Sri Lanka Institute of Information Technology

MSc in IT specialized in Information Technology

Machine Learning

Assignment 2

Abeykoon A M A C B - MS22903624

Table of Content

Table of Contents

Introduction

An indoor navigation system is a device or technology intended to assist people in navigating inside spaces, such as malls, airports, medical facilities, office buildings, or any other complex structure where standard GPS-based navigation would not be practical. Similar to how GPS navigation works in outside settings, these systems seek to give users precise and effective advice. The most common uses of indoor navigation systems are to help users locate specific places of interest (POIs), including stores, offices, bathrooms, or exits, and to determine the best or quickest paths to get there.

Currently used techniques in indoor navigation systems to determine the shortest path frequently rely on graph-based algorithms such as A* search algorithm or Dijkstra's algorithm.

Dijkstra's Method: In a network where each edge has a weight that is non-negative, this algorithm determines the shortest path between two locations. When it comes to indoor navigation, the interior area is visualised as a graph, with each room or hallway acting as a node and the relationships between them being represented by the edges. Dijkstra's method, which finds the shortest path by adding up the edge weights, is frequently employed in indoor navigation equipment.

A search is an additional graph traversal algorithm that enhances performance by integrating heuristics with the benefits of Dijkstra's algorithm. It is frequently used in interior navigation to determine the shortest path while accounting for the expected cost to get at the destination, which in some circumstances increases efficiency.

One well-known area of machine learning is called reinforcement learning (RL), which stands out for its emphasis on teaching intelligent agents how to make decisions in a certain order in an environment. These agents' main goal is to maximise a cumulative reward by interacting with their environment. In contrast to other machine learning paradigms like supervised or unsupervised learning, reinforcement learning (RL) functions inside a unique framework where agents acquire knowledge by trial and error and iteratively improving their methods.

Reinforcement Learning

RL has several noteworthy benefits. It works best in situations that require flexibility, such as those with changing conditions or situations where the best course of action may change over time. Because of its innate flexibility, RL is a good fit for real-world applications. Furthermore, RL is especially pertinent to fields like robotics and autonomous systems since it facilitates the creation of autonomous agents with the ability to make decisions on their own. Furthermore, RL demonstrates task-to-task information transfer, which facilitates effective learning in similar contexts. Finally, reinforcement learning is skilled at managing intricate decision-making tasks in high-dimensional state spaces with a multitude of possible courses of action. This adaptability finds use in a variety of fields, including as recommendation systems, control systems, and game play.

Why Q-Learning?

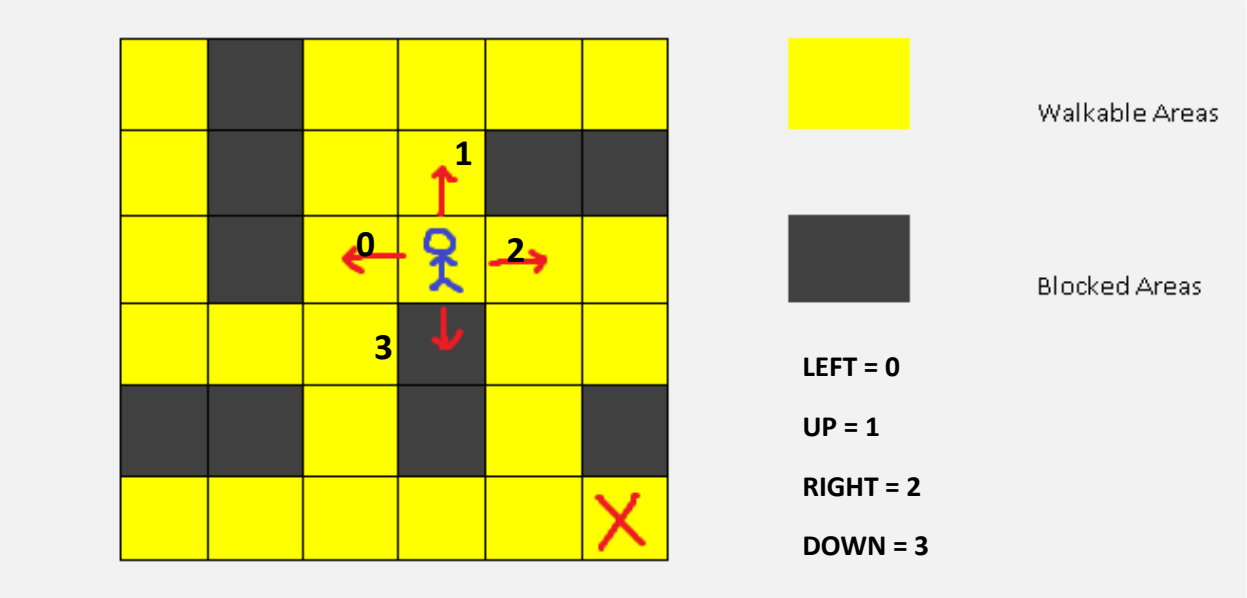
By iteratively assessing the value of state-action pairings, Q-learning is intended to assist an agent in learning the best course of action. The primary concept of Q-learning is to keep track of the predicted cumulative rewards for every action that may be taken in every condition of the environment in a Q-table. The agent eventually approaches the ideal course of action by updating the Q-values through exploration and exploitation in light of its experiences.

The scenario of indoor navigation poses unique challenges that make Q-learning an apt choice for training intelligent agents in this context.

1. 1. The spatial configuration of indoor spaces can exhibit a high degree of complexity, characterized by the presence of several passageways, impediments, and alternate routes. Q-learning is a suitable approach for managing complexity because to its ability to explore and acquire knowledge about the most advantageous actions in various conditions.
2. 2. The objective of Q-learning is to iteratively enhance Q-values in order to uncover the best path from the initial point to the goal. This is consistent with the goal of optimizing indoor navigation.
3. Limited Pre-existing information: Q-learning does not need a substantial amount of pre-existing information about the environment or explicit rules relevant to the domain. The machine possesses the ability to acquire knowledge by direct engagement with its surroundings, rendering it adaptable to a wide range of interior environments.
4. Reward-Based Learning: The Q-learning algorithm utilizes reward feedback to direct the decision-making process of the agent. The reward function within the context of indoor navigation may be strategically constructed to incentivize the agent to quickly achieve the desired destination while simultaneously avoiding obstacles. This adaptable approach allows for the pursuit of various navigation objectives.

The Q-learning method is renowned for its convergence properties, which ensure that the agent will finally attain an optimal policy. The feature possesses considerable significance in scenarios pertaining to interior navigation, particularly in instances where reliability is of paramount relevance.

Environment and Actions



Actions: Within the code, actions pertain to the choices or maneuvers that the intelligent agent, formerly denoted as the "agent," is capable of executing within the confines of the indoor setting. These activities dictate the way the agent traverses across spatial environments. The potential acts that the agent can do are as follows:

- Move Left
- Move Up
- Move Right
- Move Down

The agent's task is to select these actions strategically to reach a target location within the indoor environment efficiently.

The term "environment" in the code refers to the interior location in which the intelligent agent functions. The area in question is a representation of the agent's movement and is characterized by a structure like that of a maze. The indoor environment is shown as a grid, wherein each cell inside the grid may be classified as either open, denoting passable pathways, or blocked, indicating the presence of barriers or walls. The objective of the agent is to successfully traverse the enclosed indoor setting and ultimately arrive at a predetermined target place, sometimes referred to as the endpoint of the labyrinth.

In the context of an indoor navigation system, the actions correspond to the choices or movements available to a user or an autonomous agent when trying to navigate through an indoor space. The indoor environment represents the physical or virtual structure of the building or space where navigation is required. The agent's task is to determine a sequence of actions that will guide it to the desired destination within this indoor environment.

This part of report explains the various inputs and captured data to guide an intelligent agent (previously referred to as the "agent") through the indoor environment. This section provides an overview of these inputs and the data they capture, essential for the agent's effective navigation. These elements collectively enable the agent's successful movement within the indoor space.

Inputs and Captured Data in Indoor Navigation

Maze Structure (Input): Within the navigation system, the maze structure is the foundation. It defines the indoor environment and is represented as a grid. In the code, the maze is defined as a two-dimensional array, where each cell is marked as open (1.0) or blocked (0.0). This grid serves as the spatial framework for navigation.

Inputs

• The initial position of the agent has significant importance within the labyrinth. The code specifies the initial location of the agent. The placement of the agent at the start of its trip within the interior environment is determined by this position.

• The goal place, commonly known as the destination, has a pivotal role. In the provided code, the target position is recognized and designated as the cell that represents the objective. The primary goal of the agent is to successfully traverse the maze and ultimately arrive at a predetermined destination.

The activities that the agent can do are specified in the code. The actions encompass shifting to the left, ascending, shifting to the right, and descending. These activities determine the agent's ability to traverse the surroundings.

Captured Data

- Environmental Observations: As the agent navigates the indoor environment within the code, it consistently records observations. The observations encompass many factors such as the agent's present location, the condition of adjacent cells (whether they are accessible or obstructed), and the relative distance to the desired destination. The aforementioned observations are recorded as data and function as the foundation for the decision-making process of the agent.
- Rewards are provided to the agent as it navigates through the code. These awards serve as a means of providing feedback on the performance of the agent. Positive outcomes can signify advancement, while bad outcomes might indicate undesirable behaviours or interactions with difficulties. The reward system exerts an impact on the decision-making process of the agent.

- Visited Cells: The agent maintains a record of visited cells in the code. This record is essential to prevent revisiting the same cells, ensuring efficient exploration, and avoiding repetitive paths.
- The current state of the game is stored in the code, representing three possible outcomes: "win," "lose," or "not over." The provided status indicates the result of the navigation procedure, indicating whether the agent has achieved the intended destination or if the navigation is still in progress.

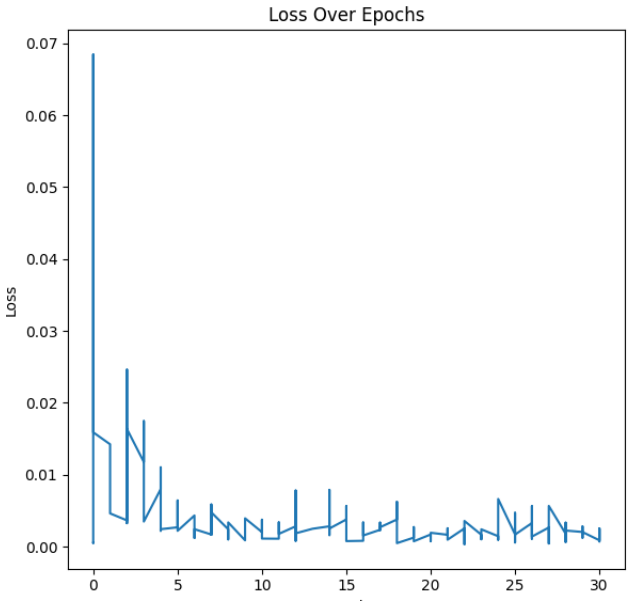
Reward function

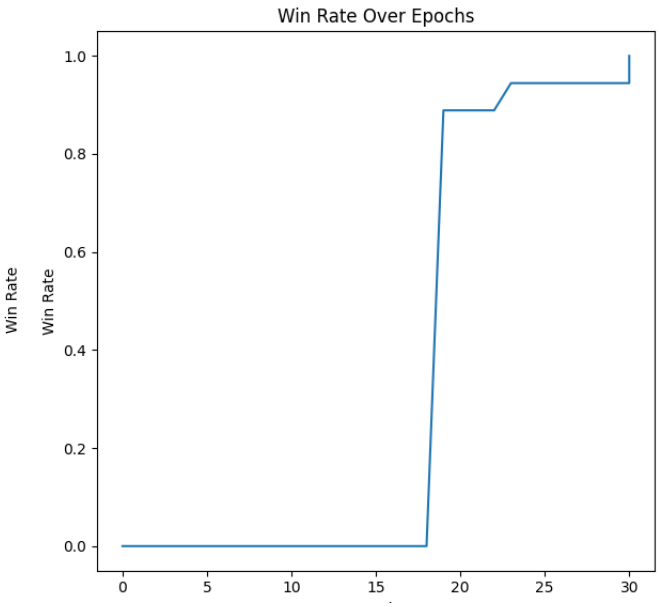
A reward function is a fundamental concept in reinforcement learning, serving as a mechanism to provide feedback to an intelligent agent as it interacts with an environment. It quantifies the immediate benefit or desirability of the agent's actions and helps guide the agent's learning process.

Rewards can be either positive or negative. The agent is awarded these reward according to how they interact with the environment. In the solution proposed by the team uses following rewarding function

1. Winning Reward: When the intelligent agent successfully reaches the target location (the destination), it receives a reward of +1.0. This positive reward reinforces the agent's behavior and encourages it to reach its goal.
2. Blocked Cell Penalty: If the agent's action leads it to a blocked cell (an obstacle or wall), it incurs a penalty. In this case, the reward is set to a low value of -0.75, discouraging the agent from making invalid moves.
3. Visited Cell Penalty: Revisiting cells is discouraged by imposing a penalty of -0.25. If the agent revisits a cell it has been to before, it receives this negative reward.
4. Valid Move Reward: If the agent makes a valid move within the maze, it receives a modest positive reward of +0.04. This encourages the agent to explore the environment while avoiding blocked cells and revisited locations.

Results





Model reached 100% success rate

Limitations

Reinforcement Learning (RL) poses substantial challenges in terms of training time, computational requirements, and convergence reliability. RL agents often necessitate prolonged interactions with the

environment, making training time-intensive, particularly in complex domains. Moreover, RL's applicability to high-dimensional state and action spaces demands significant computational resources due to the use of complex function approximation techniques. Additionally, credit assignment complexities and sensitivity to hyperparameters contribute to the protracted learning process. Furthermore, RL algorithms lack guarantees of convergence, raising concerns about their reliability in achieving optimal policies. These combined challenges emphasize the need for ongoing research and refinement to make RL more practical and efficient for real-world applications.

Analysis and Discussion

In Reinforcement Learning (RL), the process of fine tuning hyperparameters is important for expedited training and avoidance of non-convergence. Parameters like epsilon, the number of training epochs, and maximum memory size play crucial roles in this process. Adjusting epsilon helps balance exploration and exploitation, guiding efficient learning. Optimizing training epochs and memory size tailors learning, preventing overfitting. Furthermore, employing a weighted Q-table from prior training accelerates learning and adaptation, while defining rewards judiciously is fundamental for efficient training. These practices collectively expedite the training process and lead to more successful RL outcomes.

References

[1] M. A. Samsuden, N. M. Diah, and N. A. Rahman, "A Review Paper on Implementing Reinforcement Learning Technique in Optimising Games Performance," 2019 IEEE 9th International Conference on System Engineering and Technology (ICSET), Shah Alam, Malaysia, 2019, pp. 258-263, doi: 10.1109/ICSEngT.2019.8906400.

[2] R. Nian, J. Liu, and B. Huang, "A review on reinforcement learning: Introduction and applications in industrial process control," Computers & Chemical Engineering, vol. 139, p. 106886, 2020.

[3] P. Dayan and Y. Niv, "Reinforcement learning: the good, the bad and the ugly," Current opinion in neurobiology, vol. 18, no. 2, pp. 185-196, 2008.

Appendix

```
#import libraries and variable declarations
from __future__ import print_function
import os, sys, time, datetime, json, random
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt
from keras.layers import PReLU
from keras.models import model_from_json, load_model
import json
%matplotlib inline
import os
from keras.models import model_from_json
model_file_path = r'C:\Users\Legion\Desktop\New folder\model'


epoch_list = []
loss_list = []
win_rate_list = []
# Define the function to load the model and its weights
def load_model_and_weights(model_file_path):
    h5file = model_file_path + ".h5"
    json_file = model_file_path + ".json"
    if os.path.isfile(h5file) and os.path.isfile(json_file):
        try:
            # Load the model architecture from the JSON file
            with open(json_file, "r") as json_file:
                model_json = json.load(json_file)
                model = model_from_json(model_json)
            # Load the model weights from the H5 file
            model.load_weights(h5file)
            print("Model and weights loaded successfully!")
```

```
            return model
        except Exception as e:
            print("Error loading model and weights:", str(e))
            return None
    else:
        print("Model and weights files do not exist.")
        return None
    # Try to load the model and weights
loaded_model = load_model_and_weights(model_file_path)
if loaded_model is not None:
    # The model and weights were loaded successfully
    print("Model summary:")
    loaded_model.summary()
else:
    # Handle the case when the model couldn't be loaded
    print("Model and weights could not be loaded.")
#create maze using a numpy array
maze = np.array([
    [1., 0., 1., 1., 1., 1. ],
    [1., 0., 1., 1., 0., 0. ],
    [1., 0., 1., 1., 1., 1. ],
    [1., 1., 1., 0., 1., 0. ],
    [0., 0., 1., 0., 1., 1. ],
    [1., 1., 1., 1., 1., 1. ]
])
#defining actions/ visualization variables/epsilon value
visited_mark = 0.6
agent_mark = 0.3
LEFT = 0
UP = 1
RIGHT = 2
DOWN = 3
```

```
actions_dict = {
    LEFT: 'left',
    UP: 'up',
    RIGHT: 'right',
    DOWN: 'down',
}
num_actions = len(actions_dict)
epsilon = 0.2

#Qmaze class
class Qmaze(object):
    def __init__(self, maze, agent=(0, 0)):
        self._maze = np.array(maze)
        nrows, ncols = self._maze.shape
        self.target = (nrows - 1, ncols - 1)
        self.free_cells = [(r, c) for r in range(nrows) for c in range(ncols) if self._maze[r, c] == 1.0]
        self.free_cells.remove(self.target)
        if self._maze[self.target] == 0.0:
            raise Exception("Invalid maze: target cell cannot be blocked!")
        if not agent in self.free_cells:
            raise Exception("Invalid Agent Location: must sit on a free cell")
        self.reset(agent)

    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.min_reward = -0.2 * self.maze.size
        self.total_reward = 0
```

```
    self.visited = set()

def update_state(self, action):
    nrows, ncols = self.maze.shape
    nrow, ncol, nmode = agent_row, agent_col, mode = self.state

    if self.maze[agent_row, agent_col] > 0.0:
        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
        if 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 == nrows - 1 and agent_col == ncols - 1:
        return 1.0
```

```
    if mode == 'blocked':
        return self.min_reward - 1
    if (agent_row, agent_col) in self.visited:
        return -0.25
    if mode == 'invalid':
        return -0.75
    if mode == 'valid':
        return 0.04

def act(self, action):
    self.update_state(action)
    reward = self.get_reward()
    self.total_reward += reward
    status = self.game_status()
    envstate = self.observe()
    return envstate, reward, status

def observe(self):
    canvas = self.draw_env()
    envstate = canvas.reshape((1, -1))
    return envstate

def draw_env(self):
    canvas = np.copy(self.maze)
    nrows, ncols = self.maze.shape
    for r in range(nrows):
        for c in range(ncols):
            if canvas[r, c] > 0.0:
                canvas[r, c] = 1.0
    row, col, valid = self.state
    canvas[row, col] = agent_mark
    return canvas
```

```
def game_status(self):
    if self.total_reward < self.min_reward:
        return 'lose'
    agent_row, agent_col, mode = self.state
    nrows, ncols = self.maze.shape
    if agent_row == nrows - 1 and agent_col == ncols - 1:
        return 'win'
    return 'not_over'


def valid_actions(self, cell=None):
    if cell is None:
        row, col, _ = self.state
    else:
        row, col = cell
    actions = [0, 1, 2, 3]
    nrows, ncols = self.maze.shape
    if row == 0:
        actions.remove(1)
    if row == nrows - 1:
        actions.remove(3)
    if col == 0:
        actions.remove(0)
    if col == ncols - 1:
        actions.remove(2)
    if row > 0 and self.maze[row - 1, col] == 0.0:
        actions.remove(1)
    if row < nrows - 1 and self.maze[row + 1, col] == 0.0:
        actions.remove(3)
    if col > 0 and self.maze[row, col - 1] == 0.0:
        actions.remove(0)
    if col < ncols - 1 and self.maze[row, col + 1] == 0.0:
```

```
            actions.remove(2)
        return actions


# Define the Experience class for replay memory
class Experience(object):
    def __init__(self, model, max_memory=1000, discount=0.97):
        self.model = model
        self.max_memory = max_memory
        self.discount = discount
        self.memory = list()
        self.num_actions = model.output_shape[-1]

    def remember(self, episode):
        self.memory.append(episode)
        if len(self.memory) > self.max_memory:
            del self.memory[0]

    def predict(self, envstate):
        return self.model.predict(envstate)[0]

    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, next_envstate, game_over = self.memory[j]
            inputs[i:i + 1] = envstate
            # There should be no target values for actions not taken.
            targets[i] = self.predict(envstate)
            Q_sa = np.max(self.predict(next_envstate))
```

```
            if game_over:
                targets[i, action] = reward
            else:
                # reward + gamma * max_a' Q(s', a')
                targets[i, action] = reward + self.discount * Q_sa
        return inputs, targets


#model training function

def qtrain(model, maze, **opt):
    global epsilon
    n_epoch = opt.get('n_epoch', 100)
    max_memory = opt.get('max_memory', 100)
    data_size = opt.get('data_size', 40)
    weights_file = opt.get('weights_file', "")
    name = opt.get('name', 'model')
    start_time = datetime.datetime.now()

    # If you want to continue training from a previous model,
    # just supply the h5 file name to weights_file option
    if weights_file:
        print("loading weights from file: %s" % (weights_file,))
        model.load_weights(weights_file)

    # Construct environment/game from the numpy array: maze
    qmaze = Qmaze(maze)

    # Initialize the experience replay object
    experience = Experience(model, max_memory=max_memory)

    win_history = []   # history of win/lose game
```

```
n_free_cells = len(qmaze.free_cells)
hsize = qmaze.maze.size // 2   # history window size
win_rate = 0.0
imctr = 1

# Define a function to visualize the maze
def visualize_maze(qmaze):
    plt.grid('on')
    nrows, ncols = qmaze.maze.shape
    ax = plt.gca()
    ax.set_xticks(np.arange(0.5, nrows, 1))
    ax.set_yticks(np.arange(0.5, ncols, 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    canvas = np.copy(qmaze.maze)
    for row, col in qmaze.visited:
        canvas[row, col] = 0.6
    agent_row, agent_col, _ = qmaze.state
    canvas[agent_row, agent_col] = 0.3   # agent cell
    canvas[nrows - 1, ncols - 1] = 0.9  # target cell
    img = plt.imshow(canvas, interpolation='none', cmap='gray')
    plt.pause(0.01)

# Modify the training loop to visualize the maze
for epoch in range(n_epoch):
    loss = 0.0
    agent_cell = random.choice(qmaze.free_cells)
    qmaze.reset(agent_cell)
    game_over = False

    visualize_maze(qmaze)  # Initial visualization
```

```
while not game_over:
    valid_actions = qmaze.valid_actions()
    if not valid_actions:
        break
    prev_envstate = qmaze.observe()
    if np.random.rand() < epsilon:
        action = random.choice(valid_actions)
    else:
        action = np.argmax(experience.predict(prev_envstate))

    envstate, reward, game_status = qmaze.act(action)

    action_str = actions_dict[action]
    print("Action: {}, Reward: {}".format(action_str, reward))

    if game_status == 'win':
        win_history.append(1)
        game_over = True
    elif game_status == 'lose':
        win_history.append(0)
        game_over = True
    else:
        game_over = False

    episode = [prev_envstate, action, reward, envstate, game_over]
    experience.remember(episode)

    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)



        # Append statistics for this epoch
        epoch_list.append(epoch)
        loss_list.append(loss)
        win_rate_list.append(win_rate)

        visualize_maze(qmaze)  # Visualize at each step

    if len(win_history) > hsize:
        win_rate = sum(win_history[-hsize:]) / hsize

    dt = datetime.datetime.now() - start_time
    t = format_time(dt.total_seconds())
    template = "Epoch: {:03d}/{:d} | Loss: {:.4f} | Episodes: {:d} | Win count: {:d} | Win rate: {:.3f} | time: {}"
    print(template.format(epoch, n_epoch - 1, loss, len(experience.memory), sum(win_history), win_rate, t))

    if win_rate > 0.9:
        epsilon = 0.05
    if sum(win_history[-hsize:]) == hsize and qmaze.game_status() == 'win':
        print("Reached 100%% win rate at epoch: %d" % (epoch,))
        break



# Append statistics for this epoch
epoch_list.append(epoch)
loss_list.append(loss)
```

```
win_rate_list.append(win_rate)


plt.figure(figsize=(12, 6))

# Chart 1: Loss over epochs
plt.subplot(121)
plt.plot(epoch_list, loss_list)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss Over Epochs')

# Chart 2: Win rate over epochs
plt.subplot(122)
plt.plot(epoch_list, win_rate_list)
plt.xlabel('Epoch')
plt.ylabel('Win Rate')
plt.title('Win Rate Over Epochs')

# Display the charts
plt.tight_layout()
plt.show()

# Save the trained model weights and architecture, which will be used by the visualization code


# Save the entire model, including architecture and weights


h5file = model_file_path + ".h5"
kerasfile = model_file_path + ".keras"
json_file = model_file_path + ".json"
model.save_weights(h5file, overwrite=True)
```

```
    model.save(kerasfile, overwrite=True)
    with open(json_file, "w") as outfile:
        json.dump(model.to_json(), outfile)
    end_time = datetime.datetime.now()
    dt = datetime.datetime.now() - start_time
    seconds = dt.total_seconds()
    t = format_time(seconds)
    print('Files: %s, %s' % (h5file, json_file))
    print("n_epoch: %d, max_mem: %d, data: %d, time: %s" % (epoch, max_memory, data_size, t))
    return seconds

# utility for printing readable time strings:
def format_time(seconds):
    if seconds < 400:
        s = float(seconds)
        return "%.1f seconds" % (s,)
    elif seconds < 4000:
        m = seconds / 60.0
        return "%.2f minutes" % (m,)
    else:
        h = seconds / 3600.0
        return "%.2f hours" % (h,)



#function to build the model
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

weights_file = r'C:\Users\Legion\Desktop\New folder\model1.h5'



#build and train model

model = build_model(maze)
qtrain(model, maze, epochs=1000, max_memory=8*maze.size, data_size=32)

#Test trained model
########run the program from model saved in filesystem

from keras.models import load_model
import numpy as np
import matplotlib.pyplot as plt
import os

# Define the Qmaze class and other necessary functions here
# ...

# Constants for visualization
visited_mark = 0.6
agent_mark = 0.3

# Load the trained model from your computer's hard disk
model = load_model(r'C:\Users\Legion\Desktop\New folder\model.keras')

# Function to draw the maze and agent's position
def draw_maze_with_agent(qmaze):
    nrows, ncols = qmaze.maze.shape
```

```
    canvas = np.copy(qmaze.maze)
    for r, c in qmaze.visited:
        canvas[r, c] = visited_mark
    agent_row, agent_col, _ = qmaze.state
    canvas[agent_row, agent_col] = agent_mark
    canvas[nrows - 1, ncols - 1] = 1.0  # Target cell

    plt.grid('on')
    ax = plt.gca()
    ax.set_xticks(np.arange(0.5, nrows, 1))
    ax.set_yticks(np.arange(0.5, ncols, 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])

    plt.imshow(canvas, interpolation='none', cmap='gray')
    plt.show()
from keras.models import load_model
import numpy as np
import matplotlib.pyplot as plt
# Define the Qmaze class and other necessary functions here
# Constants for visualization
visited_mark = 0.6
agent_mark = 0.3
# Define the directory where you want to save the maze images
output_directory = r'C:\Users\Legion\Desktop\New folder\images'

# Load the trained model from your computer's hard disk
model = load_model(r'C:\Users\Legion\Desktop\New folder\model.keras')

# Function to draw the maze and agent's position
def draw_maze_with_agent(qmaze, step_number):
    nrows, ncols = qmaze.maze.shape
```

```
    canvas = np.copy(qmaze.maze)
    for r, c in qmaze.visited:
        canvas[r, c] = visited_mark
    agent_row, agent_col, _ = qmaze.state
    canvas[agent_row, agent_col] = agent_mark
    canvas[nrows - 1, ncols - 1] = 1.0  # Target cell

    plt.grid('on')
    ax = plt.gca()
    ax.set_xticks(np.arange(0.5, nrows, 1))
    ax.set_yticks(np.arange(0.5, ncols, 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])

    plt.imshow(canvas, interpolation='none', cmap='gray')
    plt.savefig(os.path.join(output_directory, f'step_{step_number}.png'))
    plt.close()


# Function to test a trained model with the maze
def test_trained_model_with_images(model, maze):
    qmaze = Qmaze(maze)
    step_number = 0

    while True:
        agent_cell = (0, 5)  # Start from the beginning
        qmaze.reset(agent_cell)

        while True:
            # Create an image of the current step and display it
            draw_maze_with_agent(qmaze, step_number)
            step_number += 1
```

```
            # Get the current state
            envstate = qmaze.observe()

            # Predict the best action to take using the trained model
            action = np.argmax(model.predict(envstate))

            # Perform the predicted action
            _, reward, game_status = qmaze.act(action)

            print(f"Step: {step_number} - Reward: {reward} - Status: {game_status}")

            if game_status == 'win' or game_status == 'lose':
                break

        # Ask if you want to run another test
        another_test = input("Do you want to run another test (y/n)? ")
        if another_test.lower() != 'y':
            break
# Load the maze or create it as in your previous code
maze = np.array([
    [1., 0., 1., 1., 1., 1. ],
    [1., 0., 1., 1., 0., 0. ],
    [1., 0., 1., 1., 1., 1. ],
    [1., 1., 1., 0., 1., 0. ],
    [0., 0., 1., 0., 1., 1. ],
    [1., 1., 1., 1., 1., 1. ]
])
# Call the test function with the trained model and maze
test_trained_model_with_images(model, maze)

============================================================
```

```
# make a gif
from PIL import Image, ImageSequence
import os
from IPython.display import display, Image as IPImage

# Function to create a GIF from the saved images with a specified frame duration
def create_gif(image_directory, gif_filename, frame_duration):
    images = []
    for filename in sorted(os.listdir(image_directory)):
        if filename.endswith('.png'):
            img = Image.open(os.path.join(image_directory, filename))
            images.append(img)

    # Save the images as a GIF with the specified frame duration
    images[0].save(gif_filename, save_all=True, append_images=images[1:], loop=0, duration=int(frame_duration
* 1000))

# Call the create_gif function to create the GIF
image_directory = 'C:/Users/Legion/Desktop/New folder/images'
gif_filename = 'maze_steps.gif'
frame_duration = 0.5  # Adjust the frame duration as needed (in seconds)
create_gif(image_directory, gif_filename, frame_duration)

# Display the GIF in the notebook
with open(gif_filename, 'rb') as file:
    display(IPImage(data=file.read(), format="png"))
```