# گزارش مینی پروژه دوم درس یادگیری تقویتی

**مریم واقعی**

دانشگاه فردوسی مشهد، دانشکده مهندسی

ma.vaghei78@gmail.com

اطلاعات گزارش

تاریخ:

۲۳ خرداد ۱۴۰۲

واژگان کلیدی:

Gym

Q-Learning

MC-First Visit

Sarsa

Off-policy

On-policy

Gamma

نرخ یادگیری

سیاست بهینه

سیاست هدف

سیاست رفتار

چکیده

در این پروژه، ما از محیط Taxi در کتابخانه Gym استفاده می‌کنیم تا الگوریتم‌های Q-Learning، MC First Visit و Sarsa را پیاده سازی کنیم. در ابتدا، ۴ اکشن مورب را به کشن‌های موجود اضافه می‌کنیم و سپس با استفاده از این الگوریتم‌ها، مقادیر ارزش و نمودار پاداش تجمعی را با استفاده از مقادیر گاما ۰ و ۰.۹ محاسبه و نمایش می‌دهیم.

فهرست مطالب

## ۱-مقدمه

یادگیری تقویتی یک حوزه پژوهشی مهم در علوم کامپیوتر است که به تلاش برای تعیین بهترین راه‌حل برای مسئله‌ها با تجربه و تعامل بین عامل و محیط می‌پردازد. یکی از محیط‌های معروف مورد استفاده در این حوزه، محیط Taxi می‌باشد که توسط کتابخانه Gym ارائه می‌شود.

در این گزارش، ابتدا به توضیحاتی درباره محیط Taxi و اکشن‌های موجود در آن می‌پردازیم. سپس به اضافه کردن ۴ اکشن مورب به اکشن‌های موجود می‌پردازیم. سپس، الگوریتم Q-Learning را پیاده‌سازی می‌کنیم و مقادیر ارزش را برای هر وضعیت و اکشن محاسبه می‌کنیم. همچنین نمودار پاداش تجمعی را با استفاده از مقادیر گاما ۰ و ۰.۹ رسم می‌کنیم.

همچنین الگوریتم‌های MC First Visit و Sarsa را نیز پیاده‌سازی کرده و مقادیر ارزش و نمودار پاداش تجمعی را برای آن‌ها نمایش می‌دهیم. در نهایت، نتایج و نتیجه‌گیری از پروژه را بیان می‌کنیم و ایده‌ها و پیشنهاداتی برای تحقیقات آینده در این حوزه ارائه می‌کنیم.

## ۲- افزودن حرکت های مورب تاکسی

برای تعریف اکشن های مورب، به این صورت عمل کردیم که ابتدا چک میکنیم اگر اکشن، یکی از اکشن های موجود یعنی پایین، بالا، راست، چپ، سوار کردن و پیاده کردن مسافر باشد، آنگاه همان تابع step() مربوط به env را صدا میزنیم که مشابه گذشته عمل مورد نظر را اجرا کند.

اما اگر اکشن ما یکی از اعداد ۶ (بالا-چپ)، ۷ (بالا-راست)، ۸ (پایین-راست) و ۹ (پایین-چپ) باشد به این صورت عمل میکنیم که ابتدا چک میکنیم آن سمت تاکسی آزاد باشد و مانعی نباشد، به عبارت دیگر:

برای بالا-چپ، موانع سمت چپ و بالا-چپ را چک میکنیم و اگر هر دو مانع قرار داشته باشند امکان انجام اکشن بالا-چپ نیست و لازم است حداقل یکی از این دو خانه از نقشه آزاد باشد تا امکان انجام این اکشن باشد (با توجه به ابهام در مسئله این فرض را در نظر گرفتم زیرا اگر هر دو مانع حضور داشته باشند و امکان انجام این اکشن باشد عملا تاکسی از روی موانع پریده است که خلاف واقع می باشد!).

برای بالا-راست، موانع سمت راست و بالا-راست را چک میکنیم و اگر هر دو مانع قرار داشته باشند امکان انجام اکشن بالا-راست نیست.

برای پایین-راست، موانع سمت راست و پایین-راست را چک میکنیم و اگر هر دو مانع قرار داشته باشند امکان انجام اکشن پایین-راست نیست.

در نهایت، برای پایین-چپ، موانع سمت چپ و پایین-چپ را چک میکنیم و اگر هر دو مانع قرار داشته باشند امکان انجام اکشن پایین-چپ نیست.

در نهایت با تغییر موقعیت تاکسی (با توجه به شروط گفته شده)، وضعیت جدید تاکسی تعیین میشود و به ازای این اکشن ها reward منفی ۱ در نظر میگیریم، از طرفی چون این عمل ها، پیاده کردن نیستند لذا میدانیم که به هدف نرسیده ایم بنابراین done = False می باشد.

```python
def step(self, action):
    if action in range(6):
        return self.env.step(action)
    taxi_row, taxi_col, pass_loc, dest_idx = self.decode(self.env.s)
    next_state = self.encode(taxi_row, taxi_col, pass_loc, dest_idx)

    # check if there is obstacles on left, top-left and down-left of the taxi
    top_left_free = self.desc[taxi_row, 2 * taxi_col] == b":"
    left_free = self.desc[1 + taxi_row, 2 * taxi_col] == b":"
    down_left_free = self.desc[2 + taxi_row, 2 * taxi_col] == b":"

    # check if there is obstacles on right, top-right and down-right of the taxi
    top_right_free = self.desc[taxi_row, 2 * taxi_col + 2] == b":"
    right_free = self.desc[1 + taxi_row, 2 * taxi_col + 2] == b":"
    down_right_free = self.desc[2 + taxi_row, 2 * taxi_col + 2] == b":"

    if action == 6 and (top_left_free or left_free):  # Diagonal action: top-left
        next_state = self.encode(max(taxi_row - 1, 0), max(taxi_col - 1, 0), pass_loc, dest_idx)
        # print(action, next_state, max(taxi_row - 1, 0), max(taxi_col - 1, 0), pass_loc, dest_idx)
    elif action == 7 and (top_right_free or right_free):  # Diagonal action: top-right
        next_state = self.encode(max(taxi_row - 1, 0), min(taxi_col + 1, self.num_cols - 1), pass_loc, dest_idx)
        # print(action, next_state, max(taxi_row - 1, 0), min(taxi_col + 1, self.num_cols - 1), pass_loc, dest_idx)
    elif action == 8 and (left_free or down_left_free):  # Diagonal action: down-left
        next_state = self.encode(min(taxi_row + 1, self.num_rows - 1), max(taxi_col - 1, 0), pass_loc, dest_idx)
        # print(action, next_state, min(taxi_row + 1, self.num_rows - 1), max(taxi_col - 1, 0), pass_loc, dest_idx)
    elif action == 9 and (right_free or down_right_free):  # Diagonal action: down-right
        next_state = self.encode(min(taxi_row + 1, self.num_rows - 1), min(taxi_col + 1, self.num_cols - 1), pass_loc, dest_idx)
        # print(action, next_state, min(taxi_row + 1, self.num_rows - 1), min(taxi_col + 1, self.num_cols - 1), pass_loc, dest_idx)
    reward = -1  # Assign a reward of -1 for diagonal actions
    self.env.env.s = next_state  # Update the environment state
    # Update the transition probabilities in env.P for the new actions
    return next_state, reward, False, {}
```

تصویر ۱- تعریف ۴ عمل حرکت مورب تاکسی در متد ()step

## ۳- الگوریتم Q-Learning

در این مرحله ما میخواهیم الگوریتم Q-Learning را برای محیط تاکسی پیاده کنیم. Q-Learning یک الگوریتم یادگیری تقویتی بدون مدل[۱] است که هدف آن یادگیری یک سیاست بهینه برای یک عامل در یک محیط است. این الگوریتم در دسته روش های یادگیری تفاوت زمانی[۲] (TD) قرار می گیرد.

در Q-Learning، عامل یک جدول به نام Q-table را نگه می دارد که در آن هر ورودی نشان دهنده پاداش تجمعی مورد انتظار برای انجام یک عمل خاص در یک حالت خاص است. جدول Q با مقادیر دلخواه یا صفر مقداردهی اولیه می شود. عامل با انجام اکشن ها، مشاهده پاداش ها و انتقال به حالت های جدید با محیط تعامل دارد.

الگوریتم Q-Learning از یک فرآیند تکراری برای به روز رسانی مقادیر Q بر اساس پاداش های مشاهده شده و انتقال حالت پیروی می کند. در هر مرحله، عامل اکشنی را بر اساس exploration-exploitation trade-off، اغلب با استفاده از یک epsilon-greedy policy انتخاب می کند. پس از انجام یک اکشن، عامل پاداش و حالت جدید را مشاهده می کند. سپس Q-value را برای جفت action-value قبلی با استفاده از معادله بلمن به روز می کند، که پاداش فعلی و مقدار تخمینی حالت بعدی را متعادل می کند.

معادله به روز رسانی Q-value در Q-Learning به صورت زیر است:

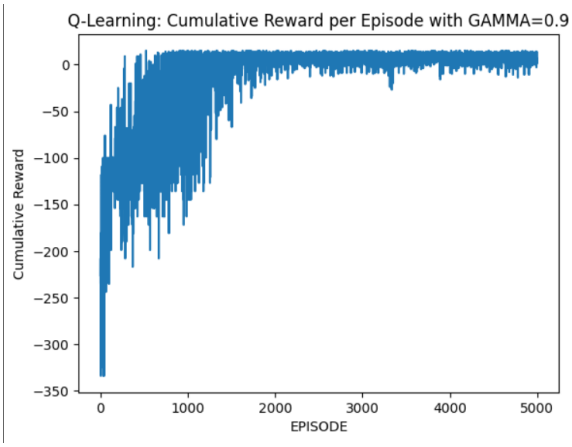
$$Q(s,a) = (1-\alpha) * Q(s,a) + \alpha * (reward + \gamma * \max_{a'} Q(s',a'))$$

[۱] Model-Free
[۲] Temporal Difference

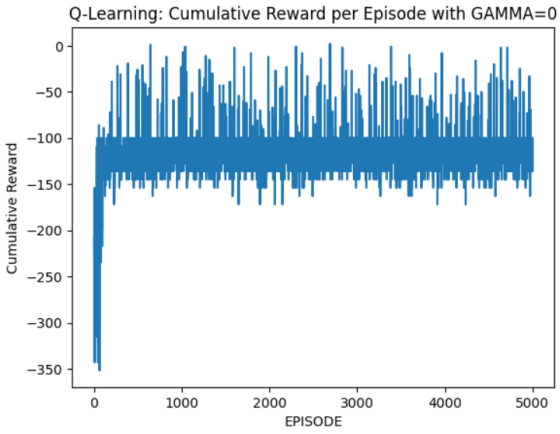
در اینجا، Q(s,a)، مقدار Q را برای حالت s و عمل a را نشان می دهد، $\alpha$ نرخ یادگیری است که وزن داده شده به اطلاعات جدید را کنترل می کند، reward پاداش مشاهده شده است، $\gamma$ عامل تخفیف[۳] است که اهمیت پاداش های آینده و $\max_{a'} Q(s', a')$ را تعیین می کند.

عامل به تعامل با محیط ادامه می دهد و مقادیر Q را به طور مکرر به روز می کند تا زمانی که به سیاست بهینه همگرا شود، جایی که مقادیر Q نشان دهنده حداکثر پاداش تجمعی مورد انتظار برای هر جفت حالت-عمل است.

حال میزان پاداش تجمعی به ازای هر episode را برای $\gamma = 0.9$ و $\gamma = 0$ نمایش میدهیم:



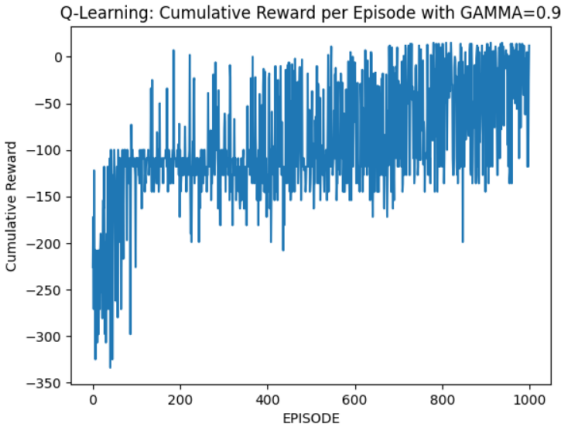

*نمودار ۱- پاداش تجمعی به ازای هر episode با gamma = 0.9 با ۵۰۰۰ episode*



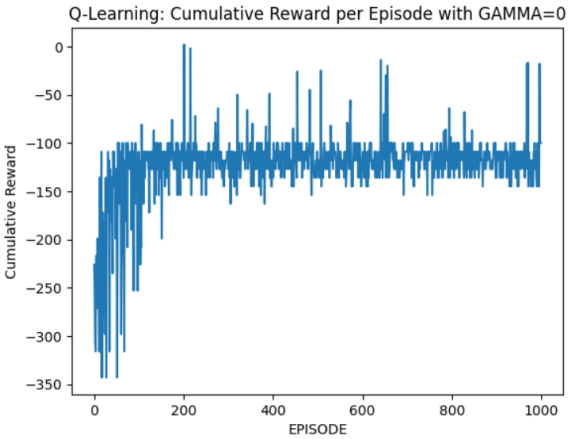

*نمودار ۲- پاداش تجمعی به ازای هر episode با gamma = 0 با ۵۰۰۰ episode*

برای مشاهده بهتر روند تغییر پاداش تجمعی، در نمودار های زیر تعداد episode را برابر با ۱۰۰۰ قرار میدهیم:

[۳] Discount factor




*نمودار ۳- پاداش تجمعی به ازای هر episode با gamma = 0.9 با ۱۰۰۰ episode*




*نمودار ۴- پاداش تجمعی به ازای هر episode با gamma = 0 با ۱۰۰۰ episode*

با توجه به دو نمودار و تحلیل آنها متوجه میشویم که:

وقتی gamma = 0.9 است، الگوریتم اهمیت نسبتاً بالایی به پاداش‌های آینده می‌دهد. در نتیجه، عامل پاداش‌های بلندمدت را در اولویت قرار می‌دهد و تصمیماتی اتخاذ می‌کند که پاداش تجمعی را در طول زمان به حداکثر برساند. نمودار پاداش تجمعی در هر episode افزایش تدریجی را در طول کل episode ها نشان می‌دهد، که نشان می‌دهد عامل در حال یادگیری و بهبود سیاست خود است. پاداش ها نوساناتی را نشان می‌دهند، اما به طور کلی، ما روند مثبتی را در پاداش‌های تجمعی مشاهده می‌کنیم زیرا عامل یاد می‌گیرد بر اساس مقادیر Q تصمیمات بهتری بگیرد.

وقتی gamma = 0 است، عامل به طور کامل پاداش های آینده را نادیده می‌گیرد و فقط بر پاداش های فوری تمرکز می کند. در نتیجه، عامل به یک عاملی تبدیل می شود که آینده نزدیک خود را می‌بیند و تمایل دارد به جای در نظر گرفتن پاداش‌های بلندمدت، دستاوردهای کوتاه مدت را در اولویت قرار دهد. پاداش تجمعی در هرepisode، نوسان بسیاری را نشان می دهد. همچنین عملکرد عامل ناپایدار است، زیرا پاداش اکشن های آینده را در نظر نمی گیرد و برای یادگیری یک سیاست بهینه یا همگرا شدن به یک راه حل پایدار تقلا می‌کند.

با مقایسه این دو متوجه می‌شویم که، با gamma = 0.9، پاداش تجمعی در هر قسمت روند ثابت‌تر و افزایشی تری را نشان می‌دهد و به پاداش تجمعی بیشتری نسبت به حالت دیگر رسیده است، که نشان‌دهنده توانایی عامل برای یادگیری و تصمیم‌گیری بر اساس پاداش‌های آینده است. از سوی دیگر، با gamma = 0، پاداش‌های بسیار نوسانی را پیش‌بینی می‌کنیم، که نشان می‌دهد تصمیمات عامل اساساً توسط پاداش‌های فوری هدایت می‌شوند و ممکن است منجر به بهینه‌سازی در طولانی‌مدت نشوند.

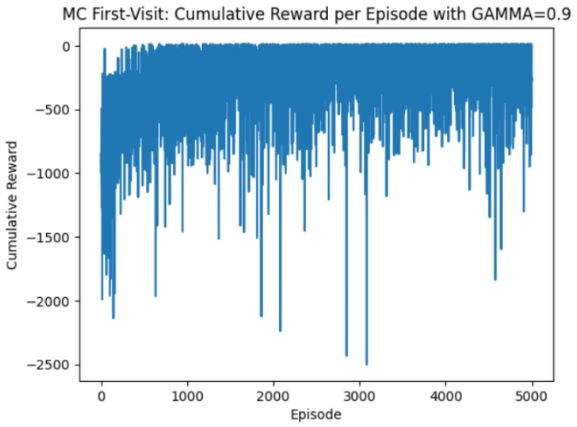
## ۴- الگوریتم First-Visit Monte Carlo

First-Visit MC یک الگوریتم یادگیری تقویتی بدون مدل است که برای تخمین تابع ارزش عمل در فرآیند تصمیم گیری مارکوف (MDP) استفاده می شود. این روش در دسته روش‌های sampling یا on-policy قرار می‌گیرد، زیرا مستقیماً از episode های نمونه‌برداری شده از تعامل با محیط یاد می‌گیرد.

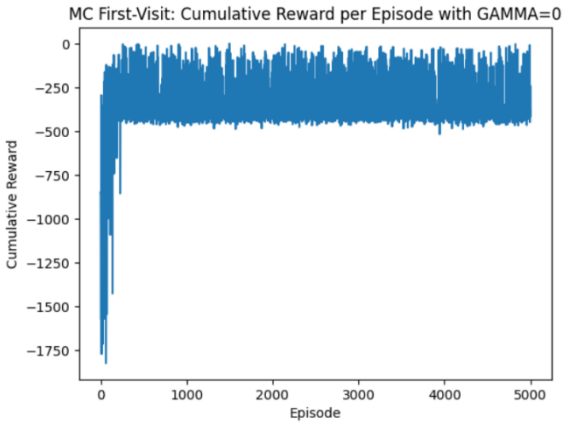
الگوریتم First-Visit MC با جمع آوری مکرر episode هایی از تجربه تعامل عامل با محیط عمل می کند. در هر episode، عامل با انجام اکشن های مبتنی بر epsilon-greedy policy، که بین exploration و exploitation تعادل برقرار می‌کند، با محیط تعامل می‌کند. عامل جفت‌های state-action، پاداش‌ها و حالت‌های بعدی مشاهده‌شده در طول episode را جمع‌آوری می‌کند.

ایده کلیدی پشت First-Visit MC این است که فقط اولین مشاهده یک جفت state-action در هر episode را در نظر می گیرد. هنگامی که یک episode کامل می شود، الگوریتم تابع ارزش عمل را با میانگین گیری return به دست آمده برای هر جفت state-action که اولین بار در هر episode بازدید می‌شود، به روز می کند.

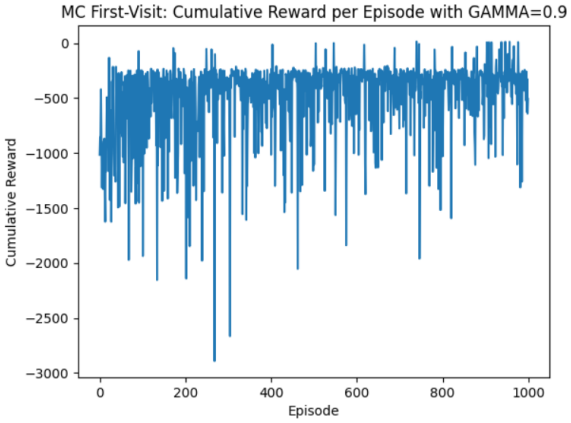
مرحله به روز رسانی شامل محاسبه return تجمعی (مجموع پاداش ها) از یک جفت state-action تا پایان episode است. سپس تابع ارزش عمل با در نظر گرفتن میانگین همه return مشاهده شده برای آن جفت state-action در چندین episode به روز می شود. با گذشت زمان، این به‌روزرسانی‌ها به مقادیر واقعی عمل همگرا می‌شوند و به عامل اجازه می‌دهند تا تصمیمات آگاهانه‌تری اتخاذ کند.



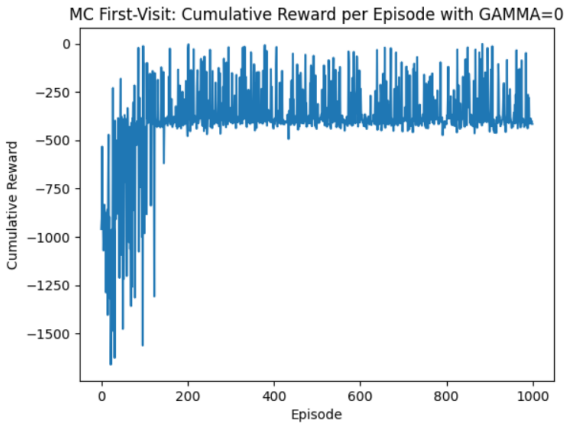

*نمودار ۵- پاداش تجمعی به ازای هر episode با gamma = 0.9 با ۵۰۰۰ episode*




*نمودار ۶- پاداش تجمعی به ازای هر episode با gamma = 0 با ۵۰۰۰ episode*




*نمودار ۷- پاداش تجمعی به ازای هر episode با gamma = 0.9 با ۱۰۰۰ episode*




*نمودار ۸- پاداش تجمعی به ازای هر episode با gamma = 0 با ۱۰۰۰ episode*

رفتار الگوریتم MC First Visit، با Q-Learning متفاوت است، زیرا روشی بدون مدل است که مستقیماً از episode های تجربه شده توسط عامل یاد می‌گیرد بدون اینکه به طور صریح تابع ارزش عمل را تخمین بزند. در عوض، تابع مقدار را بر اساس میانگین return مشاهده شده پس از بازدید از یک جفت حالت-عمل تخمین می زند، از طرفی چون الگوریتم MC First Visit آپدیت را episode به episode انجام میدهد، نسبت به الگوریتم Q-Learning که آپدیت را step به step انجام میدهد، روند یادگیری به شدت کندتر است. مثلا با مقایسه نمودار ۳ و ۷ میبینیم که در ۱۰۰۰ اپیزود تقریبا از اپیزود ۴۰۰ به بعد Q-Learning یک روند افزایشی دارد و مقادیر آن به بین -۱۰۰ و ۰ میرسد در صورتی که همین تعداد اپیزود را در مونت کارلو مشاهده کنیم، متوجه

میشویم که میانگین مقادیر نمودار روی مقدار -۲۵۰ می باشد و در انتها حدود اپیزود ۹۰۰ تا ۱۰۰۰ میبینیم که یک مقدار نمودار پاداش تجمعی وضعیت بهتری پیدا کرده و این نشان میدهد که مونت کارلو بسیار کند تر از Q-Learning است.

با بررسی دو نمودار متوجه میشویم که:

وقتی gamma = 0.9 است، الگوریتم اهمیت نسبتاً بالایی به پاداش‌های آینده می‌دهد. مشابه الگوریتم Q-Learning پاداش تجمعی در هر episode برای MC First Visit با gamma = 0.9 افزایش تدریجی را در طول کل episode ها نشان می‌دهد. با این حال، از آنجایی که MC First Visit مانند Q-Learning مقادیر عمل را به طور مکرر به روز نمی کند، روند یادگیری کندتر می‌باشد. همچنان پاداش نوساناتی دارد، اما به طور کلی، ما روند مثبتی را در پاداش‌های تجمعی مشاهده می‌کنیم زیرا عامل سیاست خود را یاد می‌گیرد و بهبود می‌بخشد.

وقتی gamma = 0 است، مشابه Q-Learning، MC First Visit فقط بر پاداش‌های فوری تمرکز می‌کند و پاداش‌های آینده را نادیده می‌گیرد. پاداش تجمعی در هر episode برای MC First Visit با gamma = 0، پاداش‌ها نوسان زیادی را نشان می‌دهند. همچنین عملکرد عامل ناپایدار باشد، زیرا پاداش اکشن های آینده را در نظر نمی گیرد.

با مقایسه این دو حالت، با gamma = 0.9، انتظار داریم که پاداش تجمعی در هر قسمت روند منسجم‌تر و افزایشی را نشان می‌دهد و به پاداش تجمعی بیشتری نسبت به حالت دیگر رسیده است، که نشان‌دهنده توانایی عامل برای یادگیری و تصمیم‌گیری بر اساس پاداش‌های آینده است. با این حال، فرآیند یادگیری در مقایسه با Q-Learning کندتر می‌باشد. از سوی دیگر، با gamma = 0، نوسان زیادی در پاداش تجمعی مشاهده می‌کنیم، که نشان می‌دهد تصمیمات عامل اساساً توسط پاداش‌های فوری هدایت می‌شوند و ممکن است منجر به بهینه‌سازی در طولانی‌مدت نشوند.

## ۵- الگوریتم Sarsa

سارسا که مخفف عبارت "State-Action-Reward-State-Action" است، نوعی الگوریتم یادگیری تقویتی است که به دسته روش های یادگیری تفاوت زمانی (TD) تعلق دارد. این یک الگوریتم on-policy است، به این معنی که سیاست خود را در حین پیروی از آن سیاست یاد می گیرد و به روز می کند.

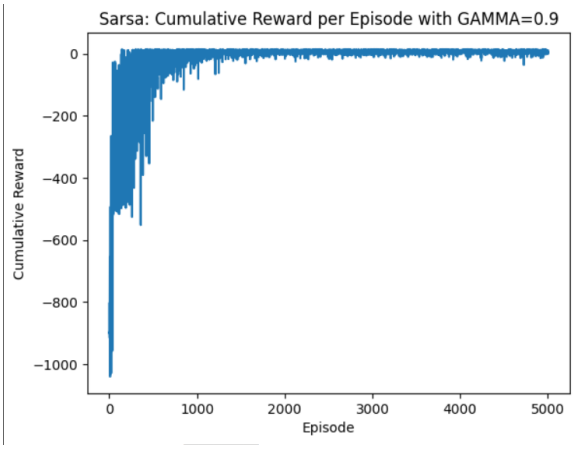
هدف الگوریتم سارسا تخمین تابع مقدار عمل است که نشان‌دهنده مقدار مورد انتظار انجام یک اکشن خاص در یک وضعیت معین و پیروی از یک سیاست خاص پس از آن است. از طریق تعامل با محیط، تجربه، انتقال حالت و دریافت پاداش یاد می گیرد.

الگوریتم با مقداردهی اولیه تابع عمل به طور دلخواه شروع می شود. سپس، به طور مکرر مقادیر عمل را بر اساس انتقال وضعیت و پاداش مشاهده شده به روز می کند. در هر مرحله، عامل بر اساس خط مشی فعلی خود اقدامی را انتخاب می کند (معمولاً یک سیاست حریصانه اپسیلونی که اکتشاف و بهره برداری را متعادل می کند). سپس آن عمل را انجام می دهد، وضعیت بعدی و پاداش را مشاهده می کند و بر اساس سیاست به روز شده اقدام بعدی را انتخاب می کند.

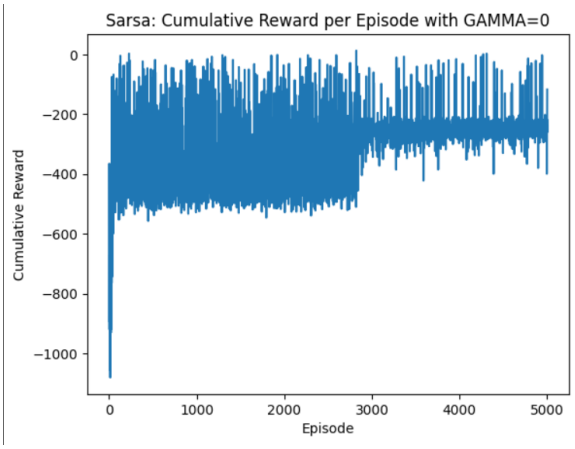
مرحله آپدیت در سارسا بر اساس تاپل state-action-reward-next state-action انجام می شود. خطای تفاوت زمانی را محاسبه می‌کند، که تفاوت بین تخمین فعلی مقدار عمل و تخمین مقدار عمل جفت state-action بعدی است. سپس تابع مقدار عمل با برداشتن گامی در جهت کاهش این خطای اختلاف زمانی، با پارامتر نرخ یادگیری که اندازه آپدیت را تعیین می کند، به روز می شود.

الگوریتم سارسا این فرآیند تکراری را ادامه می‌دهد، مقادیر عمل را به‌روزرسانی می‌کند و سیاست را بر اساس تجربیات مشاهده‌شده تنظیم می‌کند، تا زمانی که به یک سیاست بهینه همگرا می‌شود که پاداش‌های تجمعی را در طول زمان به حداکثر می‌رساند.

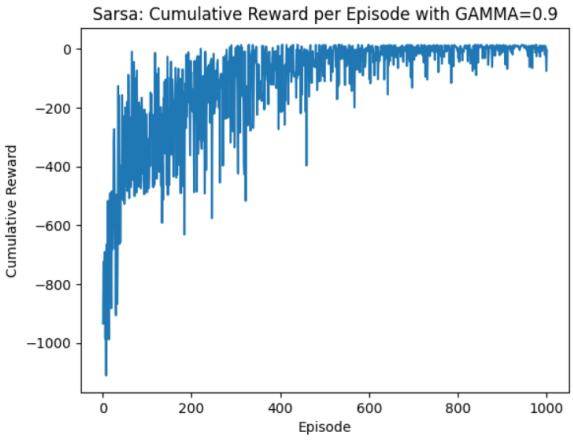
توجه به این نکته مهم است که همگرایی و عملکرد سارسا می تواند تحت تأثیر عواملی مانند نرخ یادگیری، exploration-exploitation trade-off و ضریب تخفیف باشد که اهمیت پاداش های آینده را تعیین می کند.



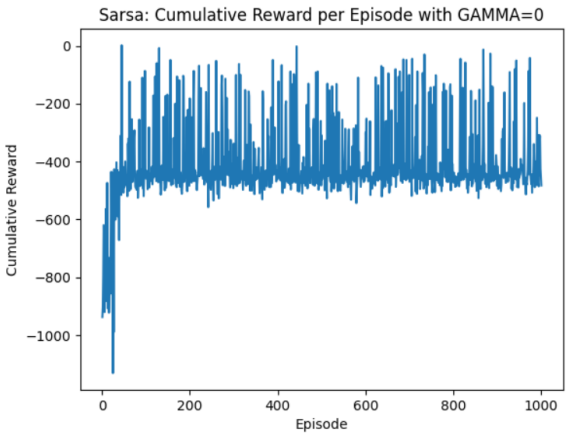

نمودار ۹- پاداش تجمعی به ازای هر episode با gamma = 0.9 با episode ۵۰۰۰




نمودار ۱۰- پاداش تجمعی به ازای هر episode با gamma = 0 با episode ۵۰۰۰




نمودار ۱۱- پاداش تجمعی به ازای هر episode با gamma = 0.9 با episode ۱۰۰۰




*نمودار ۱۲- پاداش تجمعی به ازای هر episode با gamma = 0 با ۱۰۰۰ episode*

با مشاهده نمودار های Sarsa ( و همچنین نمودار های Q-Learning و MC First-Visit ) میبینیم که در episode های اولیه، پاداش تجمعی نسبتاً کم است زیرا عامل محیط را کاوش می کند و از طریق آزمون و خطا یاد می گیرد. همانطور که عامل بیشتر با محیط تعامل می کند و ارزش های عمل خود را به روز می کند، ما انتظار داریم که شاهد بهبودی در پاداش تجمعی نسبت به تمام episode ها باشیم.

در طول فرآیند یادگیری، نمودار پاداش تجمعی Sarsa برخی از نوسانات و واریانس را نشان میدهد. این طبیعی است و می توان آن را به exploration-exploitation trade-off در الگوریتم نسبت داد. عامل گاهی اوقات ممکن است اکشن های غیربهینه را برای بررسی حالات و اکشن های جدید انتخاب کند، که می تواند در ابتدا منجر به پاداش کمتر شود. با این حال، همانطور که عامل ارزش های عمل خود را یاد می گیرد و به روز می کند، باید به تدریج اقدامات بهینه بیشتری را کشف کند و پاداش تجمعی را افزایش دهد.

در کل ما روند مثبتی را در پاداش تجمعی می‌ببینیم، که نشان می‌دهد عامل در حال یادگیری و بهبود سیاست خود در طول زمان است. نمودار ممکن است با اصلاح سیاست خود، افزایش تدریجی نشان دهد، اگرچه میزان بهبود ممکن است بسته به عواملی مانند میزان یادگیری و پیچیدگی محیط متفاوت باشد.

توجه به این نکته مهم است که همگرایی و عملکرد الگوریتم سارسا می تواند تحت تأثیر پارامترهای مختلفی مانند نرخ یادگیری، exploration-exploitation trade-off و ضریب تخفیف (گاما) قرار گیرد. . تنظیم این پارامترها می تواند بر سرعت یادگیری و کیفیت سیاست آموخته شده تأثیر بگذارد.

## ۶- تفاوت الگوریتم های Q-Learning و Sarsa

تفاوت اصلی بین Sarsa و Q-Learning در یادگیری و استراتژی های به روز رسانی آنها نهفته است. در اینجا تمایزات کلیدی وجود دارد:

۱. زمان بندی به روز رسانی:

- Sarsa: در Sarsa تابع action-value (Q-values) در طول تعامل عامل با محیط به روز می شود که هر به روز رسانی پس از انجام یک اکشن توسط عامل و مشاهده حالت و پاداش بعدی اتفاق می افتد.

- Q-Learning: در Q-Learning، تابع action-value به صورت off-policy به روز می شود، به این معنی که آپدیت ها بر اساس حداکثر Q-value وضعیت بعدی، صرف نظر از اکشن انجام شده، انجام می شود. آپدیت ها مستقل از اکشن های عامل یا تعاملات آن با محیط رخ می دهد.

۲. ارزیابی سیاست:

- سارسا: سارسا یک الگوریتم on-policy است، به این معنی که تابع مقدار عمل خود را بر اساس سیاست فعلی به روز می کند و از سیاست یکسانی برای انتخاب و ارزیابی عملکرد استفاده می کند. مقادیر Q را برای سیاست فعلی می آموزد و هدف آن بهبود آن سیاست است.

- Q-Learning: Q-Learning یک الگوریتم off-policy است. این تابع عملکرد مقدار بهینه را با تخمین حداکثر expected return برای هر جفت حالت-عمل می آموزد. با این حال، از یک سیاست متفاوت (معمولاً یک سیاست حریصانه) برای انتخاب اکشن ها در حین یادگیری استفاده می کند و به آن اجازه می دهد تا سیاست های مختلف را بررسی کند و از آنها بیاموزد.

۳. exploration در مقابل exploitation:

- سارسا: سارسا از سیاست epsilon-greedy استفاده می کند که بین exploration و exploitation تعادل برقرار می کند. این به طور تصادفی اکشن های جدید با احتمال epsilon را بررسی می کند و و با احتمال 1-epsilon اکشن های حریصانه (با بیشترین مقدار Q) را انتخاب می کند. این سیاست به Sarsa اجازه می‌دهد تا به طور مداوم اکشن های غیربهینه را بررسی کند و به طور بالقوه از آنها یاد بگیرد.

- Q-Learning: Q-Learning معمولاً از یک سیاست حریصانه استفاده می کند و اکشنی را با حداکثر Q-value انتخاب می کند. این الگوریتم بر بهره برداری از دانش آموخته شده برای انتخاب بهترین اکشن ها بر اساس تخمین فعلی سیاست بهینه تمرکز دارد. Q-Learning بیشتر به کاوش در مرحله یادگیری متکی است تا در فرآیند به روز رسانی.

از نظر پاداش تجمعی، عملکرد Sarsa و Q-Learning می تواند در سناریوهای خاصی متفاوت باشد:

می توان گفت چون الگوریتم Sarsa، on-policy است و از یک سیاست برای ارزیابی و بهبود استفاده میکند، لذا زودتر از الگوریتم Q-Learning که یک الگوریتم off-policy است، به همگرایی می رسد، زیرا در الگوریتم Q-Learning، سیاست رفتار و سیاست هدف متفاوتند لذا باید از تجربیات عامل با استفاده از سیاست رفتار برای برای یادگیری سیاست هدف استفاده کنیم نه تجربه عامل با استفاده از همان سیاست هدف، لذا دیرتر به سیاست هدف همگرا میشود.