پاسخ تکلیف چهارم

"یادگیری تعاملی"

گروه دوم

نام استاد:

دکتر نیلی

تهیه کننده :

عرفان میرزایی ۸۱۰۱۹۹۲۸۹

در این تمرین سعی شده است که با استفاده از محیط داده شده در صورت سوال الگوریتم های مربوط به یادگیری بدون در اختیار داشتن دینامیک محیط را پیاده سازی کنیم. در ابتدا الگوریتم On-policy Monte-Carlo را پیاده سازی می کنیم.

در حل این تمرین با در اختیار داشتن محیط، تنها کلاس Agent را پیاده سازی می‌کنیم که برای این منظور از پکیج Amalearn کمک می‌گیریم. به طور کلی اطلاعات کلاس Agent از سه قسمت تشکیل شده است. که بخش اول مربوط به پالیسی و الگوریتم هر عامل است که در متد Init به آن ها مقداردهی اولیه می‌کنیم. بخش دوم مربوط به متد take_action است که در هر زمان با توجه به سیاست مان و حالتی که در آن قرار داریم عملی را انتخاب می‌کنیم. بخش سوم مربوط به بروزرسانی مقادیر مورد نیاز برای سیاست است که این بخش تنها بخشی است که میان الگوریتم های مختلف متفاوت است.

- الگوریتم On-policy Monte-Carlo

با مراجعه به کتاب ساتون بارتو شبه کد این الگوریتم به شرح زیر است :

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
  $\pi \leftarrow$ an arbitrary ε-soft policy
  $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
  $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$
      $A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)
      For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

شکل ۱ - شبه کد الگوریتم On-policy Monte-Carlo

که با توجه به این الگوریتم کلاس OnPolicy_esoft_FirstVisit_MC_Agent در فایل On_FV_MC_agent_esoft2.py پیاده‌سازی شده است.

برای پیاده سازی عامل مقدار اپسیلون برای تعیین سیاست نرم و هم چنین مجموعه حالت ها و عمل ها به عنوان ورودی به عامل داده شده است. هم چنین برای تضمین همگرایی به سیاست بهینه از یک تابع کاهش اپسیلون استفاده شده است که آن را به صورت یک لیست شامل سه عضو به صورت [a,b,c] دریافت می‌کنیم.

روال کار این تابع به این صورت است که در هر اپیزود با فراخوانی متد decay_epsilon از کلاس عامل اپسیلون به صورت زیر کاهش می‌یابد.

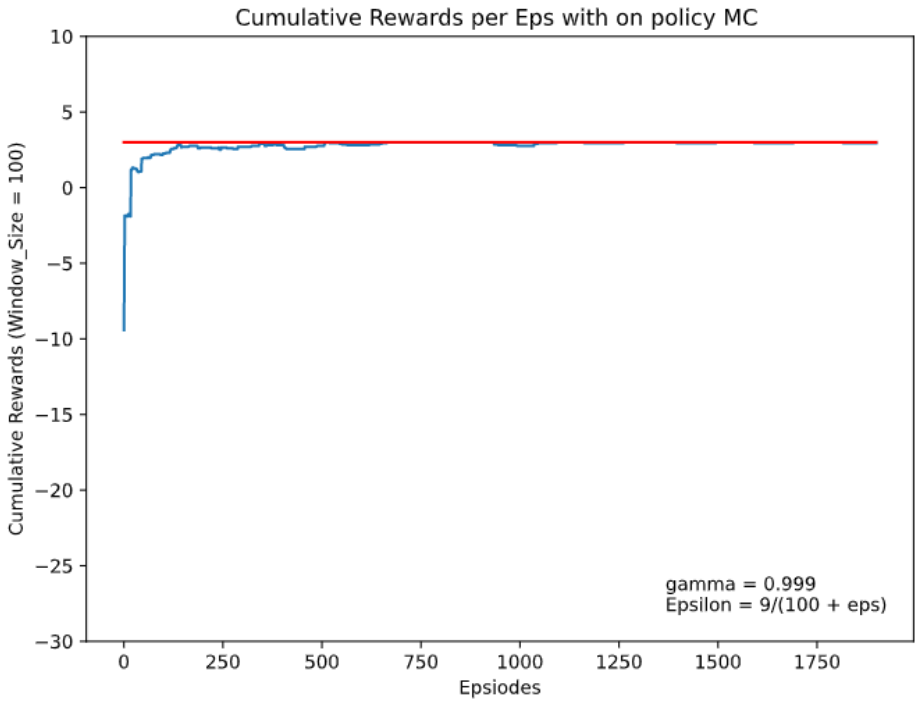
$$Epsilon = \frac{a}{b + c * episode}$$

که این روند در عامل های سایر الگوریتم ها نیز استفاده شده است.

مقادیر اولیه را همان طور که در بخش قبل توضیح دادیم در ابتدا با تعریف عامل مقداردهی می‌کنیم. سپس به کمک تابع Generate_episode اپیزود میسازیم بدین صورت که با شروع از حالت اولیه و با استفاده از متد take_action عامل هر مرحله را ادامه می‌دهیم تا به حالت نهایی برسیم. با توجه به توضیحات گفته شده در صورت سوال یک اپیزود یک مرحله دیرتر از رسیدن به جزیره به پایان می‌رسد. به همین منظور از یک شمارنده استفاده کرده ایم که وقتی مقدار آن به ۲ رسید یعنی دو بار مقدار متغییر done برابر True شد اپیزود را خاتمه دهد. در نهایت پاداش های دریافتی و هم چنین حالت ها و عمل هایی که در آن قرار گرفتیم را به عنوان خروجی باز می‌گردانیم.

در ادامه برای هر اپیزود مقدار G را به صورت بازگشتی از انتهای اپیزود حساب کرده و با استفاده از آن مقدار Q مورد نظر آن حالت-عمل را به روز رسانی می‌کنیم. در هر مرحله طول اپیزود و هم چنین مجموع ریوارد ها را نیز چاپ می‌کنیم. برای محاسبه‌ی مجموع پاداش ها بر حسب تعداد اپیزودها از پنجره ای به طول ۱۰۰ استفاده می‌کنیم بدین معنا که برای هر اپیزود میانگین پاداش در ۱۰۰ مرحله‌ی قبلی را در نظر می‌گیریم. به همین خاطر از رسم پاداش های ۱۰۰ مرحله‌ی اول صرف نظر شده است.

با توجه به نمودار و هم چنین طول اپیزود های بهینه که ۱۰ گام است(عمل مرحله آخر در نظر گرفته نشده است) نشان می دهد که این الگوریتم به خوبی همگرا شده است.



شکل ۲- پاداش تجمعی الگوریتم On policy monte carlo

• الگوریتم Off-policy Monte-Carlo

با مراجعه به کتاب ساتون بارتو شبه کد این الگوریتم به شرح زیر است :

Off-policy MC control, for estimating $\pi \approx \pi_*$

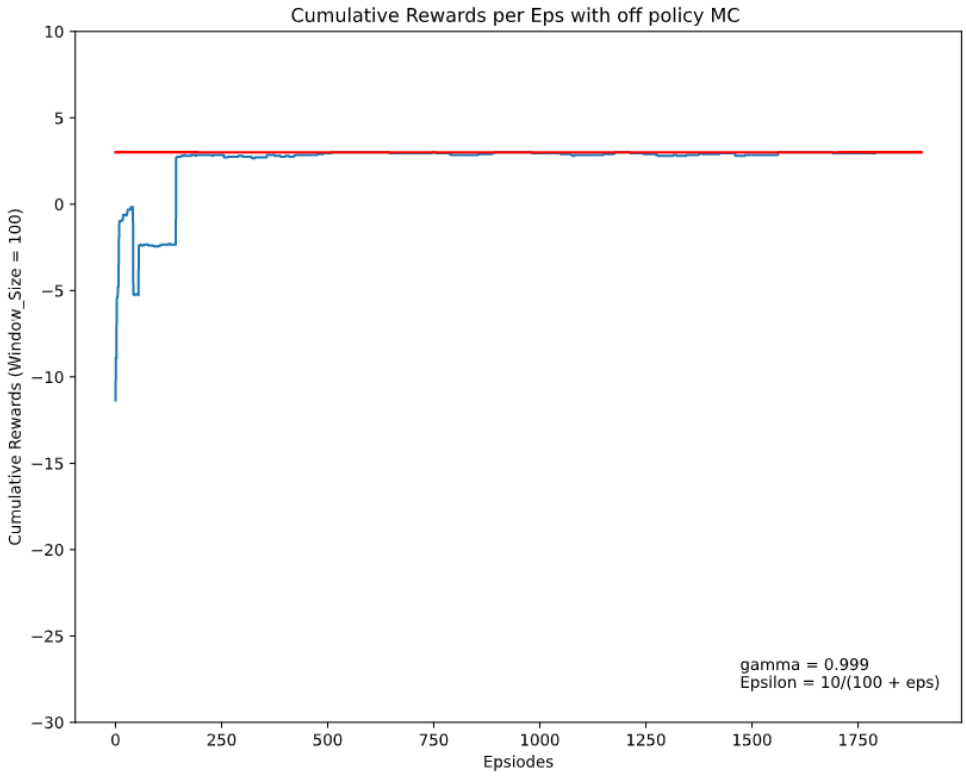
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \in \mathbb{R}$ (arbitrarily)
  $C(s,a) \leftarrow 0$
  $\pi(s) \leftarrow \text{argmax}_a\, Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
  $b \leftarrow$ any soft policy
  Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
    $\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
    If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
    $W \leftarrow W \frac{1}{b(A_t|S_t)}$

که با توجه به این الگوریتم کلاس OffPolicy_MC_Agent در فایل Off_MC_agent.py پیاده‌سازی شده است.

مقادیر اولیه را همان طور که در بخش قبل توضیح دادیم در ابتدا با تعریف عامل مقداردهی می‌کنیم. سپس به کمک تابع Generate_episode اپیزود میسازیم. مقدار نرخ تخفیف را برابر ۰,۹۹۹ قرار می‌دهیم. در ادامه برای هر اپیزود مقدار G را به صورت بازگشتی از انتهای اپیزود حساب کرده و با استفاده از آن مقدار Q مورد نظر آن حالت-عمل را تحت سیاست قاطعانه pi به روز رسانی می‌کنیم.

در پایان کار نمودار زیر از نتایج حاصل شد:



شکل ۳- پاداش های تجمعی off policy Monte Carlo

با توجه به نمودار و هم چنین طول اپیزود های بهینه که ۱۰ گام است(عمل مرحله آخر در نظر گرفته نشده است) نشان می دهد که این الگوریتم به خوبی همگرا شده است. نکته قابل توجه در این قسمت که در الگوریتم های off-policy هدف آموزش سیاستی قاطعانه است اما در این جا مشاهده می کنیم که سیاست نرم نیز همگرا

شده است اما برای اطمینان بیشتر سیاست قاطعانه را از حالت اولیه اجرا می‌کنیم که عامل به درستی و با طی ۱۰ قدم به جزیره می‌رسد.

• الگوریتم Double Q-learning

با مراجعه به کتاب ساتون بارتو شبه کد این الگوریتم به شرح زیر است :

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

شکل ۴- شبه کد الگوریتم Double Q-Learning

که با توجه به این شبه کد، کلاس Double_Q_Learning_Agent در فایل Double_Q_Learning_agent.py پیاده‌سازی شده است.

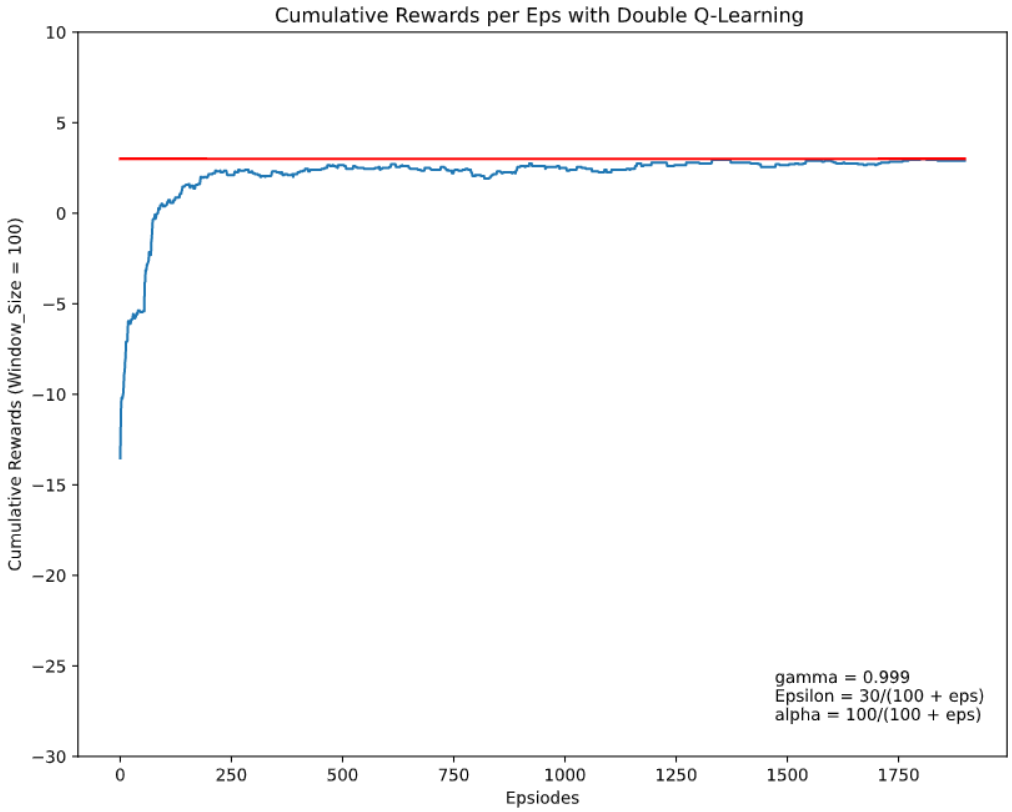
مقادیر اولیه را همان طور که در بخش قبل توضیح دادیم در ابتدا با تعریف عامل مقداردهی می‌کنیم.. مقدار نرخ تخفیف را برابر ۰,۹۹۹ قرار می‌دهیم. در ادامه با قرار گرفتم در هر حالت با دریافت پاداش و تخمین خود از مقدار Q حالت بعدی مقدار Q مورد نظر آن حالت-عمل را تحت سیاست قاطعانه pi به روز رسانی می‌کنیم.

برای این عامل از نرخ یادگیری کاهشی استفاده می‌کنیم که آن را به صورت یک لیست شامل سه عضو به صورت [a,b,c] دریافت می‌کنیم. روال کار این تابع به این صورت است که در هر اپیزود با فراخوانی متد decay_learning_rate از کلاس عامل اپسیلون به صورت زیر کاهش می‌یابد.

$$alpha = \frac{a}{b + c * episode}$$

که این روند در عامل های سایر الگوریتم ها نیز استفاده شده است.

در پایان کار نمودار زیر از نتایج حاصل شد:



شکل ۵- پاداش های تجمعی الگوریتم Double Q-learning

با توجه به نمودار این الگوریتم دیرتر از سایر الگوریتم ها به همگرایی رسیده است اما با بررسی دقیق تر طول اپیزود ها مشخص می شود که این الگوریتم بهینه سراسری را پیدا کرده است اما این نوسات به دلیل استفاده از دو پشته است. هم چنین ذکر این نکته شایان توجه است که در کلیه ی الگوریتم های پیاده سازی شده در این ترم به منظور پیدا کردن هایپرپارامتر های متناسب با هر الگوریتم در ابتدای سلول هر الگوریتم مقدار سید عدد را مشخص می‌کنیم.

• الگوریتم SARSA

با مراجعه به کتاب ساتون بارتو شبه کد این الگوریتم به شرح زیر است :

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

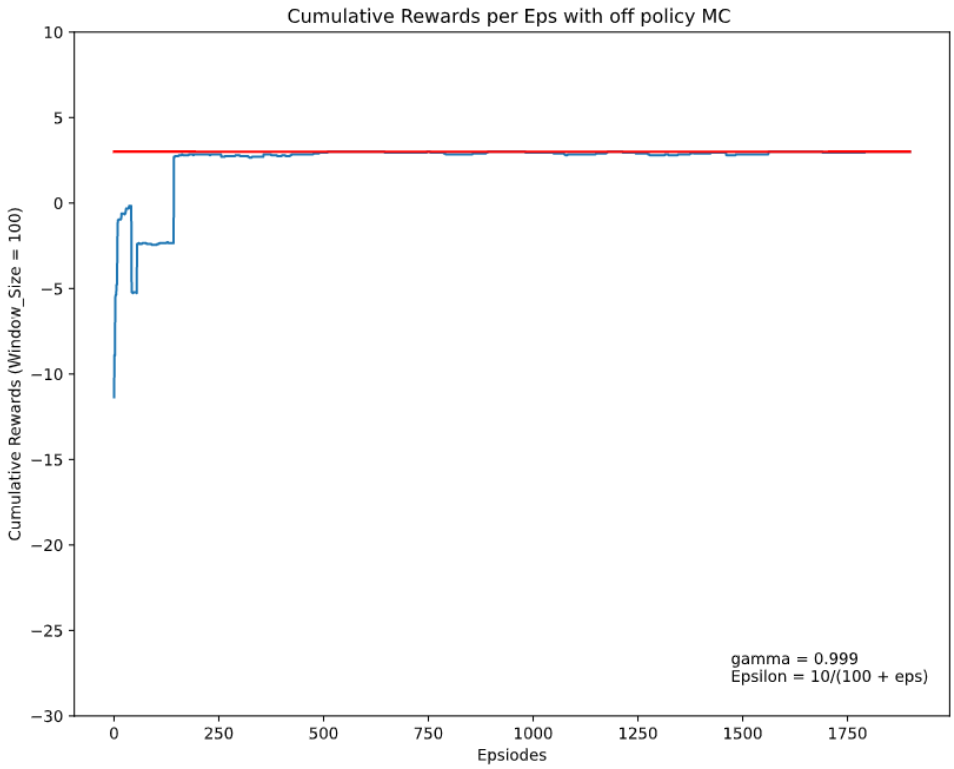
شکل ۶- شبه کد الگوریتم SARSA

که با توجه به این شبه کد، کلاس SARSA_Agent در فایل SARSA_agent.py پیاده‌سازی شده است.

مقادیر اولیه را همان طور که در بخش قبل توضیح دادیم در ابتدا با تعریف عامل مقداردهی می‌کنیم.. مقدار نرخ تخفیف را برابر ۰,۹ قرار می‌دهیم. در ادامه با قرار گرفتم در هر حالت با دریافت پاداش و تخمین خود از مقدار Q حالت بعدی مقدار Q مورد نظر آن حالت-عمل را به روز رسانی می‌کنیم.

با توجه به نمودار و هم چنین طول اپیزود های بهینه که ۱۰ گام است (عمل مرحله آخر در نظر گرفته نشده است) نشان می دهد که این الگوریتم به خوبی همگرا شده است.

در پایان کار نمودار زیر از نتایج حاصل شد:



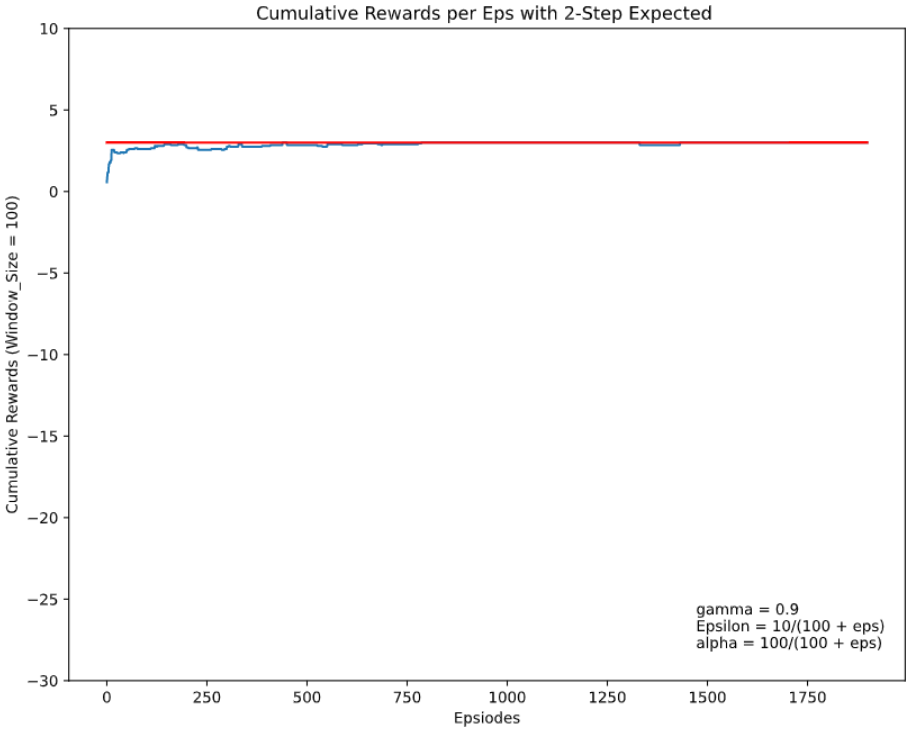
شکل ۷ - نمودار مربوط به الگوریتم SARSA

- ## الگوریتم 2-step Expected SARSA

این الگوریتم بسیار مانند الگوریتم مرحله‌ی قبل است با این تفاوت که برای بوت استرپ کردن یک عمل خاص را در نظر نمی‌گیریم بلکه امید ریاضی مقدار Q حالت-عمل های مورد نظر را در نظر می‌گیریم. هم چنین با توجه به نام الگوریتم برای بروز رسانی هر حالت از مقادیر مربوط به حالت های ۲ گام جلوتر استفاده می‌کنیم.

با این تفاسیر و با توجه به توضیحات داده شده در مرحله‌ی قبلی عامل مربوط به این الگوریتم کلاس Two_step_Exp_SARSA_Agent در فایل Two_step_Exp_SARSA_agent.py پیاده‌سازی شده است.

نتایج مربوط به این الگوریتم به شرح زیر است :



شکل ۸- نمودار مربوط به پاداش های تجمعی 2-step Expected SARSA

با توجه به نمودار و هم چنین طول اپیزود های بهینه که ۱۰ گام است (عمل مرحله آخر در نظر گرفته نشده است) نشان می دهد که این الگوریتم بسیار خوب و به سرعت به بهینه سراسری همگرا شده است.

• الگوریتم n-Step Backup tree

با مراجعه به کتاب ساتون بارتو شبه کد این الگوریتم به شرح زیر است :

n-step Tree Backup for estimating $Q \approx q_*$ or q_π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose an action A_0 arbitrarily as a function of S_0; Store A_0
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$:
      Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
      If S_{t+1} is terminal:
        $T \leftarrow t+1$
      else:
        Choose an action A_{t+1} arbitrarily as a function of S_{t+1}; Store A_{t+1}
    $\tau \leftarrow t+1-n$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      If $t+1 \geq T$:
        $G \leftarrow R_T$
      else
        $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
      Loop for $k = \min(t, T-1)$ down through $\tau + 1$:
        $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k|S_k) G$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \left[G - Q(S_\tau, A_\tau) \right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

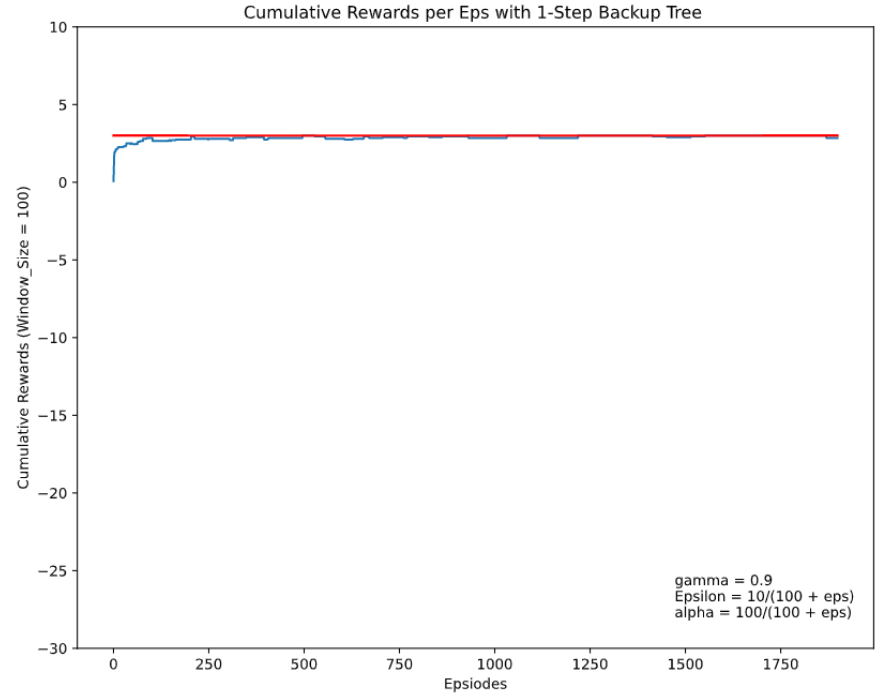
شکل ۹- شبه کد الگوریتم n-Step Backup tree

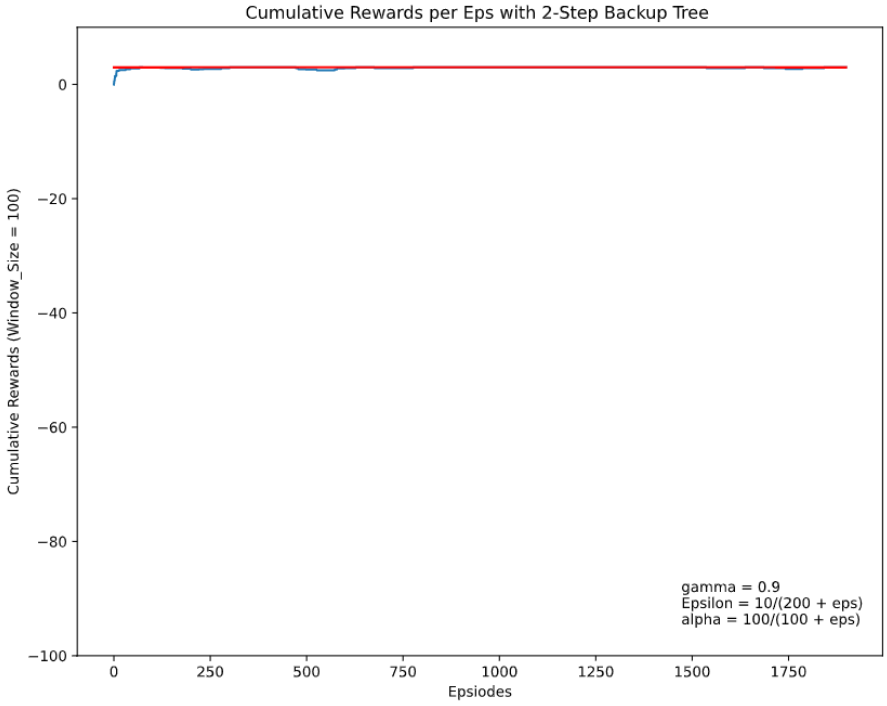
که با توجه به این شبه کد، کلاس nStep_tree_backup_Agent در فایل nStep_tree_backup_agent.py پیاده‌سازی شده است.

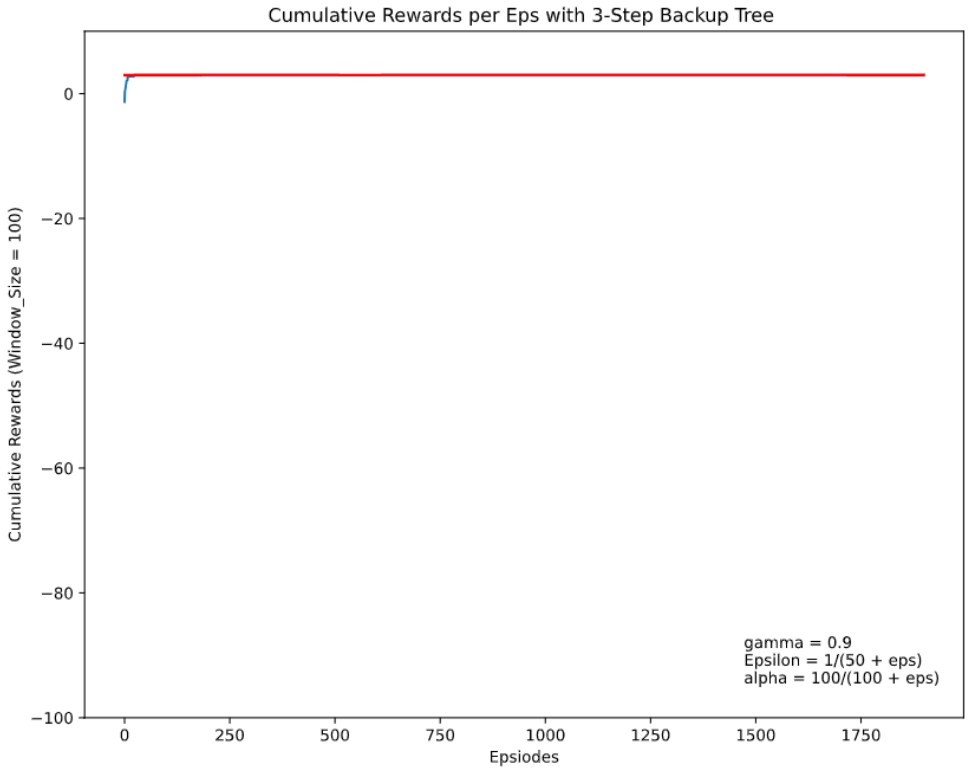
مقادیر اولیه را همان طور که در بخش قبل توضیح دادیم در ابتدا با تعریف عامل مقداردهی می‌کنیم.. مقدار نرخ تخفیف را برابر ۰,۹ قرار می‌دهیم. در ادامه با قرار گرفتم در هر حالت با دریافت پاداش و تخمین خود از مقدار Q حالت بعدی مقدار Q مورد نظر آن حالت-عمل را به روز رسانی می‌کنیم. عمل به روز رسانی شامل سه قسمت می شود که دو قسمت از آن مربوط به بروزرسانی G است و قسمت نهایی مربوط به بروزرسانی مقدار Q هاست.

که به ترتیب در متد های agent.calculate_G_Expected و agent.calculate_G_treebackup

و agent.update_Q پیاده سازی شده اند. نتایج مربوط به این الگوریتم به شرح زیر است :



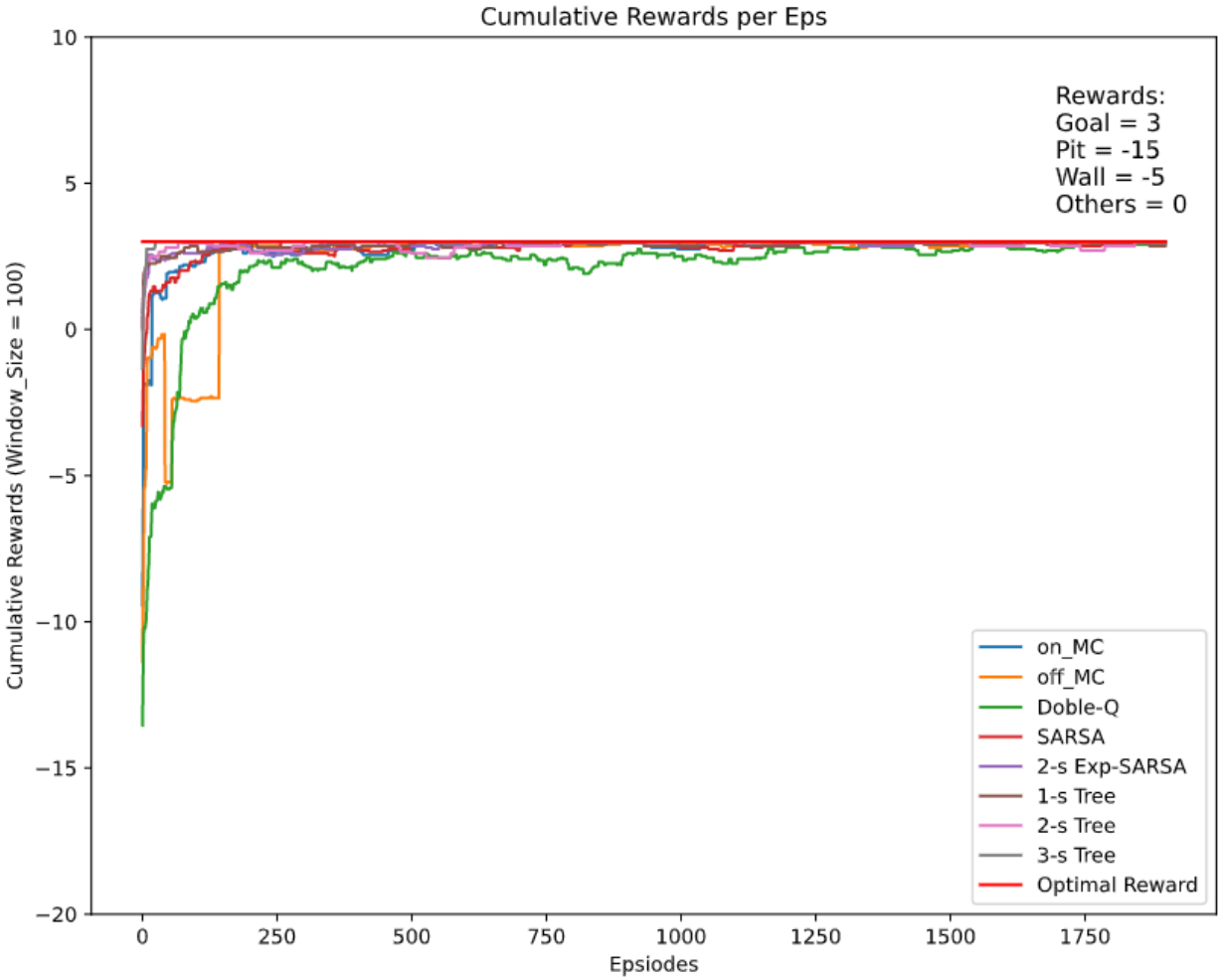




با توجه به نمودار و هم چنین طول اپیزود های بهینه که ۱۰ گام است (عمل مرحله آخر در نظر گرفته نشده است) نشان می دهد که این الگوریتم برای دو مقدار n = 1, n = 2 بسیار خوب و به سرعت به بهینه سراسری همگرا شده است. اما برای n = 3 الگوریتم به سرعت همگرا شده است اما نتوانسته تعداد گام های بهینه را پیدا کند که این امر با توجه به اینکه برای هر گام پاداش منفی درنظر نگرفته ایم می‌تواند توجیه شود.

- پاسخ پرسش ۲

برای مقایسه بین زمان همگرایی الگوریتم های مختلف تمامی نتایج مربوط به پاداش های تجمعی در هر اپیزود را در یک نمودار رسم می‌کنیم. که این نمودار در صفحه بعدی آورده شده است.



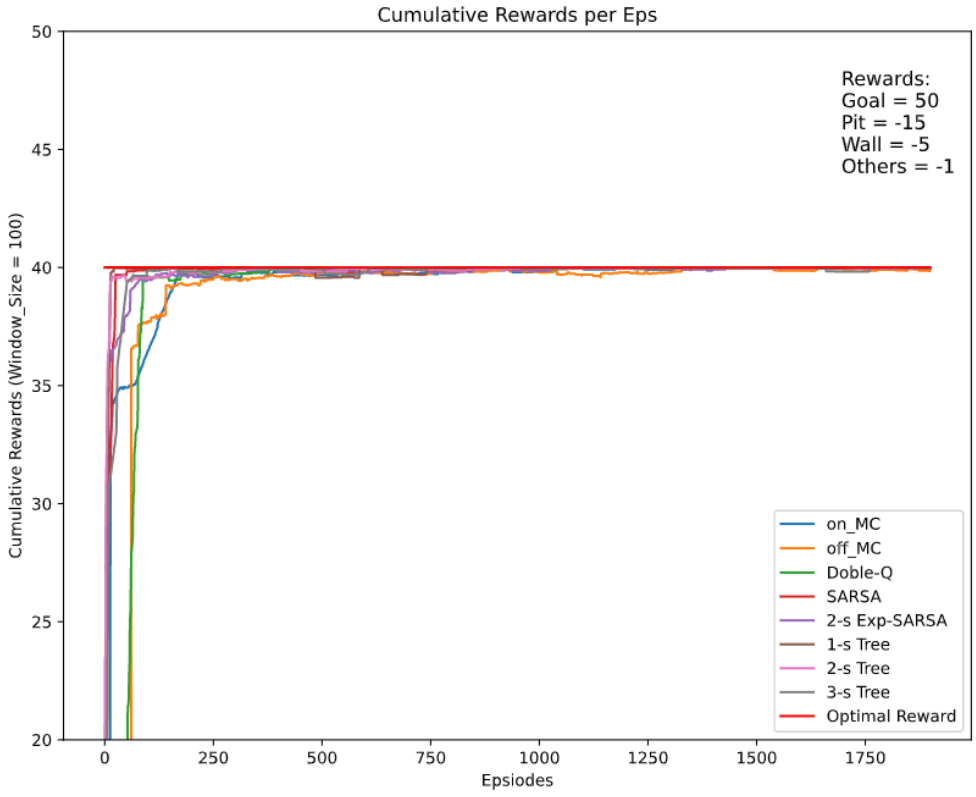
با توجه به این نمودار می‌توان به این نتیجه رسید که با در نظر داشتن پارامترهای ذکر شده در هر قسمت نمودار مربوط به Double Q-learning از همه دیرتر به سیاست بهینه همگرا شده است که از آنجایی که این الگوریتم Off – policy است و در این نوع از الگوریتم ها هدف پیدا کردن سیاست قاطعانه بهینه است و سیاست نرم می تواند دیرتر به همگرایی برسد قابل توجیه است. هم چنین در رتبه‌ی بعدی الگوریتم Off-policy MC قرار دارد که این الگوریتم نیز به دلیل مشابه ذکر شده، نتیجه آن قابل توجیه است. سایر الگوریتم ها عملکرد بسیار نزدیک به هم و خوبی را داشته اند و تقریبا با گذشت کمتر از ۲۰۰ مرحله سیاست بهینه را پیدا کرده اند.

• بخش امتیازی

در محیط داده شده در صورت سوال برای رسیدن به ساحل پاداشی معادل ۳ برای خوردن به دیوار پاداش -۵ و برای افتادن در رودخانه عمیق پاداشی برابر ۱۵- در نظر گرفته شده است و برای سایر موارد عامل پاداش صفر را دریافت می کند. مشکل این کار این است که عامل در صورت دریافت پاداش کوچک منفی در هر مرحله ترغیب می شود که سریع تر خود را به ساحل برساند و این امر منجر به یادگیری بهتر و سریع تر آن می شود.

به همین جهت در فایل Environment Modified.ipynb محیط را به صورت زیر تغییر داده ایم و تمامی مراحل بالا را تکرار کرده ایم .

در محیط تغییر داده شده در صورت سوال برای رسیدن به ساحل پاداشی معادل ۵۰، برای خوردن به دیوار پاداش -۵ و برای افتادن در رودخانه عمیق پاداشی برابر ۱۵- در نظر گرفته شده است و برای سایر موارد عامل پاداش ۱- را دریافت می کند. نتیجه آموزش عامل برای کلیه حالت مانند قسمت قبلی صورت گرفته است و نتایج مربوط به پاداش های تجمعی در هر اپیزود را برای تمامی الگوریتم ها در یک نمودار رسم می‌کنیم.



همان طور که ازین نمودار مشخص است در این حالت برای تمامی الگوریتم ها به نقطه بهینه سراسری همگرا شده ایم و پاداشی معادل ۴۰ را دریافت کرده ایم حتی برای حالت هایی که در مرحله‌ی قبلی دیرتر همگرا می‌شدیم.

اما برای مقایسه ی زمان همگرایی الگوریتم ها در این محیط و محیط قبلی می توان گفت که تفاوت چندانی وجود ندارد.