به نام خدا

دانشکده مهندسی برق

گزارش درس یادگیری ماشین

مقطع: کارشناسی ارشد گرایش: مهندسی کنترل

گزارش مینی پروژه چهارم

توسط:

مرجان محمدی

۴۰۱۱۱۵۳۴

استاد درس:

دکتر علیاری

لینک کولب

لینک گیت هاب

تابستان ۱۴۰۳

پرسش یک: حل دنیای Wumpus

Wumpus Worldیک مسئله کلاسیک در هوش مصنوعی و یادگیری تقویتی است که شامل یک محیط مبتنی بر شبکه است

که در آن یک عامل باید برای یافتن طلا حرکت کند و در عین حال از خطراتی مانند چاله ها و Wumpusاجتناب کند .

·اهداف پیمایش در شبکه Grid: عامل باید یاد بگیرد که به طور موثر در شبکه حرکت کند .

·اجتناب از خطرات: عامل باید یاد بگیرد که از چاله ها و Wumpusاجتناب کند .

·جمع آوری طلا: عامل باید طلا را پیدا کرده و جمع آوری کند .

·کشتن **Wumpus**: عامل می تواند برای کشتن **Wumpus**تیری شلیک کند و آن را به عنوان تهدید از بین ببرد .

·راه اندازی محیط شبکه: یک شبکه ۴*۴که در آن هر سلول می تواند خالی باشد، حاوی یک گودال، **Wumpus**

یا طلا باشد .

·فضای اکشن ها: حرکت به بالا، پایین، چپ، راست .

یک فلش را در هر یک از چهار جهت (بالا، پایین، چپ، راست) شلیک کنید (امتیازی).

·تصورات**Wumpus** :در شبکه با هر تغییر اکشن به اندازه یک خانه در راستای چپ، راست، بالا یا پایین حرکت می کند (امتیازی).

فضای Reward:

+ ۱۰۰-برای گرفتن طلا

– ۱۰۰۰-برای افتادن در گودال یا خورده شدن توسط **Wumpus**

+ ۵۰-برای کشتن**Wumpus** (امتیازی)

– ۱-برای هر حرکت

·تعریف محیط: یک شبکه ۴ × ۴با موقعیت های دلخواه برای چاله ها، **Wumpus**و طلا ایجاد کنید. حالت اولیه و

حالت های ممکن را بعد از هر عمل تعریف کنید .

·تنظیم پارامترها:

-نرخ یادگیری : ۱.۰

-ضریب تخفیف: ۹.۰

-نرخ اکتشاف: از ۱ . ۰شروع می شود و در طول زمان کوچک میشود.

با توجه به موارد کلی گفته شده راجع به مسئله، موارد زیر را پاسخ دهید .

آ. برای این مسئله یک بار با روش **Q-learning**و یک بار با روش **Deep Q-learning**عاملی را طراحی کرده و آموزش دهید .

ب. عملکرد **Policy**:

- پاداش تجمعی را در اپیزودها برای هر دو عامل **Q-learning** و DQN ترسیم کنید. چگونه عملکرد عامل در طول زمان بهبود می یابد؟
- میانگین پاداش در هر اپیزود را برای هر دو عامل پس از ۱۰۰۰ اپیزود مقایسه کنید. کدام الگوریتم عملکرد بهتری داشت؟

ج. بحث کنید که چگونه نرخ اکتشاف اپسیلون بر فرآیند یادگیری تأثیر می گذارد. وقتی اپسیلون بالا بود در مقابل وقتی کم بود چه چیزی را مشاهده کردید؟

د. کارایی یادگیری:

- چند اپیزود طول کشید تا عامل **Q-learning** به طور مداوم طلا را بدون افتادن در گودال یا خورده شدن توسط **Wumpus** پیدا کند؟
- کارایی یادگیری **Q-learning** و **DQN** را مقایسه کنید. کدام یک **Policy** بهینه را سریعتر یاد گرفت؟

ه. معماری شبکه عصبی مورد استفاده برای عامل **DQN** را شرح دهید. چرا این معماری را انتخاب کردید؟

پاسخ سوال

دنیای Wumpus یک محیط شبیه‌سازی کلاسیک در زمینه هوش مصنوعی است که اغلب برای آزمایش و آموزش الگوریتم‌های جستجو و یادگیری تقویتی (Reinforcement Learning) استفاده می‌شود. در این بازی، یک عامل (Agent) باید در یک شبکه مربع‌ای (Grid) حرکت کند تا طلا را پیدا کند و از Wumpus (هیولا) و چاله‌ها اجتناب کند. در ادامه، به طور کلی به حل دنیای Wumpus و چالش‌های آن می‌پردازیم:

اهداف بازی:

یافتن طلا: عامل باید طلا را پیدا کند و به دست آورد.

اجتناب از Wumpus:عامل باید از برخورد با Wumpus که می‌تواند آن را بکشد، اجتناب کند.

اجتناب از چاله‌ها: عامل باید از افتادن در چاله‌ها که باعث مرگ آن می‌شود، اجتناب کند.

بازگشت به نقطه شروع: پس از یافتن طلا، عامل باید به نقطه شروع بازگردد.

اجزاء بازی:

عامل (Agent):موجودیتی که در شبکه حرکت می‌کند و تصمیمات می‌گیرد.

Wumpus:هیولایی که می‌تواند عامل را بکشد اگر عامل به آن برخورد کند.

طلا (Gold):هدفی که عامل باید آن را پیدا کند.

چاله‌ها (Pits):مکان‌هایی که عامل نباید به آن‌ها بیفتد.

شبکه مربع‌ای (Grid):محیطی که عامل در آن حرکت می‌کند.

قوانین بازی:

حرکت: عامل می‌تواند در هر مرحله به یکی از چهار جهت (چپ، راست، بالا، پایین) حرکت کند.

شلیک: عامل می‌تواند در چهار جهت (چپ، راست، بالا، پایین) تیر شلیک کند تا Wumpus را بکشد.

حسگرها: عامل می‌تواند نشانه‌هایی از محیط دریافت کند:

بوی بد: نشان‌دهنده حضور Wumpus در یکی از خانه‌های مجاور.

نسیم: نشان‌دهنده حضور چاله در یکی از خانه‌های مجاور.

درخشش: نشان‌دهنده حضور طلا در خانه‌ای که عامل در آن قرار دارد.

استراتژی حل:

یادگیری تقویتی(Reinforcement Learning)

استفاده از الگوریتم‌های یادگیری تقویتی مانند Q-Learning یا DQN برای آموزش عامل به تصمیم‌گیری بهینه در محیط دنیای .Wumpus

عامل با استفاده از پاداش‌ها و تنبیهات یاد می‌گیرد که کدام اقدامات بهینه هستند و چگونه باید در محیط حرکت کند.

چالش‌ها:

عدم قطعیت: عامل ممکن است اطلاعات کامل و دقیقی از محیط نداشته باشد و باید با استفاده از حسگرها و نتیجه‌گیری‌های منطقی به تصمیمات خود برسد.

خطرات: عامل باید از Wumpus و چاله‌ها اجتناب کند که باعث مرگ آن می‌شوند.

مدیریت منابع: عامل ممکن است تعداد محدودی تیر داشته باشد و باید به صورت بهینه از آن‌ها استفاده کند.

دنیای Wumpus یک محیط پیچیده و چالش‌برانگیز است که به‌خوبی می‌تواند برای آزمایش و آموزش الگوریتم‌های جستجو و یادگیری تقویتی استفاده شود. حل این محیط نیازمند ترکیبی از منطق، جستجو و یادگیری است تا عامل بتواند به طور مؤثری در محیط حرکت کند، طلا را پیدا کند و از خطرات اجتناب کند.

ابتدا در سوال یک محیط شبیه‌سازی به نام "Wumpus World" را ایجاد می‌کنیم که یک بازی کلاسیک در هوش مصنوعی است. در این بازی، یک عامل (ربات) در یک شبکه مربع‌ای حرکت می‌کند و هدف آن یافتن طلا و اجتناب از Wumpus (هیولا) و چاله‌ها است. این محیط به عنوان یک محیط آموزشی برای الگوریتم‌های یادگیری تقویتی (Reinforcement Learning)استفاده می‌شود. این بخش از کد می‌تواند به عنوان یک محیط آموزشی برای الگوریتم‌های یادگیری تقویتی استفاده شود. الگوریتم‌های یادگیری تقویتی می‌توانند در این محیط تمرین کنند و یاد بگیرند که چگونه عامل را به بهترین شکل هدایت کنند تا طلا را پیدا کرده و از Wumpus و چاله‌ها اجتناب کنند.

در ادامه به ترتیب کد را توضیح می دهیم:

ابتدا پکیج های موردنیاز برای ایجاد یک نمایشگر مجازی، استفاده از محیط های بازی و شبیه سازی، رندر کردن محیط بازی و ایجاد یک نمایشگر مجازی (با اندازه ی 1400*900 پیکسل) را نصب کرده و ایجاد می کنیم، سپس محیط Wumpus World را تعریف می کنیم.

در این کلاس فضای مشاهده شامل موقعیت عامل، موقعیتWumpus ، و نشانگرهای دودویی برای طلا و چاله و فضای عمل شامل ۸ عمل مختلف حرکت به چپ، راست، بالا، پایین و شلیک به چپ، راست، بالا، پایین تعریف می شوند. سپس موقعیت‌های اولیه عامل، طلا، Wumpus، و چاله را تنظیم می‌کنیم و مقدار اولیه پاداش کل را به ۰ تنظیم می‌کنیم. در ادامه کد اگر حالت رندر humanباشد، PyGameرا راه‌اندازی می‌کند و یک پنجره بازی با عنوان "Wumpus World" ایجاد می‌کند.

در ادامه موقعیت‌های اولیه عامل و Wumpus را بازنشانی می‌کنیم، وضعیت زنده بودن Wumpus را به True تنظیم کرده و پاداش کل را به ۰ بازنشانی می‌کنیم. سپس مشاهده فعلی را برمی‌گردانیم.

سپس یک گام در محیط بازی را شبیه‌سازی کرده و عمل عامل را پردازش می‌کنیم. بسته به نوع عمل (حرکت یا شلیک)، موقعیت عامل یا وضعیت Wumpus به‌روز می‌شود و پاداش‌ها محاسبه می‌شوند. همچنین وضعیت‌های نهایی مانند پیدا کردن طلا، افتادن در چاله یا خوردن توسط Wumpus بررسی می‌شوند. در پایان، مشاهده جدید، پاداش و وضعیت پایان بازی برگردانده می‌شود.

سپس در متد های کمکی موقعیت posرا بر اساس عمل actionبه‌روز می‌کنیم و بررس میکنیم که آیا action برای موقعیت معتبر است یا نه.

همچنین بررسی می‌کنیم که آیا شلیک به جهت actionمنجر به کشتن Wumpus می‌شود یا نه.

سپس مشاهده فعلی محیط را برمی‌گردانیم که شامل موقعیت عامل، موقعیتWumpus ، و نشانگرهای دودویی برای طلا و چاله است.

متد رندر محیط را رندر می‌کند. در حالت human، شبکه بازی، عامل، Wumpus، طلا، و چاله را رسم می‌کند و صفحه نمایش را به‌روزرسانی می‌کند و متد PyGame close را در صورت استفاده از حالت humanمی‌بندد و در پایان یک نمونه از کلاس WumpusWorldEnvایجاد می‌کنیم و حالت رندر را humanتنظیم می‌کنیم.

۸ تا اکشن داریم: شلیک به چپ، راست، بالا، پایین و همچنین حرکت به بالا، پایین ، چپ و راست .

همچنین wumpus به چهار طرف به صورت رندم حرکت می کند.

موقعیت اولیه agent، gold، wumpus و pit و همچنین مقدار اولیه پاداش به صورت زیر مشخص شده است.

```
# Initialize positions of the agent, gold, wumpus, and pit
        self.agent_start_pos = np.array([0, 0])
        self.agent_pos = self.agent_start_pos.copy()
```

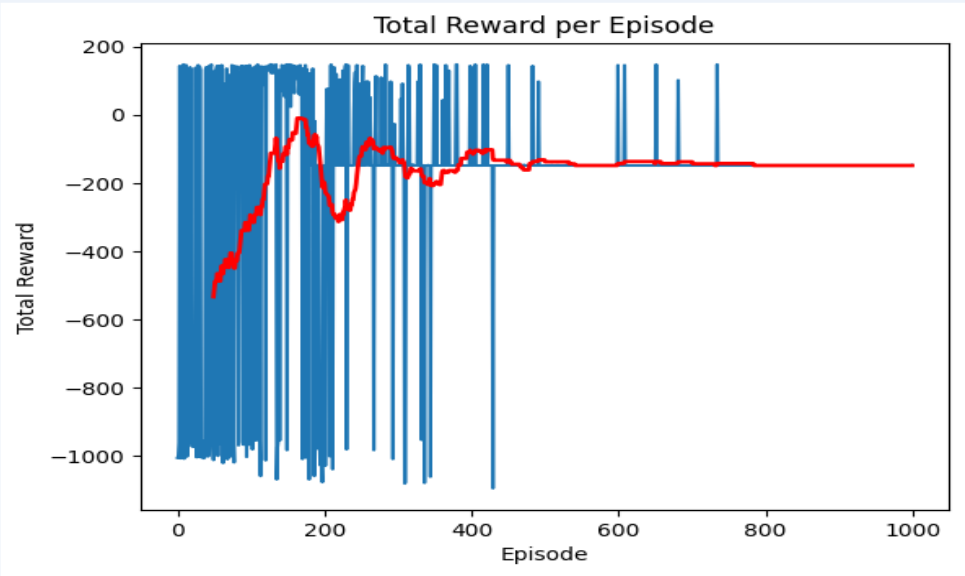
```
        self.gold_pos = np.array([3, 1])
        self.wumpus_pos = np.array([0, 3])
        self.pit_pos = np.array([1, 3])
        self.wumpus_alive = True
        self.total_reward = 0
```

همچنین مشخص میکنیم که در چهار طرف grid به بعضی از طرف ها نمیتونه حرکت کنه(مثلا در خانه ی اول نمیتونه به چپ و بالا حرکت کنه) ولی در بقیه خانه ها میتونه به چهارطرف حرکت کنه.

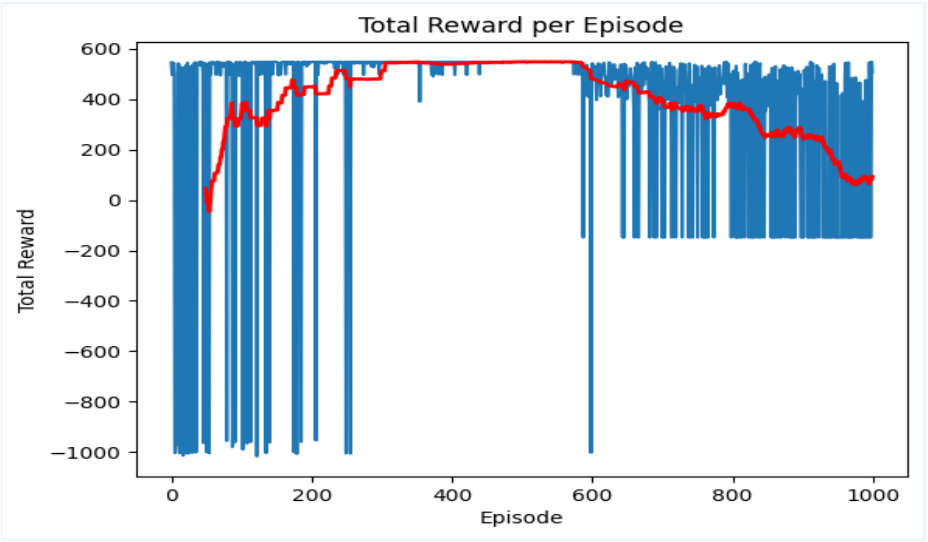
همچنین یک اپسیلون داریم که نرخ رندم بودن است یعنی agent مثلا ده استپ اول با اپسیلون یک راه میره تا اطلاعاتشو بالا ببره و ماتریس Q رو آپدیت کنه بعد اپسیلونه که از یه حدی پایین تر اومد میاد و بهترین ماتریس Q رو انتخاب میکنه که مقدار ماکسیمم یک دارد و با ضریب ۰,۹۹۵ کاهش می یابد تا آرام آرام به سمت مقدار مینیمم بره

برای قسمت Q-Learning وقتی برای گرفتن طلا پاداش +۱۰۰ در نظر میگیریم و برای هر حرکت پاداش -۱ در نظر میگیریم.

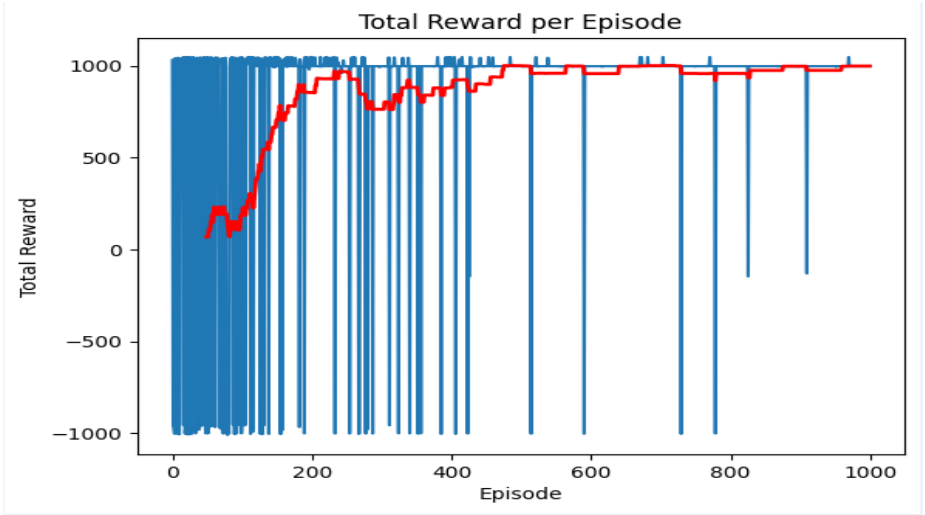
با این اعداد به جواب نمیرسیم چون داخل حلقه می افته و ۵۰۰ تا استپ راه میره و -۵۰۰ تا توبیخ میشه و چون قسمت های امتیازی رو داخل کد آوردیم محیط پیچیده میشه و اگه ۱۰۰ تا پاداش برای پیدا کردن طلا در نظر بگیریم براش مهم نیست و دنبال طلا نمیگرده و نتیجه به صورت زیر درمیاد.



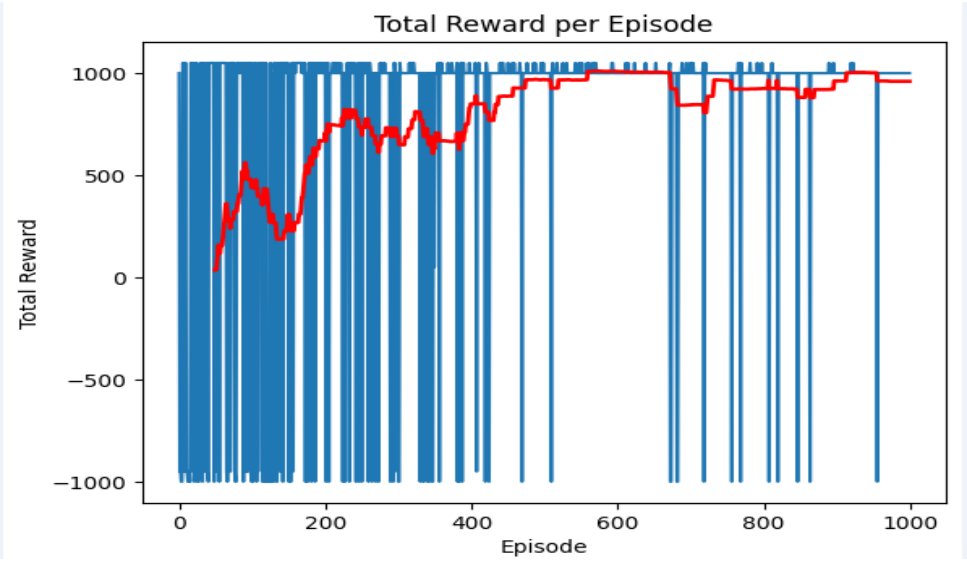
حالا میایم و به جای پاداش +۱۰۰ برای پیدا کردن طلا پاداش +۵۰۰ در نظر میگیریم و میبینیم که بهتر شده و داره تلاش میکنه طلارو پیدا کنه ولی همچنان راه بهینه رو پیدا نکرده.



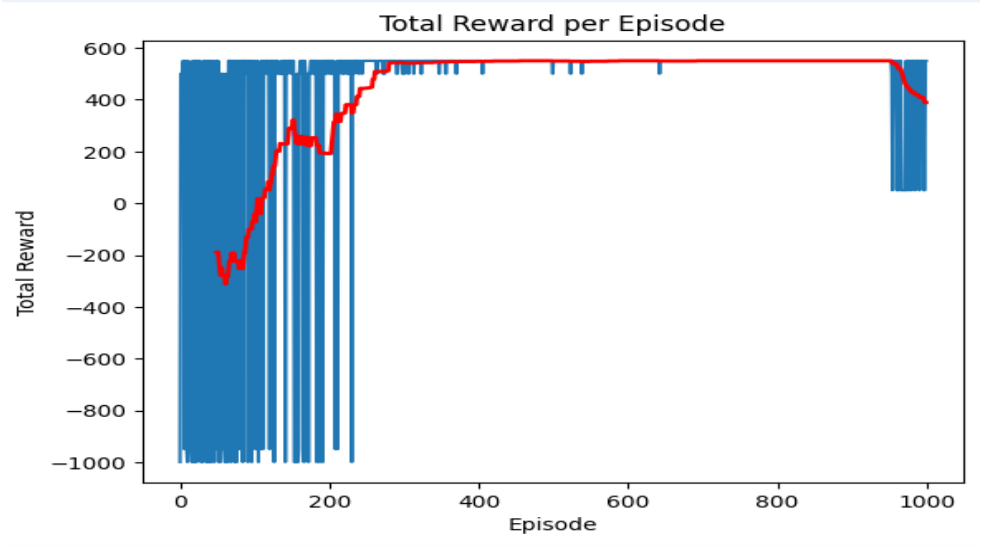
حالا میایم و به جای +۵۰۰ از ریوارد+۱۰۰۰ برای پیدا کردن طلا استفاده میکنیم. میبینیم که چون پاداش پیدا کردن طلا زیاد شده سعی کرده راه بهینه رو برای پیدا کردن طلا پیدا کنه و پیدا کرده.



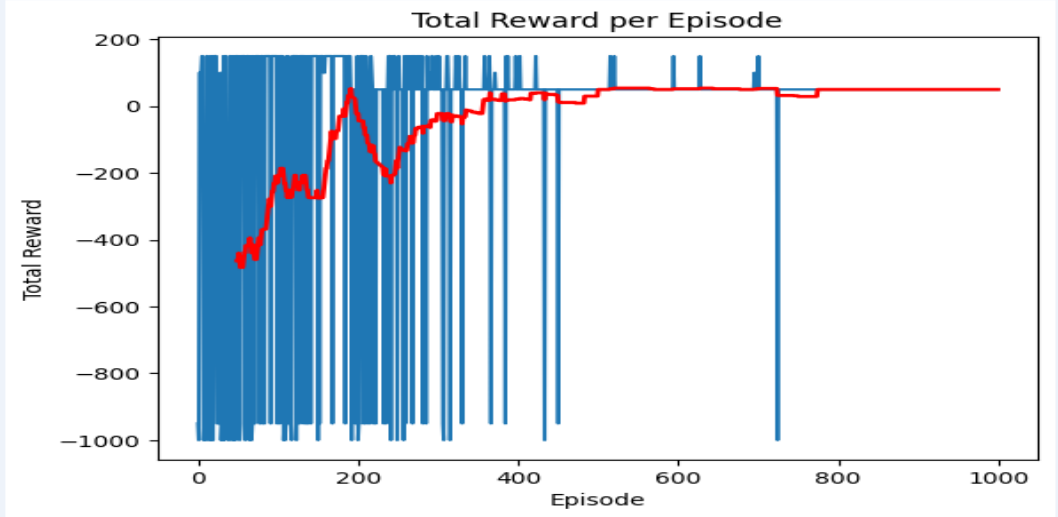
حالا میایم و پنالتی برای هر استپ حرکت رو به جای منفی یک، صفر در نظر میگیریم و پاداش برای پیدا کردن طلارو همون +۱۰۰۰ میذاریم. همانطور که مشخص است، چون برای هر حرکت به جای ریوارد منفی ما ریوارد صفر در نظر گرفته ایم سعی کرده که طلا را پیدا کند به خاطر اینکه به هر طرف حرکت کنه نگران گرفتن ریوارد منفی نیست و همچنین پاداش پیدا کردن طلا نیز زیاد است، راه رو پیدا کرده. یعنی این پارامترها یک پارامتر بهینه هستن برای agent ما برای پیدا کردن طلا و نیفتادن در چاله و کشتن wumpus.



حالا میایم و برای پیدا کردن طلا ریوارد +۵۰۰ در نظر میگیریم و پنالتی هر حرکت را صفر در نظر میگیریم. همانطور که از شکل مشخص است سعی کرده طلا رو پیدا کنه ولی نتونسته.



حالا میایم و برای پیدا کردن طلا ریوارد +۱۰۰ در نظر میگیریم و پنالتی هر حرکت را صفر در نظر میگیریم. همانطور که از شکل مشخص است سعی کرده طلا رو پیدا کنه ولی پیدا نکرده.



همونطور که از شکل ها مشخص است، اگر ریوارد پیدا کردن طلا را +۱۰۰۰ در نظر بگیریم سریع طلا را پیدا می کند.

DQN (Deep Q-Networks)

DQN یا شبکه‌های Q عمیق، یک روش یادگیری تقویتی (Reinforcement Learning) است که توسط DeepMind معرفی شده است. این روش از شبکه‌های عصبی عمیق (Deep Neural Networks)برای تقریب تابع Q استفاده می‌کند. در ادامه به توضیح مفاهیم کلیدی و روند کار DQN پرداخته می‌شود.

در DQN، از یک شبکه عصبی عمیق برای تقریب تابع Q استفاده می‌شود. این شبکه عصبی ورودی‌اش حالت s است و خروجی‌اش مقادیر Q برای هر اقدام ممکن در آن حالت است.

معماری و روند کار DQN

شبکه عصبی عمیق (Deep Neural Network)

شبکه عصبی عمیق ورودی‌اش حالت s و خروجی‌اش مقادیر Q برای هر اقدام ممکن در آن حالت است.

حافظه بازپخش تجربیات (Experience Replay)

تجربیات (s, a, r, s', done)در یک حافظه بازپخش ذخیره می‌شوند.

در هر قدم آموزش، نمونه‌ای تصادفی از تجربیات برای به‌روزرسانی شبکه استفاده می‌شود. این کار باعث می‌شود که داده‌ها همبستگی کمتری داشته باشند و شبکه پایدارتر آموزش ببیند.

شبکه هدف (Target Network)

یک نسخه کپی از شبکه Q اصلی که به صورت دوره‌ای به‌روزرسانی می‌شود.

استفاده از شبکه هدف باعث پایداری بیشتر در آموزش می‌شود.

به‌روزرسانی شبکه Q

برای به‌روزرسانی شبکه Q، از معادله به‌روزرسانی Bellman استفاده می‌شود:

$$\alpha \left(r + \gamma \max_{'a} Q'(s', a') - Q(s, a) \right) + Q(s, a) \leftarrow Q(s, a)$$

در این معادله، alpha نرخ یادگیری و gamma ضریب تنزیل است. Q' شبکه هدف است.

الگوریتم DQN به صورت خلاصه

۱. مقداردهی اولیه شبکه Q و شبکه هدف با وزن‌های تصادفی.

۲. مقداردهی اولیه حافظه بازپخش تجربیات.

۳. برای هر قسمت (Episode)

۱. تنظیم حالت اولیه s.

۲. برای هر قدم در قسمت:

- انتخاب اقدام a با استفاده از سیاست greedy – epsilon.

- انجام اقدام a و مشاهده پاداش r و حالت جدید s'.

- ذخیره تجربه (s, a, r, s', done) در حافظه بازپخش.

- نمونه‌گیری مینی‌بچ از حافظه بازپخش.

- محاسبه هدف Q برای هر نمونه:

$$\gamma \max_{'a} Q'(s', a') + r = y$$

- محاسبه از دست رفتن (Loss) و به‌روزرسانی شبکه Q.

- تنظیم حالت s به s'.

۳. به‌روزرسانی شبکه هدف به صورت دوره‌ای.

کاربردهای DQN

DQN در بسیاری از مسائل یادگیری تقویتی موفقیت‌آمیز بوده است، از جمله بازی‌های ویدیویی (مانند بازی‌های Atari)، رباتیک، کنترل‌های صنعتی و بسیاری از مسائل دیگر.

مزایا و معایب DQN

مزایا

- استفاده از شبکه عصبی عمیق برای تقریب تابع Q که امکان یادگیری در محیط‌های پیچیده را فراهم می‌کند.

- استفاده از حافظه بازپخش تجربیات برای کاهش همبستگی در داده‌های آموزشی.

- استفاده از شبکه هدف برای پایداری بیشتر در آموزش.

معایب

- نیاز به تنظیم هایپرفرامترهای مختلف مانند نرخ یادگیری، ضریب تنزیل، و اندازه حافظه بازپخش.

- حساسیت به انتخاب سیاست greedy – epsilon. و نرخ کاهش epsilon.

- می‌تواند نیاز به منابع محاسباتی بالایی داشته باشد.

کد محیط شبیه‌سازی شده از دنیای Wumpus World را با استفاده از کتابخانه Gymnasium و Pygame پیاده‌سازی می‌کند. این محیط شامل یک شبکه مربعی است که در آن یک عامل (agent)باید طلا را پیدا کند، از گودال‌ها و Wumpusاجتناب کند.

متد سازنده :__init____

اندازه شبکه مربعی، اندازه پنجره و اندازه هر سلول را تنظیم می‌کند.

فضای مشاهده شامل موقعیت عامل، Wumpus، و اندیکاتورهای باینری برای طلا و گودال را تعریف می‌کند.

فضای اقدام شامل ۸ اقدام مختلف (حرکت به چپ، راست، بالا، پایین و شلیک در چهار جهت) را تنظیم می‌کند.

موقعیت‌های اولیه عامل، طلا، Wumpus و گودال را تنظیم می‌کند. همچنین وضعیت زنده بودن Wumpus و مجموع پاداش را تعیین می‌کند.

اگر حالت رندرینگ humanباشد، PyGame را مقداردهی و یک پنجره ایجاد می‌کند.

متد reset موقعیت‌های اولیه را بازنشانی می‌کند و حالت اولیه محیط را بازمی‌گرداند.

متد step

یک اقدام را اجرا می‌کند، پاداش اولیه و وضعیت پایان را تنظیم می‌کند.

بررسی می‌کند که آیا اقدام حرکت یا شلیک است و بر اساس آن عمل می‌کند.

بررسی می‌کند که آیا عامل به طلا، گودال یا Wumpus رسیده است و وضعیت پایان را تنظیم می‌کند.

اگر Wumpus زنده است و محیط به پایان نرسیده است، Wumpus را به صورت تصادفی حرکت می‌دهد.

مجموع پاداش را به‌روزرسانی کرده و وضعیت فعلی محیط، پاداش و وضعیت پایان را بازمی‌گرداند.

متدهای کمکی

moveعامل یا Wumpus را بر اساس اقدام به موقعیت جدید حرکت می‌دهد.

valid_moveبررسی می‌کند که آیا حرکت معتبر است یا نه.

shootبررسی می‌کند که آیا شلیک عامل Wumpus را می‌کشد یا نه.

get_obsوضعیت فعلی محیط را به صورت دیکشنری بازمی‌گرداند.

متد render

محیط را در حالت human بصری یا rgb_array آرایه RGB نمایش می‌دهد. در حالت human، صفحه با رنگ سفید پر شده و شبکه، عامل، Wumpus، طلا و گودال را رسم می‌کند.

متد close

اگر حالت رندرینگ humanباشد، PyGame را خاتمه می‌دهد.

در نهایت، یک نمونه از محیط WumpusWorldEnv ایجاد می‌شود و حالت رندرینگ human تنظیم می‌شود. این محیط شبیه‌سازی به کاربر امکان می‌دهد تا با انجام اقدامات مختلف، عامل را در شبکه حرکت داده و طلا را پیدا کند در حالی که از خطرات مختلف اجتناب می‌کند.

معماری شبکه عصبی ما به صورت زیر انتخاب شده است

```
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_dim, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, action_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

یک شبکه ی عصبی با سه لایه ی کاملا متصل است که تعداد نورون های لایه های آن طبق بالا می باشد و همچنین از تابع فعالساز relu به دلیل سادگی و سرعت بالا در محاسبات و همچنین عملکرد خوب برای آن استفاده کرده ایم.

در ادامه یک کلاس به نام ReplayBufferتعریف می‌کنیم که برای ذخیره‌سازی و نمونه‌گیری تجربیات در الگوریتم‌های یادگیری تقویتی (Reinforcement Learning) استفاده می‌شود. این کلاس به منظور بهبود فرآیند یادگیری توسط شبکه‌ی عصبی، تجربیات گذشته را ذخیره و به صورت تصادفی برای به‌روزرسانی مدل انتخاب می‌کند.

تابع سازنده:

هنگام ایجاد یک نمونه از کلاس، یک شیء deque با حداکثر ظرفیت تعیین‌شده ایجاد می‌کند. این شیء تجربیات را نگه می‌دارد و در صورت پر شدن، قدیمی‌ترین تجربه را حذف می‌کند.

تابعpush:یک تجربه جدید شامل حالت، عمل، پاداش، حالت بعدی، و وضعیت نهایی بودن را به buffer اضافه می‌کند.

تابع sample: به صورت تصادفی یک batch از تجربیات را از buffer انتخاب می‌کند. تجربیات را به صورت جداگانه به متغیرهای حالت، عمل، پاداش، حالت بعدی، و وضعیت نهایی بودن تقسیم می‌کند. حالت‌ها و حالت‌های بعدی به آرایه‌های NumPy تبدیل می‌شوند و سایر متغیرها به همان صورت باز می‌گردند.

تابع len تعداد تجربیات موجود در buffer را باز می‌گرداند.

به طور کلی کلاس ReplayBuffer یک buffer حلقوی با ظرفیت مشخص فراهم می‌کند که تجربیات را ذخیره و در زمان نیاز به صورت تصادفی از آنها نمونه‌گیری می‌کند. این کار به الگوریتم‌های یادگیری تقویتی کمک می‌کند تا از تجربیات متنوع‌تری استفاده کنند و بهبود عملکرد یادگیری را ممکن می‌سازد.

کلاس DQNAgent برای تعریف یک عامل یادگیری تقویتی با استفاده از الگوریتم Deep Q-Network (DQN) طراحی شده است. این کلاس شامل تنظیمات و ویژگی‌هایی است که برای آموزش یک شبکه عصبی جهت پیش‌بینی ارزش‌های Q (Q-values) مورد نیاز هستند. در زیر توضیحی از وظایف و عملکرد این کلاس ارائه شده است:

توابع و وظایف کلاسDQNAgent

تابع سازنده:

- پارامترها:
- state_d ابعاد فضای حالت.
- action_dim ابعاد فضای عمل.
- lr نرخ یادگیری برای به‌روزرسانی وزن‌های شبکه.
- gamma نرخ تخفیف برای محاسبه پاداش‌های آینده.
- epsilon مقدار اولیه epsilon برای سیاست اکتشافی .epsilon-greedy
- epsilon_decay نرخ کاهش epsilon در طول زمان.

- epsilon_min حداقل مقدار epsilon.

ویژگی‌های تعریف‌شده:

- self.state_dim و self.action_dim ذخیره ابعاد فضای حالت و عمل.

- self.gamma, self.epsilon, self.epsilon_decay, self.epsilon_min تنظیم پارامترهای مربوط به الگوریتم یادگیری.

- self.device تعیین دستگاه (CUDA یا CPU) برای انجام محاسبات.

شبکه‌های عصبی:

- self.policy_net شبکه عصبی برای پیش‌بینی ارزش‌های Q.

- self.target_net شبکه عصبی هدف برای پایدارسازی یادگیری.

به‌روزرسانی شبکه هدف:

- پارامترهای policy_net به target_net کپی می‌شود و target_net در حالت ارزیابی قرار می‌گیرد.

بهینه‌ساز:

- self.optimizer بهینه‌ساز Adam برای به‌روزرسانی وزن‌های policy_net.

حافظه بازپخش:

- self.memory یک نمونه از کلاس ReplayBuffer برای ذخیره تجربیات گذشته.

- self.batch_size اندازه batch برای نمونه‌گیری از ReplayBuffer.

به طور کلی کلاس DQNAgent یک عامل یادگیری تقویتی با الگوریتم DQN را تعریف می‌کند. این کلاس شامل دو شبکه عصبی (یکی برای سیاست و یکی برای هدف)، یک بهینه‌ساز Adam، و یک حافظه بازپخش برای ذخیره و نمونه‌گیری تجربیات است. همچنین پارامترهای مختلفی برای کنترل فرآیند یادگیری و اکتشاف تنظیم شده‌اند.

تابع select_action در کلاس DQNAgent وظیفه انتخاب یک عمل (action) بر اساس سیاست اکتشافی epsilon-greedy را بر عهده دارد. این تابع به این صورت عمل می‌کند:

اکتشاف یا بهره‌برداری (Exploration vs. Exploitation)

- اگر یک عدد تصادفی بین ۰ و ۱ بزرگ‌تر از مقدار epsilon باشد، عامل بهره‌برداری می‌کند و عمل را بر اساس پیش‌بینی شبکه عصبی policy_net انتخاب می‌کند.

- در غیر این صورت، عامل اکتشاف می‌کند و یک عمل تصادفی را از فضای عمل انتخاب می‌کند.

بهره‌برداری:

- حالت (state)ورودی به یک torch.FloatTensor تبدیل می‌شود و یک بعد اضافی برای batch اضافه می‌شود.

- این حالت به دستگاه (CUDA یا CPU)منتقل می‌شود.

- با استفاده از torch.no_grad()، عملیات پیش‌بینی ارزش‌های Q برای این حالت بدون محاسبه گرادیان‌ها انجام می‌شود.

- بالاترین مقدار Q انتخاب می‌شود و شاخص مربوط به آن به عنوان عمل انتخاب‌شده بازگردانده می‌شود.

اکتشاف:

یک عمل تصادفی از فضای عمل بازگردانده می‌شود.

اگر random.randomیک عدد تصادفی بین ۰ و ۱بزرگ‌تر از epsilonباشد:

حالت ورودی به یک FloatTensor تبدیل می‌شود و به دستگاه مناسب منتقل می‌شود.

با استفاده از torch.no_grad شبکه عصبی policy_net ارزش‌های Q برای این حالت را محاسبه می‌کند.

عمل با بیشترین مقدار Q انتخاب می‌شود و شاخص آن بازگردانده می‌شود.

در غیر این صورت، یک عمل تصادفی از فضای عمل انتخاب و بازگردانده می‌شود.

به طور کلی تابع select_action با استفاده از سیاست اکتشافی epsilon-greedy، بین بهره‌برداری از شبکه عصبی برای انتخاب بهترین عمل و اکتشاف تصادفی عمل جدید جابه‌جا می‌شود. این کار به عامل یادگیری تقویتی کمک می‌کند تا هم به دنبال اعمال جدید باشد و هم از تجربیات گذشته برای بهبود عملکرد خود استفاده کند.

در ادامه تابع train_step در کلاس DQNAgent مسئول به‌روزرسانی وزن‌های شبکه عصبی policy_net بر اساس تجربیات ذخیره شده در ReplayBuffer است. این تابع به منظور بهبود شبکه عصبی و یادگیری ارزش‌های Q برای حالت‌های مختلف، از الگوریتم یادگیری Q استفاده می‌کند. در اینجا جزئیات مراحل اجرای این تابع آمده است:

بررسی تعداد تجربیات موجود:

ابتدا بررسی می‌کند که آیا تعداد تجربیات ذخیره‌شده در ReplayBuffer کمتر از batch_size است یا خیر. اگر کمتر باشد، تابع خاتمه می‌یابد و هیچ‌کاری انجام نمی‌شود.

نمونه‌گیری از حافظه بازپخش (Replay Buffer)

یک batch از تجربیات شامل حالت‌ها، اعمال، پاداش‌ها، حالت‌های بعدی و وضعیت نهایی بودن را از حافظه بازپخش نمونه‌گیری می‌کند.

تبدیل داده‌ها به تنسورهای PyTorch:

داده‌های نمونه‌گیری شده به FloatTensor یا LongTensor تبدیل می‌شوند و به دستگاه مناسب منتقل می‌شوند.

محاسبه ارزش‌های Q برای حالت‌های فعلی:

با استفاده از policy_net ارزش‌های Q برای حالت‌های فعلی محاسبه می‌شوند و بر اساس اعمال انجام شده از آنها نمونه‌گیری می‌شود.

محاسبه ارزش‌های Q برای حالت‌های بعدی:

با استفاده از target_net ارزش‌های Q برای حالت‌های بعدی محاسبه می‌شوند و بالاترین مقدار آنها انتخاب می‌شود.

ارزش‌های Q برای حالت‌های بعدی از گرادیان‌ها جدا می‌شوند تا در به‌روزرسانی وزن‌های شبکه شرکت نکنند.

محاسبه ارزش‌های Q مورد انتظار:

ارزش‌های Q مورد انتظار با استفاده از پاداش‌های فعلی و ارزش‌های Q برای حالت‌های بعدی محاسبه می‌شوند. این محاسبه شامل نرخ تخفیف (`gammaو وضعیت نهایی بودن تجربیات است.

محاسبه و اعمال تابع خطا Loss Function: خطا بین ارزش‌های Q محاسبه‌شده توسط policy_net و ارزش‌های Q مورد انتظار با استفاده از تابع خطای میانگین مربعات (MSE)محاسبه می‌شود.

گرادیان‌ها صفر می‌شوند، خطا به عقب انتشار داده می‌شود و وزن‌های شبکه با استفاده از بهینه‌ساز Adam به‌روزرسانی می‌شوند.

بطور کلی تابع train_step مراحل زیر را برای به‌روزرسانی وزن‌های شبکه عصبی policy_net انجام می‌دهد:

۱. بررسی تعداد تجربیات موجود در حافظه بازپخش.

۲. نمونه‌گیری یک batch از تجربیات.

۳. تبدیل داده‌ها به تنسورهای .PyTorch

۴. محاسبه ارزش‌های Q برای حالت‌های فعلی با استفاده از .policy_net

۵. محاسبه ارزش‌های Q برای حالت‌های بعدی با استفاده از .target_net

۶. محاسبه ارزش‌های Q مورد انتظار.

۷. محاسبه و اعمال تابع خطا برای به‌روزرسانی وزن‌های شبکه عصبی.

این فرآیند به عامل یادگیری تقویتی کمک می‌کند تا به مرور زمان یاد بگیرد که کدام اعمال در هر حالت بهتر است و عملکرد خود را بهبود دهد.

تابع update_target وزن‌های شبکه عصبی policy_net را به شبکه عصبی target_net کپی می‌کند. هدف این کار پایدار کردن فرآیند یادگیری است، زیرا target_net کمتر به‌روز می‌شود و به عنوان یک هدف ثابت‌تر برای محاسبه ارزش‌های Q استفاده می‌شود.

تابع train_dqn فرآیند آموزش عامل یادگیری تقویتی (agent)را در محیط (env) با استفاده از الگوریتم Deep Q-Network (DQN) برای تعداد مشخصی از اپیزودها انجام می‌دهد. مراحل این فرآیند به شرح زیر است:

تعریف متغیرهای اولیه:

reward_history لیستی برای ذخیره مجموع پاداش‌های به‌دست‌آمده در هر اپیزود.

حلقه اصلی آموزش:

برای هر اپیزود، محیط بازنشانی می‌شود و حالت اولیه بازیابی می‌شود. حالت اولیه به صورت یک بردار شامل اطلاعات مربوط به عامل، وامپوس، طلا و چاله‌ها ایجاد می‌شود.

total_rewardبرای نگهداری مجموع پاداش‌های هر اپیزود تنظیم می‌شود.

حلقه داخلی برای هر اپیزود:

- برای حداکثر ۱۰۰۰ گام:

- عامل یک عمل بر اساس سیاست اکتشافی epsilon-greedy انتخاب می‌کند.

- محیط یک گام جلو می‌رود و حالت بعدی، پاداش و وضعیت نهایی بودن بازی را بازمی‌گرداند. حالت بعدی نیز به صورت یک بردار مشابه با حالت اولیه ایجاد می‌شود.

- تجربه شامل حالت، عمل، پاداش، حالت بعدی و وضعیت نهایی بودن به حافظه بازپخش عامل اضافه می‌شود.

- تابع train_stepبرای به‌روزرسانی وزن‌های شبکه عصبی policy_net فراخوانی می‌شود.

- حالت فعلی به حالت بعدی به‌روز می‌شود و پاداش به total_reward اضافه می‌شود.

- اگر بازی تمام شود (وضعیت نهایی باشد)، حلقه شکسته می‌شود.

به‌روزرسانی شبکه هدف:

- پس از پایان هر اپیزود، تابع update_target فراخوانی می‌شود تا وزن‌های policy_net به target_netکپی شوند.

به‌روزرسانی epsilon:

- مقدار epsilon کاهش می‌یابد طبق نرخ کاهشی epsilon_decay تا عامل به تدریج کمتر اکتشاف کند و بیشتر بهره‌برداری کند. اگر مقدار epsilon به حداقل مقدار epsilon_min برسد، دیگر کاهش نمی‌یابد.

ذخیره و نمایش نتایج:

- مجموع پاداش‌های هر اپیزود به لیست reward_historyاضافه می‌شود.

- اطلاعات اپیزود شامل شماره اپیزود، مجموع پاداش و مقدار epsilon چاپ می‌شود.

به طور کلی تابع train_dqn عامل یادگیری تقویتی را در محیط آموزش می‌دهد و شامل مراحل زیر است:

۱. بازنشانی محیط و جمع‌آوری حالت اولیه.

۲. اجرای حلقه‌ای برای هر اپیزود که در آن عامل به طور متناوب عمل انتخاب می‌کند، محیط به جلو می‌رود، تجربیات به حافظه اضافه می‌شوند و شبکه عصبی به‌روزرسانی می‌شود.

۳. به‌روزرسانی شبکه هدف پس از هر اپیزود.

۴. کاهش مقدار epsilon برای کنترل فرآیند اکتشاف و بهره‌برداری.

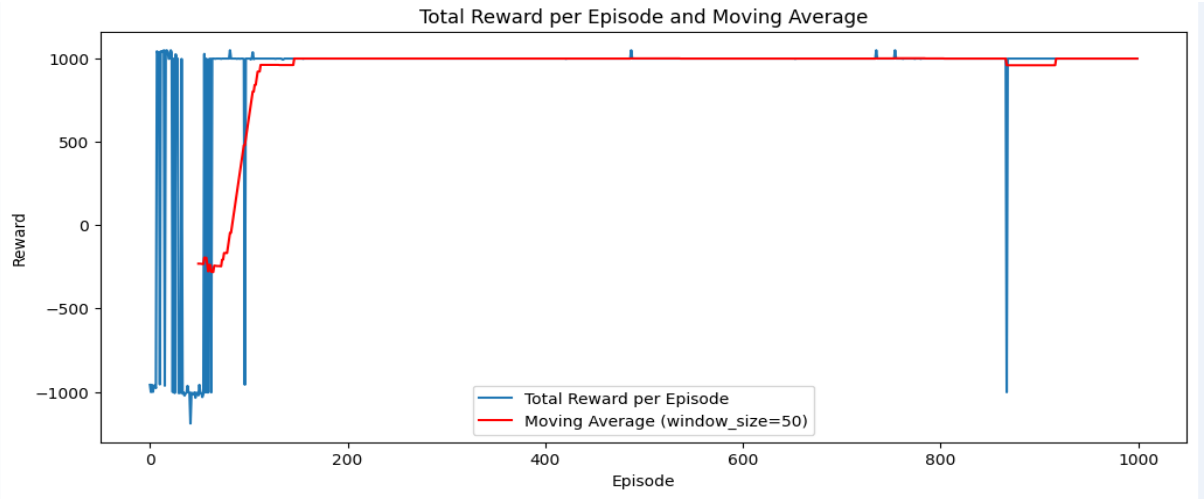
۵. ذخیره و نمایش مجموع پاداش‌های هر اپیزود.

در نهایت، این تابع لیستی از مجموع پاداش‌های هر اپیزود را بازمی‌گرداند که می‌توان از آن برای ارزیابی عملکرد عامل استفاده کرد.

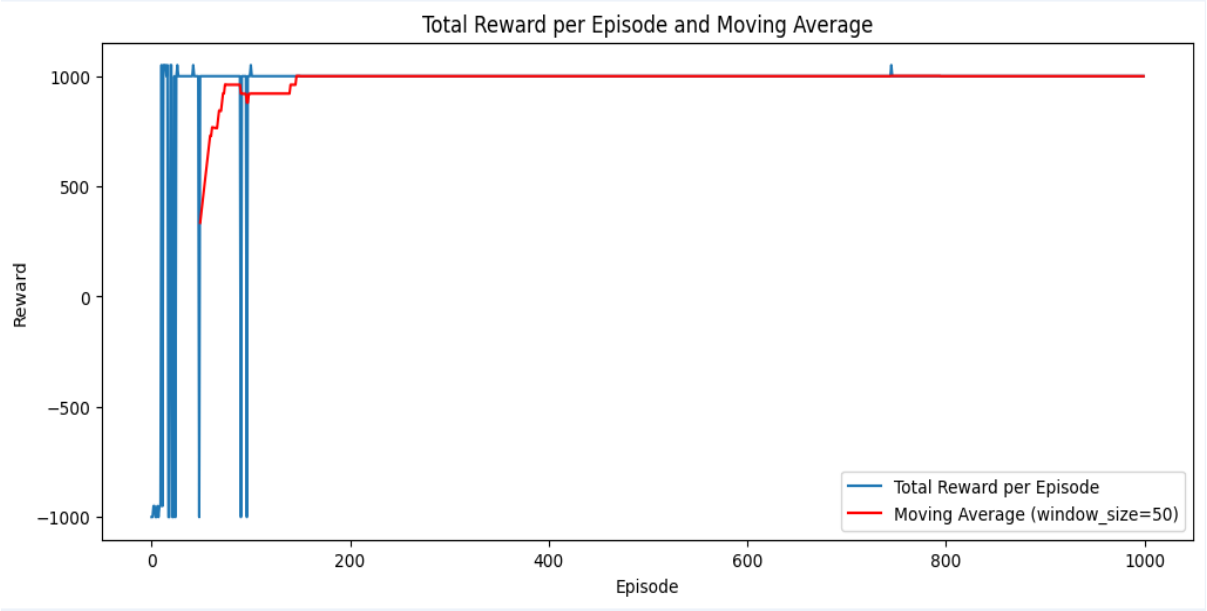
در نهایت محیط و عامل با مشخصات فضای حالت و عمل مقداردهی اولیه می شوند و عامل به مدت ۱۰۰۰ اپیزود در محیط آموزش می‌بیند سپس تاریخچه پاداش‌های هر اپیزود ذخیره می‌شود.

میانگین متحرک پاداش‌ها با پنجره ۵۰ محاسبه می‌شود و نموداری از تاریخچه پاداش‌ها و میانگین متحرک آنها ترسیم می‌شود تا روند یادگیری عامل نمایش داده شود.

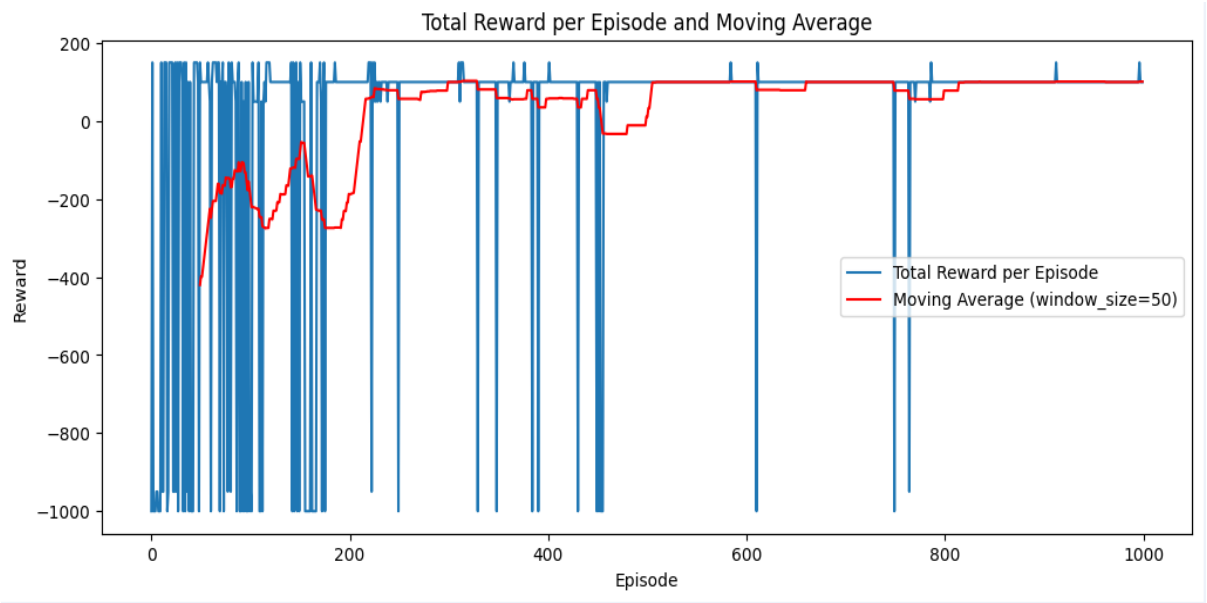
در نهایت نتیجه با ریوارد -۱ برای هر حرکت و +۱۰۰۰ برای پیدا کردن طلا به صورت زیر است.



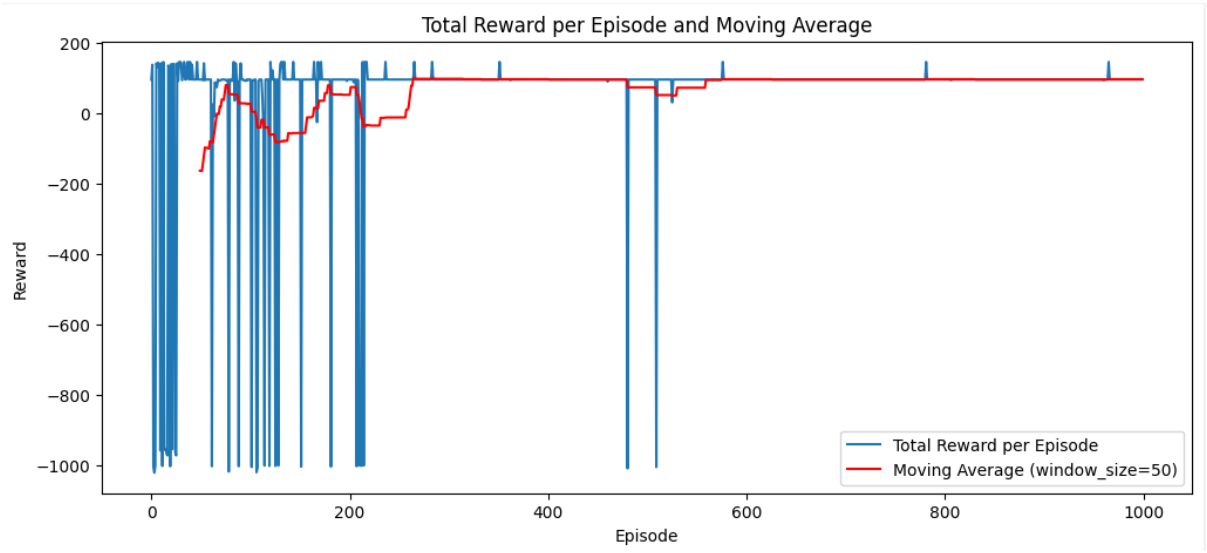
با ریوارد ۰ برای هر حرکت و +۱۰۰۰ برای پیدا کردن طلا



با ریوارد ۰ برای هر حرکت و +۱۰۰ برای پیدا کردن طلا



با ریوارد -۱ برای هر حرکت و +۱۰۰ برای پیدا کردن طلا



همانطور که میبینیم با ریوارد ۱۰۰ به نتیجه نرسیده ولی با ریوارد ۱۰۰۰ چون ریوارد بزرگتری داشته سعی کرده طلا را پیدا کند و سریع پیدا کرده است.

در Q-learning و DQN ، عملکرد عامل در طول زمان با به‌روزرسانی تدریجی تابع Q یا شبکه Q بهبود می‌یابد. این به‌روزرسانی‌ها به عامل کمک می‌کند تا بهترین اقدامات را در هر وضعیت انتخاب کند و در نتیجه پاداش بیشتری دریافت کند.

بهبود تدریجی :با گذشت زمان و تجربیات بیشتر، جدول Q یا شبکه عصبی دقیق‌تر می‌شود.

کاهش کاوش :کاهش تدریجی epsilon باعث می‌شود که عامل بیشتر بهره‌برداری کند.

به‌روزرسانی منظم :به‌روزرسانی‌های منظم شبکه سیاست و شبکه هدف در DQN به بهبود سیاست عامل کمک می‌کند.

این فرآیند تکراری باعث می‌شود که عامل به مرور زمان سیاست بهینه‌تری پیدا کند و عملکرد بهتری داشته باشد.

بر اساس دو نموداری که ارائه کردید، می‌توانیم عملکرد دو الگوریتم Q-learning و DQN را مقایسه کنیم:

در Q-learning پاداش‌ها در بسیاری از اپیزودها نوسانات زیادی دارند و برخی از اپیزودها پاداش‌های منفی بزرگی دریافت می‌کنند. با این حال، یک روند کلی به سمت بهبود دیده می‌شود و به نظر می‌رسد که الگوریتم به تدریج یاد می‌گیرد و عملکرد خود را بهبود می‌بخشد. میانگین متحرک پاداش‌ها (خط قرمز) نشان می‌دهد که الگوریتم به تدریج پاداش‌های بالاتری را کسب می‌کند، اما همچنان نوسانات زیادی دارد.

در DQN نمودار نشان می‌دهد که پس از چند اپیزود اولیه، پاداش‌ها به سرعت به مقدار ثابتی نزدیک به حداکثر پاداش (۱۰۰۰) می‌رسند و بسیار پایدار می‌شوند. الگوریتم DQN به سرعت یاد می‌گیرد و پاداش‌های بسیار بالایی را در مدت زمان کوتاهی کسب می‌کند. میانگین متحرک پاداش‌ها (خط قرمز) نشان می‌دهد که الگوریتم DQN به سرعت به حداکثر پاداش رسیده و پس از آن تقریباً ثابت باقی می‌ماند.

عملکرد بهتر DQN: به وضوح می‌توان مشاهده کرد که DQN عملکرد بسیار بهتری نسبت به Q-learning دارد. پاداش‌های DQN به سرعت به مقدار حداکثر می‌رسند و ثابت باقی می‌مانند، در حالی که Q-learning نوسانات زیادی دارد و به تدریج بهبود می‌یابد.

پایداری DQN :DQN پس از چند اپیزود اولیه به پاداش‌های ثابتی دست می‌یابد و نشان می‌دهد که الگوریتم به سرعت به یک سیاست بهینه رسیده است.

در نتیجه، DQN عملکرد بهتری نسبت به Q-learning در این محیط دارد.

نرخ اکتشاف اپسیلون در الگوریتم‌های Q-learning و DQN نقش بسیار مهمی در فرآیند یادگیری ایفا می‌کند. Epsilon سیاست انتخاب اقدام عامل را تعیین می‌کند، که بین کاوش exploration و بهره‌برداری exploitation تعادل ایجاد می‌کند.

تأثیر epsilon بالا

زمانی که epsilon بالا است، عامل بیشتر تمایل به کاوش اقدامات تصادفی دارد. این به معنای این است که عامل به جای انتخاب بهترین اقدام شناخته‌شده، اقداماتی را امتحان می‌کند که ممکن است کمتر شناخته شده باشند.

epsilon بالا باعث می‌شود که عامل تجربیات متنوع‌تری کسب کند. این امر می‌تواند به کشف سیاست‌های بهتر کمک کند، به خصوص در مراحل اولیه یادگیری که عامل اطلاعات کمی در مورد محیط دارد.

به دلیل انتخاب‌های تصادفی، پاداش‌ها ممکن است نوسانات زیادی داشته باشند. این نوسانات می‌توانند منجر به پاداش‌های منفی یا پایین‌تر در برخی اپیزودها شوند، اما در عین حال، عامل ممکن است به سیاست‌های بهینه جدیدی دست یابد.

تأثیر epsilonپایین

زمانی که epsilon پایین است، عامل بیشتر از اقدامات بهترین شناخته‌شده استفاده می‌کند و کمتر تمایل به کاوش اقدامات جدید دارد.

بهره‌برداری بیشتر باعث می‌شود که پاداش‌ها پایدارتر و کمتر نوسان داشته باشند. عامل از سیاست‌های اثبات‌شده و موفق خود استفاده می‌کند و این منجر به کسب پاداش‌های بالاتر می‌شود.

با این حال، اگر epsilon بیش از حد پایین باشد، عامل ممکن است از کاوش و کشف سیاست‌های جدید باز بماند. این امر می‌تواند منجر به گیر افتادن در سیاست‌های زیر بهینه شود و عملکرد کلی عامل را محدود کند.

در نمودار Q-learning نوسانات زیاد در پاداش‌ها در ابتدای یادگیری، به خصوص زمانی که epsilon بالا است. این نشان‌دهنده کاوش زیاد عامل و کسب تجربیات متنوع است.

به مرور زمان و با کاهش epsilon عامل شروع به بهره‌برداری بیشتر از تجربیات خود می‌کند و پاداش‌ها به تدریج افزایش می‌یابند.

در نمودار DQN بهبود سریع در اوایل یادگیری، که نشان می‌دهد عامل با کاوش اولیه به سرعت سیاست‌های بهینه را کشف می‌کند.

پس از کاهش epsilon عامل به سرعت به پاداش‌های ثابت و بالا دست می‌یابد، که نشان‌دهنده بهره‌برداری بیشتر از سیاست‌های بهینه است.

تنظیم مناسب epsilon بسیار حیاتی است. در مراحل اولیه یادگیری، epsilon بالا برای کاوش محیط و کسب تجربیات متنوع مفید است. اما به مرور زمان، کاهش epsilon و افزایش بهره‌برداری از سیاست‌های یادگرفته‌شده برای بهبود عملکرد عامل ضروری است.

استفاده از سیاست‌های تطبیقی که به صورت پویا epsilon را تنظیم می‌کنند، می‌تواند کمک کند تا عامل بهترین تعادل را بین کاوش و بهره‌برداری پیدا کند و به سرعت به سیاست‌های بهینه دست یابد.

این مشاهدات نشان می‌دهد که epsilon چگونه بر فرآیند یادگیری تأثیر می‌گذارد و چرا تنظیم مناسب آن برای دستیابی به بهترین عملکرد عامل حیاتی است.

تجزیه و تحلیل عملکرد Q-learning و DQN

در نمودار Q-learning، مشاهده می‌شود که پاداش‌ها به طور قابل ملاحظه‌ای نوسان دارند. این نشان می‌دهد که عامل در ابتدا با مشکلات زیادی روبرو شده است و چندین بار دچار افتادن در گودال یا خورده شدن توسط Wumpus شده است.

به نظر می‌رسد که پس از حدود ۴۰۰ تا ۵۰۰ اپیزود، عامل شروع به عملکرد پایدارتر می‌کند و به طور مداوم پاداش‌های مثبت (نزدیک به حداکثر پاداش) را دریافت می‌کند. این نشان می‌دهد که عامل Q-learning پس از حدود ۴۰۰ تا ۵۰۰ اپیزود به سیاستی دست یافته است که به طور مداوم طلا را پیدا می‌کند بدون افتادن در گودال یا خورده شدن توسط .Wumpus

روند یادگیری Q-learning آهسته‌تر و با نوسانات بیشتری همراه است. این نوسانات نشان می‌دهد که عامل زمان بیشتری برای کشف سیاست بهینه نیاز داشته است و در ابتدا با مشکلات زیادی مواجه بوده است.

در نمودار DQN، مشاهده می‌شود که عامل بسیار سریع‌تر یاد می‌گیرد. پس از چند اپیزود اولیه (حدود ۵۰ اپیزود)، پاداش‌ها به سرعت به حداکثر مقدار ممکن نزدیک می‌شوند و این پاداش‌ها بسیار پایدار می‌شوند.

این نشان می‌دهد که عامل DQN پس از حدود ۵۰ تا ۱۰۰ اپیزود به سیاست بهینه دست یافته است که به طور مداوم طلا را پیدا می‌کند بدون افتادن در گودال یا خورده شدن توسط .Wumpus

روند یادگیری DQN بسیار سریع‌تر و پایدارتر است. عامل DQN به سرعت سیاست بهینه را کشف کرده و پاداش‌های ثابت و بالایی را دریافت می‌کند. این عملکرد به دلیل استفاده از شبکه عصبی و تجربه بازپخش است که به عامل کمک می‌کند تا تجربیات متنوعی را پردازش کرده و به سرعت یاد بگیرد.

به صورت کلی DQN به طور قابل ملاحظه‌ای سریع‌تر از Q-learning سیاست بهینه را یاد می‌گیرد. در حالی که Q-learning حدود ۴۰۰ تا ۵۰۰ اپیزود برای یادگیری سیاست بهینه نیاز دارد، DQN تنها پس از ۵۰ تا ۱۰۰ اپیزود به سیاست بهینه دست می‌یابد.

عملکرد DQN پس از یادگیری سیاست بهینه بسیار پایدارتر است. پاداش‌های DQN به سرعت به حداکثر مقدار ممکن نزدیک می‌شوند و ثابت باقی می‌مانند، در حالی که Q-learning همچنان نوساناتی در پاداش‌ها دارد.

DQN از تجربه بازپخش و شبکه‌های عصبی برای یادگیری استفاده می‌کند که به آن اجازه می‌دهد تجربیات بیشتری را پردازش کرده و سیاست بهینه را سریع‌تر یاد بگیرد. در مقابل، Q-learning به دلیل استفاده از یک جدول Q و کاوش تصادفی بیشتر زمان می‌برد تا به سیاست بهینه دست یابد.

در نتیجه، DQN در مقایسه با Q-learning سیاست بهینه را سریع‌تر و پایدارتر یاد می‌گیرد و عملکرد بهتری دارد.

طبق قبل شبکه عصبی ما سه لایه است و دلیل انتخاب آن به دلایل زیر است:

این معماری نسبتاً ساده است و از دو لایه پنهان با تعداد متوسطی نورون استفاده می‌کند. این باعث می‌شود که شبکه به اندازه کافی قوی باشد تا ویژگی‌های پیچیده را یاد بگیرد، ولی در عین حال بیش از حد پیچیده و سنگین نباشد.

استفاده از توابع فعال‌سازی ReLU به شبکه اجازه می‌دهد تا غیرخطی بودن داده‌ها را مدل‌سازی کند .ReLU به دلیل سادگی و کارایی محاسباتی انتخاب شده است و مشکلات مانند ناپدید شدن گرادیان‌ها را کاهش می‌دهد.

معماری‌هایی با دو یا سه لایه پنهان در بسیاری از مسائل تقویتی موفق بوده‌اند. این معماری نیز بر اساس تجربیات موفق قبلی در مسائل مشابه انتخاب شده است.

تعداد نورون‌ها در هر لایه به تدریج کاهش می‌یابد (از ۶۴ به ۳۲)، که به تجمیع و استخراج اطلاعات مهم‌تر کمک می‌کند و پیچیدگی شبکه را کنترل می‌کند.