Machine Learning - Homework 3

Tintrim Dwi Ary Widhianingsih

December 5, 2018

Problem 3a. Running the policy iteration code of Reinforcement Learning in the following URL: `https://github.com/rlcode/reinforcement-learning/tree/master/1-grid-world/1-policy-iteration`.

1. Program analyzation

   - `environment.py`

     This code is basically built for the graphic display of the Grid World Game, see Fig.1. In this module, there is a code for making the grid, which is 5×5 grids in the original code. There is also code for making 4 buttons (evaluate, improve, move, and reset) for showing the main action in the game. The detail information about each function inside this module, including the two function mentioned before and some other functions, will be explain in this section.

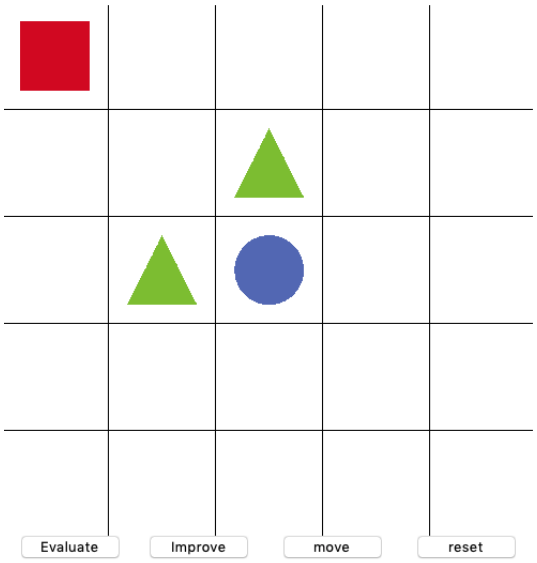


Figure 1: Interface of Grid World Game

The modules that is needed to built the environment here is `tkinter`, for making a GUI (Graphical User Interfaces) in Python, `time`, `numpy`, and `PIL`. Line 7 in

Listing 1 defines the function to open the image. This variable will be used in `load_images()` function. `UNIT`, `HEIGHT`, and `WIDTH` in Line 7–8 assigns the number of pixels in each grid, grid height, and grid width. Line 12 defines the possible encoded direction, which are left (0), right (1), up (2), and down (3). The actions in coordinates is expressed by `ACTIONS` in Line 13. In the `ACTIONS` input, there are two values in each direction. The first value is denoted the left-right movement and the second value is denoted the up-down movement. So, $(-1, 0)$ is left, $(1, 0)$ is right, $(0, -1)$ is up, and $(0, 1)$ is down.

```
1  import tkinter as tk
2  from tkinter import Button
3  import time
4  import numpy as np
5  from PIL import ImageTk, Image
6
7  PhotoImage = ImageTk.PhotoImage
8  UNIT = 100
9  HEIGHT = 5
10 WIDTH = 5
11 TRANSITION_PROB = 1
12 POSSIBLE_ACTIONS = [0, 1, 2, 3]
13 ACTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]
14 REWARDS = []
```

Listing 1: Environment – Import Modules and Assign Fix Variables

Listing 2 is the function for initializing the input for the other function in this module. Line 14 until 16 give the initial reward, that for grid $(2, 2)$, which is our goal, the initial reward is 1.0, and for both grid $(2, 1)$ and $(1, 2)$, which are the obstacles that we have to avoid, the initial reward is -1.0.

```
1  class GraphicDisplay(tk.Tk):
2      def __init__(self, agent):
3          super(GraphicDisplay, self).__init__()
4          self.title('Policy Iteration')
5          self.geometry('{0}x{1}'.format(HEIGHT * UNIT, HEIGHT * UNIT +
               50))
6          self.texts = []
7          self.arrows = []
8          self.env = Env()
9          self.agent = agent
10         self.evaluation_count = 0
11         self.improvement_count = 0
12         self.is_moving = 0
13         (self.up, self.down, self.left, self.right), self.shapes = self
               .load_images()
14         self.canvas = self._build_canvas()
15         self.text_reward(2, 2, "R : 1.0")
16         self.text_reward(1, 2, "R : -1.0")
17         self.text_reward(2, 1, "R : -1.0")
```

Listing 2: Environment – Initialization

For building the canvas, we use the function appeared in Listing 3. The canvas is set to be 500×500 of the size and white colored background. This setting is

defined in Line 2 until 4, which the `HEIGHT`, `WIDTH`, and `UNIT` has been defined in Listing 1. The are 3 buttons inside the GUI: evaluate, improve, move, and reset. The evaluate button is to evaluate the policy. The function to activate this button is `evaluate_policy()`. The improve button is to improve the policy. Inside this button, the function that is used is `improve_policy()`. The move button is to move the agent, in this game we use the rectangle, to move based on the policy that that has been calculated. This button is activated by `move_by_policy()`. The last is reset button. This button is for reseting the action of the agent. It is activated by `reset()`. All of the function to activate the button will be loaded in `policy_iteration.py`.

```
1  def _build_canvas(self):
2      canvas = tk.Canvas(self, bg='white',
3                         height=HEIGHT * UNIT,
4                         width=WIDTH * UNIT)
5      iteration_button = Button(self, text="Evaluate",
6                                command=self.evaluate_policy)
7      iteration_button.configure(width=10, activebackground="#33B5E5")
8      canvas.create_window(WIDTH * UNIT * 0.13, HEIGHT * UNIT + 10,
9                           window=iteration_button)
10     policy_button = Button(self, text="Improve",
11                            command=self.improve_policy)
12     policy_button.configure(width=10, activebackground="#33B5E5")
13     canvas.create_window(WIDTH * UNIT * 0.37, HEIGHT * UNIT + 10,
14                          window=policy_button)
15     policy_button = Button(self, text="move", command=self.
           move_by_policy)
16     policy_button.configure(width=10, activebackground="#33B5E5")
17     canvas.create_window(WIDTH * UNIT * 0.62, HEIGHT * UNIT + 10,
18                          window=policy_button)
19     policy_button = Button(self, text="reset", command=self.reset)
20     policy_button.configure(width=10, activebackground="#33B5E5")
21     canvas.create_window(WIDTH * UNIT * 0.87, HEIGHT * UNIT + 10,
22                          window=policy_button)
23     for col in range(0, WIDTH * UNIT, UNIT):
24         x0, y0, x1, y1 = col, 0, col, HEIGHT * UNIT
25         canvas.create_line(x0, y0, x1, y1)
26     for row in range(0, HEIGHT * UNIT, UNIT):
27         x0, y0, x1, y1 = 0, row, HEIGHT * UNIT, row
28         canvas.create_line(x0, y0, x1, y1)
29     self.rectangle = canvas.create_image(50, 50, image=self.shapes[0])
30     canvas.create_image(250, 150, image=self.shapes[1])
31     canvas.create_image(150, 250, image=self.shapes[1])
32     canvas.create_image(250, 250, image=self.shapes[2])
33
34     canvas.pack()
35
36     return canvas
```

Listing 3: Environment – Building the Canvas

The following code is used for importing the images: circle, rectangle, triangle, and the arrows for guiding the action of the agent. The images that will be

loaded using this function is shown in Fig.2.

```
def load_images(self):
    up = PhotoImage(Image.open("img/up.png").resize((13, 13)))
    right = PhotoImage(Image.open("img/right.png").resize((13, 13)))
    left = PhotoImage(Image.open("img/left.png").resize((13, 13)))
    down = PhotoImage(Image.open("img/down.png").resize((13, 13)))
    rectangle = PhotoImage(Image.open("img/rectangle.png").resize((65,
        65)))
    triangle = PhotoImage(Image.open("img/triangle.png").resize((65,
        65)))
    circle = PhotoImage(Image.open("img/circle.png").resize((65, 65)))
    return (up, down, left, right), (rectangle, triangle, circle)
```

Listing 4: Environment – Loading Images

In this image, we use the red rectangle image as the agent. We want to move the agent from $(0, 0)$ coordinate (as a default starting point) to the position where the circle image is located by avoiding the grid where triangles are located. For the sign images, these are used for guiding the movement of our agent.

(a) (b) (c) (d) (e) (f) (g)

Figure 2: Imported Images

The following function is used to activate reset button. In this function all previous movement and calculation of the agent are reseted to be in the onset position which is coordinate $(0, 0)$ and the evaluation and improvement count will be assign as 0. Then the arrows that appear on the canvas will be deleted.

```
def reset(self):
    if self.is_moving == 0:
        self.evaluation_count = 0
        self.improvement_count = 0

        for i in self.texts:
            self.canvas.delete(i)

        for i in self.arrows:
            self.canvas.delete(i)
        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] = []
        x, y = self.canvas.coords(self.rectangle)
        self.canvas.move(self.rectangle, UNIT / 2 - x, UNIT / 2 - y)
```

Listing 5: Environment – Reset

Listing 6 is to show the reward and updated value in the canvas.

```
def text_value(self, row, col, contents, font='Helvetica', size=10,
               style='normal', anchor="nw"):
    origin_x, origin_y = 85, 70
    x, y = origin_y + (UNIT * col), origin_x + (UNIT * row)
    font = (font, str(size), style)
    text = self.canvas.create_text(x, y, fill="black", text=contents,
                                   font=font, anchor=anchor)
    return self.texts.append(text)

def text_reward(self, row, col, contents, font='Helvetica', size=10,
                style='normal', anchor="nw"):
    origin_x, origin_y = 5, 5
    x, y = origin_y + (UNIT * col), origin_x + (UNIT * row)
    font = (font, str(size), style)
    text = self.canvas.create_text(x, y, fill="black", text=contents,
                                   font=font, anchor=anchor)
    return self.texts.append(text)
```

Listing 6: Environment – Text Values and Reward

The movement of the agent is managed by the function in Listing 7. Function `find_rectangle()` is used for assigning the position of the agent in realtime. The movement of our agent is always based on the coordinate obtained from `find_rectangle()`.

```
def rectangle_move(self, action):
    base_action = np.array([0, 0])
    location = self.find_rectangle()
    self.render()
    if action == 0 and location[0] > 0:
        base_action[1] -= UNIT
    elif action == 1 and location[0] < HEIGHT - 1:
        base_action[1] += UNIT
    elif action == 2 and location[1] > 0:
        base_action[0] -= UNIT
    elif action == 3 and location[1] < WIDTH - 1:
        base_action[0] += UNIT
    self.canvas.move(self.rectangle, base_action[0], base_action[1])

def find_rectangle(self):
    temp = self.canvas.coords(self.rectangle)
    x = (temp[0] / 100) - 0.5
    y = (temp[1] / 100) - 0.5
    return int(y), int(x)
```

Listing 7: Environment – Agent Movement

The movement of the rectangle is not solely random. To get the fastest and nearest path, its movement have to be guided by something. In this case, the movement of our agent is conducted by using policy. Line 11 in Listing **??** shows the movement of the agent based on the value that is got from `agent.get_action()`. This value basically is the action for the next movement according to the current policy. This function will be appear in `policy_iteration.py`.

```
def move_by_policy(self):
    if self.improvement_count != 0 and self.is_moving != 1:
        self.is_moving = 1

        x, y = self.canvas.coords(self.rectangle)
        self.canvas.move(self.rectangle, UNIT / 2 - x, UNIT / 2 - y)

        x, y = self.find_rectangle()
        while len(self.agent.policy_table[x][y]) != 0:
            self.after(100,
                       self.rectangle_move(self.agent.get_action([x, y
                           ])))
            x, y = self.find_rectangle()
        self.is_moving = 0
```

Listing 8: Environment – Movement by Policy

The next functions are to choose one of the four possible arrows for moving the agent. This part will be decided based on the policy value.

```
def draw_one_arrow(self, col, row, policy):
    if col == 2 and row == 2:
        return
    if policy[0] > 0:
        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:
        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:
        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:
        origin_x, origin_y = 90 + (UNIT * row), 50 + (UNIT * col)
        self.arrows.append(self.canvas.create_image(origin_x, origin_y,
                                                    image=self.right))

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])
```

Listing 9: Environment – Draw One Arrow

```
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])
def render(self):
    time.sleep(0.1)
    self.canvas.tag_raise(self.rectangle)
    self.update()
```

Listing 10: Environment – Print Values as a Table

The following function is for printing the values that have been got from `text_value` as a table with `HEIGHT`×`WIDTH` of the size. The `render` function is use for showing every movement of our agent.

Listing 11 is used for changing the values and and arrows. These functions are used to activate the evaluate and improve button respectively.

```
def evaluate_policy(self):
    self.evaluation_count += 1
    for i in self.texts:
        self.canvas.delete(i)
    self.agent.policy_evaluation()
    self.print_value_table(self.agent.value_table)

def improve_policy(self):
    self.improvement_count += 1
    for i in self.arrows:
        self.canvas.delete(i)
    self.agent.policy_improvement()
    self.draw_from_policy(self.agent.policy_table)
```

Listing 11: Environment – Loading Images

The next code is inside different class, which is `Env`. This class is basically used for defining the current state and reward. The first function in Listing 12 is the intialization part. In this part, reward of each grid is fixed, which are for grid where the triangles located, the reward is -1, for grid where the circle is in, the reward is 1, and the reward of other grids are 0.

```
class Env:
def __init__(self):
    self.transition_probability = TRANSITION_PROB
    self.width = WIDTH
    self.height = HEIGHT
    self.reward = [[0] * WIDTH for _ in range(HEIGHT)]
    self.possible_actions = POSSIBLE_ACTIONS
    self.reward[2][2] = 1
    self.reward[1][2] = -1
    self.reward[2][1] = -1
    self.all_state = []

    for x in range(WIDTH):
        for y in range(HEIGHT):
            state = [x, y]
            self.all_state.append(state)

def get_reward(self, state, action):
    next_state = self.state_after_action(state, action)
    return self.reward[next_state[0]][next_state[1]]

def state_after_action(self, state, action_index):
    action = ACTIONS[action_index]
    return self.check_boundary([state[0] + action[0], state[1] + action
        [1]])
```

Listing 12: Environment – Initialization of the Reward and Getting the State After Action

The `get_reward()` function is used for calculating the reward that agent get after doing an action. The calculation basically is did inside in `state_after_acion()`. Finally the Listing 13 is used for checking the current state after the agent do an action.

```
@staticmethod
def check_boundary(state):
    state[0] = (0 if state[0] < 0 else WIDTH - 1
                if state[0] > WIDTH - 1 else state[0])
    state[1] = (0 if state[1] < 0 else HEIGHT - 1
                if state[1] > HEIGHT - 1 else state[1])
    return state

def get_transition_prob(self, state, action):
    return self.transition_probability

def get_all_states(self):
    return self.all_state
```

Listing 13: Environment – Loading Images

- `policy_iteration.py`

  This module contains the process of policy improvement and evaluation. The following function defines the initialization for the input in the next function. Line 7 in Listing 14 denotes the initialization of the value in each grid, which is 0. Line 8 denotes the initialization of the policy value, which are got from the uniform distribution, 0.25 for all possible actions. Then the discount factor that is defined in this problem is 0.9.

```
import random
from environment import GraphicDisplay, Env

class PolicyIteration:
def __init__(self, env):
    self.env = env
    self.value_table = [[0.0] * env.width for _ in range(env.height)]
    self.policy_table = [[[0.25, 0.25, 0.25, 0.25]] * env.width
                                for _ in range(env.height)]
    self.policy_table[2][2] = []
    self.discount_factor = 0.9
```

Listing 14: Policy Iteration – Initialization

The following function is for policy evaluation. This code is basically built based on the Bellman Expectation Equation, that is expressed by Eq.1.

$$V_\pi(s) = \sum_{a \in A} \pi(a|s)(R_{t+1} + \gamma \sum_{s' \in S} P^a_{ss'} V_\pi(s')) \tag{1}$$

where $V_\pi(s)$ is the value function for the current state, $\pi(a|s)$ is the policy value for a action from the current state, R_{t+1} is the reward that agent get, γ is the discount factor, and the last summation denotes the next value function. The

calculation of this equation is denoted in Line 15–16 in Listing 15. The result of this calculation is appeared in the canvas after we click evaluate button.

```
1  def policy_evaluation(self):
2      next_value_table = [[0.00] * self.env.width
3                                  for _ in range(self.env.height)]
4
5      for state in self.env.get_all_states():
6          value = 0.0
7          if state == [2, 2]:
8              next_value_table[state[0]][state[1]] = value
9              continue
10
11         for action in self.env.possible_actions:
12             next_state = self.env.state_after_action(state, action)
13             reward = self.env.get_reward(state, action)
14             next_value = self.get_value(next_state)
15             value += (self.get_policy(state)[action] *
16                       (reward + self.discount_factor * next_value))
17
18         next_value_table[state[0]][state[1]] = round(value, 2)
19
20     self.value_table = next_value_table
```

Listing 15: Policy Iteration – Policy Evaluation

```
1  def policy_improvement(self):
2      next_policy = self.policy_table
3      for state in self.env.get_all_states():
4          if state == [2, 2]:
5              continue
6          value = -99999
7          max_index = []
8          result = [0.0, 0.0, 0.0, 0.0]
9          for index, action in enumerate(self.env.possible_actions):
10             next_state = self.env.state_after_action(state, action)
11             reward = self.env.get_reward(state, action)
12             next_value = self.get_value(next_state)
13             temp = reward + self.discount_factor * next_value
14             if temp == value:
15                 max_index.append(index)
16             elif temp > value:
17                 value = temp
18                 max_index.clear()
19                 max_index.append(index)
20         prob = 1 / len(max_index)
21         for index in max_index:
22             result[index] = prob
23         next_policy[state[0]][state[1]] = result
24     self.policy_table = next_policy
```

Listing 16: Policy Iteration – Policy Improvement

The next one is the function for the policy improvement. This function will be used inside improve button. The function is basically for getting the maximum policy value, so that when we click the improve button, the appear arrows will be

changed. The appear arrows has the maximum value of $R_{t+1}+\gamma \sum_{s' \in S} P^a_{ss'} V_\pi(s')$. The choosing maximum policy value is done in Line 14–19. Then after clicking improve button, the probability of each possible actions (left, right, up, and down) will also be updated. The update value is calculated based on the number of the possible actions that has maximum value of $R_{t+1}+\gamma \sum_{s' \in S} P^a_{ss'} V_\pi(s')$. This calculation is expressed in Line 20.

The next function is for deciding the next action of the agent.

```
1  def get_action(self, state):
2      random_pick = random.randrange(100) / 100
3
4      policy = self.get_policy(state)
5      policy_sum = 0.0
6      for index, value in enumerate(policy):
7          policy_sum += value
8          if random_pick < policy_sum:
9              return index
```

Listing 17: Policy Iteration – Get Action from The Current Policy

The next function is for getting the policy, to where the agent will go and also getting the value of the next state.

```
1  def get_policy(self, state):
2      if state == [2, 2]:
3          return 0.0
4      return self.policy_table[state[0]][state[1]]
5
6  def get_value(self, state):
7      return round(self.value_table[state[0]][state[1]], 2)
8
9  if __name__ == "__main__":
10     env = Env()
11     policy_iteration = PolicyIteration(env)
12     grid_world = GraphicDisplay(policy_iteration)
13     grid_world.mainloop()
```

Listing 18: Policy Iteration – Get Policy of the Spesific State

The figures below is the capture image when the code is executed. The first figure is the initial display when we execute the code. There is showed 1 rectangle in the top left side or coordinate $(0, 0)$, then a circle in coordinate $(2, 2)$, and two triangles as the obstacles in coordinate $(1, 2)$ and $(2, 1)$. The second figure is the figure when we click evaluate and improvement button once. Then, the third figure is the converge converge path that we have have got. In this figure, the optimal path has been shown, which are $(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 2)$ or $(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (2, 2)$. Using these path, our agent can easily reach the goal, which is in $(2, 2)$.

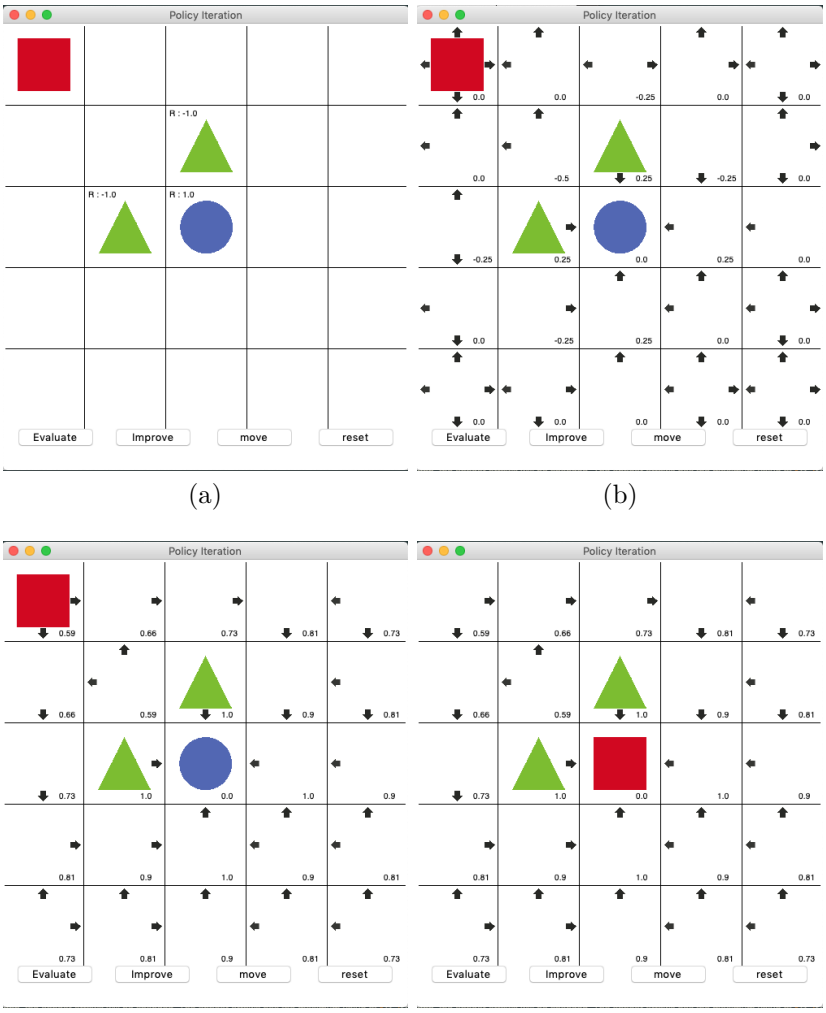


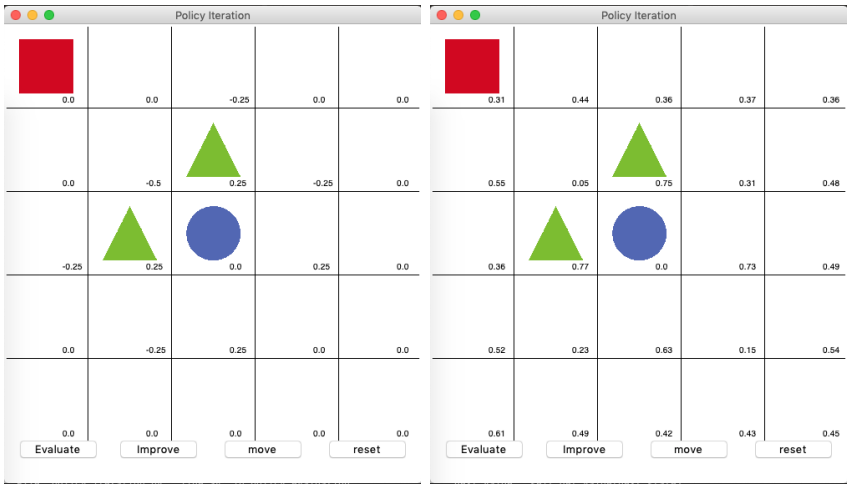
Figure 3: Code Execution

2. Changing the initial value of value function

   In this part, the initial value of value function is change to be random. The random value is drawn from uniform distribution with 0 and 1 as the minimum and maximum value.

```
def __init__(self, env):
    self.env = env
    self.value_table = np.random.uniform(0,1,25).reshape(5,5)
    self.policy_table = [[[0.25, 0.25, 0.25, 0.25]] * env.width
                         for _ in range(env.height)]
    self.policy_table[2][2] = []
    self.discount_factor = 0.9
```

Listing 19: Policy Iteration – Change the Initialization



(a) (b)

Figure 4: Code Execution – Initialization Scenario

In this case we need `numpy` module. From Listing 21, we can see the changing part is in Line 3. The execution example is shown in Fig.4. We can see that in the first click of evaluate button, the value function that are appeared in the canvas are different between the first and the second figure. First figure is using 0 in each grid, while the second one is using the random value.

The optimal value function we get from this scenario is shown in Fig.5. We can see that even if the initial value is changed to be random, the optimum value function we get is still same. The optimum path for reaching the goal also same with the original one.

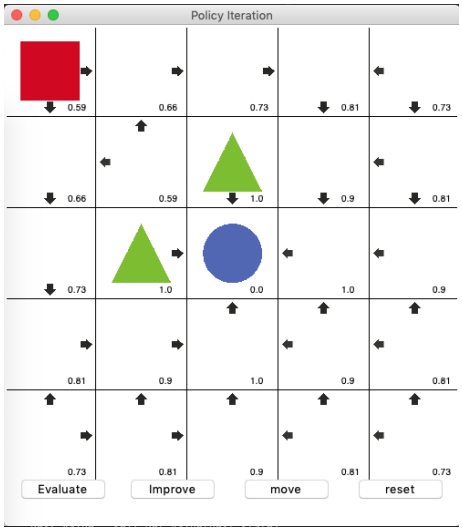


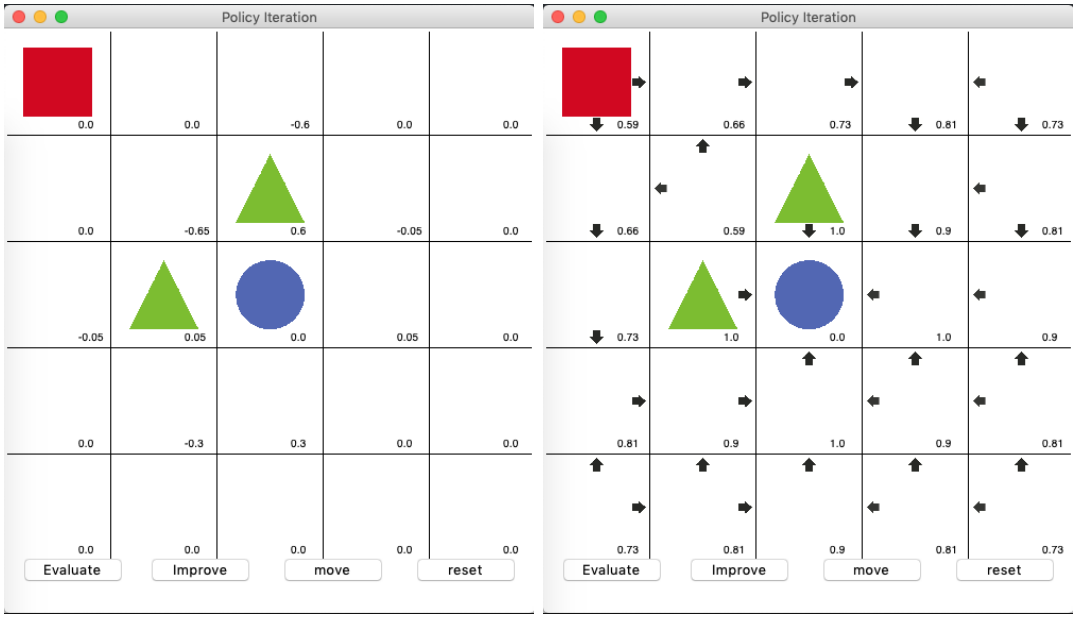
Figure 5: Optimal Value Function in Changing Initialization Scenario

3. Changing the initial value of policy

   In this part, the initial value of policy is change to be random probabilities. The random value we use in this part is 0.3, 0.6, 0.05, and 0.05.

```
1 def __init__(self, env):
2     self.env = env
3     self.value_table = [[0.0] * env.width for _ in range(env.height)]
4     self.policy_table = [[[0.3, 0.6, 0.05, 0.05]] * env.width
5                                 for _ in range(env.height)]
6     self.policy_table[2][2] = []
7     self.discount_factor = 0.9
```

Listing 20: Policy Iteration – Change the Policy Initialization



(a) (b)

Figure 6: Code Execution – Initialization Scenario

From Listing 20, we can see the changing part is in Line 4. The execution example is shown in Fig.6. We can see that in the first click of evaluate button, the value function that are appeared in the canvas are different with the original one. We also can see that even if the initial value is changed to be random, the optimum value function we get is still same. The optimum path for reaching the goal also same with the original one.

4. Adding one more triangle

   In this scenario, we add one more triangle. We locate the adding triangle in coordinate $(3, 2)$. In the code, we add more row in `_build_canvas` function (see Line 2–6) and `__init__`, see Line 9–12 and Line 15–18 in the following Listing.

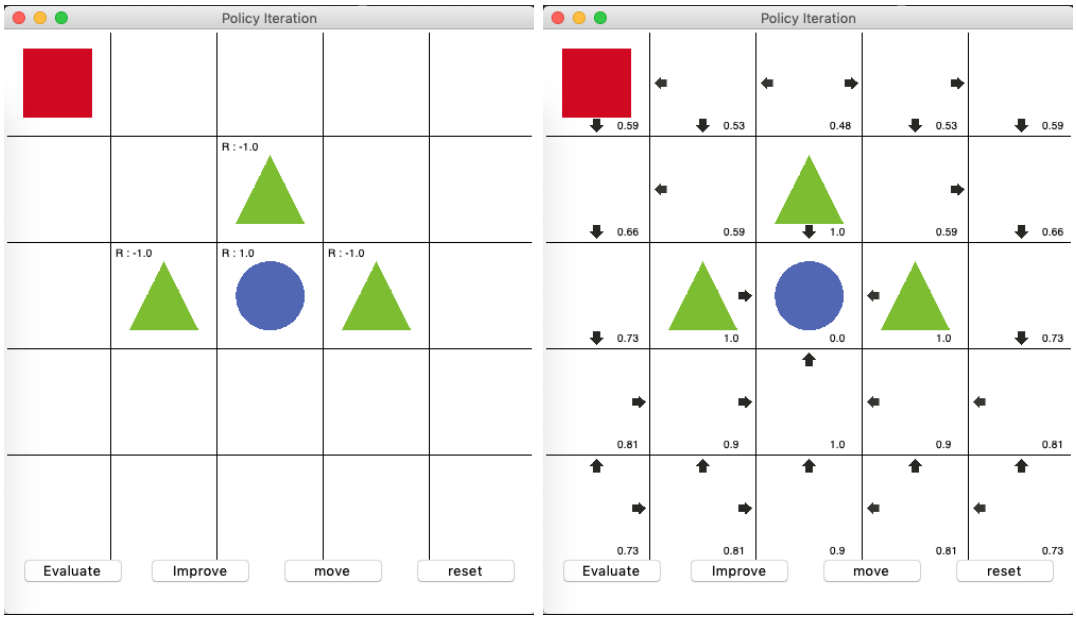
```
#The added row in _build_canvas function (GraphicDisplay module)
self.rectangle = canvas.create_image(50, 50, image=self.shapes[0])
canvas.create_image(250, 150, image=self.shapes[1])
canvas.create_image(150, 250, image=self.shapes[1])
canvas.create_image(350, 250, image=self.shapes[1])
canvas.create_image(250, 250, image=self.shapes[2])

#The added row in __init__ function (GraphicDisplay module)
self.text_reward(2, 2, "R : 1.0")
self.text_reward(1, 2, "R : -1.0")
self.text_reward(2, 1, "R : -1.0")
self.text_reward(2, 3, "R : -1.0")

#The added row in __init__ function (Env module)
self.reward[2][2] = 1  # reward 1 for circle
self.reward[1][2] = -1  # reward -1 for triangle
self.reward[2][1] = -1  # reward -1 for triangle
self.reward[2][3] = -1
```

Listing 21: Policy Iteration – Change the Initialization

The display of our grid world after adding one more triangle is shown in Fig.7. We can see in the right figure that the one optimum path is same with the original one. In this scenario, because we add the new triangle in coordinate $(3, 2)$, the optimum path is just through $(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 2)$.



(a) (b)

Figure 7: Code Execution – Adding Triangle Scenario

Problem 3b. Running the value iteration code of Reinforcement Learning in the following URL: `https://github.com/rlcode/reinforcement-learning/tree/master/1-grid-world/2-value-iteration`.

1. Program analyzation

   - `environment.py`

     The analyzation of this program is mostly same with the previous one.

   - `value_iteration.py`

     The program inside `value_iteration.py` is mostly same with `policy_iteration.py`, except the value iteration part. Listing 22 is the code for this section. The rule for choosing the next state in this method is simpler that in model that is based on the policy. The chosen policy in this method just based on the maximum value of $R_{t+1} + \gamma \sum_{s' \in S} P^a_{ss'} V_\pi(s')$, which expressed in Line 14.

```
1  def value_iteration(self):
2      next_value_table = [[0.0] * self.env.width
3                                 for _ in range(self.env.height)]
4      for state in self.env.get_all_states():
5          if state == [2, 2]:
6              next_value_table[state[0]][state[1]] = 0.0
7              continue
8          value_list = []
9
10         for action in self.env.possible_actions:
11             next_state = self.env.state_after_action(state, action)
12             reward = self.env.get_reward(state, action)
13             next_value = self.get_value(next_state)
14             value_list.append((reward + self.discount_factor *
                   next_value))
15         # return the maximum value(it is the optimality equation!!)
16         next_value_table[state[0]][state[1]] = round(max(value_list),
               2)
17     self.value_table = next_value_table
```

Listing 22: Policy Iteration – Change the Initialization

The figures below is the capture image when the code is executed. The first figure is the initial display when we execute the code. There is showed 1 rectangle in the top left side or coordinate $(0, 0)$, then a circle in coordinate $(2, 2)$, and two triangles as the obstacles in coordinate $(1, 2)$ and $(2, 1)$. The second figure is the figure when we click evaluate and improvement button once. Then, the third figure is the converge converge path that we have have got. In this figure, the optimal path has been shown, which are $(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 2)$ or $(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (2, 2)$. Using these path, our agent can easily reach the goal, which is in $(2, 2)$.

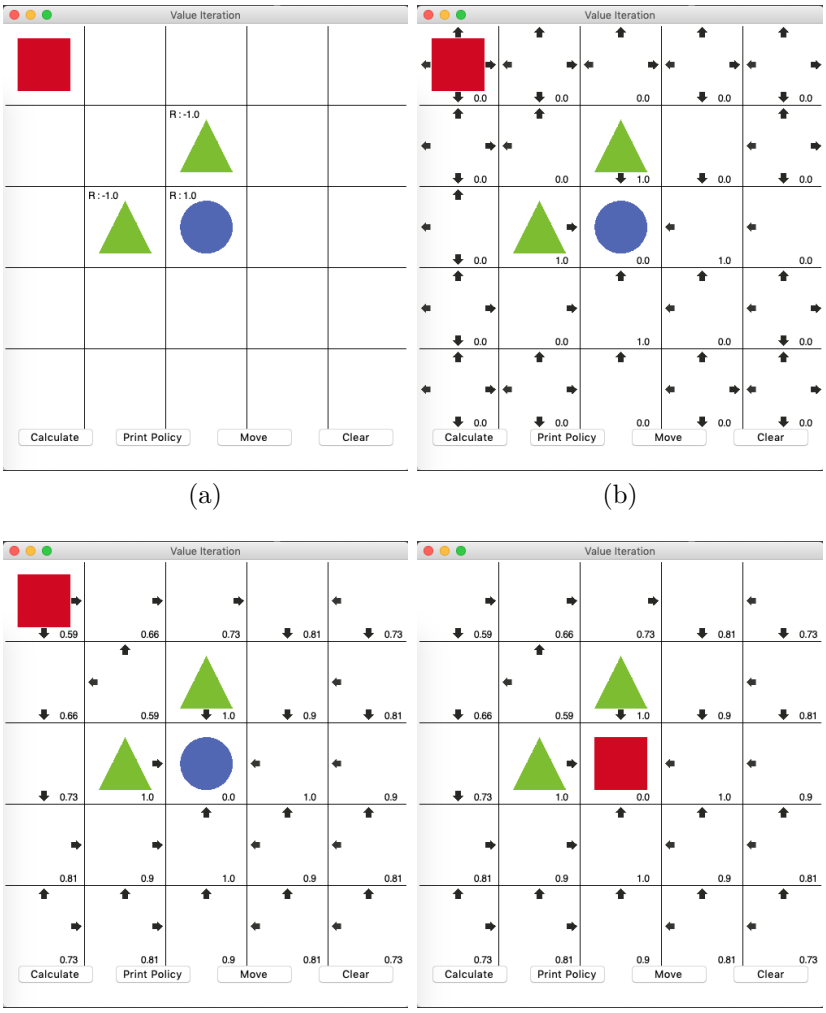


Figure 8: Code Execution

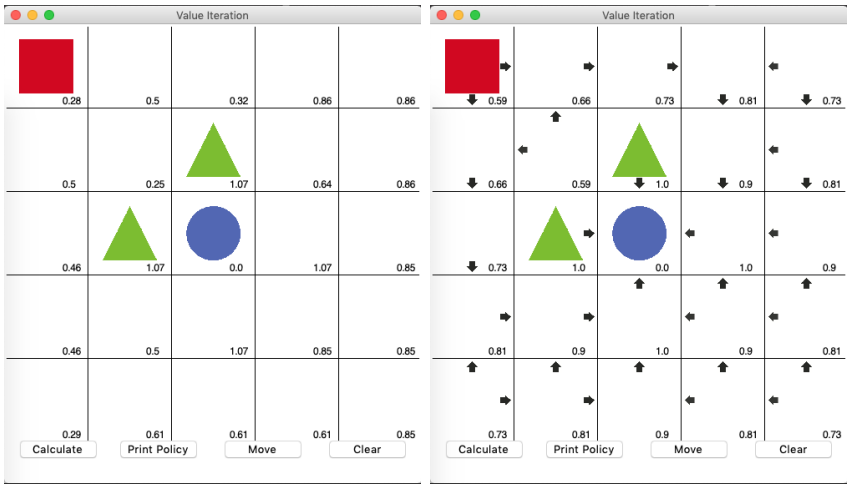
2. Changing the initial value of value function

   In this part, the initial value of value function is change to be random. The random value is drawn from uniform distribution with 0 and 1 as the minimum and maximum value.

```
1 def __init__(self, env):
2     self.env = env
3     # 2-d list for the value function
4     self.value_table = np.random.uniform(0,1,25).reshape(5,5)
5     self.discount_factor = 0.9
```

Listing 23: Policy Iteration – Change the Initialization

In this case we need `numpy` module. From Listing 23, we can see the changing part is in Line 4. The execution example is shown in Fig.9. We can see that in the first click



(a) (b)

Figure 9: Code Execution – Initialization Scenario

of evaluate button, the value function that are appeared in the canvas are different with the original one. The optimal value function we get from this scenario is shown in right side. We can see that even if the initial value is changed to be random, the optimum value function we get is still same. The optimum path for reaching the goal also same with the original one.

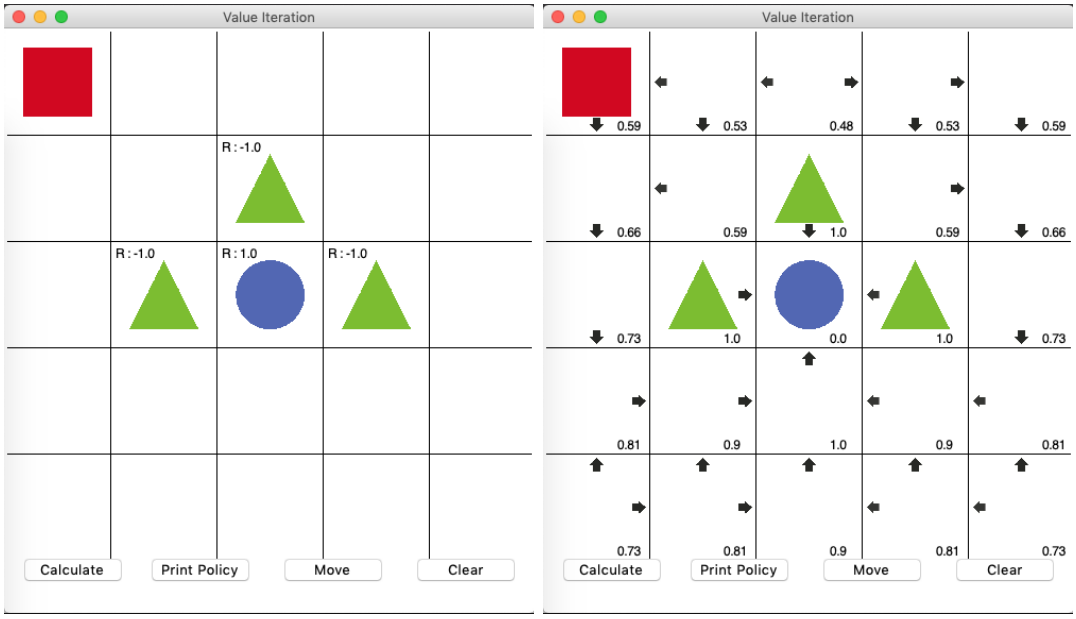
3. Adding one more triangle

   In this scenario, we add one more triangle. We locate the adding triangle in coordinate $(3, 2)$. In the code, we add more row in `_build_canvas` function (see Line 2–6) and `__init__`, see Line 9–12 and Line 15–18 in the following Listing.

```
1  #The added row in _build_canvas function (GraphicDisplay module)
2  self.rectangle = canvas.create_image(50, 50, image=self.shapes[0])
3  canvas.create_image(250, 150, image=self.shapes[1])
4  canvas.create_image(150, 250, image=self.shapes[1])
5  canvas.create_image(350, 250, image=self.shapes[1])
6  canvas.create_image(250, 250, image=self.shapes[2])
7
8  #The added row in __init__ function (GraphicDisplay module)
9  self.text_reward(2, 2, "R : 1.0")
10 self.text_reward(1, 2, "R : -1.0")
11 self.text_reward(2, 1, "R : -1.0")
12 self.text_reward(2, 3, "R : -1.0")
13
14 #The added row in __init__ function (Env module)
15 self.reward[2][2] = 1  # reward 1 for circle
16 self.reward[1][2] = -1  # reward -1 for triangle
17 self.reward[2][1] = -1  # reward -1 for triangle
18 self.reward[2][3] = -1  # reward -1 for triangle
```

Listing 24: Policy Iteration – Change the Initialization

The display of our grid world after adding one more triangle is shown in Fig.10. We can see in the right figure that the one optimum path is same with the original one. In this scenario, because we add the new triangle in coordinate $(3, 2)$, the optimum path is just through $(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 2)$.



(a) (b)

Figure 10: Code Execution – Adding Triangle Scenario