دانشگاه اصفهان
دانشکده مهندسی کامپیوتر

مستند پروژه سوم (هزارتو - یادگیری تقویتی)

درس مبانی و کاربردهای هوش مصنوعی

دکتر کارشناس

اعضای گروه 6:

امیرعلی لطفی (۴۰۰۳۶۱۳۰۵۳)

متین اعظمی (۴۰۰۳۶۲۳۰۰۳)

زهرا معصومی (۴۰۰۳۶۲۳۰۳۴)

پاییز ۱۴۰۲

یادگیری تقویتی

یادگیری تقویتی یکی از زیرشاخه های یادگیری ماشین است که در آن یک عامل یادگیرنده در تعامل با محیط سعی می‌کند به یک سیاست بهینه دست یابد. عامل یادگیرنده با مشاهده وضعیت سیستم (S)، کنش (A) را انتخاب می‌نماید. محیط، بازخورد این کنش را در قالب پاداش (R) و حالت بعدی سیستم به عامل بازمی‌گرداند. عامل مجددا با مشاهده پاداش و حالت سیستم، کنش بعدی را انتخاب می‌کند و این فرآیند تا زمان رسیدن به سیاست بهینه ادامه پیدا می‌کند.

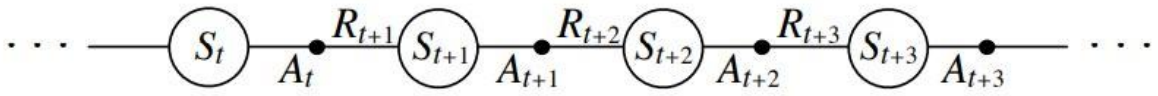
در یک دسته بندی کلی می‌توان الگوریتم‌های یادگیری تقویتی را به دو دسته الگوریتم‌های On-policy و الگوریتم‌های Off-policy تقسیم‌بندی نمود:

- الگوریتم‌های یادگیری تقویتی On-policy: در این الگوریتم‌ها تابع ارزش بر اساس سیاست و کنش فعلی عامل به‌روز می‌شود.
- الگوریتم‌های یادگیری تقویتی Off-policy: در این الگوریتم‌ها تابع ارزش مستقل از سیاست و کنش فعلی عامل به‌روز می‌شود.

الگوریتم سارسا SARSA یکی از معروف‌ترین الگوریتم‌های on-policy یادگیری تقویتی است و الگوریتم Q-learning نیز یکی از معروف‌ترین الگوریتم‌های off-policy یادگیری تقویتی است که در این پروژه استفاده شده‌اند.

الگوریتم سارسا (SARSA)

در این الگوریتم ابتدا عامل یادگیرنده با مشاهده حالت سیستم (S) و بر اساس سیاست مشخص، کنش (A) را انتخاب می‌کند. در ادامه بعد از انتخاب کنش، محیط حالت بعدی سیستم و پاداش را مشخص می‌کند. عامل با مشاهده وضعیت بعدی سیستم و پاداش دریافتی، مقدار تابع ارزش کنش (action-value function) را محاسبه و به‌روز می‌کند. این روند تا زمانی که مقدار تابع ارزش کنش به مقدار بهینه آن همگرا شود ادامه خواهد یافت.



الگوریتم Q-Learning

در الگوریتم یادگیری کیو (Q-Learning)، عامل یادگیرنده مشابه الگوریتم سارسا، بعد از مشاهده محیط کنشی (A) را انتخاب می‌کند. سپس محیط به عامل، حالت بعدی سیستم و پاداش حاصل از کنش انتخاب شده را بر می‌گرداند. عامل با مشاهده اطلاعات دریافتی از محیط، اقدام بعدی را انتخاب می‌کند و این فرآیند تا زمان رسیدن به سیاست بهینه ادامه پیدا می‌کند.

آماده‌سازی محیط

در ابتدا محیط بازی با دستور زیر ساخته شده و تعداد سطر، ستون و خانه‌های آن مشخص می‌شود:

```
env = gym.make("maze-random-10x10-plus-v0")
# Gets the size of the maze
COLUMN, ROW = env.maze_size
CELLS = COLUMN * ROW
```

توابع جانبی

1. تابع $plot\_success\_history$: برای رسم نمودار تعداد موفقیت‌ها (رسیدن به مقصد در هر 10,000 گام) در هر 20 اپیزود
2. تابع $plot\_average\_rewards$: برای رسم نمودار میانگین مجموع پاداش‌ها در هر 2500 اپیزود
3. تابع $draw\_values$: برای رسم نمودار گرافیکی مقادیر Q-Value هر خانه
4. تابع get_pi_v : برای به دست آوردن سیاست به دست آمده و مقادیر آن از جدول Q
5. تابع $draw\_policy$: برای به تصویر کشیدن نمودار سیاست، رسم جهتی که در هر خانه بر اساس آن سیاست به دست آمده است.
6. تابع map_state : برای تبدیل مختصات هر خانه به شماره آن خانه
7. تابع get_col_row : برای تبدیل شماره هر خانه به مختصات آن خانه

کلاس Agent

کلاس عامل ویژگی‌های زیر را دارد:

```
class Agent():
    def __init__(
        self,
        env,
        NUM_TRAIN_STEP = (1000000, 1000000),
        NUM_TEST_STEP = (100000, 100000),
        alpha = (0.09, 0.09),
        gamma = (0.99,0.99)
    ):
        self.env = env
        self.NUM_TRAIN_STEP = NUM_TRAIN_STEP
        self.NUM_TEST_STEP = NUM_TEST_STEP
        self.alpha = alpha
        self.gamma = gamma
```

توضیح ویژگی‌های کلاس:

- env: محیطی که عامل در آن فعالیت می‌کند.
- NUM_TRAIN_STEP: نشان‌دهنده تعداد گام‌هایی که آموزش از طریق trade off بین exploitation و exploration طول می‌کشد است. این ویژگی دو مقدار دارد که خانه اول آن برای الگوریتم یادگیری Q و خانه دوم آن برای الگوریتم SARSA استفاده می‌شود.
- NUM_TEST_STEP: نشان‌دهنده تعداد گام‌هایی که عامل بدون احتمال و غیر تصادفی در محیط کنش انجام می‌دهد است. این ویژگی نیز مشابه مورد قبل، دو مقدار برای دو الگوریتم دارد.
- alpha: نرخ یادگیری (Learning Rate) در آموزش مدل
- gamma: ضریب تخفیف (Discount Factor) عامل

همچنین در این کلاس توابع زیر را تعریف می‌کنیم:

1. تابع select_action

برای انتخاب عمل در هر شرایط، این تابع به روش $epsilon - greedy$ در هر لحظه یک حرکت را (در بازه 0 تا 3 که همان جهت‌های جغرافیایی هستند و در ابتدای کد تعریف شده‌اند) انتخاب می‌کند:

```
def select_action(self, Q, s, epsilon):
        if(random.random() < epsilon):
            return random.randint(0, 3)
        else:
            action = 0
            for i in range(4):
                if Q[s][i] > Q[s][action]:
                    action = i

            return action
```

عملکرد این روش به این صورت است که به احتمال ε یک حرکت تصادفی و به احتمال $1 - \varepsilon$ حرکتی که بیشینه پاداش مورد انتظار در آینده را خواهد داشت (در جدول Q ذخیره شده)، با توجه به حالت داده شده انتخاب کرده و برمی‌گرداند.

علت انتخاب حرکت تصادفی این است که به هر کنشی شانس انتخاب شدن بدهیم تا تعادل بین Exploration و Exploitation برقرار شود و کنشی که ممکن است در آینده بهتر از کنش‌های کنونی عمل کند نیز احتمال انتخاب شدن داشته باشد. در بقیه حالات نیز با توجه به سیاست حریصانه، بهترین کنش ممکن را انتخاب می‌کنیم.

لازم به ذکر است که مقدار ε در ابتدای یادگیری مقداری نزدیک به 1 (کنش کاملاً تصادفی) و در انتها مقداری نزدیک به 0 (بهترین کنش) دارد. این متغیر تعادل بین جستجو و بهره‌برداری را تنظیم می‌کند.

---

2. تابع q_learning

در این تابع ابتدا از طریق ویژگی‌های کلاس، مقادیر تعداد گام‌های آموزشی و آزمایشی را در متغیرهای مربوطه ذخیره می‌کنیم و alpha (نرخ یادگیری) و gamma (ضریب تخفیف) را تعیین می‌کنیم. (مقدار این دو ضریب برای الگوریتم اول (Q-Learning) در ویژگی‌های کلاس و در خانه صفرم آن‌ها تعیین شده است.)

سپس Q که جدول ارزش‌های خانه‌ها است را با صفر مقداردهی اولیه می‌کنیم.

مقادیر اولیه state را با حالت اولیه محیط، مقدار success (تعداد موفقیت‌ها در رسیدن به مقصد) را با صفر، مقدار EPISODE (نشان‌دهنده شماره اپیزود) را با صفر و epsilon را با مقدار اولیه 1 تنظیم می‌کنیم. (علت این مقدار و در نتیجه کنش تصادفی ناشی از آن، در توضیح تابع select_action و در ادامه نیز ذکر شده است.)

بقیه متغیرها (success_per_part و success_history و total_rewards و episode_total_reward) برای ثبت اطلاعات و رسم نمودارها تعریف شده‌اند.

```
def q_learning(self):
    NUM_TRAIN_STEP = self.NUM_TRAIN_STEP[0]
    NUM_TEST_STEP = self.NUM_TEST_STEP[0]

    alpha  = self.alpha[0]
    gamma = self.gamma[0]

    Q = {}
    for i in range(CELLS):
        Q[i] = [0 for j in range(4)]

     state = self.env.reset()
     success = 0
     success_per_part = 0
     EPISODE = 0
     epsilon = 1

     success_history = []
     total_rewards = []
     episode_total_reward = 0
```

الگوریتم Q-Learning را با این مقادیر اولیه شروع می‌کنیم. این فرآیند تکراری شامل یادگیری عامل با کاوش (Exploration) در محیط و به‌روزرسانی مدل در ادامه کاوش است. اجزای مورد نیاز یادگیری Q شامل موارد زیر است:

- **عامل:** موجودی که در محیط عمل می‌کند یا از آن اطلاعاتی کسب می‌کند. در این مسئله، محیط ما به عنوان env در کلاس تعریف شده است و عامل با صدا زدن توابع آن، کنشی روی آن انجام داده و حالت بعدی را از آن دریافت می‌کند.
- **حالت‌ها:** حالت متغیری است که موقعیت فعلی را در محیط یک عامل مشخص می کند. حالت‌ها در این مسئله یک خانه از جدول هزارتو هستند و در محیط تعریف شده‌اند. با انجام هر کنش، می‌توانیم به حالت بعدی دسترسی داشته باشیم. (`state` و `next_state`: این متغیرها مختصات هر حالت را با دو عدد نشان می‌دهند.)
- **کنش‌ها:** عمل عامل در زمانی که در یک حالت خاص باشد. در این کد ما کنش در هر گام را در متغیر `action` ذخیره می‌کنیم که یک عدد بین 0 تا 3 (جهات جغرافیایی تعریف شده در ابتدای کد) است.
- **پاداش:** بازخوردی است که از طرف محیط به عامل داده می‌شود تا کنشی که انجام داده ارزیابی شود. در این کد ما پاداش را با `reward` نمایش داده‌ایم.
- **اپیزودها:** هر اپیزود شامل حرکاتی از ابتدا تا رسیدن به مقصد یا timeout شدن است.
- **نرخ یادگیری** α: در کد با `alpha` نشان داده شده است.
- **ضریب تخفیف** γ: در کد با `gamma` نشان داده شده است.
- **مقادیر Q-value:** Q معیاری است که برای اندازه‌گیری کیفیت یک عمل در یک حالت خاص استفاده می‌شود و پیش‌بینی‌ای از بیشینه پاداشی که می‌توان از این حرکت در آینده به دست آید، به ما می‌دهد. در این پروژه، در جدول Q این ارزش‌ها را به ازای هر حالت و کنش‌های ممکن، ذخیره می‌کنیم.

مدل‌های Q-Learning از طریق تجربیات آزمون و خطا کار می‌کنند تا رفتار بهینه برای یک کار را یاد بگیرند.

برای انجام این کار، ابتدا تمام مقادیر Q را در جدولی ذخیره می کنیم که در هر مرحله با استفاده از تکرار Q-Learning آن را به‌روز می‌کنیم.

در ابتدا، عامل هیچ ایده‌ای از محیط ندارد. او بیشتر احتمال دارد چیزهای جدید را کشف کند (به دلیل `epsilon=1`، و در نتیجه حتماً کنش تصادفی) تا این که از دانش خود بهره ببرد؛ چون در ابتدا خودش دانشی ندارد. برای هر اپیزود مراحل زیر را تکرار می‌کنیم:

```
for step in range(NUM_TRAIN_STEP + NUM_TEST_STEP):
    mapped_state = map_state(state)

    action = self.select_action(Q, mapped_state, epsilon)
    next_state, reward, done, truncated = env.step(action)
    mapped_next_state = map_state(next_state)
```

در ابتدا برای راحت‌تر شدن محاسبات و شهود بهتر، state (که در آغاز با حالت اولیه محیط (0,0) مقداردهی شده) را از حالت مختصاتی به شماره خانه تبدیل می‌کنیم. این کار با تابع map_state انجام شده و نتیجه آن در mapped_state ذخیره می‌شود.
سپس با توجه به ارزش‌های کنونی، حالتی که در آن قرار داریم و مقدار epsilon یک حرکت را انتخاب می‌کنیم. این کار با استفاده از تابع select_action انجام می‌شود که قبلاً عملکرد آن را شرح داده‌ایم.
حرکت انتخاب شده با تابع step روی محیط اعمال می‌شود و تابع اطلاعات حالت بعدی، پاداش دریافت شده در ازای کنش و بررسی حالات پایانی (رسیدن به مقصد (done) یا ارور (truncated)) را برمی‌گرداند. مانند حالت قبل، next_state را هم از حالت مختصاتی به شماره خانه تبدیل می‌کنیم.

می‌دانیم از طریق مراحل تکرار، عامل اطلاعات بیشتر و بیشتری در مورد نحوه عملکرد محیط به دست می آورد و پس از تعدادی تکرار، احتمال بیشتری دارد که از دانش خود بهره ببرد تا کاوش چیزهای جدید در محیط. به همین دلیل، epsilon را به‌صورت خطی در طول گام‌های آموزش کاهش می‌دهیم.

```
# Linear Decay of epsilon
epsilon -= (1 / NUM_TRAIN_STEP)
```

هدف این الگوریتم در واقع یادگیری تکراری Q-value بهینه با استفاده از معادله بلمن است.

Bellman's equation: $Q(s,a) = Q(s,a) + \alpha\,.\,(reward + \gamma\,.\,max(Q(s',a')) - Q(s,a))$

در این معادله، $Q(s,\ a)$ نشان دهنده پاداش مورد انتظار برای انجام کنش a در حالت s است. α نرخ یادگیری و γ ضریب تخفیف است. بیشینه پاداش مورد انتظار برای تمام کنش‌های ممکن a' در حالت s' با $max\ (Q(s',\ a'))$ نشان داده می‌شود.

پس با استفاده از مقادیر مشاهده شده، این فرمول را در کد به شکل زیر پیاده‌سازی و $Q(s, a)$ را به‌روزرسانی کرده‌ایم:

```
Q[mapped_state][action] += alpha *
 (reward + gamma * max(Q[mapped_next_state])-Q[mapped_state][action])
```

پس از محاسبه ارزش، اطلاعات آماری مورد نیاز را ثبت می‌کنیم و به حالت بعدی می‌رویم:

```
episode_total_reward += reward if mapped_next_state != CELLS - 1 else 1

if mapped_next_state == CELLS - 1:
    if step > NUM_TRAIN_STEP:
        success += 1

        success_per_part += 1
        total_rewards.append(episode_total_reward)
        episode_total_reward = 0

if step % 10000 == 0:
    success_history.append(success_per_part)
    success_per_part = 0

state = next_state
```

حالا بررسی می‌کنیم اگر به حالت پایانی رسیده بودیم یا خطایی وجود داشت، و تعداد گام‌های لازم برای یک اپیزود طی شده بود، به تعداد اپیزودها یکی اضافه می‌کنیم، محیط را به حالت اولیه برمی‌گردانیم و حلقه تکرار الگوریتم Q-Learning به پایان می‌رسد:

```
if done or truncated:
    if step > NUM_TRAIN_STEP:
        EPISODE += 1
    observation = self.env.reset()
    state = observation
```

در پایان تابع نیز، محیط را نمایش داده و آن را به پایان می‌رسانیم، اطلاعات آماری اجرای تابع را در خروجی چاپ می‌کنیم و جدول ارزش‌ها و اطلاعات مورد نیاز را برمی‌گردانیم:

```
self.env.render()
time.sleep(5)
self.env.close()

print(f"{EPISODE=}")
print(f"RATE WIN = {success / EPISODE * 100} ")
print(f"AVG STEP for Win = {NUM_TEST_STEP / success}")
print(f"CONVERGED REWARD = {total_rewards[-1]}")

return Q, success_history, total_rewards
```

در پایان این مستند، نتایج اجرای الگوریتم و نمودارهای آن رسم شده است.

---

3. تابع SARSA

SARSA یک روش مبتنی بر سیاست محسوب می‌شود که در آن ارزش‌ها را براساس عمل کنونی a که از سیاست کنونی آن مشتق شده می‌آموزد. تفاوت عمده‌ای که الگوریتم SARSA با الگوریتم Q-learning دارد این است که برای محاسبه پاداش حالت‌های بعدی، نیازی به داشتن تمام Q-table نیست.
در این تابع نیز مانند تابع قبلی، متغیرهای مورد نیاز را مقداردهی اولیه می‌کنیم. تعداد گام‌ها برای این الگوریتم، در کلاس Agent و در خانه‌های یکم ویژگی‌های NUM_TRAIN_STEP و NUM_TEST_STEP و alpha و gamma قرار گرفته است. فرآیند SARSA با مقداردهی اولیه $Q(S,\ A)$ به مقادیر دلخواه شروع می‌شود. ما این مقادیر را صفر قرار دادیم:

```
def SARSA(self):
    NUM_TRAIN_STEP = self.NUM_TRAIN_STEP[1]
    NUM_TEST_STEP = self.NUM_TEST_STEP[1]

    alpha  = self.alpha[1]
    gamma = self.gamma[1]

    Q = {}
    for i in range(CELLS):
        Q[i] = [0 for j in range(4)]

    success = 0
    success_per_part = 0
    EPISODE = 0
    epsilon = 1

    success_history = []
    total_rewards = []
    episode_total_reward = 0
```

مانند الگوریتم قبلی، مقدار epsilon برابر یک (برای حرکت‌های اولیه‌ی کاملاً تصادفی) و بقیه متغیرها برای اندازه‌گیری و مشاهده عملکرد الگوریتم تعریف و مقداردهی اولیه می‌شوند.

در این الگوریتم ابتدا عامل یادگیرنده با مشاهده حالت سیستم (S) و بر اساس سیاست مشخص، کنش (A) را انتخاب می‌کند. در ادامه بعد از انتخاب کنش، محیط حالت بعدی سیستم و پاداش را مشخص می‌کند. عامل با مشاهده وضعیت بعدی سیستم و پاداش دریافتی، مقدار تابع ارزش کنش

(action-value function) را محاسبه و به‌روز می‌کند. این روند تا زمانی که مقدار تابع ارزش کنش به مقدار بهینه آن همگرا شود ادامه خواهد یافت.
در این مرحله، وضعیت فعلی اولیه (S) تنظیم می شود و کنش اولیه (action) با استفاده از الگوریتم epsilon-greedy بر اساس مقادیر Q فعلی انتخاب می‌شود. (الگوریتم اپسیلون حریصانه که در تابع `select_action` پیاده‌سازی شده و قبلاً توضیح داده‌ایم، استفاده از روش‌های بهره‌برداری و اکتشاف را در فرآیند یادگیری متعادل می‌کند تا کنشی با بالاترین پاداش تخمینی انتخاب شود).

```
state = self.env.reset()
mapped_state = map_state(state)
action = self.select_action(Q, mapped_state, epsilon)
```

حالا تکرار الگوریتم SARSA برای یافتن سیاست بهینه را شروع می‌کنیم:

```
for step in range(NUM_TRAIN_STEP + NUM_TEST_STEP):
    mapped_state = map_state(state)
    next_state, reward, done, truncated = env.step(action)
    mapped_next_state = map_state(next_state)

    next_action = self.select_action(Q, mapped_next_state, epsilon)
```

در این الگوریتم با توجه به سیاست و حالت فعلی و کنش انجام شده، ارزش را به روزرسانی می‌کنیم:

$$Q^{new}(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1})\right]$$

این فرمول را به صورت زیر پیاده‌سازی کرده‌ایم:

```
# learn
Q[mapped_state][action] = (1-alpha) * (Q[mapped_state][action]) +
            alpha*(reward + gamma*Q[mapped_next_state][next_action])
```

ثبت اطلاعات آماری برای رسم نمودارها (مشابه قسمت قبل):

```
episode_total_reward += reward if mapped_next_state != CELLS - 1 else 1
```

```
if mapped_next_state == CELLS - 1:
    if step > NUM_TRAIN_STEP:
        success += 1

    total_rewards.append(episode_total_reward)
    episode_total_reward = 0
    success_per_part += 1

if step % 10000 == 0:
    success_history.append(success_per_part)
    success_per_part = 0
```

کاهش خطی مقدار epsilon (علت آن قبلاً ذکر شده) و تبدیل حالت و اکشن فعلی به حالت و اکشن بعدی:

```
# Linear Decay of epsilon
epsilon -= (1 / NUM_TRAIN_STEP)

state = next_state
action = next_action
```

بررسی پایان اپیزود و شروع مجدد، در صورتی که اپیزود جدید داشته باشیم (یعنی برگشت به حالت اولیه در محیط و انتخاب یک کنش با الگوریتم اپسیلون حریصانه) و پایان حلقه تکرار الگوریتم:

```
if done or truncated:
    if step > NUM_TRAIN_STEP:
        EPISODE += 1

    observation = self.env.reset()
    state = observation
    mapped_state = map_state(state)
    action = self.select_action(Q, mapped_state, epsilon)
```

نمایش گرافیکی محیط، بستن آن و چاپ اطلاعات اجرای الگوریتم در خروجی و پایان تابع با بازگرداندن متغیرهای مورد نیاز:

```
self.env.render()
time.sleep(5)
self.env.close()

print(f"{EPISODE=}")
print(f"RATE WIN = {success / EPISODE * 100} ")
print(f"AVG STEP for Win = {NUM_TEST_STEP / success}")
print(f"CONVERGED REWARD = {total_rewards[-1]}")

return Q, success_history, total_rewards
```

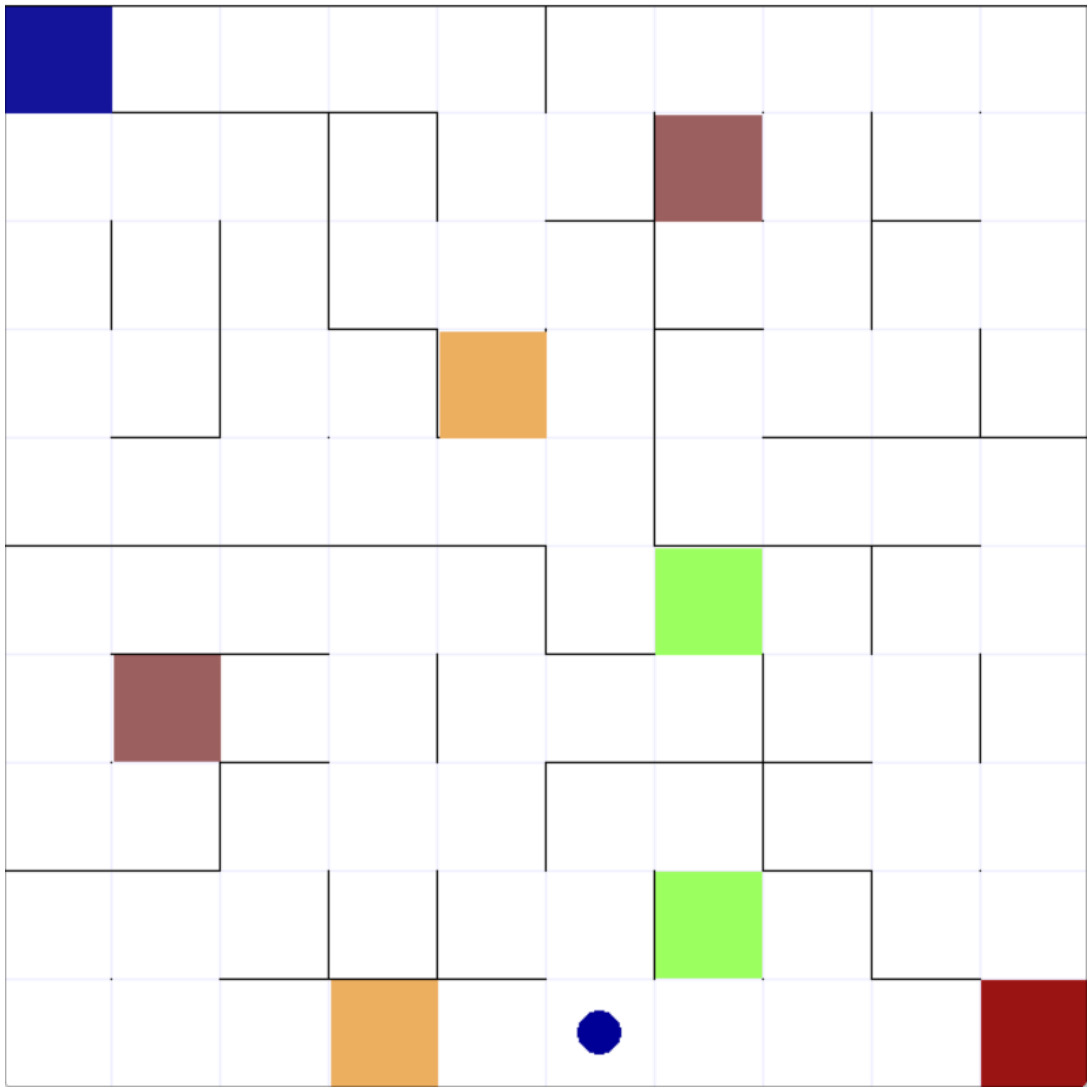
در پایان این مستند، نتایج اجرای الگوریتم و نمودارهای آن رسم شده است.

اجرای الگوریتم‌ها

در نهایت یک نمونه از Agent ساخته و توابع را اجرا می‌کنیم:

```
agent = Agent(env, NUM_TRAIN_STEP=(1000000, 1000000))
Q_SARSA, history_SARSA, total_rewards_SARSA = agent.SARSA()
print('-----------------------------------------')
Q_Q_Learning, history_Q, total_rewards_Q_Learning =
agent.q_learning()
```

نتیجه یک اجرا بر روی هزارتوی تولید شده زیر:



خروجی‌های چاپ شده حاصل از اجرای هر دو الگوریتم:

```
EPISODE=3617
RATE WIN = 100.0
AVG STEP for Win = 27.647221454243848
CONVERGED REWARD = 0.976
-----------------------------------------
EPISODE=3577
RATE WIN = 100.0
AVG STEP for Win = 27.95638803466592
CONVERGED REWARD = 0.977
```

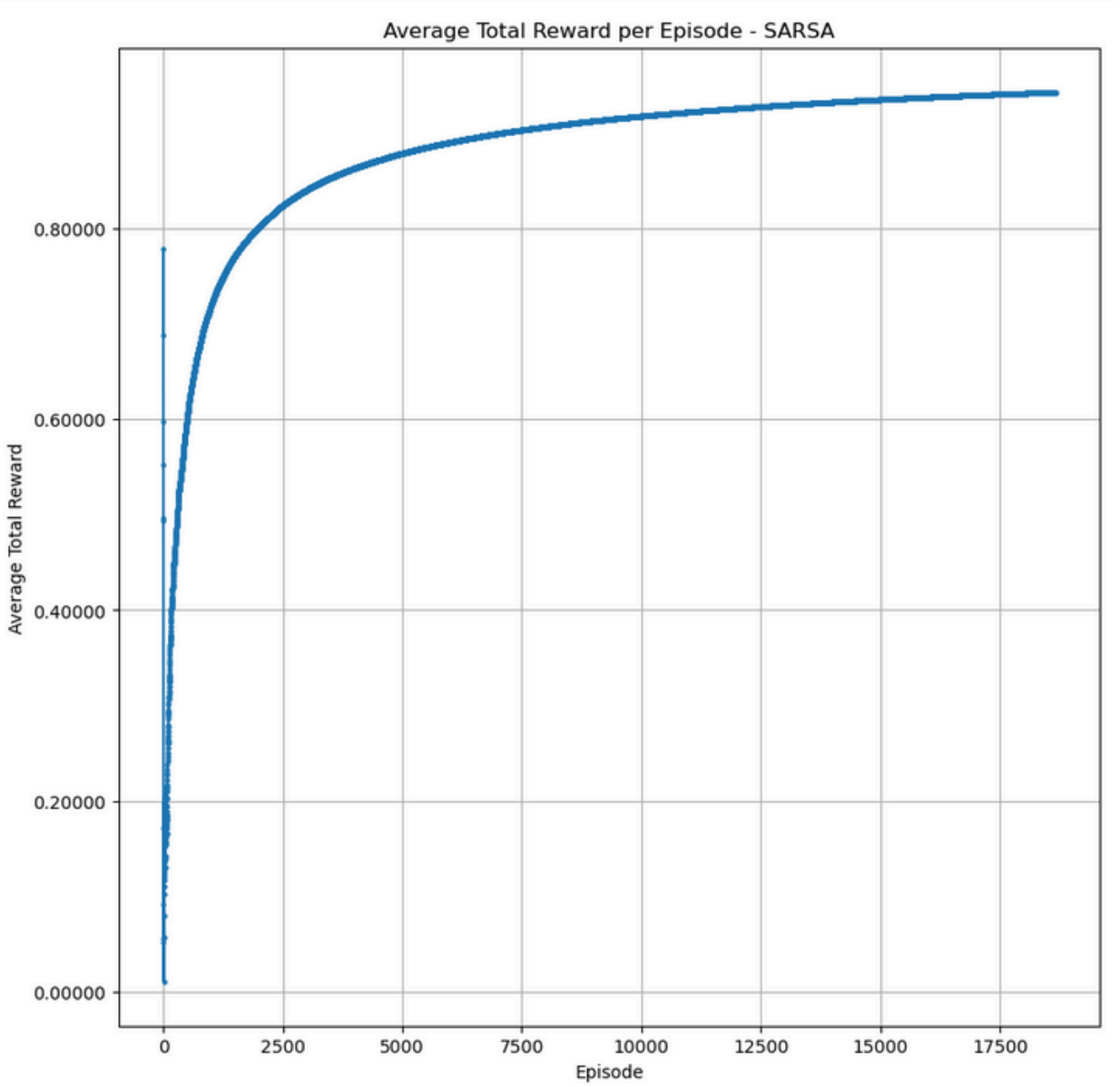
همانطور که مشاهده می‌کنیم، در هر دو الگوریتم بسیار نزدیک به هم عمل می‌کنند.
مقادیر RATE WIN و AVG STEP for Win برای بازه‌ی تست محاسبه شده‌اند. حدودا هر دو الگوریتم به طور متوسط با ۲۷ مرحله به حالت هدف می‌رسند و در ۱۰۰ درصد اپیزودها عامل به هدف می‌رسد.

نتیجه اجرای الگوریتم SARSA

با توابع کمکی ذکر شده در قسمت اول، نمودارهای مربوط به الگوریتم SARSA را رسم می‌کنیم:

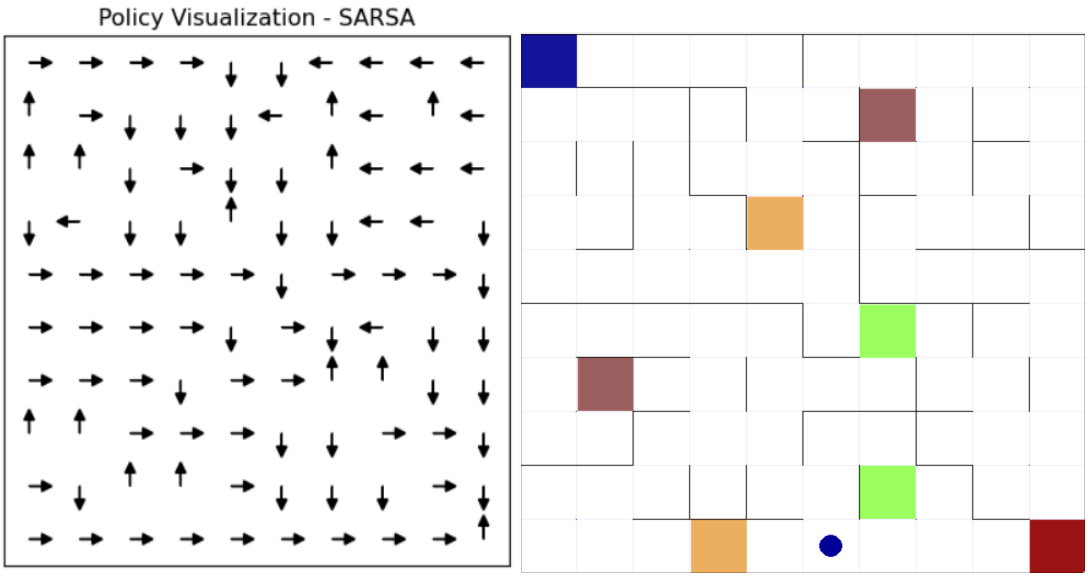
```
algorithm = "SARSA"
pi_SARSA, V_SARSA = get_pi_v(Q_SARSA)
draw_policy(pi_SARSA, title=algorithm)
draw_values(V_SARSA, title=algorithm)
plot_average_rewards(total_rewards_SARSA, title=algorithm)
plot_success_history(history_SARSA, title=algorithm)
```

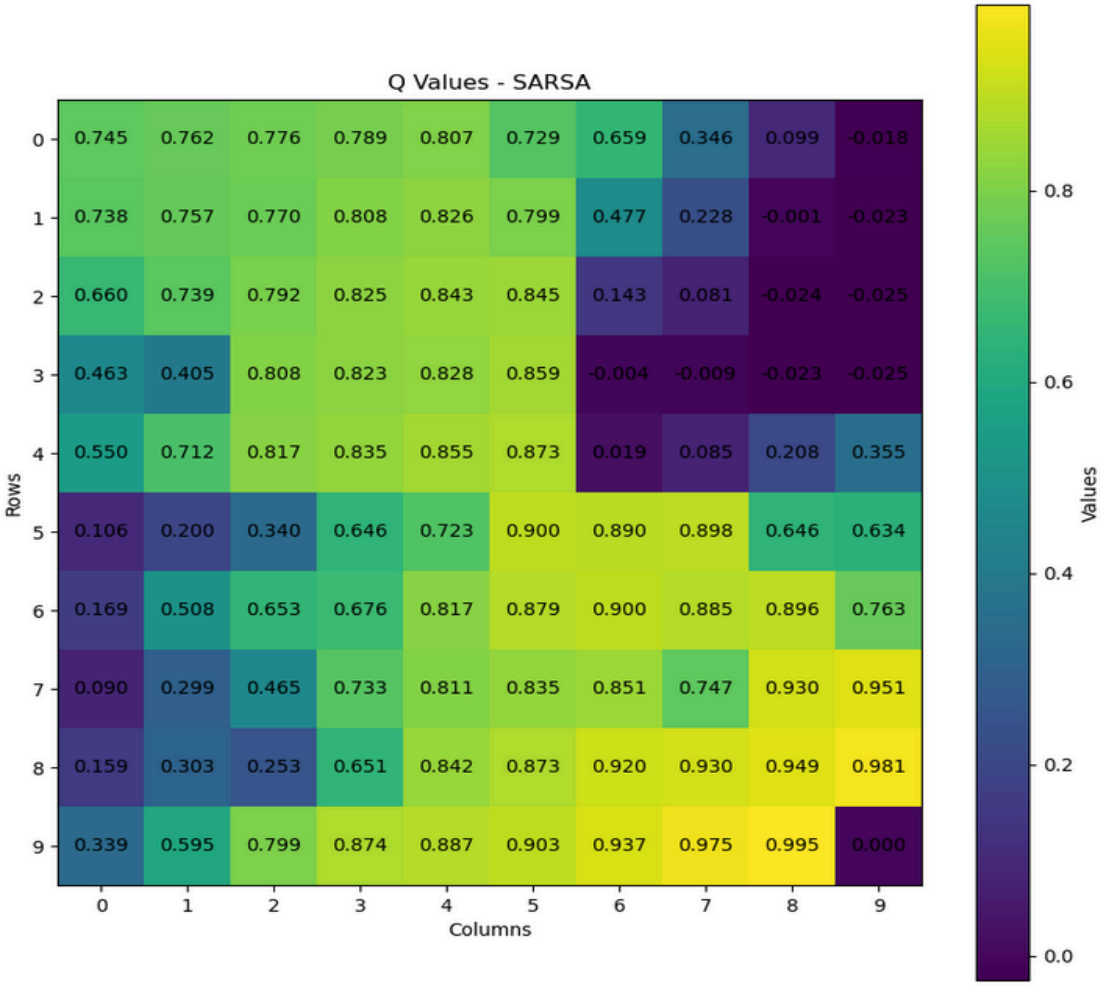
نمودار میانگین پاداش‌ها در هر 2500 اپیزود:



نمودار تعداد موفقیت‌ها (در هر 10,000 گام) در هر 20 اپیزود:

نمودارهای سیاست:



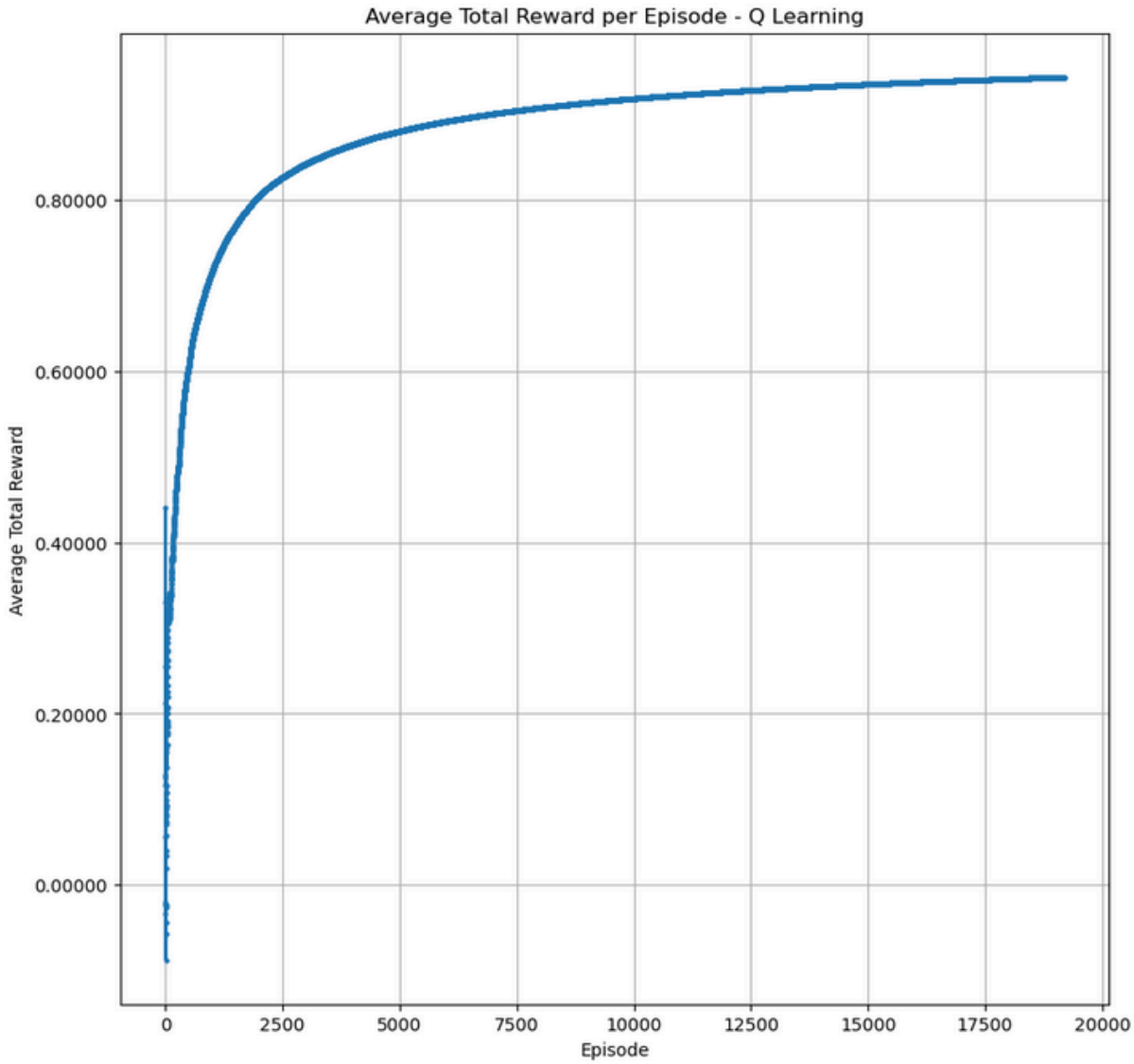


نتیجه اجرای الگوریتم Q-Learning

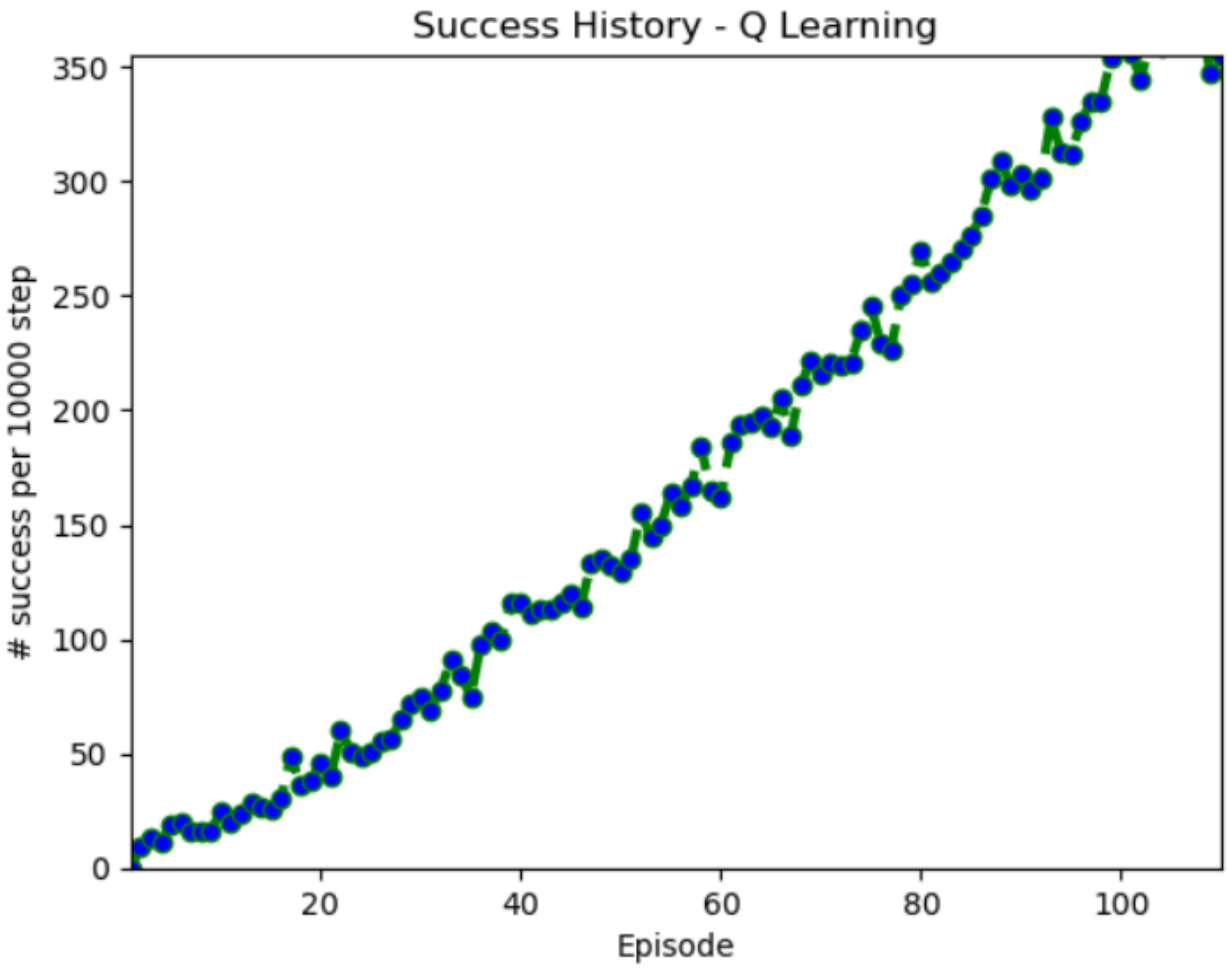
مشابه نتایج و نمودارهای بالا، برای الگوریتم Q-Learning:

```
algorithm = "Q Learning"
pi_Q_learning, V_Q_learning = get_pi_v(Q_Q_Learning)
draw_policy(pi_Q_learning, title=algorithm)
draw_values(V_Q_learning, title=algorithm)
plot_average_rewards(total_rewards_Q_Learning, title=algorithm)
plot_success_history(history_Q, title=algorithm)
```

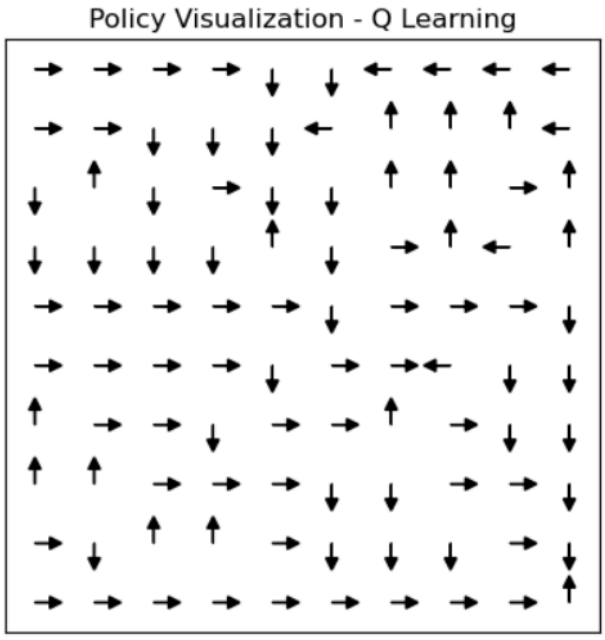
نمودار میانگین پاداش‌ها در هر 2500 اپیزود:

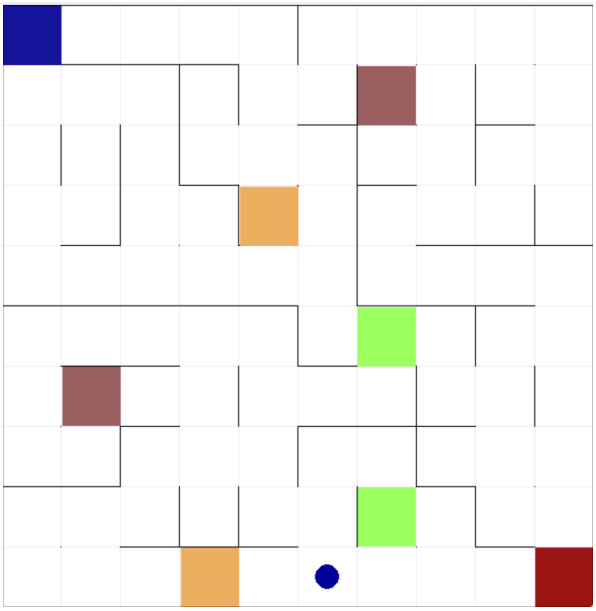


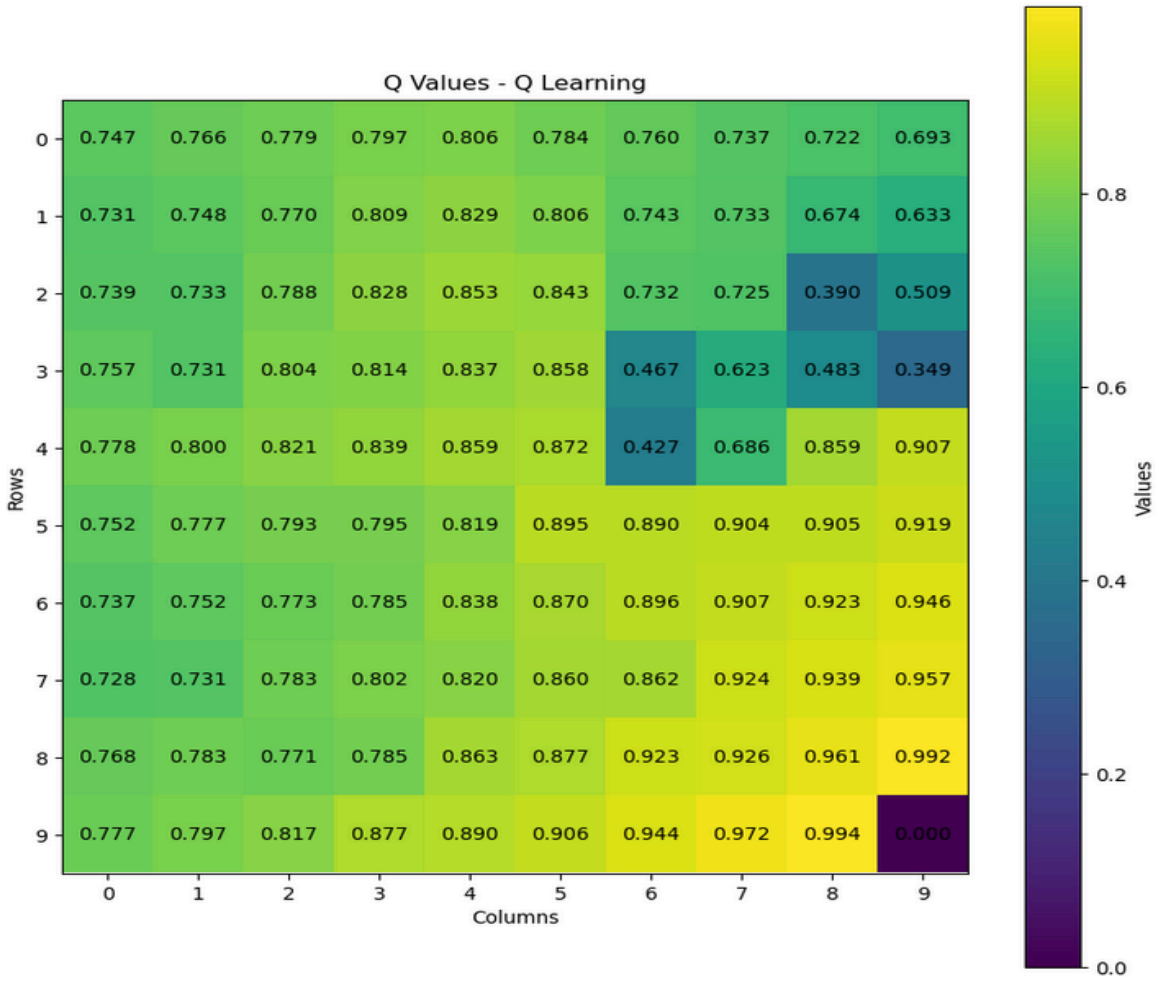
نمودار تعداد موفقیت‌ها (در هر 10,000 گام) در هر 20 اپیزود:



نمودارهای سیاست:







مقایسه الگوریتم‌ها

الگوریتم Q-Learning بر خلاف الگوریتم سارسا، یک الگوریتم Off-policy است که این موضوع در فرمول ارائه شده برای به روزرسانی مقدار ارزش – کنش $Q(s,a)$ مشخص است. مقدار به‌روزرسانی بر اساس بیشترین مقدار ارزش-اقدام انجام می‌گیرد $(Max\ Q(s',a))$ و نه بر اساس $Q(s',a)$.

تفاوت عمده‌ای که الگوریتم SARSA با الگوریتم Q-learning دارد این است که برای محاسبه پاداش حالت‌های بعدی، نیازی به داشتن تمام Q-table نیست.

بنابراین، الگوریتم SARSA مقدار Q-value را با توجه به اقدامی که ناشی از سیاست فعلی است محاسبه می‌کند نه اقدام ناشی از سیاست حریصانه.

1. **Q-Learning:**
$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1},a) - Q(s_t,a_t))$$
2. **SARSA:**
$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t))$$

در برخی از مقالات اشاره شده است که الگوریتم SARSA سرعت هم‌گرایی بیشتری نسبت به الگوریتم Q-learning دارد. همچنین در الگوریتم سارسا پردازش کمتری نسبت به الگوریتم یادگیری Q احتیاج است. البته بیان شده است که در صورتی که نیاز است تا در زمان کم و با هزینه کمتری سیاست بهینه به دست آید (مثلاً برنامه ریزی یک ربات در محیط واقعی)، بهتر است از الگوریتم SARSA استفاده شود. در غیر این‌صورت و در صورتی که یک مدل شبیه‌سازی از سیستم وجود دارد و تعداد تکرار بالا هزینه‌ای را ایجاد نمی‌کند، الگوریتم Q-learning مناسب‌تر است.

همانطور که در نتایج اجرا و نمودارها نیز نشان داده شده است، سرعت هم‌گرایی SARSA بهتر است. ولی در برخی از مقالات بیان شده است که Q-Learning تخمین بهتری از سیاست بهینه به دست می‌آورد.

کتابخانه‌های مورد استفاده

- numpy
- gym==0.23.1
- gym_maze
- matplotlib.pyplot, matplotlib.ticker
- time, random

منابع

- دوره هوش‌مصنوعی دانشگاه شریف (دکتر رهبان)
- دوره Reinforcement Learning کورسرا (Andrew NG)
- درس هوش مصنوعی دانشگاه برکلی (CS 188 Fall 2022)
- Towardsdatascience
- Builtin
- Techtarget
- Baeldung
- Faradars
- Shabihpardazan
- ChatGPT (برای تابع‌های ترسیم نمودار)