"بسمه تعالی"

## گزارش پروژه سوم هوش مصنوعی – زهرا لطیفی – ۹۹۲۳۰۶۹

### الگوریتم ۱: Iterative Policy Evaluation

مطابق آنچه در صورت سوال گفته شده، این الگوریتم از یک حدس اولیه نسبت به تابع ارزش شروع کرده و بعد با استفاده از Bellman Equation مقادیر ارزش را به مرور به‌روزرسانی می‌کند تا وقتی که الگوریتم همگرا شود. این تابع دینامیک، سیاست انتخابی، پارامتر گاما و تعداد iteration ها را به عنوان ورودی گرفته و مقادیر تمام ۱۶ حالت را بر می‌گرداند.

به این صورت که ابتدا تعداد حالات و اعمال تعریف شده و آرایه خروجی به طول ۱۶ ایجاد و با صفر مقداردهی اولیه شده است. سپس به تعداد خواسته شده حلقه for تکرار انجام می‌دهد. به ازای هر state یک متغیر موقت به نام new_s تعریف شده که ابتدا صفر مقداردهی می‌شود. عمل هر مرحله را policy در آن state تعیین می‌کند. متغیر transition_prob احتمال رفتن به هر state از حالت فعلی با عمل انتخاب شده را محاسبه و ذخیره می‌کند. مقادیر حالت بعدی و پاداش هم با فراخوانی تابع dynamics.next_state_reward(s, a) مشخص می‌شود. حال با رابطه زیر، مقدار هر حالت محاسبه شده و در متغیر new_s جمع می‌شود.

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$$

نهایتا بررسی می‌شود که حالت در دست بررسی ترمینال استیت غیر از خانه آخر (چاله) هست یا نه. اگر نبود، value آن به روز می‌شود و اگر بود، صفر باقی می‌ماند.

```
def policy_evaluation(dynamics, policy, gamma=0.9, num_iter=10):

    # TODO: implement Iterative Policy Evaluation algorithm
    state_len = 16
    action_len = 4
    s_value_function = np.zeros(16, dtype=float)
    for i in range(num_iter):
        # for all state
        for s in range(state_len):
            # we will get new state value
            new_s = 0
            a = policy[s]
            transition_prob = dynamics.trans_prob(s, a)
            next_state, reward = dynamics.next_state_reward(s, a)
            new_s +=
transition_prob[next_state]*(reward+gamma*s_value_function[next_state])
            if dynamics.is_terminal(s) == 1 and s != 15:
                s_value_function[s] = 0
            else:
                s_value_function[s] = new_s

    return s_value_function
```

حال دو بار تابع نوشته شده را به ازای سیاست‌های go-right و shortest-path فراخوانی می‌کنیم:

```
s_value_function1 = policy_evaluation(dynamics, policy1)
s_value_function2 = policy_evaluation(dynamics, policy2)
```

پس از آن مقادیر هر حالت را چاپ کردیم:

```
# TODO: print and analyze the state value function
print("State values for go-right policy:")
print(np.round(s_value_function1.reshape(4,4), 3))
print("\nState values for shortest-path policy:")
print(np.round(s_value_function2.reshape(4,4), 3))
```

```
State values for go-right policy:
[[0.    0.    0.    0.   ]
 [0.    0.    0.    0.   ]
 [0.    0.    0.    0.   ]
 [0.    5.513 6.513 6.513]]

State values for shortest-path policy:
[[2.418 3.074 3.803 3.423]
 [3.074 0.    4.613 0.   ]
 [3.803 4.613 5.513 0.   ]
 [0.    5.513 6.513 6.513]]
```

مطابق انتظار، مقدار حالت‌های مربوط به چاله صفر باقی ماند و مقادیر هم هرچه به حالت پایانی که دارای پاداش ۱ است نزدیک‌تر می‌شویم، بیشتر می‌شوند. با توجه به صفر بودن همیشگی مقادیر حالات دارای چاله، در سیاست go-right مقادیر حالات ۰ تا ۱۲ صفر باقی خواهند ماند.

## الگوریتم۲: Policy Iteration

بناست سیاست بهینه را بدست آوریم. مطابق صورت سوال، این الگوریتم از یک سیاست رندوم شروع کرده و آن را با تابع قسمت قبل ارزیابی می‌کند. با توجه به ارزش حالت‌ها، سیاست را به صورت حریصانه بروزرسانی می‌کند تا اعمال بهتری انتخاب شوند. این کار تکرار می‌شود تا سیاست ما بهینه شود.

ابتدا تابع `greedy_policy_improvement` پیاده سازی شد. ورودی‌ها دینامیک، آرایه مقادیر ارزش‌های حالات و پارامتر گاما هستند و سیاست بهبود یافته را بر می‌گرداند. در این تابع ابتدا تعداد حالات و اعمال تعریف شده سپس آرایه `policy` با اعمال رندوم مقداردهی اولیه شده است. پس از آن به ازای هر حالت، بردار `q_values` به طول تعداد اعمال ایجاد و با صفر مقداردهی اولیه می‌شود. به ازای هر عمل ممکن، متغیر `transition_prob` احتمال رفتن به هر `state` از حالت فعلی با عمل انتخاب شده را محاسبه و ذخیره می‌کند. مقادیر حالت بعدی و پاداش هم با فراخوانی تابع `dynamics.next_state_reward(s, a)` مشخص می‌شود. حال با رابطه زیر، `q-value` ها محاسبه شده و در متغیر `q_values[a]` جمع می‌شود.

$$\sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$$

پس از آن اعمالی که به ازای آن‌ها `Q-Value` بیشینه شده، آرایه `policy` را تشکیل می‌دهند.

```
def greedy_policy_improvement(dynamics, s_value_function, gamma=0.9):

    # TODO: implement Greedy Policy Improvement algorithm
    state_len = 16
    action_len = 4
    policy = np.random.randint(0, 4, size=16)
    for s in range(state_len):
        q_values = np.zeros(action_len)
        for a in range(action_len):
          transition_prob = dynamics.trans_prob(s, a)
          next_state, reward = dynamics.next_state_reward(s, a)
          q_values[a] += transition_prob[next_state]*(reward + gamma *
s_value_function[next_state])
        policy[s] = np.argmax(q_values)

    return policy
```

پس از آن تابع policy_iteration پیاده سازی شد. ورودی ها دینامیک، پارامتر گاما و تعداد iteration های حلقه درونی و بیرونی هستند و سیاست بهینه را بر می‌گرداند. در این تابع هم تعداد حالات، اعمال و آرایه s_value_function به طول ۱۶ تعریف شده و مقداردهی اولیه شده‌اند. آرایه policy هم با اعمال رندوم مقداردهی شده. حال حلقه for به تعداد iterationهای خارجی اجرا می‌شود. در این حلقه یک کپی از سیاست گرفته می‌شود سپس با فراخوانی تابع polivy_evaluation، این سیاست ارزیابی می‌شود و آرایه V(s) را بر می‌گرداند. حال با فراخوانی تابع greedy_policy_improvement و دادن آرایه محاسبه شده به آن، سیاست را بهبود می‌دهیم و تنها در صورتی که سیاست با سیاست قبلی تفاوت داشت، آن را به روز می‌کنیم.

```
def policy_iteration(dynamics, gamma=0.9, outer_iter=100, inner_iter=100):

    # TODO: implement Policy Iteration algorithm
    state_len = 16
    action_len = 4
    s_value_function = np.zeros(state_len)
    policy = np.random.randint(0, 4, size=16)
    for i in range(outer_iter):
        old_policy = policy.copy()
        s_value_function = policy_evaluation(dynamics, policy, gamma,
inner_iter)
        policy = greedy_policy_improvement(dynamics, s_value_function, gamma)
        if np.all(old_policy == policy):
            break
    return policy
```

نوبت به تست توابع رسیده است. تابع policy_iteration را فرا می‌خوانیم تا سیاست بهینه یافته شده را ببینیم.

```
# TODO: test and analyze the algorithm
dynamics = FrozenLakeMDP(is_slippery=False)
policy = policy_iteration(dynamics)
policy
```

سیاست را بر روی محیط با `is_slippery=False` اجرا می‌کنیم. نتیجه به شرح زیر است:

```
array([1, 2, 1, 0, 1, 1, 1, 0, 2, 1, 1, 1, 2, 2, 2, 1])
```

```
action: 1, state: 4, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 1, state: 8, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 2, state: 9, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 1, state: 13, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 2, state: 14, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 2, state: 15, reward: 1.0, done: True, truncated: False, info:
{'prob': 1.0}
total reward: 1.0
```

ویدیوی Policy_iteration مربوط به این بخش است.

یک بار هم با `is_slippery=True` اجرا شد که عامل نتوانست به مقصد برسد. چون در این حالت ممکن است هیچ‌گاه عامل در جهتی که انتخاب می‌شود حرکت نکند، هیچ تضمینی مبنی بر اینکه بتوانیم به مقصد برسیم وجود ندارد. به علاوه چون زمان هم محدود است، احتمال رسیدن به مقصد کمتر هم خواهد شد.

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 1])
```

```
action: 0, state: 0, reward: 0.0, done: False, truncated: False, info: {'prob': 0.333333
3333333333}
action: 0, state: 4, reward: 0.0, done: False, truncated: False, info: {'prob': 0.333333
3333333333}
action: 0, state: 4, reward: 0.0, done: False, truncated: False, info: {'prob': 0.333333
3333333333}
action: 0, state: 0, reward: 0.0, done: False, truncated: False, info: {'prob': 0.333333
3333333333}
action: 0, state: 4, reward: 0.0, done: False, truncated: False, info: {'prob': 0.333333
3333333333}
action: 0, state: 8, reward: 0.0, done: False, truncated: False, info: {'prob': 0.333333
3333333333}
action: 0, state: 8, reward: 0.0, done: False, truncated: False, info: {'prob': 0.333333
3333333333}
action: 0, state: 8, reward: 0.0, done: False, truncated: False, info: {'prob': 0.333333
3333333333}
action: 0, state: 12, reward: 0.0, done: True, truncated: False, info: {'prob': 0.333333
3333333333}
total reward: 0.0
```

ویدیوی Policy_iteration_True مربوط به این بخش است.

## الگوریتم۳: Q-Learning

در این بخش طبق صورت سوال دیگر عامل از پیش محیط را نمی‌شناسد. پس از الگوریتم Model-free تحت عنوان temporal-difference استفاده می‌کنیم تا عامل، خود، محیط را یاد بگیرد.

ابتدا تابع choose_action را نوشتیم. این تابع حالت فعلی را به عنوان ورودی می‌گیرد و عمل را با الگوریتم epsilon-greedy انتخاب می‌کند و بر می‌گرداند. به این صورت که در هر بار فراخوانی عدد رندمی تولید کرده و با پارامتر تعریف شده epsilon که در ابتدا برابر یک است مقایسه می‌کند. اگر از اپسیلون کمتر بود، یک عمل رندم انتخاب می‌کند و در غیر این صورت، عملی که به ازای آن q-value بیشینه می‌شود را انتخاب خواهد کرد.

```
def choose_action(self, state):
        """
        chooses an action in an epsilon-greedy manner.

        Args:
            state (int): current state of the agent.

        Returns:
            int: the chosen action
        """

        # TODO: implement epsilon-greedy action selection
        if np.random.rand() < self.epsilon:
            action = np.random.randint(self.q_table.shape[1])
        else:
            action = np.argmax(self.q_table[state])
        return action
```

تابع بعدی تابع learn است که حالت، عمل، پاداش و حالت بعدی را گرفته و به ازای این تک ارتباط عامل با محیط، با رابطه زیر، مقادیر q-table را به روز رسانی می‌کند. البته مقادیر برای چاله ها همچنان صفر باقی خواهند ماند.

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$$

```
def learn(self, state, action, reward, next_state):

        # TODO: implement Q-table update
        if state not in [5, 7, 11, 12]:
            self.q_table[state, action] = self.q_table[state, action] + \
                                          self.alpha * (reward + self.gamma *
np.max(self.q_table[next_state]) - self.q_table[state, action])

        # epsilon decay
        self.epsilon = self.epsilon - self.eps_decay if self.epsilon >
self.eps_end else self.eps_end
```

در هر بار اجرای این تابع، مقدار اپسیلون کاهش پیدا خواهد کرد تا عامل خیلی حریصانه رفتار نکند. (دو خط آخر)

نوبت به تابع train می‌رسد.این تابع محیط و عامل را به عنوان ورودی گرفته، ۱۰۰۰۰۰ اپیسود مختلف را طی می‌کند تا عامل با محیط آشنا شود. در هر اپیسود، یک بار محیط با دستور env.reset ریست می‌شود تا حالت ابتدایی عامل که درایه صفرم خروجی این تابع است مشخص شود. سپس تا رسیدن به خانه مقصد، حرکاتش را با تابع choose_action انتخاب می‌کند. اینکه پس از هر عمل، حالت بعدی کدام خواهد بود و چه پاداشی دارد، با فراخوانی تابع env.step برای هر عمل، مشخص می‌شود. نهایتا مقادیر مشخص شده را به تابع learn می‌دهیم و حالت را به روز می‌کنیم.

```
      def train(env, agent, n_episodes=100000):


    for i in range(n_episodes):
        # TODO: implement the training loop for Q-learning
        done = False
        state = env.reset()[0]
        # Until the agent reaches the goal, keep training it
        while not done:
            # Choose the action with the highest value in the current state
            action = agent.choose_action(state)
            # Implement this action and move the agent in the desired
direction
            next_state, reward, done, _, _ = env.step(action)
            agent.learn(state, action, reward, next_state)
            # Update our current state
            state = next_state
```

محیط و عامل تعریف شده و تابع train را یک بار برایشان فراخواندیم تا QAgent ما محیط را در حالت is_slippery=False یاد بگیرد. حال با یک بار argmax گرفتن بر روی q-table ایجاد شده، سیاست بهینه را استخراج کردیم.

```
        array([2, 2, 1, 0, 1, 0, 1, 0, 2, 2, 1, 0, 0, 2, 2, 0],dtype=int64)
```

همینطور q-table به روز شده را چاپ کردیم.

```
           array([[0.447101  , 0.1981795 , 0.59049   , 0.44657874],
              [0.43522562, 0.        , 0.6561    , 0.4969679 ],
              [0.49017621, 0.729     , 0.34779662, 0.57369182],
              [0.54932219, 0.        , 0.06464323, 0.07053423],
              [0.03646578, 0.34795446, 0.        , 0.1034192 ],
              [0.        , 0.        , 0.        , 0.        ],
              [0.        , 0.81      , 0.        , 0.53493683],
              [0.        , 0.        , 0.        , 0.        ],
              [0.06828995, 0.        , 0.55598094, 0.03492287],
              [0.08336845, 0.18013486, 0.78047148, 0.        ],
              [0.51925757, 0.9       , 0.        , 0.60205992],
              [0.        , 0.        , 0.        , 0.        ],
              [0.        , 0.        , 0.        , 0.        ],
              [0.        , 0.09648051, 0.77284388, 0.10238245],
              [0.48017031, 0.77804729, 1.        , 0.66859993],
             [0.        , 0.        , 0.        , 0.        ]])
```

سیاست بدست آمده را بر روی محیط اجرا کردیم، نتیجه به شرح زیر است:

```
action: 2, state: 1, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 2, state: 2, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 1, state: 6, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 1, state: 10, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 1, state: 14, reward: 0.0, done: False, truncated: False, info:
{'prob': 1.0}
action: 2, state: 15, reward: 1.0, done: True, truncated: False, info:
{'prob': 1.0}
total reward: 1.0
```

شاهدیم که عامل از کوتاهترین مسیر ممکن به سمت جایی که پاداش برابر یک دارد رفته و از خانه های دارای q-value = 0 اجتناب می‌کند. ویدیوی Q_learning مربوط به این بخش است.

یک بار هم با `is_slippery=True` اجرا شد که عامل نتوانست به مقصد برسد. چون در این حالت ممکن است هیچ‌گاه عامل در جهتی که انتخاب می‌شود حرکت نکند، هیچ تضمینی مبنی بر اینکه بتوانیم به مقصد برسیم وجود ندارد. به علاوه چون زمان هم محدود است، احتمال رسیدن به مقصد کمتر هم خواهد شد.

```
action: 1, state: 4, reward: 0.0, done: False, truncated: False, info:
{'prob': 0.3333333333333333}
action: 1, state: 8, reward: 0.0, done: False, truncated: False, info:
{'prob': 0.3333333333333333}
action: 2, state: 4, reward: 0.0, done: False, truncated: False, info:
{'prob': 0.3333333333333333}
action: 1, state: 8, reward: 0.0, done: False, truncated: False, info:
{'prob': 0.3333333333333333}
action: 2, state: 12, reward: 0.0, done: True, truncated: False, info:
{'prob': 0.3333333333333333}
total reward: 0.0
```

ویدیوی Q_learning_True مربوط به این بخش است.