Assignment 4

Experiment 10

Title: Q-Learning and Backpropagation

Name of Student: Sangeet Agrawal PRN No. 21070122140

DoP: 7 Oct DoS: 9 Oct

Aim: Study and Implementation of Q-learning

Problem Statement: Develop a game using Reinforcement learning (Q learning).

Rules are as follows:

- Travelers have 8 degrees of movement. Up, down, left, and right.
- Create a maze as discussed in the class.

* Theory :

Reinforcement learning (RL) involves agents learning to take actions in an environment to maximize rewards. Q-learning is a popular RL algorithm that updates a Q-value table represent the best actions for each state. The agent balances exploration (trying new actions) and exploitation (choosing the best-known action) to learn the optimal policy.

Problem Statement 1 :

To demonstrate Reinforcement Learning at work, You can develop an applet that uses RL methods and learns to play a game. The particular game was Cat and Mouse. For those of you unfamiliar with the game, it is a simple game. There is a cat, a mouse, a piece of cheese as well as some obstacles in the cat and mouse world. The mouse tries to avoid getting caught by the cat, at the same time trying to get to the cheese to eat it. The mouse is the one learning in the applet, the cat is already programmed to go for the mouse. When the applet loads, either choose one of the pre-specified obstacle layouts or have one randomly generated for you.

You can use any other platform for the same.

The rules of the Cat and Mouse game are:

- Both the cat and mouse have 8 degrees of movement. Up, down, left and right, as well as the four diagonals.
- The mouse scores a point for getting the cheese. The mouse gets the cheese when it is in the same square as the cheese.
- The cat scores a point for catching the mouse, by simply moving to the same square as the mouse.
- If the mouse gets the cheese, a new piece is placed randomly while the cat and mouse keep their positions.
- The game is over when the cat catches the mouse. The scores are then updated and a new game can begin.

Hardware/Software:

- Processors. Minimum: Any Intel or AMD x86-64 processor. Recommended: Any Intel or AMD x86-64 processor with four logical cores and AVX2 instruction set support.
- Disk. Minimum: 3.4 GB of disk space for MATLAB only, 5-8 GB for a typical installation.
- Recommended: An SSD is recommended
- RAM. Minimum: 4 GB. Recommended: 8 GB.
- Software: MATLAB R2023a.

* Procedure :

1) Created a grid-based Cat & Mouse game where the mouse seeks cheese while avoiding the cat, both capable of moving in 8 directions.

2) Implemented a Q-learning agent using a Q-table to store state-action values and defined parameters for learning rate (α), discount factor (γ), and exploration factor (ϵ).

3) Represented the state as a tuple of the positions of the mouse, cat & cheese.

4) Developed a visual simulation using pygame, where the cat moves toward the mouse, ~~and the~~ and the mouse learns to avoid the cat while reaching the cheese.

5) The agent chooses actions based on the Q-table, updates Q-values after each action, and renders the game environment.

Code:

```
import numpy as np
import random
import pygame

# Constants
GRID_SIZE = 10
ACTIONS = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1),
(-1, 1), (1, -1)]  # 8 degrees of movement
ALPHA = 0.1  # Learning rate
GAMMA = 0.9  # Discount factor
EPSILON = 0.1  # Exploration factor
CELL_SIZE = 50  # Size of each cell in the grid

# Initialize pygame
pygame.init()
screen = pygame.display.set_mode((GRID_SIZE * CELL_SIZE, GRID_SIZE
* CELL_SIZE))
pygame.display.set_caption("Cat and Mouse Game")
clock = pygame.time.Clock()

# Define colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
BLUE = (0, 0, 255)
YELLOW = (255, 255, 0)

# Define the environment (Cat and Mouse Grid)
class CatMouseEnv:
   def __init__(self, grid_size):
       self.grid_size = grid_size
       self.reset()

   def reset(self):
       # Place mouse, cat, and cheese in random positions
       self.mouse_pos = [random.randint(0, self.grid_size - 1),
random.randint(0, self.grid_size - 1)]
       self.cat_pos = [random.randint(0, self.grid_size - 1),
random.randint(0, self.grid_size - 1)]
       self.cheese_pos = [random.randint(0, self.grid_size - 1),
```

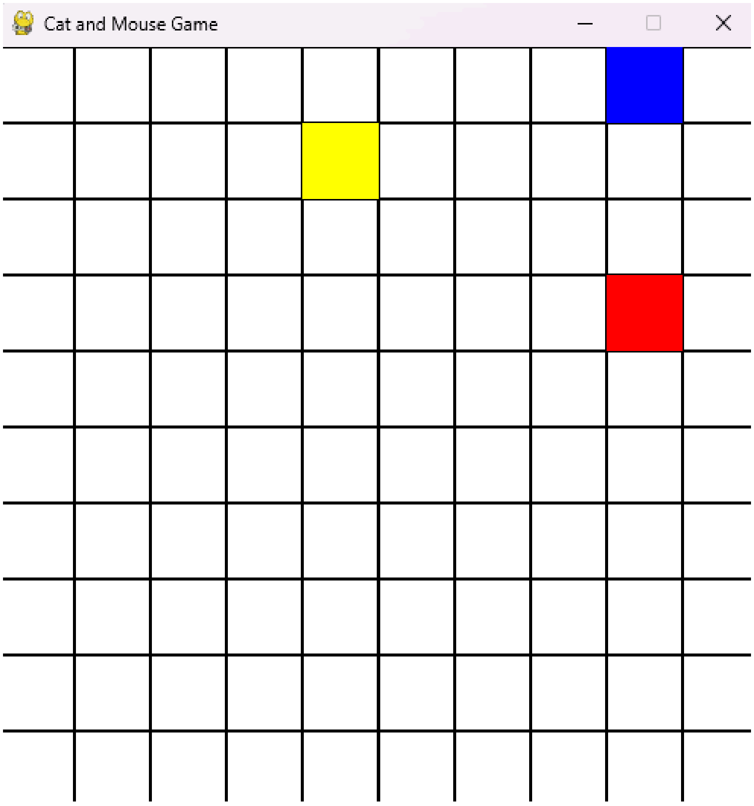
```
random.randint(0, self.grid_size - 1)]
        return self.get_state()

    def get_state(self):
        return tuple(self.mouse_pos + self.cat_pos +
self.cheese_pos)

    def is_valid(self, pos):
        return 0 <= pos[0] < self.grid_size and 0 <= pos[1] <
self.grid_size

    def step(self, action):
        # Move mouse
        new_mouse_pos = [self.mouse_pos[0] + action[0],
self.mouse_pos[1] + action[1]]
        if self.is_valid(new_mouse_pos):
            self.mouse_pos = new_mouse_pos

        # Move cat towards mouse
        self.move_cat()

        # Check for win/lose condition
        if self.mouse_pos == self.cheese_pos:
            reward = 1  # Mouse gets cheese
            done = False  # Game continues after getting the cheese
            self.cheese_pos = [random.randint(0, self.grid_size -
1), random.randint(0, self.grid_size - 1)]  # New cheese position
        elif self.mouse_pos == self.cat_pos:
            reward = -1  # Cat catches mouse
            done = True  # Game ends when the cat catches the mouse
        else:
            reward = 0  # No one has won yet
            done = False

        return self.get_state(), reward, done

    def move_cat(self):
        if self.cat_pos[0] < self.mouse_pos[0]:
            self.cat_pos[0] += 1
        elif self.cat_pos[0] > self.mouse_pos[0]:
            self.cat_pos[0] -= 1
```

```
        if self.cat_pos[1] < self.mouse_pos[1]:
            self.cat_pos[1] += 1
        elif self.cat_pos[1] > self.mouse_pos[1]:
            self.cat_pos[1] -= 1

    def render(self):
        # Draw grid
        screen.fill(WHITE)
        for row in range(self.grid_size):
            for col in range(self.grid_size):
                pygame.draw.rect(screen, BLACK, pygame.Rect(col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE, CELL_SIZE), 1)

        # Draw mouse, cat, and cheese
        pygame.draw.rect(screen, BLUE, pygame.Rect(self.mouse_pos[1] * CELL_SIZE, self.mouse_pos[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
        pygame.draw.rect(screen, RED, pygame.Rect(self.cat_pos[1] * CELL_SIZE, self.cat_pos[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
        pygame.draw.rect(screen, YELLOW, pygame.Rect(self.cheese_pos[1] * CELL_SIZE, self.cheese_pos[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
        pygame.display.flip()

# Q-learning Agent
class QLearningAgent:
    def __init__(self, actions, alpha=ALPHA, gamma=GAMMA, epsilon=EPSILON):
        self.q_table = {}  # Q-table: maps state-action pairs to rewards
        self.actions = actions
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon

    def get_q_value(self, state, action):
        return self.q_table.get((state, action), 0.0)

    def update_q_value(self, state, action, reward, next_state):
        old_q_value = self.get_q_value(state, action)
        max_next_q_value = max([self.get_q_value(next_state, a) for a in self.actions])
```

```
        new_q_value = old_q_value + self.alpha * (reward +
self.gamma * max_next_q_value - old_q_value)
        self.q_table[(state, action)] = new_q_value

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:  # Explore
            return random.choice(self.actions)
        else:  # Exploit
            q_values = [self.get_q_value(state, action) for action
in self.actions]
            max_q = max(q_values)
            return self.actions[q_values.index(max_q)]

# Main loop to run the simulation with pygame visualization
def run_simulation(episodes=500):
    env = CatMouseEnv(GRID_SIZE)
    agent = QLearningAgent(ACTIONS)

    for episode in range(episodes):
        state = env.reset()
        done = False
        while not done:
            # Handle Pygame events (for closing the window)
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    return

            # Choose action and take a step in the environment
            action = agent.choose_action(state)
            next_state, reward, done = env.step(action)
            agent.update_q_value(state, action, reward, next_state)
            state = next_state

            # Render the environment
            env.render()

            # Control the frame rate
            clock.tick(5)

            if done:
                break  # Terminate the game if the cat catches the
```

```
mouse

        if episode % 50 == 0:
            print(f"Episode {episode}: Q-table size
{len(agent.q_table)}")


if __name__ == "__main__":
    run_simulation()
```

Output:



* Conclusion:

The Q-learning agent successfully learns to play the Cat & Mouse game, improving its strategy over time by balancing exploration and exploitation to maximize rewards.