دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

حل کردن مسئله یادگیری تقویتی با استفاده از الگوریتم MDP

(به کمک رابط گرافیکی ماژول Gymnasium)

آرین جعفری (4003613014)

محمدحسن حیدری (4003613025)

- توضیحات کلی

در فاز قبل (پیدا کردن بهترین مسیر پروازی) ، محیطی که با آن کار می‌کردیم محیطی قطعی بود. اما محیطی که در این فاز پیش روی ماست محیطی غیر قطعی است. در این پروژه ما از کتابخانه Gymnasium استفاده میکنیم . با استفاده از Gymnasium علاوه بر پیاده سازی الگوریتم ، میتوانیم به صورت گرافیکی نحوه اجرای الگوریتم را به صورت بصری ببینیم . محیطی که این پروژه در آن پیاده سازی شده و عامل قصد یادگیری آن را دارد ، محیط Cliff Walking است . این محیط امکان فعالیت عامل در فضای دو بعدی را فراهم می سازد

- جزئیات محیط و پیاده سازی

محیط Cliff Walking محیطی است برای یادگیری تقویتی در فضایی دو بعدی که در این پروژه به گونه‌ای شخصی‌سازی شده که تعداد 48 خانه مشخص داریم و در هر بار اجرادر آن تعداد صخره‌ی مشخصی به صورت رندوم در آن قرار می‌گیرند . خانه ای که عامل از آن جست و جو را شروع میکند [3, 0] و خانه‌ی هدف [3, 11] است . عامل ما می‌تواند با کلیدهای 0، 1، 2و 3 به ترتیب به ترتیب و در صورت امکان به بالا، راست، پایین و چپ می‌رود . در ضمن عامل به صورت غیر قطعی عمل می‌کند و در صورت قصد برای یک حرکت تنها با احتمال 1/3 به آن سمت می‌رود و با احتمال 1/3 به دو سمت مجاور آن خواهد رفت . بازی در در صورت ورود به خانه‌ی صخره یا رسیدن به حالت هدف ، پایان می‌یابد و محیط دوباره بارگذاری میشود . امتیاز هر حرکت در این محیط برای خانه های غیر صخره 1- و امتیاز خانه‌ی صخره 100- است .

- الگوریتم حل مسئله

مسائل یادگیری تقویتی عموما با الگوریتم هایی حل میشوند که پیاده سازی آنها باید به گونه ای باشد که عامل بتواند در محیطی با ویژگی مشاهده پذیری جزئی به خوبی فعالیت کند . با توجه به محتوای درس ، الگوریتم MDP (Markov Decision Process) برای حل این مسئله از یادگیری تقویتی در محیط دو بعدی انتخاب شد . فرایند تصمیم مارکوف الگوریتم و مدلی ریاضی برای حل مسائل ترتیبی است . در این الگوریتم یک عامل در هر مرحله با مجموعه ای از احتمالات و حالات روبرو است و کنشی را انتخاب میکند . انتخاب کنش موجب تغییر حالت و دریافت پاداش میشود . هدف عامل انتخاب سیاستی است که پاداش کل را به حداکثر برساند .

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

- پیاده سازی

```
# Import nessary libraries
import numpy as np
import gymnasium as gym
from gymnasium.envs.toy_text.cliffwalking import CliffWalkingEnv
from gymnasium.error import DependencyNotInstalled
from os import path
```

در ابتدای پیاده سازی این فاز به زبان پایتون , ماژول ها و کتابخانه های مورد نیاز را دخیل میکنیم . استفاده از کتابخانه Numpy روشی مرسوم و کارآمد برای انجام عملیات های عددی و ماتریسی است . به کمک این کتابخانه پیاده سازی الگوریتم های مختلف آسان میشود و در عین حال به دلیل استفاده از تکنیک های چند پردازشی ، سرعت اجرای دستورات و به تبع الگوریتم ها بالاتر خواهد رفت .

استفاده از کتابخانه Gymnasium همانطور که قبلا گفته شد ، پیاده سازی الگوریتم های یادگیری تقویتی در قالبی آماده و سهل انجام میشود . توابع آماده این کتابخانه به توسعه دهنده کمک میکند تمرکز خود را روی انتخاب سیاست بهینه برای حل مسئله بگذارد و وقت زیادی جهت تعریف محیط و پاداش از دست ندهد .

```
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3
image_path = path.join(path.dirname(gym.__file__), "envs", "toy_text")
```

کنش های ممکن برای عامل به صورت حرکات در دو بعد ممکن هستند . بالا ، راست ، پایین و چپ

```
class CliffWalking(CliffWalkingEnv):
    def __init__(self, is_hardmode=True, num_cliffs=10, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.is_hardmode = is_hardmode

        # Generate random cliff positions
        if self.is_hardmode:
            self.num_cliffs = num_cliffs
            self._cliff = np.zeros(self.shape, dtype=bool)
            self.start_state = (3, 0)
            self.terminal_state = (self.shape[0] - 1, self.shape[1] - 1)

            self.cliff_positions = []
            while len(self.cliff_positions) < self.num_cliffs:
                new_row = np.random.randint(0, 4)
                new_col = np.random.randint(0, 11)
                state = (new_row, new_col)
                if (
                    (state not in self.cliff_positions)
                    and (state != self.start_state)
                    and (state != self.terminal_state)
                ):
                    self._cliff[new_row, new_col] = True
                    if not self.is_valid():
                        self._cliff[new_row, new_col] = False
                        continue
                    self.cliff_positions.append(state)

        # Calculate transition probabilities and rewards
        self.P = {}
        for s in range(self.nS):
            position = np.unravel_index(s, self.shape)
            self.P[s] = {a: [] for a in range(self.nA)}
            self.P[s][UP] = self._calculate_transition_prob(position, [-1,
0])
            self.P[s][RIGHT] = self._calculate_transition_prob(position,
[0, 1])
            self.P[s][DOWN] = self._calculate_transition_prob(position, [1,
0])
            self.P[s][LEFT] = self._calculate_transition_prob(position, [0,
-1])
```

کلاس Cliff Walking از قبل پیاده شده و شروط پاداش ها و خانه ها و شروع و پایان در آن مشخض شده اند . محیط 48 خانه دارد که تعداد 10 خانه از آن به صورت صخره یا Cliff و به صورت تصادفی از بین این 48 خانه انتخاب میشوند .

```
def step(self, action):
    if action not in [0, 1, 2, 3]:
        raise ValueError(f"Invalid action {action}   must be in [0, 1, 2,
3]")

    if self.is_hardmode:
        match action:
            case 0:
                action = np.random.choice([0, 1, 3], p=[1 / 3, 1 / 3, 1 /
3])
            case 1:
                action = np.random.choice([0, 1, 2], p=[1 / 3, 1 / 3, 1 /
3])
            case 2:
                action = np.random.choice([1, 2, 3], p=[1 / 3, 1 / 3, 1 /
3])
            case 3:
                action = np.random.choice([0, 2, 3], p=[1 / 3, 1 / 3, 1 /
3])

    return super().step(action)
```

با توجه به محیط غیر قطعی تعریف شده , هر کنش عامل به احتمال 1/3 انجام خواهد شد . این ویژگی محیط به ویژگی های تابع step از کلاس پدر Cliff Walking اضافه شده تا به محیط چالش بیشتری ببخشد .

سایر توابع مربوط به پیاده سازی رابط گرافیکی را از این لینک میتوان دنبال کرد .

در نهایت پیاده سازی تابع Value Iteration برای پیدا کردن بهترین سیاست و پیاده سازی این سیاست در حلقه اصلی برنامه به شیوه پیش رو انجام خواهد شد . عامل با توجه به این سیاست در هر مرحله ، کنش خود را انتخاب میکند

```
def value_iteration(env, gamma=0.9, theta=1e-50):

    V = np.zeros(env.nS)  # Initialize the state-value function with zeros
    Q = np.zeros((env.nS, env.nA))  # Initialize the action-value function
with zeros
    i = 0

    while True:
        delta = 0  # Initialize delta to track changes in the value
function
        for s in range(env.nS):  # For each state in the environment
            v = V[s]  # Store the current value of the state

            for a in range(env.nA):  # For each action in the environment

                if a == 0 or a == 2 :

                    left= sum([p * (r + gamma * V[s_]) for p, s_, r, _ in
env.P[s][3]])
                    right = sum([p * (r + gamma * V[s_]) for p, s_, r, _ in
env.P[s][1]])
                    action = sum([p * (r + gamma * V[s_]) for p, s_, r, _
in env.P[s][a]])
                    summ = left + right + action

                elif a == 1 or a == 3 :

                    up= sum([p * (r + gamma * V[s_]) for p, s_, r, _ in
env.P[s][0]])
                    down = sum([p * (r + gamma * V[s_]) for p, s_, r, _ in
env.P[s][2]])
                    action = sum([p * (r + gamma * V[s_]) for p, s_, r, _
in env.P[s][a]])
                    summ = up + down + action

                Q[s][a] = summ

            V[s] = max(Q[s])  # Update the value function
            delta = max(delta, abs(v - V[s]))  # Update delta

        i+=1

        if delta < theta:
            # If the change in the value function is less than the
threshold
            print('converged at ', i)
            break


    return V, Q
```

به ازای هر کنش در هر حالت ، به علت غیر قطعی بودن محیط ، معادله بلمن (Bellman) برای کنش ها مجاور نیز محاسبه میشود . برای مثال اگر کنش به سمت بالا باشد , معادله بلمن را برای کنش های چپ و راست نیز محاسبه میکنیم . و یا اگر کنش به سمت چپ باشد ، نتیجه معادله بلمن را برای کنش های بالا و پایین نیز محاسبه میکنیم . در نهایت سیاست

انتخاب شده در سطر ماتریس Q ذخیره میشود . این کار را برای تمامی کنش ها در تمامی حالت ها انجام میدهیم .

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right)$$

ماتریس Q ماتریسی است با تعداد سطر هایی متناسب با تعداد حالات و ستون هایی متناسب با تعداد کنش های ممکن . با استفاده از این ماتریس ، در هر مرحله ما بهترین کنش ممکن (بزرگترین عنصر هر سطر در هر کنش) را انتخاب کرده و به عنوان کنش عامل انجام میدهیم . هر چند به احتمال 1/3 این کنش انجام خواهد شد .

در حلقه اصلی برنامه و برای پیاده سازی الگوریتم ذکر شده ، با Q بدست آمده ، انتخاب بهترین کنش ممکن بر اساس سیاست استخراج شده ، کار دشواری نخواهد بود . هر چند همانطور که گفته شد ، با توجه به عدم قطعیت کنش ها در محیط ، پیدا کردن خانه هدف برای عامل اندکی چالشی خواهد بود .

```
# Create an instance of your CliffWalking environment
env = CliffWalking(render_mode="human")
observation, info = env.reset(seed=30)

# Run value iteration
V_star, Q_star = value_iteration(env)

# Define the maximum number of iterations
max_iter_number = 1000
wins = 0
```

```python
for i in range(max_iter_number):

    # Choose an action according to the policy
    action = np.argmax(Q_star[env.s])

    # Perform the action and receive feedback from the environment
    next_state, reward, done, truncated, info = env.step(action)


    # Update the current state
    env.s = next_state

    if done or truncated:
        print(f'At Epoch : {i} Agent Succeeded !')
        wins += 1
        print(f'Wins : {wins}')
        observation, info = env.reset()

# Close the environment
env.close()
```

- کتابخانه های استفاده شده

Numpy -

Gymnasium -

- زمان تقریبی رسیدن به هدف

- بین تکرار های 80 تا 200 اولین موفقیت

- بین تکرار های 220 تا 600 موفقیت دوم

- به علت غیر قطعی بودن محیط ، گاهی اولین موفقیت حتی تا تکرار 800 هم رخ نمیدهد (به ندرت)

- منابع مورد استفاده

AI Modern Approach (Russell, Norvig) -

Stanford Machine Learning lecture -

Microsoft Copilot -