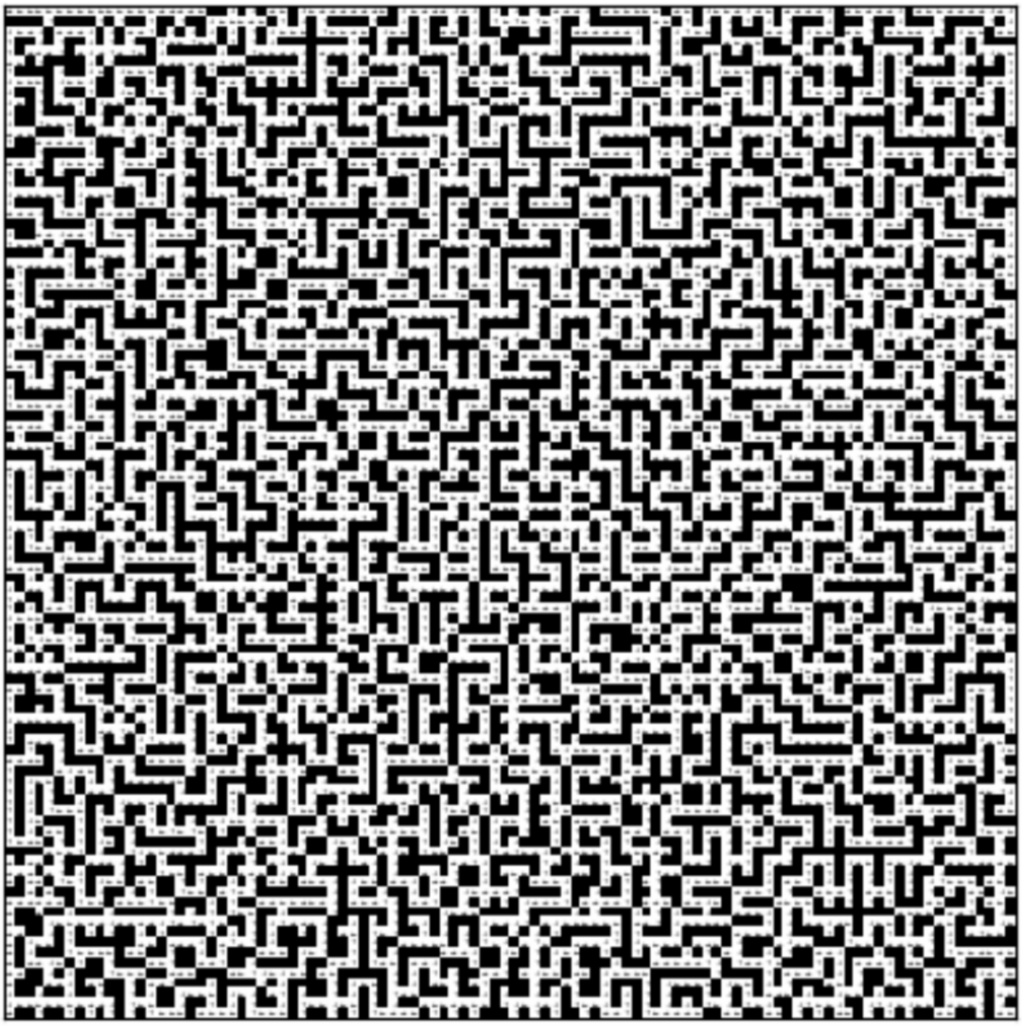

Figure 6: Policy Map Plot After q-learning algorithm for 100×100 Generated Maze