ASSIGNMENT – 2
Reinforcement Learning
Lakshmi Chandrika Yarlagadda(lvy5215)

ABSTRACT:

We solve a navigation problem using SARSA(State-Action-Reward-State-Action) agent which learns the optimum path within a grid environment with reinforcement learning. A Tkinter-generated environment consists of a 5x5 grid where there should be an agent (a rectangle) that must reach some goal (a circle), while avoiding obstacles (triangles). The agent interacts with the environment by taking actions up, down, left, and right, and for reaching the goal, getting positive rewards and in cases of collision, the agent gets negative rewards and neutral rewards otherwise. An agent repeatedly trials, and based on the outcome of its actions, updates knowledge and policy until it learns to navigate the grid effectively.

The SARSA algorithm is one on-policy reinforcement learning approach. An on-policy approach means that the agent learns by following its current policy, taking into consideration exactly the actions that it is going to take and not only the best ones, as it's done in Q-learning. Instead, SARSA makes use of a Q-table that maps state-action pairs into the Q-value, or numerical estimates of the expected future rewards of moving into a particular state by taking an action. The agent updates his Q-values at every step with $Q(s,a)=Q(s,a)+\alpha(r+\gamma Q(s',a')-Q(s,a))$, where α is the learning rate, γ is the discount factor, r is the reward, and $Q(s',a')$ is the Q-value of the next state-action pair. This allows the agent to iteratively converge towards an optimal policy, following the actions taken by the agent on each state.

The agent will follow an epsilon greedy exploration strategy where the agent mostly exploits its current knowledge, taking those actions which have the highest Q-values, while occasionally exploring new actions by selecting them at random. This balance between exploration and exploitation allows the agent to find new paths and improve its policy iteratively. As episodes progress, the Q-values in the table increasingly reflect the best possible action to take in any given state, allowing the agent to navigate to the goal without hitting obstacles. This dynamic interaction between the agent's actions, the feedback from the environment, and the continuous updates of the Q-table expresses the essence of reinforcement learning in a bounded gridded environment.

AGENT:

```
import numpy as np
import random
# Importing defaultdict for storing default values in q_table
from collections import defaultdict
# Importing the custom environment class
from environment import Env  # Importing the custom environment class

## Agent

# SARSA agent class, which learns from every state-action-reward-next_state-next_action tuple
class SARSAgent:

    def __init__(self, actions):
        # List of possible actions the agent can take
        self.actions = actions
        # Learning rate for updating Q-values
        self.learning_rate = 0.01
        # Discount factor for future rewards
        self.discount_factor = 0.9
```

```
        # Epsilon value for epsilon-greedy policy
        self.epsilon = 0.1
        # Q-table where each state-action pair has a default Q-value of 0.0
        self.q_table = defaultdict(lambda: [0.0] * len(actions))

    # Function to upsate the Q-value based on the SARSA algorithm
    def learn(self, state, action, reward, next_state, next_action):
        # Current Q-value for the state-action pair
        current_q = self.q_table[state][action]
        # Q-value of the next state-action pair
        next_state_q = self.q_table[next_state][next_action]
        # Calculate the updated Q-value using SARSA formula
        new_q = (current_q + self.learning_rate *
                (reward + self.discount_factor * next_state_q - current_q))
        # Update the Q-table with the new Q-value
        self.q_table[state][action] = new_q

    # Function to choose an action using an epsilon-greedy policy
    def get_action(self, state):
        if np.random.rand() < self.epsilon:
            # With probability epsilon, choose a random action (exploration part)
            action = np.random.choice(self.actions)
        else:
            # With probability 1 - epsilon, choose the action with the highest Q-value (exploitation part)
            state_action = self.q_table[state]
            action = self.arg_max(state_action)
        return action

## Helper methods

    # Helper function to return the index of the acion with the highest Q-value
    @staticmethod
    def arg_max(state_action):
        # List to store indices of the max Q-value actions
        max_index_list = []
        # Start with the first action's Q-value as the max
        max_value = state_action[0]
        for index, value in enumerate(state_action):
            if value > max_value:
                # Clear previous max indices if a new max is found
                max_index_list.clear()
                # Update max value
                max_value = value
                # Add new max index to the list
                max_index_list.append(index)
            # Add index to list if Q-value equals current max
            elif value == max_value:
                max_index_list.append(index)
        # Return a random index from the max Q-value actions
        return random.choice(max_index_list)
```

```
## Training

# Main loop to run SARSA agent in the environment
if __name__ == "__main__":
    env = Env()
    # Initialize the agent with action space
    agent = SARSAgent(actions=list(range(env.n_actions)))

    # Run episodes for training
    for episode in range(1000):
        # Reset environment and get initial state
        state = env.reset()
        # Choose initial action based on the current state
        action = agent.get_action(str(state))

        while True:
            # Render the environment
            env.render()

            # Take action in the environment and observe the outcome
            next_state, reward, done = env.step(action)
            # Choose next action based on the next state
            next_action = agent.get_action(str(next_state))

            # Update the Q-value based on the SARSA formula
            agent.learn(str(state), action, reward, str(next_state), next_action)

            # Update state and action to the next state and action
            state = next_state
            action = next_action

            # Print all Q-values for states for debugging or analysis
            env.print_value_all(agent.q_table)

            # End the episode if the environment signals done
            if done:
                break
```

ENVIRONMENT:

```
import time
import numpy as np
# Import tkinter for GUI rendering
import tkinter as tk
# Import PIL for image manipulation in tkinter
from PIL import ImageTk, Image

# Set a random seed for reproducibility
np.random.seed(1)
# Aliasing for PhotoImage class from PIL
PhotoImage = ImageTk.PhotoImage
```

```python
# Constants for environment configuration
UNIT = 100        # Pixel size of each grid cell
HEIGHT = 5        # Grid height in cells
WIDTH = 5         # Grid width in cells

## Environment Class

# Environment class inherited from tkinter's Tk class for a graphical interface
class Env(tk.Tk):
    def __init__(self):
        # Initialize tkinter base class
        super(Env, self).__init__()
        # Define action space: up, down, left, right
        self.action_space = ['u', 'd', 'l', 'r']
        # Number of possible actions
        self.n_actions = len(self.action_space)
        self.title('SARSA')
        # Set window size based on grid and unit size
        self.geometry(f'{HEIGHT * UNIT}x{HEIGHT * UNIT}')
        # Load images for different objects
        self.shapes = self.load_images()
        # Create the canvas for grid and objects
        self.canvas = self._build_canvas()
        # List to store text elements for displaying Q-values
        self.texts = []

## Building canvas and images

    # Builds the graphical canvas
    def _build_canvas(self):
        # Create a canvas with a white background
        canvas = tk.Canvas(self, bg='white', height=HEIGHT * UNIT, width=WIDTH * UNIT)

        # Create grid lines for visual separation of cells
        for c in range(0, WIDTH * UNIT, UNIT):    # Vertical grid lines
            x0, y0, x1, y1 = c, 0, c, HEIGHT * UNIT
            canvas.create_line(x0, y0, x1, y1)
        for r in range(0, HEIGHT * UNIT, UNIT):   # Horizontal grid lines
            x0, y0, x1, y1 = 0, r, HEIGHT * UNIT, r
            canvas.create_line(x0, y0, x1, y1)

        # Place images on the grid for the agent, obstacles, and goal
        self.rectangle = canvas.create_image(50, 50, image=self.shapes[0])        # Agent
        self.triangle1 = canvas.create_image(250, 150, image=self.shapes[1])     # Obstacle 1
        self.triangle2 = canvas.create_image(150, 250, image=self.shapes[1])     # Obstacle 2
        self.circle = canvas.create_image(250, 250, image=self.shapes[2])        # Goal

        # Pack the canvas to display it in the tkinter window
        canvas.pack()

        return canvas
```

```python
    # Loads images for the agent, obstacles, and goal
    def load_images(self):
        rectangle = PhotoImage(Image.open("../img/rectangle.png").resize((65, 65)))  # Agent image
        triangle = PhotoImage(Image.open("../img/triangle.png").resize((65, 65)))    # Obstacle image
        circle = PhotoImage(Image.open("../img/circle.png").resize((65, 65)))        # Goal image

        return rectangle, triangle, circle

    # Adds text to a cell to display Q-values for each action in the cell
    def text_value(self, row, col, contents, action, font='Helvetica', size=10, style='normal', anchor="nw"):
        # Define coordinates based on action direction to position text
        if action == 0:            # Up
            origin_x, origin_y = 7, 42
        elif action == 1:          # Down
            origin_x, origin_y = 85, 42
        elif action == 2:          # Left
            origin_x, origin_y = 42, 5
        else:                      # Right
            origin_x, origin_y = 42, 77

        # Calculate final coordinates in the cell for text placement
        x, y = origin_y + (UNIT * col), origin_x + (UNIT * row)
        # Set text font, size, and style
        font = (font, str(size), style)
        # Add text
        text = self.canvas.create_text(x, y, fill="black", text=contents, font=font, anchor=anchor)
        # Add text element to list for later deletion
        return self.texts.append(text)

## Displaying Q- values

    # Display all Q-values in the grid by calling text_value on each cell and action
    def print_value_all(self, q_table):
        # Remove any existing text on the canvas
        for i in self.texts:
            self.canvas.delete(i)
        # Clear the list of text elements
        self.texts.clear()

        # Loop over each cell and action to display the Q-value if it exists in the Q-table
        for x in range(HEIGHT):
            for y in range(WIDTH):
                # Four possible actions
                for action in range(4):
                    state = [x, y]
                    if str(state) in q_table.keys():
                        temp = q_table[str(state)][action]
                        self.text_value(y, x, round(temp, 2), action)

    # Converts canvas coordinates to grid cell state
    def coords_to_state(self, coords):
        # Convert x-coordinate to cell index
```

```
        x = int((coords[0] - 50) / 100)
        # Convert y-coordinate to cell index
        y = int((coords[1] - 50) / 100)
        return [x, y]

## Resetting environment

    # Resets the environment to the initial state
    def reset(self):
        # Update tkinter window
        self.update()
        # Pause briefly to visualize reset

        time.sleep(0.5)
        # Get current agent position
        x, y = self.canvas.coords(self.rectangle)
        # Move agent back to starting position
        self.canvas.move(self.rectangle, UNIT / 2 - x, UNIT / 2 - y)
        # Render the updated position
        self.render()
        # Return initial state
        return self.coords_to_state(self.canvas.coords(self.rectangle))

## Agent step execution

    # Takes an action and updates the environment
    def step(self, action):
        # Get current agent position
        state = self.canvas.coords(self.rectangle)
        # Initialize movement
        base_action = np.array([0, 0])
        # Render environment
        self.render()

        # Define action-based movement
        if action == 0:                         # Up
            if state[1] > UNIT:
                base_action[1] -= UNIT
        elif action == 1:                       # Down
            if state[1] < (HEIGHT - 1) * UNIT:
                base_action[1] += UNIT
        elif action == 2:                       # Left
            if state[0] > UNIT:
                base_action[0] -= UNIT
        elif action == 3:                       # Right
            if state[0] < (WIDTH - 1) * UNIT:
                base_action[0] += UNIT

        # Move the agent based on the chosen action
        self.canvas.move(self.rectangle, base_action[0], base_action[1])
        # Keep agent above other objects on canvas
        self.canvas.tag_raise(self.rectangle)
```

```
    # Get new position
    next_state = self.canvas.coords(self.rectangle)

    # Define rewards based on the agent's new position
    if next_state == self.canvas.coords(self.circle):   # Goal reached
        reward = 100
        done = True
    # Hit obstacle
    elif next_state in [self.canvas.coords(self.triangle1), self.canvas.coords(self.triangle2)]:
        reward = -100
        done = True
    # Regular move
    else:
        reward = 0
        done = False

    # Convert coordinates to grid state and return
    next_state = self.coords_to_state(next_state)
    return next_state, reward, done

## Rendering environment

  # Render the canvas and introduce a delay for simulation speed
  def render(self):
    time.sleep(0.03)
    self.update()
```

References:
Code: [GitHub - rlcode/reinforcement-learning: Minimal and Clean Reinforcement Learning Examples]