Homework 2 ML CSE-584

Name: Balaji Udayagiri
username: gbu5048

Please find the code for the github link to the original code here: link

Abstract
The code implements a reinforcement learning algorithm using Q-learning in a simulated environment. It would be navigating an agent mouse through the grid, avoiding cats and in search of cheese. The main goal of this code is to efficiently train a mouse agent so as to maximize rewards by appropriately exploring the environment, interacting with it, and updating a learned strategy on experiences gained so far. A grid with diverse entities-mouse, cat, cheese, obstacles-building walls-constitutes a dynamic scenario for reinforcement learning.

Code Structure Overview
The code realizes two major components: the Q-learning algorithm encapsulated in the qlearn_mod_random.py file and the simulation environment with the main script defining agents' behaviors.

Q-learning Algorithm qlearn_mod_random.py:

The class QLearn implements all functionalities of the Q-learning methodology, from managing the Q-table to action selection and learning from rewards. The structure of this class may help to emphasize several aspects of exploration-exploitation trade-off that are important for efficient reinforcement learning.

Simulation Environment and Agent Behaviors:

Then, it sets up the environment, followed by importing the libraries and reloading the modules in case anything has changed. The grid is then filled with cells, and the agents such as mouse, cat, and cheese are initialized. Each agent's respective class describes the behavior of that particular agent; the mouse implements the Q-learning algorithm to navigate and learn.
Key Components of Q-learning Implementation Highlighted
QLearn Class:

Initialisation: The constructor will initialize the key parameters that characterize the learner, including learning rate α, discount factor γ, and exploration rate ε. The Q-table will be a dictionary for efficient lookups of state-action pairs.
Q-value Management: The getQ returns the Q-value of a given state-action pair. learnQ updates the Q-values using the Q-learning update formula, moving the Q-value toward a reward observed plus gamma times the estimate of the value of the next state.
Action Selection: The chooseAction method implements the epsilon-greedy strategy in itself as

a balance between exploration and exploitation. This randomly selects actions based on the threshold of epsilon while the agent has to make sure that it leverages learned Q-values for optimal decision-making.
Learning Process: It implements the learn method by incorporating observed rewards with future expectations to update Q-values, thereby consolidating the learning of the agent over time.
Agent Classes:

Each agent is implemented as a class, which in turn is derived from a base class. The Mouse class implements the Q-learning approach to make decisions based on the current state of the environment and dynamically adapts its strategy through interactions, i.e., every time it encounters the cat or cheese. The Cat and Cheese classes implement basic behaviors for the cat's pursuit of and the cheese's static presence on the grid.
Interaction and Learning Dynamics
This Q-learning framework does cover majorly the agent-environment interaction. It perceives the current state the mouse is in, gets a reinforcing signal-rewards or penalties for its actions -and updates the Q-table. The grid structure provides a defined area for exploration that the mouse has to go through intelligently to maximize rewards while avoiding dangers.

Conclusion
The Q-learning algorithm implemented above forms the essential elements of reinforcement learning in a simplified yet illustrative environment. The mouse agent has been trained to learn from experience and adjust its behavior accordingly; therefore, this simulation is one of many ways Q-learning can be applied to solving practical, real-world decision-making problems. The overall structure of the code is modular, separating the Q-learning logic from the simulation dynamics.

Below is the fully commented code of the egoMouseLook.py file.

```
import importlib
import random
import cellular
import qlearn_mod_random as qlearn # Q-learning module with exploration
strategy

importlib.reload(cellular)  # Reload cellular module to apply changes
# Constants defining the environment and agent parameters
directions = 8  # Number of possible movement directions for agents (N, NE,
E, etc.)
lookdist = 2    # Distance the mouse can observe surrounding cells for
state calculation
```

```
# Define a list of relative cell positions within `lookdist` range for
state representation
lookcells = []
for i in range(-lookdist, lookdist + 1):
    for j in range(-lookdist, lookdist + 1):
        # Include cells within a manhattan distance and exclude the mouse's
current cell
        if (abs(i) + abs(j) <= lookdist) and (i != 0 or j != 0):
            lookcells.append((i, j))

# Function to randomly place an agent on a free cell in the grid
def pickRandomLocation():
    while True:
        x = random.randrange(world.width)
        y = random.randrange(world.height)
        cell = world.getCell(x, y)
        # Check if cell is free from walls and other agents
        if not (cell.wall or len(cell.agents) > 0):
            return cell

# Define cell properties in the environment
class Cell(cellular.Cell):
    wall = False  # Each cell can either be a wall or empty

    # Method to set cell color for visual representation
    def colour(self):
        return 'black' if self.wall else 'white'

    # Load cell type from external data (wall or empty)
    def load(self, data):
        self.wall = (data == 'X')

# Define the adversarial "cat" agent behavior
class Cat(cellular.Agent):
    colour = 'orange'

    # Update method moves the cat towards the mouse, randomly choosing
directions to reach it
    def update(self):
        if self.cell != mouse.cell:
            self.goTowards(mouse.cell)  # Move towards mouse
            # If already in same cell as mouse, pick a new random direction
            while self.cell == mouse.cell:
```

```
                self.goInDirection(random.randrange(directions))

# Define the "cheese" agent as a static reward target
class Cheese(cellular.Agent):
    colour = 'yellow'

    def update(self):
        pass  # Cheese does not move

# Define the "mouse" agent behavior using Q-learning for decision-making
class Mouse(cellular.Agent):
    colour = 'gray'

    def __init__(self):
        # Initialize Q-learning agent with action space (8 directions) and
hyperparameters
        self.ai = qlearn.QLearn(actions=range(directions),
                                alpha=0.1, gamma=0.9, epsilon=0.1)
        self.eaten = 0   # Counter for times mouse is caught by cat
        self.fed = 0     # Counter for times mouse reaches cheese
        self.lastState = None  # Previous state to update Q-values
        self.lastAction = None  # Previous action to update Q-values

    # Main update function to manage the Q-learning cycle and movement
    def update(self):
        # 1. Calculate the current state based on surrounding cells
        state = self.calcState()
        reward = -1  # Default small negative reward to encourage
efficiency

        # 2. Check for reward or penalty conditions
        if self.cell == cat.cell:
            self.eaten += 1
            reward = -100  # Large penalty for getting caught
            # Update Q-table with large penalty before reset
            if self.lastState is not None:
                self.ai.learn(self.lastState, self.lastAction, reward,
state)
            self.lastState = None
            self.cell = pickRandomLocation()  # Reset mouse to random
location
            return  # End update if caught
```

```
        if self.cell == cheese.cell:
            self.fed += 1
            reward = 50  # Reward for reaching cheese
            cheese.cell = pickRandomLocation()  # Reset cheese location

        # 3. Update Q-table for the last action taken (using Q-learning
rule)
        if self.lastState is not None:
            self.ai.learn(self.lastState, self.lastAction, reward, state)

        # 4. Choose new action using epsilon-greedy strategy from
Q-learning
        action = self.ai.chooseAction(state)
        self.lastState = state
        self.lastAction = action

        # 5. Execute the chosen action, moving in the selected direction
        self.goInDirection(action)

    # Function to calculate the current state as observed by the mouse
    def calcState(self):
        # Inner function to assign values to each surrounding cell
        def cellvalue(cell):
            if cat.cell is not None and (cell.x == cat.cell.x and cell.y ==
cat.cell.y):
                return 3  # Value for cat's position
            elif cheese.cell is not None and (cell.x == cheese.cell.x and
cell.y == cheese.cell.y):
                return 2  # Value for cheese's position
            else:
                return 1 if cell.wall else 0  # Value for wall or empty
cell
        # Represent the state as a tuple of values for each observed cell
        return tuple([cellvalue(self.world.getWrappedCell(self.cell.x + j,
self.cell.y + i))
                      for i, j in lookcells])

# Create instances of each agent and initialize the grid world
mouse = Mouse()
cat = Cat()
cheese = Cheese()

# Define the world with the grid dimensions and load the map layout
```

```
world = cellular.World(Cell, directions=directions,
filename='../worlds/waco.txt')
world.age = 0

# Add agents to the world at random locations
world.addAgent(cheese, cell=pickRandomLocation())
world.addAgent(cat)
world.addAgent(mouse)

# Epsilon decay parameters to adjust exploration rate over time
epsilonx = (0, 100000)
epsilony = (0.1, 0)
epsilonm = (epsilony[1] - epsilony[0]) / (epsilonx[1] - epsilonx[0])

# Set the maximum duration for the simulation
endAge = world.age + 100000

# Main simulation loop for agent updates
while world.age < endAge:
    world.update()  # Update all agents in the world

    # Periodically adjust epsilon to reduce exploration rate
    if world.age % 100 == 0:
        mouse.ai.epsilon = (epsilony[0] if world.age < epsilonx[0] else
                            epsilony[1] if world.age > epsilonx[1] else
                            epsilonm * (world.age - epsilonx[0]) +
epsilony[0])

    # Print status every 10,000 steps to monitor learning progress
    if world.age % 10000 == 0:
        print("{:d}, e: {:0.2f}, W: {:d}, L: {:d}"\
            .format(world.age, mouse.ai.epsilon, mouse.fed, mouse.eaten))
        mouse.eaten = 0
        mouse.fed = 0

# Activate display to visualize the world and agents
world.display.activate(size=30)
world.display.delay = 1

# Continuous update loop for the simulation, printing the size of the
Q-table periodically
while True:
    world.update(mouse.fed, mouse.eaten)
```

```
    bytes = sys.getsizeof(mouse.ai.q)
    print("Bytes: {:d} ({:.2f} KB)".format(bytes, bytes / 1024))
```

Below is the fully commented code of q_learn_mod_random

```
import random

# QLearn class to implement Q-learning with optional exploration strategies
class QLearn:
    def __init__(self, actions, epsilon=0.1, alpha=0.2, gamma=0.9):
        """
        Initialize Q-learning parameters:
        - actions: List of possible actions (e.g., directions in grid).
        - epsilon: Exploration rate (probability of random action).
        - alpha: Learning rate (controls the update of Q-values).
        - gamma: Discount factor (controls how much future rewards are
valued).
        """
        self.q = {}  # Q-table as a dictionary to store state-action values
        self.epsilon = epsilon  # Exploration rate for epsilon-greedy
        self.alpha = alpha      # Learning rate for Q-value updates
        self.gamma = gamma      # Discount factor for future rewards
        self.actions = actions  # List of all possible actions

    def getQ(self, state, action):
        """
        Retrieve the Q-value for a given (state, action) pair.
        If the Q-value doesn't exist in the table, it returns 0.
        """
        return self.q.get((state, action), 0.0)

    def learnQ(self, state, action, reward, value):
        """
        Update the Q-value for a given (state, action) pair.
        Q-learning rule: Q(s, a) += alpha * (reward + gamma * maxQ(s') -
Q(s, a))
        - reward: Observed reward for the (state, action) pair.
        - value: Value estimate for the next state (reward + gamma *
maxQ(s')).
        """
```

```
        oldv = self.q.get((state, action), None)
        # If the Q-value for the (state, action) pair is not initialized
        if oldv is None:
            self.q[(state, action)] = reward  # Initialize Q-value directly
        else:
            # Update Q-value based on the difference between target value
and old value
            self.q[(state, action)] = oldv + self.alpha * (value - oldv)

    def chooseAction(self, state, return_q=False):
        """
        Select an action for the given state based on epsilon-greedy
policy:
        - With probability epsilon, choose a random action (exploration).
        - Otherwise, choose the action with the highest Q-value
(exploitation).
        """
        q = [self.getQ(state, a) for a in self.actions]  # Get Q-values for
all actions
        maxQ = max(q)  # Identify the maximum Q-value for the current state

        if random.random() < self.epsilon:  # Exploration: Choose a random
action
            # Introduce randomization in Q-values to shake up action
selection
            minQ = min(q)
            mag = max(abs(minQ), abs(maxQ))
            q = [q[i] + random.random() * mag - 0.5 * mag for i in
range(len(self.actions))]
            maxQ = max(q)

        count = q.count(maxQ)
        # Handle tie-breaking if multiple actions have the same max Q-value
        if count > 1:
            best = [i for i in range(len(self.actions)) if q[i] == maxQ]
            i = random.choice(best)  # Randomly choose among best actions
        else:
            i = q.index(maxQ)  # Single best action with highest Q-value

        action = self.actions[i]  # Chosen action based on epsilon-greedy
        if return_q:  # Optionally return Q-values
            return action, q
        return action
```

```
    def learn(self, state1, action1, reward, state2):
        """
        Update Q-value for the (state1, action1) based on observed reward
and transition:
        - Computes max Q-value for the next state (state2).
        - Uses Q-learning update rule to adjust the Q-value for the current
state-action.
        """
        maxqnew = max([self.getQ(state2, a) for a in self.actions])
        # Target for Q-value is the immediate reward + discounted max
future reward
        self.learnQ(state1, action1, reward, reward + self.gamma * maxqnew)

# Helper function for formatted output, used in the main program to display
values.
import math
def ff(f, n):
    fs = "{:f}".format(f)  # Format number as floating point
    if len(fs) < n:  # Pad with spaces if needed
        return ("{:" + n + "s}").format(fs)
    else:  # Trim if the formatted string is too long
        return fs[:n]
```