دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

درس مبانی و کاربردهای هوش مصنوعی

گزارش پروژه دوم(Frozen Lake Game)

نام و نام‌خانوادگی : مهدی رهبر

شماره دانشجویی : ۴۰۱۳۶۱۳۰۳۶

پاییز ۱۴۰۳

- **فرآیند تصمیم‌گیری مارکوف (MDP)**

فرآیند تصمیم‌گیری مارکوف (MDP) یک مدل ریاضی برای تصمیم‌گیری در محیط‌های نامعین است که شامل مجموعه‌ای از حالت‌ها، کنش‌ها، انتقال‌ها، پاداش‌ها و عامل تخفیف می‌باشد. دو الگوریتم رایج برای حل MDP عبارت‌اند از: Value Iteration که مستقیماً با بهینه‌سازی ارزش هر حالت، سیاست بهینه را استخراج می‌کند و Policy Iteration که با تکرار بین ارزیابی سیاست و بهبود آن، سیاست بهینه را می‌یابد. در این پروژه، این دو الگوریتم به عامل کمک می‌کنند تا بهترین مسیر را از حالت شروع به سمت هدف پیدا کند و از چاه‌ها اجتناب کند، در حالی که پاداش‌ها و احتمالات لغزش محیط نیز در نظر گرفته می‌شوند.

۱) پیاده سازی الگوریتم Value Iteration:

```
3 usages
def create_reward_map(env):
    reward_map = np.zeros(env.nS)
    for s in range(env.nS):
        row, col = np.unravel_index(s, env.shape)

        if env._lake[row, col]:
            reward_map[s] = -10
        elif s == env.nS - 1:
            reward_map[s] = 10
        else:
            reward_map[s] = -1
    return reward_map
```

تابع **create_reward_map** نقشه‌ای از پاداش‌ها برای تمامی حالت‌های محیط ایجاد می‌کند. با پیمایش تمامی حالت‌ها، پاداش هر حالت بر اساس نوع آن تعیین می‌شود: چاه‌ها پاداش ۱۰-، هدف پاداش ۱۰، و سایر خانه‌ها جریمه ۱- برای حرکت دارند. این تابع نقش کلیدی در الگوریتم‌های Value Iteration و Policy

Iteration دارد، زیرا با تعریف پاداش‌های محیط، به محاسبه ارزش حالت‌ها و استخراج سیاست بهینه کمک می‌کند.

```python
def value_iteration(env, gamma=0.85, threshold=1e-6):

    reward_map = create_reward_map(env)
    value_table = np.zeros(env.nS)

    slip_probs = {
        0: [(0.5, 0), (0.25, 1), (0.25, 3)],
        1: [(0.5, 1), (0.25, 0), (0.25, 2)],
        2: [(0.5, 2), (0.25, 1), (0.25, 3)],
        3: [(0.5, 3), (0.25, 0), (0.25, 2)]
    }


    iteration_count = 0

    for _ in range(max_iter_number):
        updated_value_table = np.copy(value_table)
        iteration_count += 1

        for s in range(env.nS):

            Q_values = []
            for a in range(env.nA):
                q_value = 0
                for slip_prob, slip_action in slip_probs[a]:
                    for prob, s_, _, _ in env.P[s][slip_action]:
                        if isinstance(s_, tuple):
                            s_ = 8 * s_[0] + s_[1]
                        q_value += slip_prob * (reward_map[s] + gamma * updated_value_table[s_])
                Q_values.append(q_value)
            value_table[s] = max(Q_values)

        if np.sum(np.abs(updated_value_table - value_table)) <= threshold:
            break

    print(f"Value Iteration converged in {iteration_count} iterations.")
    return value_table


```

تابع **value_iteration** الگوریتم تکرار ارزش را برای یافتن سیاست بهینه در محیط اجرا می‌کند. ابتدا نقشه پاداش‌ها (reward_map) و جدول ارزش اولیه مقداردهی می‌شوند. در هر تکرار، با استفاده از احتمالات لغزش تعریف‌شده، مقدار Q برای هر کنش محاسبه و حداکثر آن به‌عنوان ارزش جدید حالت ثبت می‌شود. فرآیند تا زمانی ادامه می‌یابد که تغییرات جدول ارزش کمتر از آستانه همگرایی باشد یا به حداکثر تعداد تکرار برسد. این تابع ارزش نهایی هر حالت را بازمی‌گرداند.

```python
def extract_policy(env, value_table, gamma=0.85):
    policy = np.zeros(env.nS, dtype=int)

    reward_map = create_reward_map(env)

    slip_probs = {
        0: [(0.5, 0), (0.25, 1), (0.25, 3)],
        1: [(0.5, 1), (0.25, 0), (0.25, 2)],
        2: [(0.5, 2), (0.25, 1), (0.25, 3)],
        3: [(0.5, 3), (0.25, 0), (0.25, 2)]
    }

    for s in range(env.nS):
        Q_values = []
        for a in range(env.nA):
            q_value = 0
            for slip_prob, slip_action in slip_probs[a]:
                for prob, s_, _, _ in env.P[s][slip_action]:

                    if isinstance(s_, tuple):
                        s_ = 8 * s_[0] + s_[1]

                    q_value += slip_prob * (reward_map[s] + gamma * value_table[s_])
            Q_values.append(q_value)
        policy[s] = np.argmax(Q_values)

    return policy
```

تابع **extract_policy** سیاست بهینه را از جدول ارزش (value_table) استخراج می‌کند. برای هر حالت، مقادیر Q برای تمامی کنش‌ها با استفاده از احتمالات لغزش، نقشه پاداش‌ها (reward_map) و ارزش حالت‌های بعدی محاسبه می‌شود. سپس کنشی که بیشترین مقدار Q را دارد، به‌عنوان سیاست بهینه برای آن حالت انتخاب می‌شود. این تابع نقش کلیدی در هر دو الگوریتم Value Iteration و Policy Iteration دارد، زیرا بر اساس ارزش‌های محاسبه‌شده، سیاست بهینه را برای هدایت عامل به سمت هدف استخراج می‌کند.

```python
def plot_heatmap(value_table, env_shape):

    reshaped_table = value_table.reshape(env_shape)

    plt.figure(figsize=(8, 8))
    plt.imshow(reshaped_table, cmap='RdYlGn', interpolation='nearest')
    plt.colorbar(label='Value')
    plt.title('Value Heatmap')

    for i in range(env_shape[0]):
        for j in range(env_shape[1]):
            plt.text(j, i, s: f'{reshaped_table[i, j]:.2f}',
                     ha='center', va='center', color='black', fontsize=10)

    plt.show()
```

تابع **plot_heatmap** یک نمودار Heatmap از جدول ارزش (value_table) ترسیم می‌کند تا ارزش حالت‌ها را در محیط به‌صورت تصویری نمایش دهد. ابتدا جدول ارزش به شکل محیط تغییر داده می‌شود و با استفاده از کتابخانه Matplotlib، مقادیر ارزش به‌صورت رنگی (سبز برای مقادیر بالا و قرمز برای مقادیر پایین) نمایش داده می‌شود. همچنین، مقدار عددی هر خانه در مرکز آن درج می‌شود. این تابع نقش مهمی در تحلیل و ارزیابی الگوریتم‌ها دارد، زیرا وضعیت ارزش حالت‌ها را به‌صورت گرافیکی نمایش می‌دهد و به درک بهتر نتایج کمک می‌کند.

۲) پیاده سازی الگوریتم Policy Iteration:

```python
def compute_value_function(policy, env, gamma=0.85, threshold=1e-6):

    value_table = np.zeros(env.nS)

    reward_map = create_reward_map(env)

    slip_probs = {
        0: [(0.5, 0), (0.25, 1), (0.25, 3)],
        1: [(0.5, 1), (0.25, 0), (0.25, 2)],
        2: [(0.5, 2), (0.25, 1), (0.25, 3)],
        3: [(0.5, 3), (0.25, 0), (0.25, 2)]
    }

    while True:
        updated_value_table = np.copy(value_table)

        for state in range(env.nS):

            action = policy[state]
            q_value = 0

            for slip_prob, slip_action in slip_probs[action]:

                for trans_prob, next_state, _, _ in env.P[state][slip_action]:

                    if isinstance(next_state, tuple):
                        next_state = 8 * next_state[0] + next_state[1]

                    q_value += slip_prob * (reward_map[state] + gamma * updated_value_table[next_state])

            value_table[state] = q_value

        if np.sum(np.abs(updated_value_table - value_table)) <= threshold:
            break

    return value_table
```

تابع **compute_value_function** ارزش حالت‌ها را برای یک سیاست داده‌شده محاسبه می‌کند. ابتدا جدول ارزش با مقادیر صفر مقداردهی می‌شود و با استفاده از نقشه پاداش (reward_map) و احتمالات لغزش، مقدار Q برای هر حالت محاسبه می‌شود. در هر تکرار، ارزش حالت فعلی بر اساس کنش انتخاب‌شده در سیاست، پاداش حالت فعلی و ارزش حالت‌های بعدی به‌روزرسانی می‌شود. فرآیند تا زمانی ادامه می‌یابد که تغییرات ارزش‌ها کمتر از مقدار آستانه (threshold) شود. این تابع در الگوریتم Policy Iteration نقش ارزیابی سیاست (Policy Evaluation) را ایفا می‌کند و ارزش حالت‌ها را برای سیاست فعلی برآورد می‌کند.

```python
def policy_iteration(env, gamma=0.85, threshold=1e-6):

    policy = np.zeros(env.nS, dtype=int)

    iteration_count = 0

    for i in range(max_iter_number):

        iteration_count += 1

        value_table = compute_value_function(policy, env, gamma, threshold)

        new_policy = extract_policy(env, value_table, gamma)

        if np.array_equal(policy, new_policy):
            print(f"Policy Iteration converged in {iteration_count} iterations.")
            break

        policy = new_policy

    return policy, value_table
```

تابع **policy_iteration** الگوریتم تکرار سیاست را برای یافتن سیاست بهینه اجرا می‌کند. ابتدا سیاست اولیه مقداردهی می‌شود و در هر تکرار، ارزش حالت‌ها برای سیاست فعلی با استفاده از compute_value_function محاسبه می‌شود. سپس سیاست جدید با استفاده از extract_policy استخراج می‌گردد. اگر سیاست جدید با سیاست قبلی یکسان باشد (همگرایی رخ داده باشد)، الگوریتم متوقف می‌شود و تعداد تکرارها چاپ می‌شود. در غیر این صورت، سیاست به‌روزرسانی شده و تکرار ادامه می‌یابد. این تابع با ارزیابی و بهبود متوالی سیاست، به سیاست بهینه و ارزش متناظر آن دست می‌یابد.

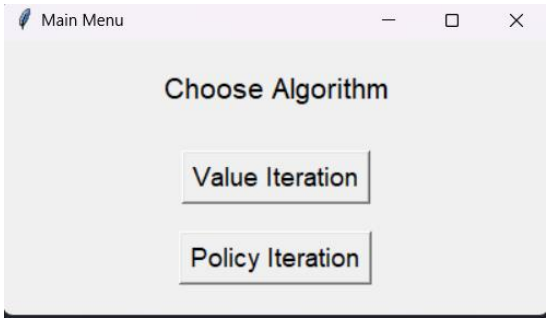
```python
def run_value_iteration():
    global observation
    optimal_value_table = value_iteration(env)
    optimal_policy = extract_policy(env, optimal_value_table)
    plot_heatmap(optimal_value_table, env.shape)

    while True:
        action = optimal_policy[observation]
        next_state, reward, done, truncated, info = env.step(action)

        observation = 8 * next_state[0] + next_state[1]

        if observation == 63:
            print("You reached the goal! Returning to the main menu.")
            env.close()
            main_menu()
            break
        elif env._lake[next_state]:
            print("You fell into a hole! Restarting the game.")
            observation, info = env.reset()
```

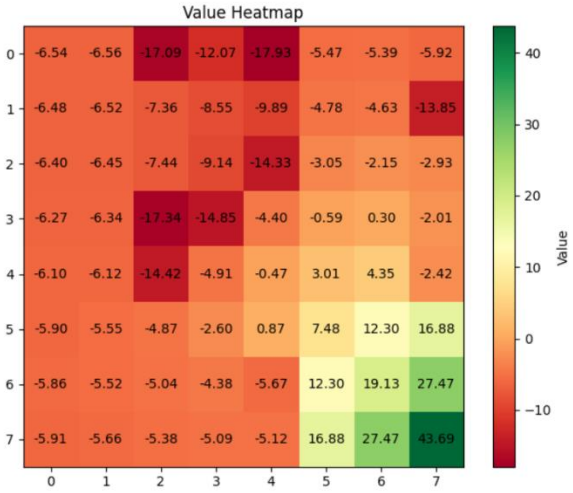
```python
def run_policy_iteration():
    global observation

    optimal_policy, value_table = policy_iteration(env)

    plot_heatmap(value_table, env.shape)

    while True:
        action = optimal_policy[observation]
        next_state, reward, done, truncated, info = env.step(action)

        observation = 8 * next_state[0] + next_state[1]

        if observation == 63:
            print("You reached the goal! Returning to the main menu.")
            env.close()
            main_menu()
            break
        elif env._lake[next_state]:
            print("You fell into a hole! Restarting the game.")
            observation, info = env.reset()
```

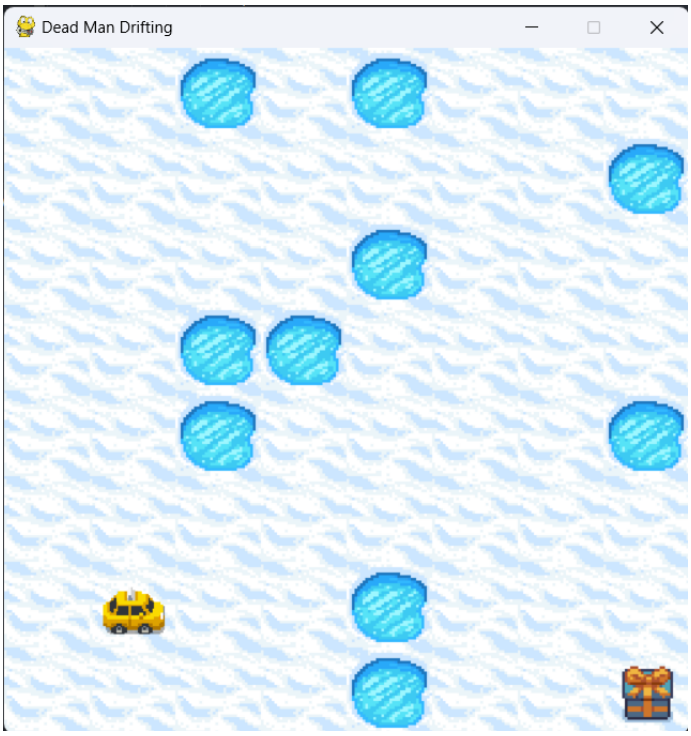
```python
def main_menu():
    global env, observation

    def start_value_iteration():
        root.destroy()
        global env, observation
        env = FrozenLake(render_mode="human", map_name="8x8")
        observation, info = env.reset(seed=30)
        run_value_iteration()

    def start_policy_iteration():
        root.destroy()
        global env, observation
        env = FrozenLake(render_mode="human", map_name="8x8")
        observation, info = env.reset(seed=30)
        run_policy_iteration()


    root = tk.Tk()
    root.title("Main Menu")
    root.geometry("400x200")


    tk.Label(root, text="Choose Algorithm", font=("Arial", 16)).pack(pady=20)
    tk.Button(root, text="Value Iteration", font=("Arial", 14), command=start_value_iteration).pack(pady=10)
    tk.Button(root, text="Policy Iteration", font=("Arial", 14), command=start_policy_iteration).pack(pady=10)

    root.mainloop()
```

سه تابع بالا برای مدیریت اجرا الگوریتم‌ها در منو پیاده سازی شده اند.

خب بعد از بررسی کدها پروژه را اجرا می‌کنیم. وقتی پروژه اجرا می شود یک منو ظاهر شده و می توانیم از بین دو الگوریتم MDP انتخاب کنیم. با انتخاب هر کدام از الگوریتم ها ابتدا نقشه Heatmap نمایش داده می‌شود.سپس با بستن نقشه، عامل(تاکسی) شروع به جست‌وجو کرده و به هدف می‌رسد. البته به دلیل انتخاب های تصادفی امکان افتادن در چاه ها هم وجود دارد. به صورت کلی عملکرد دو الگوریتم با بررسی خطاها و پیروزی ها خوب ارزیابی می‌شود با این تفاوت که الگوریتم policy iteration با تعداد تکرار به مراتب کم تری همگرا می‌شود. در ادامه تصاویر خروجی و نقشه heatmap نشان داده شده است:


Main Menu
Choose Algorithm
Value Iteration
Policy Iteration


Value Heatmap
0 -6.54 -6.56 -17.09 -12.07 -17.93 -5.47 -5.39 -5.92
1 -6.48 -6.52 -7.36 -8.55 -9.89 -4.78 -4.63 -13.85
2 -6.40 -6.45 -7.44 -9.14 -14.33 -3.05 -2.15 -2.93
3 -6.27 -6.34 -17.34 -14.85 -4.40 -0.59 0.30 -2.01
4 -6.10 -6.12 -14.42 -4.91 -0.47 3.01 4.35 -2.42
5 -5.90 -5.55 -4.87 -2.60 0.87 7.48 12.30 16.88
6 -5.86 -5.52 -5.04 -4.38 -5.67 12.30 19.13 27.47
7 -5.91 -5.66 -5.38 -5.09 -5.12 16.88 27.47 43.69
0 1 2 3 4 5 6 7
40
30
20
10
0
−10
Value


Dead Man Drifting