CSE 584 Machine Learning: Tools and Techniques
Homework - 2

Submitted By: Naga Sri Hita Veleti (nkv5154)

Understanding Policy Iteration in Grid World Problem

I have taken a piece of code from GitHub for the problem *grid-world* and the *policy-iteration* algorithm.

Grid-World

The Grid World environment consists of a simple grid or matrix of cells to represent a physical, constrained environment in which some sort of agent—such as a robot or another AI model—moves about in space to achieve a goal by reaching a certain cell. Each cell can be considered a state, and normally an agent would be given choices that typically include moving up, down, left, or right. Cells can introduce rewards or penalties, and there could be obstacles or boundaries. This is a common setup in the study of reinforcement learning—the concepts of RL in this bounded and easily visualizable phase space.

Policy Iteration

Policy iteration is a major RL algorithm used in the grid world and similar environments to find the optimal policy—that is, the most efficient set of actions to reach maximum rewards. The algorithm works by cycling between two steps:

- **Policy Evaluation**: For a given policy, this step calculates the expected cumulative reward (or value) for each state by repeatedly simulating the policy.
- **Policy Improvement**: Based on the evaluated values, this step updates the policy by selecting actions that maximize the expected reward for each state.

Both of these steps repeat until convergence, meaning that the policy no longer changes and is, as a result, optimal for the environment. Policy iteration is very effective in problems like Grid World, where the agent has to find shortest or safest paths to reach its goals.

Abstract

This codebase solves a grid world navigation problem by using a reinforcement learning solution with the policy iteration algorithm, which is a dynamic programming method commonly used in RL for finding optimal solutions to MDPs. In this 5x5 grid world, each cell has an associated reward or penalty that will inform the movements of the agent and allow the learning of an optimal policy—the best action at each state with the goal of maximizing long-term cumulative rewards. This algorithm works iteratively, with two major phases: first, policy evaluation, where the agent computes the expected reward for each state; then policy improvement, during which time the agent updates its actions to achieve maximum return.

Core-Section Step-By-Step Understanding

1. Policy Evaluation

The `policy_evaluation` function iteratively updates the value of each state based on the current policy, applying the Bellman Expectation Equation to calculate expected rewards for following the policy in each state.

```
def policy_evaluation(self):
    # Initialize a new table to store updated values for each state
    next_value_table = [[0.00] * self.env.width for _ in range(self.env.height)]

    # Iterate over all states in the grid world
    for state in self.env.get_all_states():
        value = 0.0  # Initialize the cumulative value for this state

        # Keep the value function of terminal states as 0 (no future rewards
            expected)
        if state == [2, 2]:  # Assuming [2, 2] is the terminal state
            next_value_table[state[0]][state[1]] = value
            continue  # Skip further calculations for terminal state

        # Loop through each action available in the environment
        for action in self.env.possible_actions:
            # Get the next state and reward after taking the action from the
                current state
            next_state = self.env.state_after_action(state, action)
            reward = self.env.get_reward(state, action)
            next_value = self.get_value(next_state)  # Get the value of the next
                state

            # Update the value using the probability of action and Bellman equation
            # Formula: value += P(a|s) * [R(s, a) +    * V(s')]
            value += (self.get_policy(state)[action] *
                      (reward + self.discount_factor * next_value))

        # Store the computed value for this state in the new value table
        next_value_table[state[0]][state[1]] = round(value, 2)

    # Update the current value table with the computed values from this iteration
    self.value_table = next_value_table
```

A new table is initialized to store updated values for each state, where, for each non-terminal state, the expected cumulative value is calculated by iterating over possible actions. Each action's contribution is weighted according to the policy probability and depends on the immediate reward and the discounted value of the future state. This process updates the value for each state in the `value_table`.

2. Policy Improvement

The `policy_improvement` function updates the policy by selecting the best action for each state. This step maximizes expected rewards for each state, given the most recent value estimates.

```
def policy_improvement(self):
    # Initialize the next policy as the current policy table
    next_policy = self.policy_table

    # Iterate over all states in the grid world
    for state in self.env.get_all_states():
        # Skip the terminal state since no actions are needed there
        if state == [2, 2]:
            continue

        value = -99999  # Start with a very low initial value to compare actions
        max_index = []  # List to hold indices of actions with the max value
        result = [0.0, 0.0, 0.0, 0.0]  # Initialize policy probabilities for
            actions

        # For each action, calculate expected reward + discounted future value
        for index, action in enumerate(self.env.possible_actions):
            # Determine the next state and reward for taking this action
            next_state = self.env.state_after_action(state, action)
            reward = self.env.get_reward(state, action)
```

```
                next_value = self.get_value(next_state)  # Get the estimated value of
                    the next state

                # Calculate the action value (reward + discounted future value)
                temp = reward + self.discount_factor * next_value

                # If this actions value is equal to the max, add it to max_index (
                    ties allowed)
                if temp == value:
                    max_index.append(index)
                # If this actions value is greater, update max and reset max_index
                elif temp > value:
                    value = temp
                    max_index.clear()  # Clear previous max actions
                    max_index.append(index)

            # Set the probability for the best actions found (evenly if theres a tie
                )
            prob = 1 / len(max_index)
            for index in max_index:
                result[index] = prob

            # Update the policy at the current state to the calculated result
            next_policy[state[0]][state[1]] = result

        # Update the current policy with the newly computed policy
        self.policy_table = next_policy
```

The process evaluates each state's possible actions to identify those that yield the highest value. For each state, actions with the maximum expected value are selected, allowing for ties by distributing the probability evenly among the top actions. The policy is then updated to prioritize actions that maximize expected returns.

Environment - `environment.py`

```
import tkinter as tk  # Import the Tkinter library for GUI components
from tkinter import Button  # Import the Button widget from Tkinter
import time  # Import the time module for delays
import numpy as np  # Import numpy for numerical operations
from PIL import ImageTk, Image  # Import Pillow for image handling in Tkinter

PhotoImage = ImageTk.PhotoImage  # Alias for Tkinter PhotoImage using Pillow
UNIT = 100  # Pixel size of each grid cell
HEIGHT = 5  # Number of rows in the grid
WIDTH = 5  # Number of columns in the grid
TRANSITION_PROB = 1  # Probability for state transition
POSSIBLE_ACTIONS = [0, 1, 2, 3]  # Define actions: up, down, left, right
ACTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Coordinate changes for each action
REWARDS = []  # Initialize an empty list for rewards

# Class for displaying the environment and agent's actions
class GraphicDisplay(tk.Tk):
    def __init__(self, agent):
        super(GraphicDisplay, self).__init__()  # Initialize the Tkinter window
        self.title('Policy Iteration')  # Set the window title
        self.geometry(f'{HEIGHT * UNIT}x{HEIGHT * UNIT + 50}')  # Set window size
        self.texts = []  # List to store text elements
        self.arrows = []  # List to store arrow elements
        self.env = Env()  # Initialize the environment
        self.agent = agent  # Store the agent object
        self.evaluation_count = 0  # Counter for evaluations
        self.improvement_count = 0  # Counter for improvements
        self.is_moving = 0  # Movement flag for the agent
```

```
        (self.up, self.down, self.left, self.right), self.shapes = self.load_images
            ()  # Load images
        self.canvas = self._build_canvas()  # Build and assign the canvas
        self.text_reward(2, 2, "R␣:␣1.0")  # Set a reward of 1.0 at (2,2)
        self.text_reward(1, 2, "R␣:␣-1.0")  # Set a reward of -1.0 at (1,2)
        self.text_reward(2, 1, "R␣:␣-1.0")  # Set a reward of -1.0 at (2,1)

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

        # Add buttons for actions: Evaluate, Improve, Move, and Reset
        iteration_button = Button(self, text="Evaluate", command=self.
            evaluate_policy)
        iteration_button.configure(width=10, activebackground="#33B5E5")
        canvas.create_window(WIDTH * UNIT * 0.13, HEIGHT * UNIT + 10, window=
            iteration_button)

        policy_button = Button(self, text="Improve", command=self.improve_policy)
        policy_button.configure(width=10, activebackground="#33B5E5")
        canvas.create_window(WIDTH * UNIT * 0.37, HEIGHT * UNIT + 10, window=
            policy_button)

        policy_button = Button(self, text="move", command=self.move_by_policy)
        policy_button.configure(width=10, activebackground="#33B5E5")
        canvas.create_window(WIDTH * UNIT * 0.62, HEIGHT * UNIT + 10, window=
            policy_button)

        policy_button = Button(self, text="reset", command=self.reset)
        policy_button.configure(width=10, activebackground="#33B5E5")
        canvas.create_window(WIDTH * UNIT * 0.87, HEIGHT * UNIT + 10, window=
            policy_button)

        # Create grid lines for the canvas
        for col in range(0, WIDTH * UNIT, UNIT):  # Loop to create vertical lines
            x0, y0, x1, y1 = col, 0, col, HEIGHT * UNIT
            canvas.create_line(x0, y0, x1, y1)  # Draw vertical line

        for row in range(0, HEIGHT * UNIT, UNIT):  # Loop to create horizontal
            lines
            x0, y0, x1, y1 = 0, row, HEIGHT * UNIT, row
            canvas.create_line(x0, y0, x1, y1)  # Draw horizontal line

        # Place images on the canvas
        self.rectangle = canvas.create_image(50, 50, image=self.shapes[0])  # Place
             rectangle image
        canvas.create_image(250, 150, image=self.shapes[1])  # Place triangle image
        canvas.create_image(150, 250, image=self.shapes[1])  # Place another
            triangle image
        canvas.create_image(250, 250, image=self.shapes[2])  # Place circle image

        # Finalize canvas setup
        canvas.pack()

        return canvas  # Return the created canvas

    # Load images for agent movements and objects in the grid
    def load_images(self):
        up = PhotoImage(Image.open("../img/up.png").resize((13, 13)))  # Load 'up'
            arrow image
        right = PhotoImage(Image.open("../img/right.png").resize((13, 13)))  # Load
             'right' arrow image
```

```
        left = PhotoImage(Image.open("../img/left.png").resize((13, 13)))  # Load 'left' arrow image
        down = PhotoImage(Image.open("../img/down.png").resize((13, 13)))  # Load 'down' arrow image
        rectangle = PhotoImage(Image.open("../img/rectangle.png").resize((65, 65)))  # Load rectangle image
        triangle = PhotoImage(Image.open("../img/triangle.png").resize((65, 65)))  # Load triangle image
        circle = PhotoImage(Image.open("../img/circle.png").resize((65, 65)))  # Load circle image
        return (up, down, left, right), (rectangle, triangle, circle)  # Return loaded images

    # Reset the environment to the initial state
    def reset(self):
        if self.is_moving == 0:
            self.evaluation_count = 0  # Reset evaluation count
            self.improvement_count = 0  # Reset improvement count
            for i in self.texts:  # Remove all text elements
                self.canvas.delete(i)
            for i in self.arrows:  # Remove all arrow elements
                self.canvas.delete(i)
            # Reset agent's value and policy tables
            self.agent.value_table = [[0.0] * WIDTH for _ in range(HEIGHT)]
            self.agent.policy_table = ([[[0.25, 0.25, 0.25, 0.25]] * WIDTH for _ in range(HEIGHT)])
            self.agent.policy_table[2][2] = []  # Clear policy at the reward location
            # Move rectangle to initial position
            x, y = self.canvas.coords(self.rectangle)
            self.canvas.move(self.rectangle, UNIT / 2 - x, UNIT / 2 - y)

    # Display value for a specific grid cell
    def text_value(self, row, col, contents, font='Helvetica', size=10, style='normal', anchor="nw"):
        origin_x, origin_y = 85, 70  # Base coordinates for text
        x, y = origin_y + (UNIT * col), origin_x + (UNIT * row)  # Adjusted coordinates
        font = (font, str(size), style)  # Font properties
        text = self.canvas.create_text(x, y, fill="black", text=contents, font=font, anchor=anchor)  # Create text
        return self.texts.append(text)  # Add text to list

    # Display reward for a specific grid cell
    def text_reward(self, row, col, contents, font='Helvetica', size=10, style='normal', anchor="nw"):
        origin_x, origin_y = 5, 5  # Base coordinates for reward text
        x, y = origin_y + (UNIT * col), origin_x + (UNIT * row)  # Adjusted coordinates
        font = (font, str(size), style)  # Font properties
        text = self.canvas.create_text(x, y, fill="black", text=contents, font=font, anchor=anchor)  # Create reward text
        return self.texts.append(text)  # Add text to list

    # Move the rectangle based on action
    def rectangle_move(self, action):
        base_action = np.array([0, 0])  # Starting action is no movement
        location = self.find_rectangle()  # Get rectangle's location
        self.render()  # Update canvas
        if action == 0 and location[0] > 0:  # Up action if within bounds
            base_action[1] -= UNIT
        elif action == 1 and location[0] < HEIGHT - 1:  # Down action if within bounds
```

```
            base_action[1] += UNIT
        elif action == 2 and location[1] > 0:  # Left action if within bounds
            base_action[0] -= UNIT
        elif action == 3 and location[1] < WIDTH - 1:  # Right action if within
            bounds
            base_action[0] += UNIT
        # Move rectangle by the base action
        self.canvas.move(self.rectangle, base_action[0], base_action[1])

    # Find the current position of the rectangle in the grid
    def find_rectangle(self):
        temp = self.canvas.coords(self.rectangle)  # Get rectangle coordinates
        x = (temp[0] / 100) - 0.5  # Adjust x-coordinate to grid index
        y = (temp[1] / 100) - 0.5  # Adjust y-coordinate to grid index
        return int(y), int(x)  # Return as grid indices

    # Move the rectangle according to the policy
    def move_by_policy(self):
        if self.improvement_count != 0 and self.is_moving != 1:
            self.is_moving = 1  # Mark as moving
            x, y = self.canvas.coords(self.rectangle)
            self.canvas.move(self.rectangle, UNIT / 2 - x, UNIT / 2 - y)  # Reset
                to center
            x, y = self.find_rectangle()  # Get position
            while len(self.agent.policy_table[x][y]) != 0:
                self.after(100, self.rectangle_move(self.agent.get_action([x, y])))
                      # Move by policy action
                x, y = self.find_rectangle()  # Update position
            self.is_moving = 0  # Mark as stopped moving

    # Draw arrow based on policy probabilities for each direction
    def draw_one_arrow(self, col, row, policy):
        if col == 2 and row == 2:  # Skip for reward cell
            return
        if policy[0] > 0:  # Draw up arrow if policy weight is positive
            origin_x, origin_y = 50 + (UNIT * row), 10 + (UNIT * col)
            self.arrows.append(self.canvas.create_image(origin_x, origin_y, image=
                self.up))
        if policy[1] > 0:  # Draw down arrow if policy weight is positive
            origin_x, origin_y = 50 + (UNIT * row), 90 + (UNIT * col)
            self.arrows.append(self.canvas.create_image(origin_x, origin_y, image=
                self.down))
        if policy[2] > 0:  # Draw left arrow if policy weight is positive
            origin_x, origin_y = 10 + (UNIT * row), 50 + (UNIT * col)
            self.arrows.append(self.canvas.create_image(origin_x, origin_y, image=
                self.left))
        if policy[3] > 0:  # Draw right arrow if policy weight is positive
            origin_x, origin_y = 90 + (UNIT * row), 50 + (UNIT * col)
            self.arrows.append(self.canvas.create_image(origin_x, origin_y, image=
                self.right))

    # Draw arrows for the entire grid based on the policy table
    def draw_from_policy(self, policy_table):
        for i in range(HEIGHT):
            for j in range(WIDTH):
                self.draw_one_arrow(i, j, policy_table[i][j])  # Draw arrow for
                    each cell

    # Print value table on the grid cells
    def print_value_table(self, value_table):
        for i in range(WIDTH):
            for j in range(HEIGHT):
                self.text_value(i, j, value_table[i][j])  # Display each value on
```

```
                    the grid

    # Render updates to the GUI
    def render(self):
        time.sleep(0.1)  # Brief delay for better visualization
        self.canvas.tag_raise(self.rectangle)  # Bring rectangle to the front
        self.update()  # Update the GUI display

    # Evaluate the current policy by displaying the value table
    def evaluate_policy(self):
        self.evaluation_count += 1  # Increase evaluation count
        for i in self.texts:  # Remove existing texts
            self.canvas.delete(i)
        self.agent.policy_evaluation()  # Evaluate agent's policy
        self.print_value_table(self.agent.value_table)  # Print value table

    # Improve the current policy by drawing arrows based on updated policies
    def improve_policy(self):
        self.improvement_count += 1  # Increase improvement count
        for i in self.arrows:  # Remove existing arrows
            self.canvas.delete(i)
        self.agent.policy_improvement()  # Improve agent's policy
        self.draw_from_policy(self.agent.policy_table)  # Draw new policy arrows

# Class to represent the environment for the agent
class Env:
    def __init__(self):
        self.transition_probability = TRANSITION_PROB  # Probability for
            transitions
        self.width = WIDTH  # Width of the grid
        self.height = HEIGHT  # Height of the grid
        self.reward = [[0] * WIDTH for _ in range(HEIGHT)]  # Initialize rewards
            with 0
        self.possible_actions = POSSIBLE_ACTIONS  # List of possible actions
        self.reward[2][2] = 1  # Reward of 1 at cell (2,2)
        self.reward[1][2] = -1  # Reward of -1 at cell (1,2)
        self.reward[2][1] = -1  # Reward of -1 at cell (2,1)
        self.all_state = []  # Initialize list of all states
        # Create all grid states
        for x in range(WIDTH):
            for y in range(HEIGHT):
                state = [x, y]  # Define a state as a coordinate pair
                self.all_state.append(state)  # Add state to list

    # Get reward for taking a specific action in a given state
    def get_reward(self, state, action):
        next_state = self.state_after_action(state, action)  # Get resulting state
        return self.reward[next_state[0]][next_state[1]]  # Return the reward at
            that state

    # Determine resulting state after taking an action
    def state_after_action(self, state, action_index):
        action = ACTIONS[action_index]  # Map action index to action
        return self.check_boundary([state[0] + action[0], state[1] + action[1]])  #
             Check boundaries

    # Ensure state remains within grid boundaries
    @staticmethod
    def check_boundary(state):
        state[0] = (0 if state[0] < 0 else WIDTH - 1 if state[0] > WIDTH - 1 else
            state[0])  # Bound x
        state[1] = (0 if state[1] < 0 else HEIGHT - 1 if state[1] > HEIGHT - 1 else
             state[1])  # Bound y
```

```
        return state  # Return bounded state

    # Get probability of transitioning from a state given an action
    def get_transition_prob(self, state, action):
        return self.transition_probability  # Always return set transition
            probability

    # Return all possible states in the grid
    def get_all_states(self):
        return self.all_state
```

Agent - **policy_iteration.py**

```
import random  # Import the random module for random selections
from environment import GraphicDisplay, Env  # Import environment classes for
    display and environment setup

# Class for Policy Iteration to find optimal policy and value function
class PolicyIteration:
    def __init__(self, env):
        self.env = env  # Store the environment instance
        # Initialize a 2D list to store the value function, initially zero for all
            states
        self.value_table = [[0.0] * env.width for _ in range(env.height)]
        # Initialize the policy table with equal probability for all actions in
            each state
        self.policy_table = [[[0.25, 0.25, 0.25, 0.25]] * env.width for _ in range(
            env.height)]
        # Set the terminal state (goal state) with no policy
        self.policy_table[2][2] = []
        # Discount factor for future rewards
        self.discount_factor = 0.9

    # Evaluate the policy by updating the value table based on current policy
    def policy_evaluation(self):
        # Create a new value table to store updated values
        next_value_table = [[0.00] * self.env.width for _ in range(self.env.height)
            ]

        # Apply Bellman's Expectation Equation to update each state's value
        for state in self.env.get_all_states():
            value = 0.0  # Initialize value for the state

            # Keep the value of terminal state as 0
            if state == [2, 2]:
                next_value_table[state[0]][state[1]] = value
                continue  # Skip to next state

            # Calculate the expected value for each action
            for action in self.env.possible_actions:
                next_state = self.env.state_after_action(state, action)  # Get the
                    next state
                reward = self.env.get_reward(state, action)  # Get the reward for
                    the action
                next_value = self.get_value(next_state)  # Get the value of the
                    next state
                # Calculate the expected value using the policy, reward, and
                    discounted next value
                value += (self.get_policy(state)[action] * (reward + self.
                    discount_factor * next_value))

            # Round the value to two decimal places and update it in the value
```

```
            table
            next_value_table[state[0]][state[1]] = round(value, 2)

        # Update the value table with the new values
        self.value_table = next_value_table

    # Improve the policy based on the updated value table
    def policy_improvement(self):
        next_policy = self.policy_table  # Initialize next policy table as current
            policy

        # Loop through each state to update its policy
        for state in self.env.get_all_states():
            if state == [2, 2]:  # Skip the terminal state
                continue

            value = -99999  # Set an initial low value for comparison
            max_index = []  # List to store indices of actions with max value
            result = [0.0, 0.0, 0.0, 0.0]  # Initialize new policy for this state

            # Calculate value for each possible action
            for index, action in enumerate(self.env.possible_actions):
                next_state = self.env.state_after_action(state, action)  # Get the
                    next state for action
                reward = self.env.get_reward(state, action)  # Get the reward for
                    the action
                next_value = self.get_value(next_state)  # Get the value of the
                    next state
                temp = reward + self.discount_factor * next_value  # Calculate the
                    action's expected value

                # Allow multiple actions with the same maximum value
                if temp == value:
                    max_index.append(index)  # Add index to max_index list
                elif temp > value:  # If a new maximum is found
                    value = temp  # Update max value
                    max_index.clear()  # Clear previous max indices
                    max_index.append(index)  # Add new max index

            # Calculate probability for each action with the maximum value
            prob = 1 / len(max_index)  # Probability if multiple actions have max
                value

            # Assign calculated probability to each action with max value
            for index in max_index:
                result[index] = prob

            # Update the policy table for the state with the new policy
            next_policy[state[0]][state[1]] = result

        # Replace the old policy table with the updated one
        self.policy_table = next_policy

    # Get an action based on the policy for a given state
    def get_action(self, state):
        random_pick = random.randrange(100) / 100  # Random number between 0 and 1
        policy = self.get_policy(state)  # Retrieve policy for the state
        policy_sum = 0.0  # Initialize cumulative probability

        # Return the action based on cumulative probability
        for index, value in enumerate(policy):
            policy_sum += value  # Add policy value
            if random_pick < policy_sum:  # Check if random number is within
```

```
                cumulative range
                return index  # Return the corresponding action index

    # Get the policy of a specific state
    def get_policy(self, state):
        if state == [2, 2]:  # Return 0 for terminal state as it has no actions
            return 0.0
        return self.policy_table[state[0]][state[1]]  # Return the policy for the
            given state

    # Get the value of a specific state, rounded to two decimal places
    def get_value(self, state):
        return round(self.value_table[state[0]][state[1]], 2)

# Main block to run policy iteration with GUI display
if __name__ == "__main__":
    env = Env()  # Create the environment
    policy_iteration = PolicyIteration(env)  # Create the PolicyIteration instance
    grid_world = GraphicDisplay(policy_iteration)  # Create the display instance
    grid_world.mainloop()  # Run the GUI loop to display
```