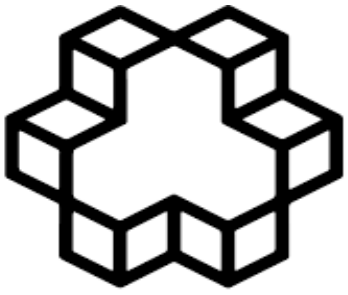


دانشکده مهندسی برق _گرایش کنترل

مینی پروژه شماره چهار

یادگیری ماشین

نگارش

فاطمه امیری

۴۰۲۰۲۴۲۴

لینک گوگل کولب

لینک گیت هاب

استاد مربوطه

جناب آقای دکتر علیاری

تیر ماه ۱۴۰۳


بسم الله الرحمن الرحيم

فهرست مطالب

سوال اول)

۱ پرسش یک: حل دنیای Wumpus

Wumpus World یک 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 Wumpus اجتناب 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 Wumpus اجتناب 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*۴ که 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 می‌کند (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 ۴x۴ با 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 : ۰.۱
  - ضریب تخفیف: ۰.۹
  - نرخ اکتشاف: از ۱.۰ شروع می شود و در طول زمان کوچک میشود.

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

در ابتدا و قبل از آن که به حل موارد خواسته شده و کد نویسی بپردازیم، به توضیحاتی راجع به صورت مسئله می پردازیم.

توضیح دنیای وومپوس (Wumpus World): یک سناریوی هوش مصنوعی

دنیای وومپوس به عنوان یک مدل آموزشی کلاسیک در حوزه هوش مصنوعی شناخته می‌شود که برای اولین بار در دهه ۱۹۷۰ توسط پیشگامان این رشته، جان مک‌کارتی و ماروین مینسکی، معرفی گردید. این مسئله از طریق برنامه‌های درسی و کتاب‌های استاندارد هوش مصنوعی برای آموزش مفاهیم پیچیده مورد استفاده قرار می‌گیرد و شامل یک محیط شبیه‌سازی شده است که در آن اکتشافگر باید طلا کشف کرده و با موفقیت از محیط خارج شود.

دنیای وومپوس در یک شبکه دوبعدی مربعی مانند چهار در چهار سازماندهی شده است، که هر خانه می‌تواند شامل عناصر مختلفی باشد که در ادامه بیان می شوند :

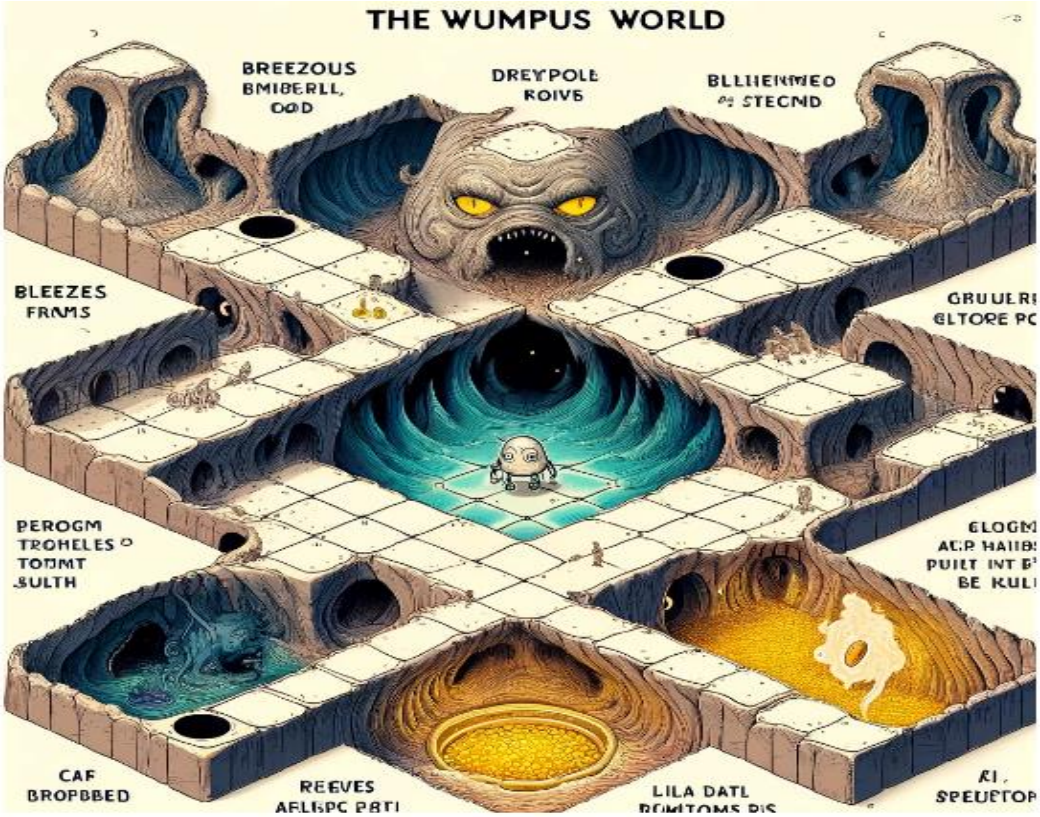
- اکتشاف گر (Explorer) : شخصیت اصلی بازی که در نقطه شروع (۰،۰) قرار دارد و هدفش کشف طلا و خروج موفق از محیط است.
- طلا (Gold) : در یکی از خانه‌ها به صورت تصادفی قرار داده شده و وظیفه اکتشاف گر است که آن را پیدا کند.
- وومپوس (Wumpus) : موجودی خطرناک که در خانه‌ای نهفته است و مواجهه با آن به معنای شکست اکتشافگر است.
- حفره‌ها (Pits) : موانعی که در خانه‌های مختلف قرار دارند و باعث شکست فوری اکتشافگر می‌شوند.
- خانه‌های خالی (Empty Cells) : خانه‌هایی که فاقد هرگونه خطر یا طلایی هستند.

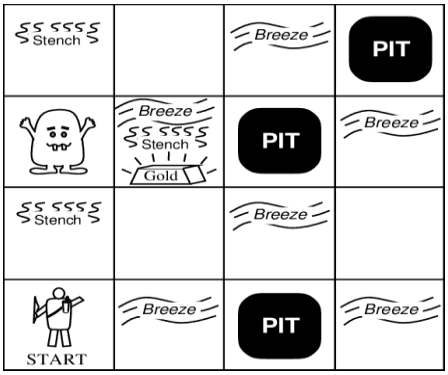
محیط دنیای وومپوس ثابت و قطعی است، به این معنی که تغییرات در محیط تنها به دنبال اقدامات اکتشاف گر رخ می‌دهد. اکتشاف گر توانایی‌هایی دارد که شامل حرکت به چهار جهت اصلی، تیراندازی برای کشتن وومپوس، برداشتن طلا، و خروج از محیط پس از جمع‌آوری طلا می‌شود.

سیستم پاداش در دنیای وومپوس به گونه‌ای طراحی شده که اکتشاف گر را به انجام عملیاتی که منجر به کشف طلا و خروج ایمن از محیط می‌شود تشویق کند. پاداش‌های مثبت برای کشف طلا و پاداش‌های منفی برای شکست در اختیار اکتشاف گر قرار داده می‌شوند. برای حل مسئله دنیای وومپوس، از مجموعه‌ای از رویکردهای پیشرفته هوش مصنوعی استفاده می‌شود. این روش‌ها شامل الگوریتم‌های جستجوی کلاسیک مانند جستجوی عمقی و سطحی، الگوریتم‌های یادگیری تقویتی مانند Q-Learning، DQN برای یادگیری سیاست بهینه بر اساس سیستم پاداش‌ها و روش‌های منطقی برای مدل‌سازی و استنتاج از وضعیت‌های مختلف هستند.

[1]https://dl.ebooksworld.ir/books/Artificial.Intelligence.A.Modern.Approach.4th.Edition.Peter.Norvig.%20Stuart.Russell.Pearson.9780134610993.EBooksWorld.ir.pdf

دنیای وومپوس بیشتر به عنوان یک ابزار آموزشی در زمینه هوش مصنوعی به کار می‌رود و به ما اجازه می‌دهد تا با اصول اساسی AI و الگوریتم‌های پیچیده در محیطی کنترل‌شده تمرین کنیم. این سناریو همچنین به عنوان یک معیار برای ارزیابی و آزمون الگوریتم‌های نوین استفاده می‌شود. در زیر تصاویری از Wumpus World اورده شده است.

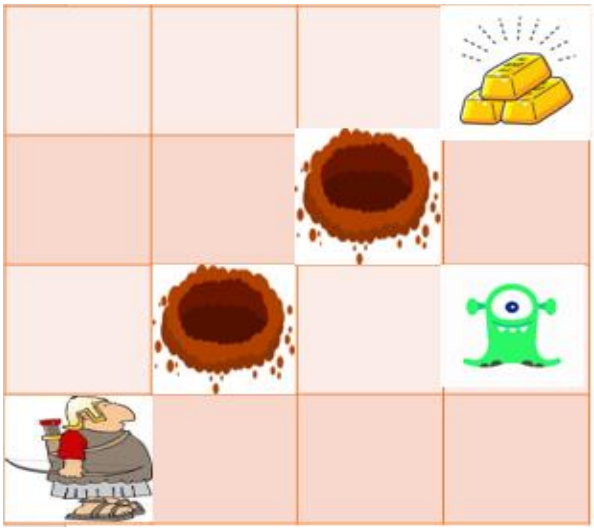




همان طور که بیان شد، مسئله Wumpus World یک نمونه کلاسیک و تأثیرگذار در میان چالش‌های هوش مصنوعی به شمار می‌رود. این مسئله به طور مفصل نشان می‌دهد که چطور می‌توان با استفاده از تنوع الگوریتم‌ها و تکنیک‌های مختلف، مشکلات پیچیده را حل کرد و کارآمدی عملکرد عامل‌ها را در محیط‌های دینامیک و

چالشی ارتقا داد. ارزیابی کدهای ارائه شده و توضیحات آن‌ها به درک عمیق‌تری از کاربرد الگوریتم‌ها در دنیای Wumpus کمک می‌کند و امکان تحلیل و مقایسه نتایج حاصل از این پیاده‌سازی‌ها را فراهم می‌آورد.

در شروع کار، با توجه به توضیح مسئله، به ساخت محیطی برای این چالش و تعریف قابلیت‌های لازم برای حرکت عامل در آن می‌پردازیم. محیط به صورت یک جدول چهار در چهار تعریف شده است که صفحه بازی ما را می‌سازد و دارای نقطه شروعی در مختصات (۰،۰) است. در این صفحه، علاوه بر عامل، اشیاء دیگری مانند طلا نیز وجود دارند که عامل در تلاش برای یافتن آن است و مکان آن توسط ما در نقطه (۳،۳) مشخص شده است.



برای رسیدن به طلا، عامل باید حداقل شش حرکت انجام دهد. در ادامه، عامل برای هر حرکت که منجر به مرگ یا یافتن طلا نشود، امتیاز منفی (-۱) دریافت می‌کند و در بهترین حالت، با کشتن Wumpus و دریافت جایزه +۵۰ و همچنین یافتن طلا و دریافت جایزه +۱۰۰، امکان رسیدن به مجموع امتیازات ۱۴۵ وجود دارد. تغییرات در محیط و مکان اشیاء می‌تواند بر این حداکثر امتیاز تأثیر بگذارد. همچنین، تنظیم میزان Exploration الگوریتم به گونه‌ای که بتواند پس از مدتی به Exploitation بیشتری برسد، بر نتایج تأثیرگذار است و ممکن است عامل تنها به جای جستجوی حداکثر امتیاز، به یافتن طلا اکتفا کند که در این صورت حداکثر امتیاز قابل کسب +۹۵ خواهد بود. این دلیلی است که الگوریتم Q-learning به امتیاز +۹۵ همگرا می‌شود. در مراحل بعدی، با تغییر نحوه کاهش نرخ Exploration، این جنبه بیشتر مورد بررسی قرار می‌گیرد و انتظار می‌رود که با تنظیم مناسب، عامل بتواند به بیشترین امتیاز ممکن دست یابد، همانند آنچه در الگوریتم DQN رخ می‌دهد.

در ادامه، به شرح نحوه تعریف محیط و قابلیت‌های عامل می‌پردازیم.

معرفی محیط یا Environment

در این بخش، محیط بازی Wumpus World را تعریف می‌کنیم و مکان‌های مختلف اشیاء را مشخص می‌کنیم. در طول توسعه، بعضی از مکان‌های اشیاء پس از هر دوره عملیاتی تغییر می‌کردند. نتایج این تغییرات در بخش جداگانه‌ای گزارش خواهد شد (در ادامه بررسی می شوند) . به دلیل پیچیدگی زیاد مسئله و مشکلات در اجرای کد الگوریتم DQN، تصمیم گرفتیم که مکان‌های اشیاء در طول آموزش هر دو الگوریتم ثابت باقی بمانند تا عامل بتواند بهتر بازی را یاد بگیرد. در نسخه‌های مختلف پیاده‌سازی این بازی که در GitHub دیده بودم[2] مکان‌های اشیاء گاهی متغیر بودند و با هر اجرا تغییر می‌کردند. به عنوان مثال، در اطراف چاه‌ها و وامپوس، اشیاء قرار داشت که عامل با ورود به آنها کشته نمی‌شد، اما در نزدیکی اشیاء خطرناک بود. در برخی حالات، صدای آب در کنار چاه‌ها شنیده می‌شد و بوی وامپوس با ورود به اطراف احساس می‌شد که عامل به خطر نزدیک شده است. این وضعیت در حالتی رخ می‌داد که وامپوس با هر حرکت عامل نیز یک حرکت در جهات مجاز انجام می‌داد. حل این مسئله با الگوریتم‌های یادگیری تقویتی کاملاً ممکن است، اما برای اجرای آن نیازمند دستگاه‌های قدرتمند و زمان زیادی هستیم، زیرا این مسائل به عنوان یکی از پیچیده‌ترین و با هزینه محاسباتی زیاد شناخته می‌شوند. *به دلیل محدودیت‌های زمانی و پیچیدگی مسئله، این امکانات را در نظر نگرفته و یک حالت ساده‌تر برای بازی انتخاب کرده‌ایم. در این حالت، مکان‌های اشیاء در طول حل مسئله ثابت هستند و حرکت نمی‌کنند. با هر اجرا، یک دوره جدید شروع می‌شود، که در آن عامل قابلیت شلیک دارد و وامپوس به حالت زنده برگردانده می‌شود. همچنین، عامل در نقطه شروع قرار می‌گیرد.* پس کلاس زیر را تعریف می کنیم :

class GridEnvironment:

```
class GridEnvironment:
    def __init__(self):
        self.grid_size = 4
        self.grid = np.zeros((self.grid_size, self.grid_size))
        self.agent_position = [0, 0]  # Starting position of the agent
        self.gold_position = [3, 3]  # Position of the gold
        self.pits = [[1, 1], [2, 2]]  # Positions of the pits
        self.wumpus_position = [1, 3]  # Position of the Wumpus
        self.wumpus_alive = True  # Status of the Wumpus (alive or dead)
        self.arrow_available = True  # Status of the arrow (available or not)

        # Setting up the grid
        self.grid[self.gold_position[0], self.gold_position[1]] = 1  # Gold
        for pit in self.pits:
            self.grid[pit[0], pit[1]] = -1  # Pits
        self.grid[self.wumpus_position[0], self.wumpus_position[1]] = -2  # Wumpus

    def reset(self):
        # Reset the positions and statuses
        self.agent_position = [0, 0]
        self.wumpus_position = [1, 3]
        self.wumpus_alive = True
        self.arrow_available = True
        return tuple(self.agent_position)
```

[2] https://github.com/nowke/wumpus-rl - https://github.com/R-Moraes/Wumpus_World-RL - https://github.com/piotrSobie/WumpusWorld-RL - https://github.com/erikphillips/wumpus_world

کد فوق یک کلاس به نام GridEnvironment تعریف می کند که یک محیط شبیه‌سازی شده برای بازی Wumpus World را ایجاد می‌کند. در این محیط، ابتدا ابعاد شبکه (grid)به اندازه ۴x۴ و تمامی مکان‌های آن به صورت آرایه‌ای numpy از اعداد صفر (خانه‌های خالی) مقداردهی می‌شود. سپس، موقعیت‌های اولیه عناصر مختلف از جمله عامل (در خانه [۰, ۰])، طلا (در خانه [۳, ۳])، چاه‌ها (در خانه‌های [۱, ۱] و [۲, ۲])، و وامپوس (در خانه [۱, ۳]) تعیین می‌شوند. وضعیت اولیه وامپوس (زنده بودن یا مرده بودن) و موجودیت تیر نیز به True قرار داده می‌شود. در نهایت، با مقداردهی مجدد به grid بر اساس موقعیت‌های تعیین شده، محیط آماده استفاده و شبیه‌سازی بازی Wumpus World می‌شود. تابع reset نیز برای بازگرداندن محیط به حالت اولیه و بازگرداندن موقعیت فعلی عامل به نقطه شروع استفاده می‌شود، همچنین وضعیت وامپوس و موجودیت تیر نیز به حالت اولیه بازمی‌گردانده می‌شود.

معرفی مجموعه عملیات یا Action Set و مجموعه پاداش یا Reward Set :

از کد های زیر استفاده می کنیم که در واقع دو تابع است که به محیط شبیه‌سازی Wumpus World کمک می‌کنند. ابتدا تابع step به عنوان تابع اصلی برای انجام یک عمل در محیط و ارزیابی اثرات آن عمل بر محیط استفاده می‌شود. در این تابع، ابتدا جریمه حرکت به اندازه -۱ تعیین می‌شود. سپس بر اساس عمل انتخاب شده (مانند حرکت به چپ، راست، بالا، پایین یا شلیک کردن تیر به یکی از جهات)، موقعیت عامل تغییر می‌کند و متغیرهای محیطی مانند وضعیت وامپوس و موجودیت تیر نیز به‌روزرسانی می‌شوند. در صورتی که عامل به مقصد (طلا) برسد، جایزه ۱۰۰ امتیاز داده می‌شود و بازی به پایان می‌رسد. اگر عامل در چاه بیفتد، مجازات -۱۰۰۰ امتیاز اختصاص داده می‌شود و بازی به پایان می‌رسد. اگر موفق به کشتن Wumpus شود به میزان ۵۰ امتیاز مثبت دریافت میکند. همچنین، اگر عامل با وامپوس زنده برخورد کند، مجازات -۱۰۰۰ امتیاز اعمال می‌شود و بازی به پایان می‌رسد. (قسمت های امتیازی هم مدنظر قرار گرفت)

سپس تابع get_possible_actions نیز لیستی از عملیات‌های ممکن را بازمی‌گرداند که شامل حرکت به چپ، راست، بالا، پایین و شلیک کردن تیر به هر یک از جهات است :

```
['up', 'down', 'left', 'right', 'shoot_up', 'shoot_down', 'shoot_left', 'shoot_right']
```

این توابع به طور کلی برای مدیریت و بررسی عملکرد عامل در محیط Wumpus World طراحی شده‌اند و می‌توانند به عنوان پایه‌ای برای پیاده‌سازی الگوریتم‌های یادگیری تقویتی و ارزیابی عملکرد عامل در محیط مورد استفاده قرار گیرند. در زیر کد پایتون این دو تابع آمده است :

```
def step(self, action):
    reward = -1  # Movement penalty
    done = False

    # Move the agent based on the action
    if action == 'up':
        self.agent_position[0] = max(0, self.agent_position[0] - 1)
    elif action == 'down':
        self.agent_position[0] = min(self.grid_size - 1, self.agent_position[0] + 1)
    elif action == 'left':
        self.agent_position[1] = max(0, self.agent_position[1] - 1)
    elif action == 'right':
        self.agent_position[1] = min(self.grid_size - 1, self.agent_position[1] + 1)
    elif action == 'shoot_up' and self.arrow_available:
        # Shoot the arrow upwards
        if self.wumpus_position[0] < self.agent_position[0]:
            reward = 50
            self.wumpus_alive = False
        self.arrow_available = False
    elif action == 'shoot_down' and self.arrow_available:
        # Shoot the arrow downwards
        if self.wumpus_position[0] > self.agent_position[0]:
            reward = 50
            self.wumpus_alive = False
        self.arrow_available = False
    elif action == 'shoot_left' and self.arrow_available:
        # Shoot the arrow to the left
        if self.wumpus_position[1] < self.agent_position[1]:
            reward = 50
            self.wumpus_alive = False
        self.arrow_available = False
    elif action == 'shoot_right' and self.arrow_available:
        # Shoot the arrow to the right
        if self.wumpus_position[1] > self.agent_position[1]:
            reward = 50
            self.wumpus_alive = False
        self.arrow_available = False
```

```
    # Check if the game has ended
    if self.agent_position == self.gold_position:
        reward = 100  # Reward for finding the gold
        done = True
    elif self.agent_position in self.pits:
        reward = -1000  # Penalty for falling into a pit
        done = True
    elif self.agent_position == self.wumpus_position and self.wumpus_alive:
        reward = -1000  # Penalty for encountering a live Wumpus
        done = True

    return tuple(self.agent_position), reward, done
```

```
def get_possible_actions(self):
    # Get a list of possible actions
    return ['up', 'down', 'left', 'right', 'shoot_up', 'shoot_down', 'shoot_left', 'shoot_right']
```

اکنون باید به حل این مسئله بپردازیم. برای این منظور، از دو روش Q-learning و DQN استفاده خواهیم کرد. ابتدا هر یک از این روش‌ها را به طور جداگانه پیاده‌سازی کرده و همانند قبل، توضیحات مربوطه را ارائه خواهیم داد. در پایان نیز، نتایج به‌دست‌آمده از این دو الگوریتم را با یکدیگر مقایسه می‌کنیم.

در این مینی پروژه، تمام موارد امتیازی مطرح شده در سوال پیاده‌سازی شده‌اند. به جز این که Wumpus به‌عنوان یک عنصر ثابت در نظر گرفته شده است. بنابراین، در راه‌حل ما، Wumpus ثابت است، در حالی که سایر جنبه‌های امتیازی، مانند توانایی شلیک توسط عامل و امتیاز مرتبط با کشتن Wumpus، لحاظ شده است. همچنین، الگوریتم DQN به‌طور کامل بررسی و تمامی سوالات مربوط به آن به دقت پاسخ داده شده است.

آ)

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.

الگوریتم Q-learning

روش Q-learning یکی از تکنیک‌های یادگیری تقویتی است که به‌طور گسترده برای حل مسائل پیچیده در حوزه هوش مصنوعی استفاده می‌شود. این روش به‌ویژه در مسائلی کاربرد دارد که در آن‌ها یک عامل باید از طریق تعامل با محیط، استراتژی بهینه‌ای برای رسیدن به هدف بیابد . Q-learning یک روش یادگیری بدون مدل است که به عامل امکان می‌دهد بدون داشتن اطلاعات قبلی از محیط، یک سیاست برای رسیدن به هدف یاد بگیرد. این روش بر اساس یادگیری مقادیر عمل-وضعیت (state-action values)یا Q-values عمل می‌کند. Q-value نشان‌دهنده امتیاز پیش‌بینی‌شده‌ای است که عامل می‌تواند با انجام یک عمل خاص در یک وضعیت خاص انتظار داشته باشد.

هدف اصلی در Q-learning، به‌روزرسانی مقدار Q برای هر جفت وضعیت-عمل است تا Q-value به مقدار واقعی نزدیک شود. معادله به‌روزرسانی Q-value به این صورت است:

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

که در این معادله، Q(s,a) نمایانگر مقدار Q برای وضعیت sو عمل aاست. همچنین، alpha نرخ یادگیری است که مشخص می‌کند مقادیر Q تا چه اندازه به‌روزرسانی شوند و معمولاً بین ۰ و ۱ تنظیم می‌شود. rپاداشی

است که پس از انجام عمل aدر وضعیت s دریافت می‌شود. علاوه بر این gamma عامل تخفیف است که مقدار پاداش‌های آینده را مشخص می‌کند و همچنین معمولاً بین ۰ و ۱ قرار دارد. در نهایت، $max_{\acute{a}}Q(\acute{s};\acute{a})$ به بیشینه مقدار Q برای وضعیت بعدی 's و تمامی اعمال ممکن 'a اشاره دارد.

در مسئله Wumpus World، عامل باید یاد بگیرد چگونه در محیط حرکت کند، طلا را پیدا کند و از خطرات اجتناب کند. برای این منظور، از روش Q-learning استفاده می‌شود تا عامل بتواند سیاست بهینه‌ای برای رسیدن به هدف بیاموزد.

تنظیم پارامترها: ابتدا نرخ یادگیری alpha، عامل تخفیف gamma و نرخ کاوش یا exploration rateکه با استفاده از $greedy-\varepsilon$ کنترل می‌شود، تنظیم می‌شوند.

مقداردهی اولیه :Q-tableیک جدول Q با ابعاد مناسب (تعداد وضعیت‌ها × تعداد اعمال) ایجاد و با مقادیر اولیه (معمولاً صفر) مقداردهی می‌شود.

چرخه آموزش: عامل در محیط به مدت چند اپیزود آموزش داده می‌شود. در هر اپیزود، عامل از وضعیت اولیه شروع به حرکت می‌کند و تا زمانی که به وضعیت پایان برسد یا کشته شود، ادامه می‌دهد.

در هر قدم، عامل با استفاده از سیاست $greedy-\varepsilon$ یک عمل انتخاب می‌کند. این سیاست به عامل اجازه می‌دهد تا با احتمالε عمل تصادفی انجام دهد (برای کاوش محیط) و با احتمال $1-\varepsilon$ عمل بهینه را انتخاب کند (بر اساس مقادیر فعلی Q).

پس از انجام عمل، عامل پاداش مربوطه را دریافت می‌کند و به وضعیت بعدی منتقل می‌شود. مقدار Q برای جفت وضعیت-عمل قبلی با استفاده از معادله به‌روزرسانی Q-value به‌روزرسانی می‌شود.

به‌روزرسانی نرخ کاوش: پس از هر اپیزود، نرخ کاوش ε با استفاده از یک ضریب کاهش (exploration decay) کاهش می‌یابد تا عامل به مرور زمان بیشتر به سمت بهره‌برداری (exploitation)از سیاست آموخته شده تمایل پیدا کند.

در مسئله Wumpus World، عامل در محیطی ۴×۴ قرار دارد که شامل عناصر مختلفی مانند طلا، وومپوس و حفره‌ها است. عامل با انجام اعمال مختلف مانند حرکت به جهات مختلف و شلیک تیر، یاد می‌گیرد چگونه به هدف برسد. پس از چندین اپیزود، عامل سیاست بهینه‌ای برای رسیدن به هدف و اجتناب از خطرات می‌آموزد.

مزایا و معایب روش Q-learning : روش Q-learning به دلیل سادگی و قابلیت اعمال در مسائل مختلف، بسیار محبوب است. این روش نیازی به مدل محیط ندارد و می‌تواند با استفاده از تعاملات مستقیم با محیط،

سیاست بهینه را یاد بگیرد. با این حال، یکی از معایب اصلی این روش این است که نیاز به حافظه زیادی برای ذخیره‌سازی جدول Q دارد و در محیط‌های بزرگتر ممکن است عملکرد آن کاهش یابد.

در مسئله Wumpus World، روش Q-learning به عامل اجازه می‌دهد تا با کاوش و بهره‌برداری از محیط، سیاست بهینه‌ای برای رسیدن به هدف پیدا کند. این روش به خوبی نشان می‌دهد که چگونه می‌توان از یادگیری تقویتی برای حل مسائل پیچیده و بهینه‌سازی رفتار عامل در محیط‌های پویا استفاده کرد.

اکنون به بررسی کد نویسی این روش می پردازیم :

برای کد نویسی کلاس class QLearningAgent را تعریف می کنیم که شامل چندین متد و اتربیوت است و در ادامه به بررسی دقیق تر این کلاس و تابع ها می پردازیم.

کلاس QLearningAgent یک الگوریتم یادگیری تقویتی است که برای آموزش عامل در محیط‌های مشخص طراحی شده است. در تابع سازنده __init، ورودی‌هایی شامل محیط env، نرخ یادگیری learning_rate، عامل تخفیف discount_factor، نرخ اکتشاف اولیه exploration_rate، و نرخ کاهش اکتشاف exploration_decay دریافت می‌کند. این کلاس دارای یک جدول Q (q_table) است که به عنوان یک دیکشنری خالی آغاز می‌شود و برای ذخیره مقادیر Q برای جفت‌های حالت-عمل استفاده می‌شود. همچنین، متد get_q_value برای بازیابی مقدار Q مربوط به یک جفت حالت و عمل خاص طراحی شده است و در صورتی که این جفت در جدول موجود نباشد، مقدار پیش‌فرض ۰,۰ را باز می‌گرداند. نکته‌ای که باید به آن توجه کرد این است که با انتخاب این هایپرپارامترها که در زیر می بینیم، توانستیم به نتایج نسبتاً خوبی دست پیدا کنیم. این انتخاب‌ها نتیجه تلاش‌ها و آزمایش‌های مکرر ما بوده و اگر برخی از این مقادیر با هایپرپارامترهای پیشنهادی متفاوت است، به این دلیل است که با این تنظیمات عملکرد بهتری داشته‌ایم. ما می‌توانیم با بررسی وضعیت‌های مختلف در یک حلقه، بهترین پارامترها را شناسایی کنیم یا از روش‌های خاص این حوزه استفاده کنیم، اما در حال حاضر به یک نتیجه منطقی و رضایت‌بخش قناعت کرده‌ایم. این امر به ما این امکان را می‌دهد که دو الگوریتم را بهتر مقایسه کنیم و اعتبار این مقایسه افزایش یابد.

```python
class QLearningAgent:
    def __init__(self, env, learning_rate=0.1, discount_factor=0.9, exploration_rate=1.0, exploration_decay=0.995):
        self.env = env
        self.q_table = {}  # Initialize the Q-table as an empty dictionary
        self.learning_rate = learning_rate  # Learning rate for Q-learning updates
        self.discount_factor = discount_factor  # Discount factor for future rewards
        self.exploration_rate = exploration_rate  # Initial exploration rate for epsilon-greedy policy
        self.exploration_decay = exploration_decay  # Decay rate for exploration rate

    def get_q_value(self, state, action):
        # Retrieve the Q-value for a given state-action pair from the Q-table, default to 0.0 if not found
        return self.q_table.get((state, action), 0.0)
```

همچنین متد update_q_value در کلاس QLearningAgent برای به‌روزرسانی مقدار Q یک جفت حالت-عمل استفاده می‌شود. ابتدا بهترین عمل ممکن برای حالت بعدی بر اساس مقادیر Q فعلی محاسبه می‌شود. سپس هدف Q با استفاده از پاداش و مقدار Q تخفیف‌یافته عمل بهترین محاسبه می‌شود. خطای تفاضل زمانی (temporal difference error) نیز محاسبه و مقدار Q برای جفت حالت-عمل به‌روزرسانی می‌شود. متد choose_action از سیاست اکتشافی $greedy - \varepsilon$ استفاده می‌کند، به این صورت که با احتمال مشخصی عمل تصادفی انتخاب می‌کند (اکتشاف) و در غیر این صورت بهترین عمل بر اساس مقادیر Q را انتخاب می‌کند (استفاده از اطلاعات قبلی). این دو متد به عامل کمک می‌کنند تا به‌طور مؤثری در محیط یاد بگیرد و تصمیم‌گیری کند. در واقع در این بخش، الگوریتم Q-learning پیاده‌سازی شده و از مقدار صفر شروع به به‌روزرسانی Q-table می‌شود. فرمول به‌روزرسانی Q در بالاتر بیان شد. به این ترتیب، در هر بار به‌روزرسانی، مقدار قبلی Q با ضریبی از پاداش حرکت و بهترین مقدار Q از حالت بعدی جمع می‌شود. این فرآیند در کد نیز پیاده‌سازی شده و در نهایت Q-table جدید برای اپیزود بعدی ذخیره می‌شود.

```python
def update_q_value(self, state, action, reward, next_state):
    # Get the best next action based on the current Q-values
    best_next_action = max(self.env.get_possible_actions(), key=lambda a: self.get_q_value(next_state, a))

    # Calculate the target Q-value using the reward and the discounted Q-value of the best next action
    td_target = reward + self.discount_factor * self.get_q_value(next_state, best_next_action)

    # Calculate the temporal difference error
    td_error = td_target - self.get_q_value(state, action)

    # Update the Q-value for the state-action pair using the learning rate
    new_q_value = self.get_q_value(state, action) + self.learning_rate * td_error

    # Store the updated Q-value in the Q-table
    self.q_table[(state, action)] = new_q_value

def choose_action(self, state):
    # Choose an action using epsilon-greedy policy
    if random.uniform(0, 1) < self.exploration_rate:
        return random.choice(self.env.get_possible_actions())  # Explore: choose a random action
    else:
        return max(self.env.get_possible_actions(), key=lambda a: self.get_q_value(state, a))  # Exploit: choose the best action
```

همچنین متد trainدر کلاس QLearningAgent برای آموزش عامل از طریق تعدادی اپیزود طراحی شده است. در ابتدا، یک لیست برای ذخیره جوایز کل هر اپیزود ایجاد می‌شود. در هر اپیزود، محیط بازنشانی شده و عامل در حالتی ابتدایی شروع می‌کند. حلقه‌ای اجرا می‌شود تا زمانی که اپیزود به پایان برسد. در هر گام، عامل یک عمل را با استفاده از متد choose_action انتخاب کرده و سپس با اجرای این عمل، وضعیت بعدی، پاداش و وضعیت پایان (done)از محیط دریافت می‌شود. پس از آن، مقدار Q برای جفت حالت-عمل به‌روزرسانی می‌شود و وضعیت فعلی به وضعیت بعدی تغییر می‌کند. مجموع پاداش‌ها در هر اپیزود جمع‌آوری شده و پس از پایان اپیزود، نرخ اکتشاف کاهش می‌یابد. در نهایت، لیست جوایز کل به عنوان خروجی برگردانده می‌شود.

```
def train(self, episodes):
    total_rewards = []
    for episode in range(episodes):
        state = self.env.reset()
        total_reward = 0
        done = False
        while not done:
            action = self.choose_action(state)
            next_state, reward, done = self.env.step(action)
            self.update_q_value(state, action, reward, next_state)
            state = next_state
            total_reward += reward
        total_rewards.append(total_reward)
        self.exploration_rate *= self.exploration_decay  # Decay the exploration rate
    return total_rewards
```

در انتهای کد، تابعی برای نمایش نتایج الگوریتم با استفاده از سه نمودار تعریف شده است.

تابع plot_rewards برای ترسیم و نمایش نتایج الگوریتم Q-learning طراحی شده است. این تابع سه نمودار مختلف را به تصویر می‌کشد که شامل مجموع پاداش‌ها، پاداش‌های تجمعی و میانگین پاداش‌ها در طول اپیزودها است. این نمودارها به ما کمک می‌کنند تا روند یادگیری و عملکرد عامل را در طول زمان مشاهده کنیم.

```
def plot_rewards(total_rewards, cumulative_rewards, mean_rewards, title, filename):
    fig, axs = plt.subplots(3, 1, figsize=(12, 9))
```

نمودار اول، مجموع پاداش‌ها (Total Reward)را در هر اپیزود نمایش می‌دهد. این نمودار نشان می‌دهد که عامل در هر اپیزود چه مقدار پاداش کسب کرده است و می‌تواند به ما اطلاعاتی درباره‌ی بهبود عملکرد یا عدم پیشرفت عامل ارائه دهد. همچنین، با افزودن علامت به آخرین مقدار، می‌توانیم آخرین نتیجه را به راحتی شناسایی کنیم.

```
# Plot total rewards per episode
axs[0].plot(total_rewards, label='Total Reward', color='blue', linestyle='-', linewidth=0.8)
axs[0].set_xlabel('Episode')
axs[0].set_ylabel('Total Reward')
axs[0].set_title(f'Total Reward per Episode ({title})')
axs[0].legend()
axs[0].grid(True)
axs[0].annotate(f'Last value: {total_rewards[-1]}', xy=(len(total_rewards)-1, total_rewards[-1]),
                xytext=(len(total_rewards)-1, total_rewards[-1]), textcoords='data',
                arrowprops=dict(arrowstyle='->', color='blue'))
```

نمودار دوم، پاداش‌های تجمعی (Cumulative Reward)را نمایش می‌دهد که مجموع پاداش‌ها را در طول اپیزودها جمع‌آوری می‌کند. این نمودار به وضوح روند کلی پیشرفت عامل را در طول زمان نشان می‌دهد و می‌تواند برای شناسایی روندهای بلندمدت و نوسانات در عملکرد مورد استفاده قرار گیرد.

```
# Plot cumulative rewards
axs[1].plot(cumulative_rewards, label='Cumulative Reward', color='green', linestyle='-', linewidth=0.8)
axs[1].set_xlabel('Episode')
axs[1].set_ylabel('Cumulative Reward')
axs[1].set_title(f'Cumulative Reward per Episode ({title})')
axs[1].legend()
axs[1].grid(True)
axs[1].annotate(f'Last value: {cumulative_rewards[-1]}', xy=(len(cumulative_rewards)-1, cumulative_rewards[-1]),
                xytext=(len(cumulative_rewards)-1, cumulative_rewards[-1]), textcoords='data',
                arrowprops=dict(arrowstyle='->', color='green'))
```

نمودار سوم، میانگین پاداش‌ها (Mean Reward)را در طول اپیزودها ترسیم می‌کند. این نمودار به ما این امکان را می‌دهد که با یک دید کلی‌تر، تغییرات در عملکرد عامل را مشاهده کنیم و نقاط قوت و ضعف را شناسایی کنیم. به علاوه، نمایش آخرین مقدار به ما کمک می‌کند تا ارزیابی دقیقی از وضعیت فعلی عامل داشته باشیم.

```
# Plot mean rewards
axs[2].plot(mean_rewards, label='Mean Reward', color='red', linestyle='-', linewidth=0.8)
axs[2].set_xlabel('Episode')
axs[2].set_ylabel('Mean Reward')
axs[2].set_title(f'Mean Reward per Episode ({title})')
axs[2].legend()
axs[2].grid(True)
axs[2].annotate(f'Last value: {mean_rewards[-1]}', xy=(len(mean_rewards)-1, mean_rewards[-1]),
                xytext=(len(mean_rewards)-1, mean_rewards[-1]), textcoords='data',
                arrowprops=dict(arrowstyle='->', color='red'))
```

در نهایت، تمام این نمودارها در یک چیدمان منظم نمایش داده شده و با استفاده از تابع plt.savefig می‌توان آنها را ذخیره کرد.

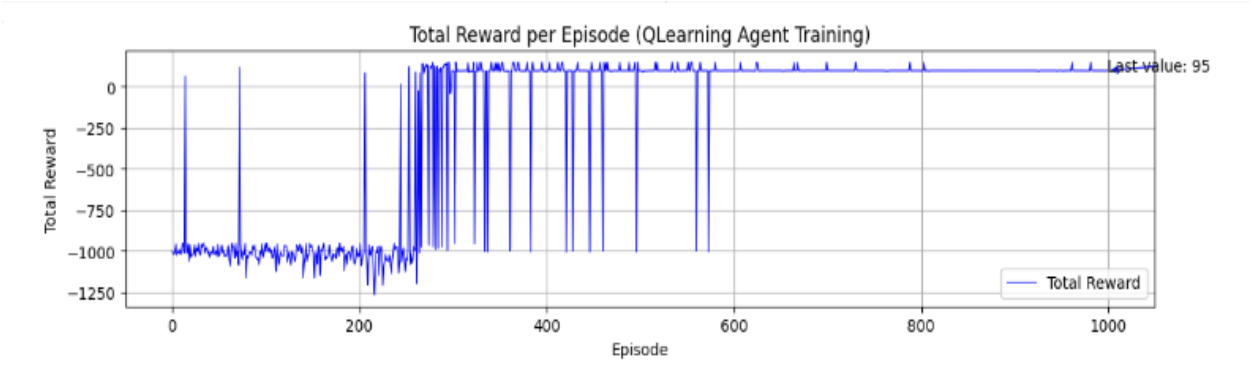
پس در واقع یکی از نمودارها مجموع امتیازهای جمع‌شده در هر اپیزود را نمایش می‌دهد و نشان می‌دهد که آیا عامل در اپیزودهای پایانی توانسته مسیر بهینه را شناسایی کند یا خیر، همچنین تأثیر کاهش نرخ اکتشاف را نیز

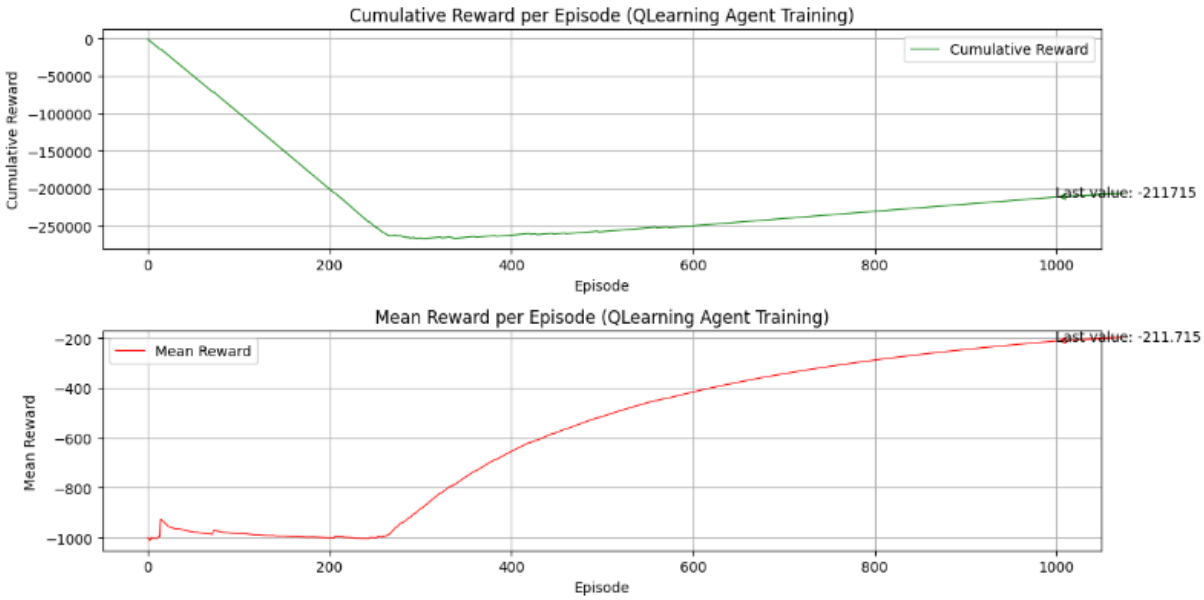
مشخص می‌کند. با افزایش تعداد اپیزودها، عامل به یادگیری مسیرهای بهینه نزدیک می‌شود، در حالی که در الگوریتم DQN این افزایش ممکن است بار محاسباتی سنگینی ایجاد کند. نمودار دوم امتیازهای جمع‌شده از ابتدای برنامه تا اپیزود جاری را نشان می‌دهد و نقاطی که امتیازها به طور قابل توجهی مثبت می‌شوند، حائز اهمیت هستند، زیرا نشان‌دهنده‌ی زمانی است که عامل به جای اکتشاف، بهترین مسیرها را انتخاب می‌کند. در نهایت، نمودار سوم میانگین امتیازهای جمع‌شده را ترسیم می‌کند و به ما می‌گوید که این مقدار به کدام سطح همگرا می‌شود، که نمایانگر بیشترین امتیازی است که عامل به عنوان سیاست بهینه یادگرفته و در آینده تکرار خواهد کرد.

در اخر هم یک محیط GridEnvironment و یک عامل Q-learning (QLearningAgent) ایجاد می‌شود. عامل با نرخ کاهش اکتشاف سریع‌تر (۰,۹۹۵) تنظیم شده است تا به سرعت به سمت سیاست‌های بهینه همگرا شود. سپس، عامل برای آموزش به مدت مثلا ۱۰۰۰ اپیزود فراخوانی می‌شود(که می توان این مقدار را تغییر داد). در این فرآیند، عامل تجربیات خود را جمع‌آوری کرده و مقادیر Q را به‌روزرسانی می‌کند تا در نهایت عملکرد بهتری در محیط داشته باشد. نتایج کل پاداش‌ها از هر اپیزود در متغیر total_rewards زخیره می‌شود.

```
# Initialize environment and agent
env = GridEnvironment()
agent = QLearningAgent(env, exploration_decay=0.995)  # Faster decay rate

# Train agent
episodes = 1000
total_rewards = agent.train(episodes)
```

اکنون الگوریتم را به ازای ۱۰۰۰ اپیزود یادگیری ران می کنیم و داریم :





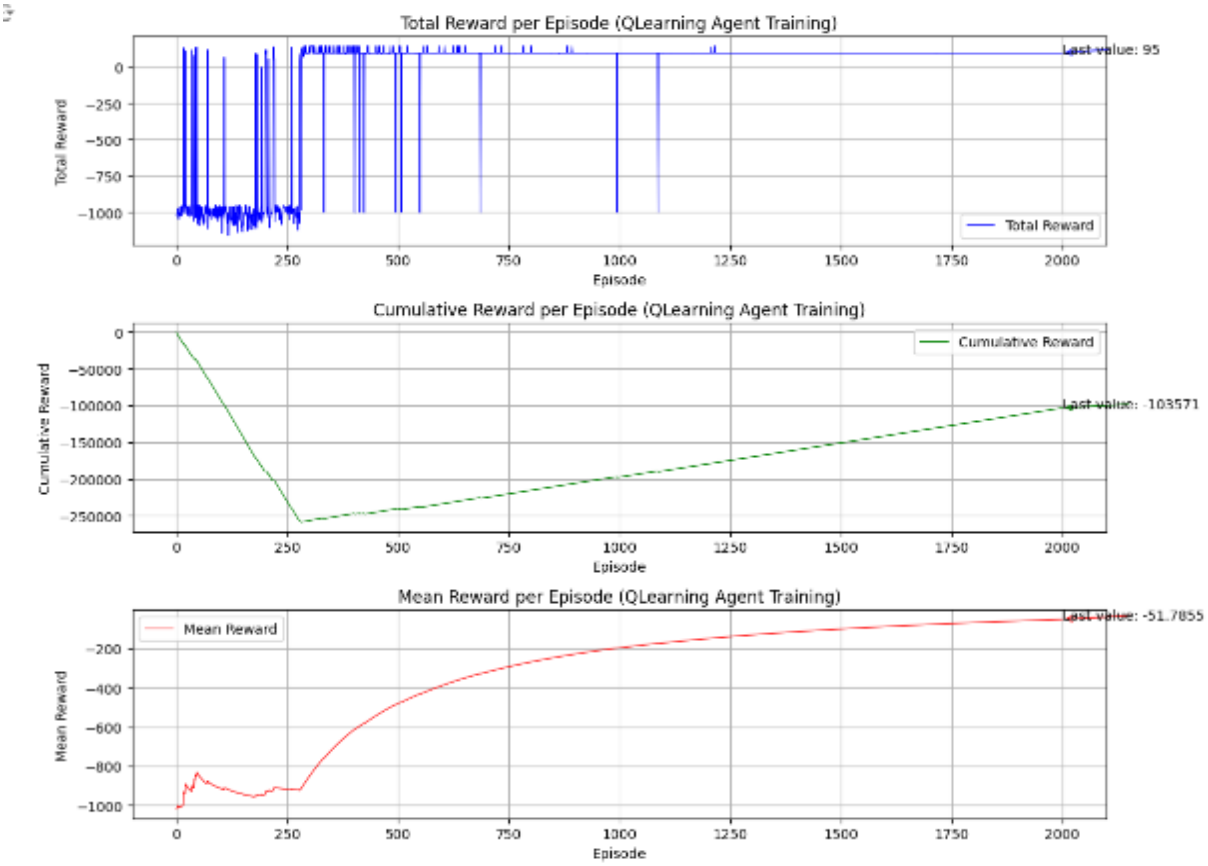
و خلاصه اطلاعات این نمودار ها به صورت زیر است :

```
=== Training Summary ===
Number of Deaths :        280
Total Rewards    :    -211715
Highest Reward   :        145
Lowest Reward    :      -1265
Accuracy         :     72.00%
========================
```

در این بررسی، از ۱۰۰۰ اپیزود، عامل ۲۸۰ بار کشته شده و بالاترین امتیاز کسب‌شده ۱۴۵ بوده است. همان‌طور که اشاره کردیم، برای کشتن Wumpus و رسیدن به طلا، ۵ حرکت لازم است که مجموع منطقی امتیازها به ۱۴۵ می‌رسد، بنابراین این نتیجه صحیح است. کمترین امتیاز به میزان -۱۲۶۵ ثبت شده که نشان دهنده ۲۶۵ حرکت در محیط بدون کشته شدن است. با این حال، معیار دقت به تنهایی نمی‌تواند عملکرد مدل را به خوبی نشان دهد، زیرا پس از مدتی، عامل یاد می‌گیرد و بهتر است عملکرد از آن نقطه به بعد ارزیابی شود. در حدود ۷۲ درصد اپیزودها با موفقیت و بدون کشته شدن به پایان رسیده‌اند. نکته مهم این است که روند صعودی نمودار میانگین امتیازها (نمودار سوم) همچنان ادامه دارد و هنوز به همگرایی نرسیده است، بنابراین نمایش ریوارد منفی به عنوان آخرین مقدار چندان معتبر نیست. به همین دلیل، قصد داریم الگوریتم را برای تعداد بیشتری اپیزود اجرا کنیم و نتایج را تحلیل کنیم.

نتایج حل سوال با الگوریتم Q-learning به ازای ۲۰۰۰ و 10,000 و 700,000 اپیزود یادگیری را در ادامه میبینیم.

نتایج برای ۲۰۰۰ اپیزود یادگیری :




```
=== Training Summary ===
Number of Deaths :        269
Total Rewards    :    -103571
Highest Reward   :        145
Lowest Reward    :      -1160
Accuracy         :     86.55%
========================
```

برای بررسی دقیق تر باز هم تعداد اپیزود را افزایش می دهیم.

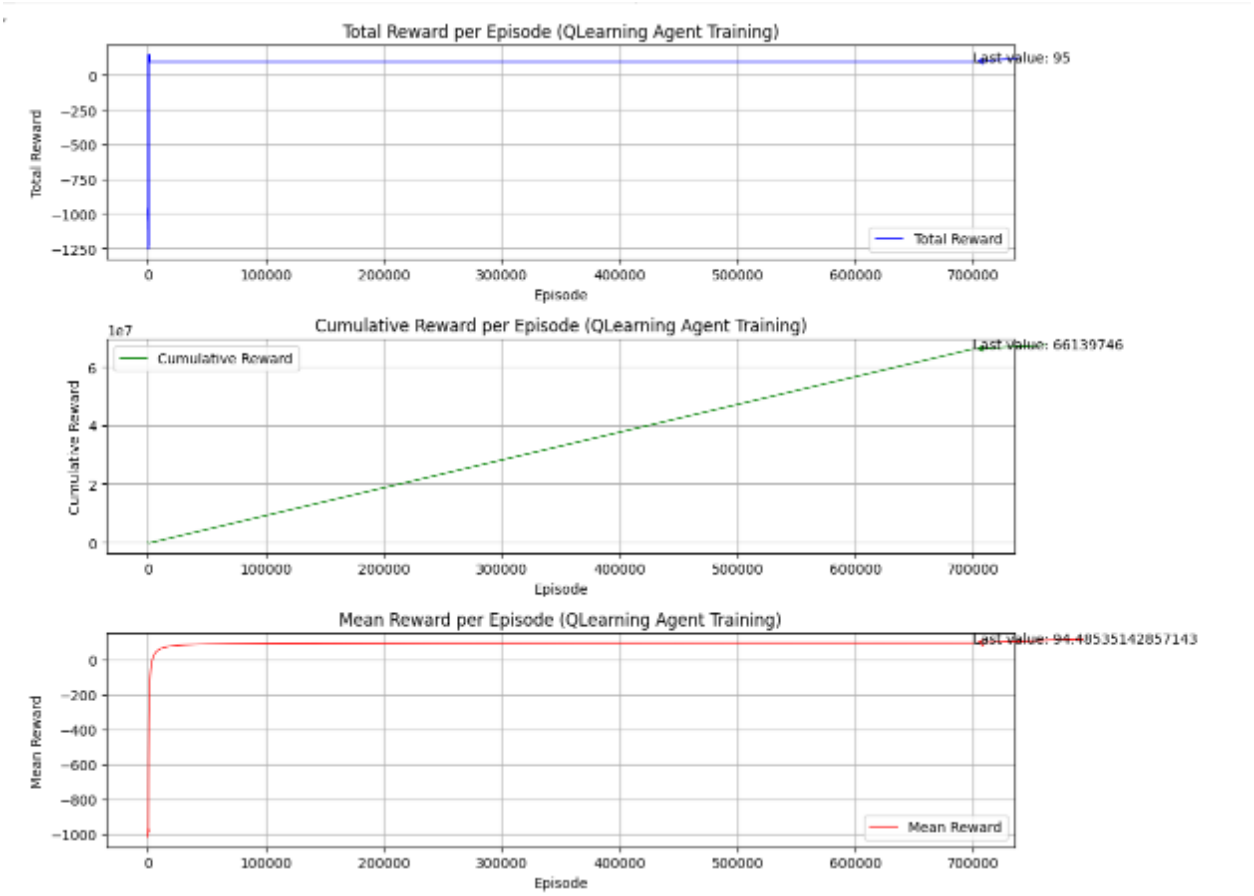
نتایج برای 10,000 اپیزود یادگیری :


```
=== Training Summary ===
Number of Deaths :        311
Total Rewards    :     608750
Highest Reward   :        145
Lowest Reward    :      -1222
Accuracy         :     96.89%
========================
```

با توجه به نمودار های دو حالت بالا ، با افزایش تعداد اپیزودها، مشاهده می‌شود که از حدود اپیزود ۱۵۰۰ به بعد، عامل (Agent)دیگر کشته نمی‌شود و تنها بهترین سیاست را انتخاب می‌کند. این امر به دلیل انتخاب نرخ اکتشاف (Exploration Rate)و کاهش تدریجی آن، همراه با موقعیت اشیاء و ماهیت جدول جستجوی (Lookup Table)در این الگوریتم است. در این شرایط، عامل نتوانسته است سیاستی را پیدا کند که منجر به کسب بیشترین امتیاز ممکن، یعنی 145شود. این وضعیت مشابه به این است که الگوریتم در یک مینیمم محلی گرفتار شده است. با انجام تعداد بیشتری از آزمون‌ها، ممکن است به سیاست بهینه‌ای دست یابیم که به کسب امتیاز ۱۴۵ منجر می‌شود.

علاوه بر این، مشاهده می‌شود که از همان نقطه‌ای که عامل دیگر کشته نمی‌شود، شیب نمودار دوم تغییر کرده است، که نشان‌دهنده بهبود عملکرد عامل است. همچنین، مقدار نهایی نمودار میانگین امتیازها مثبت شده است. برای بررسی دقیق‌تر اینکه آیا این مقدار نهایی است یا خیر، تعداد اپیزودها را به ۷۰۰۰۰۰ افزایش می‌دهیم که عددی بسیار بزرگ است. این بررسی به ما کمک می‌کند تا ببینیم آیا عامل به یک همگرایی بهتر دست می‌یابد یا خیر. لازم به ذکر است که این نوع آزمایش برای الگوریتم DQN امکان‌پذیر نیست و برای رسیدن به نتایج مشابه، نیاز به رویکردهای متفاوت و پیچیده‌تری داریم.

نتایج برای 700,000 اپیزود یادگیری :




```
=== Training Summary ===
Number of Deaths :        327
Total Rewards    :   66139746
Highest Reward   :        145
Lowest Reward    :      -1255
Accuracy         :     99.95%
========================
```

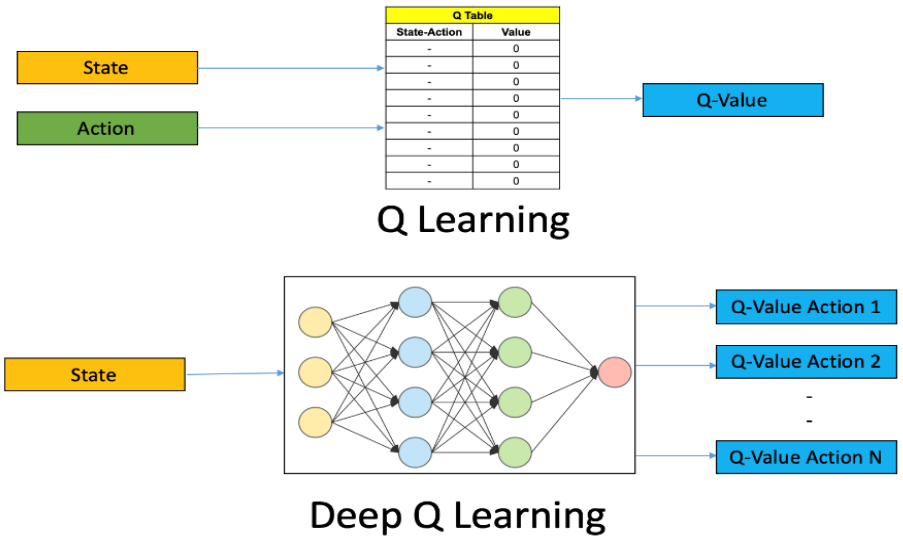
مشاهده میکنید که نمودار سوم به **مقداری نزدیک به ۹۵ همگرا شده است** که بهترین سیاستی است که Agent یادگرفته است.

همچنین خلاصه اطلاعات اموزش مدل برای هر اپیزود هم پایین نمودار ها قرار گرفته است.

الگوریتم DQN

Deep Q-Networks (DQN) یکی از الگوریتم‌های نوآورانه در حوزه یادگیری تقویتی است که توسط محققان DeepMind توسعه یافته است. این الگوریتم برای اولین بار در سال ۲۰۱۳ در مقاله‌ای تحت عنوان "Playing Atari with Deep Reinforcement Learning" معرفی شد. هدف اصلی DQN، غلبه بر محدودیت‌های الگوریتم‌های سنتی Q-learning در محیط‌های پیچیده و بزرگ است که در آن‌ها فضای حالت به‌طور قابل توجهی وسیع است. یکی از تفاوت‌های کلیدی بین DQN و Q-learning در نحوه ذخیره مقادیر Q-value است. در حالی که Q-learning از یک جدول Q برای نگهداری این مقادیر استفاده می‌کند، **DQN به جای آن از شبکه‌های عصبی عمیق بهره می‌برد.** این تغییر بنیادین به DQN این امکان را می‌دهد که با محیط‌های پیچیده‌تری مانند بازی‌های ویدیویی بزرگ و چالش‌برانگیز کار کند و به طور کلی، DQN قادر است سیاست‌های پیچیده‌تری را نسبت به Q-learning یاد بگیرد.

DQN دارای مزایای متعددی است که آن را از الگوریتم‌های سنتی متمایز می‌کند. این الگوریتم به‌ویژه در محیط‌هایی با فضای حالت وسیع عملکرد بهتری دارد و می‌تواند توابع Q-value پیچیده و غیرخطی را با استفاده از شبکه‌های عصبی عمیق یاد بگیرد. همچنین، با به‌کارگیری تکنیک تجربه تکراری، DQN قادر است از داده‌های قبلی بهره‌برداری کند و این موضوع به کاهش وابستگی به ترتیب تجربیات و بهبود کارایی یادگیری کمک می‌کند. با این حال، DQN نیز معایبی دارد؛ از جمله نیاز به قدرت محاسباتی بالا به دلیل استفاده از شبکه‌های عصبی عمیق و پیچیدگی در تنظیم هایپرپارامترها که ممکن است نیاز به سعی و خطا و زمان‌بر باشد. علاوه بر این، برخلاف Q-learning، همگرایی DQN به‌طور قطع تضمین نمی‌شود، که این نکته نیازمند بررسی دقیق‌تری در مقایسه با سایر الگوریتم‌هاست. در شکل زیر رویکرد حل مسئله با دو الگوریتم را میبینیم: (مقایسه دو روش)



شرح معادلات ریاضی DQN : تابع Q-value : در معادله $Q(s; a; \theta)$، θ وزن‌های شبکه عصبی هستند که باید آموزش داده شوند و معمولاً با w نمایش داده می‌شوند.

تابع هزینه:

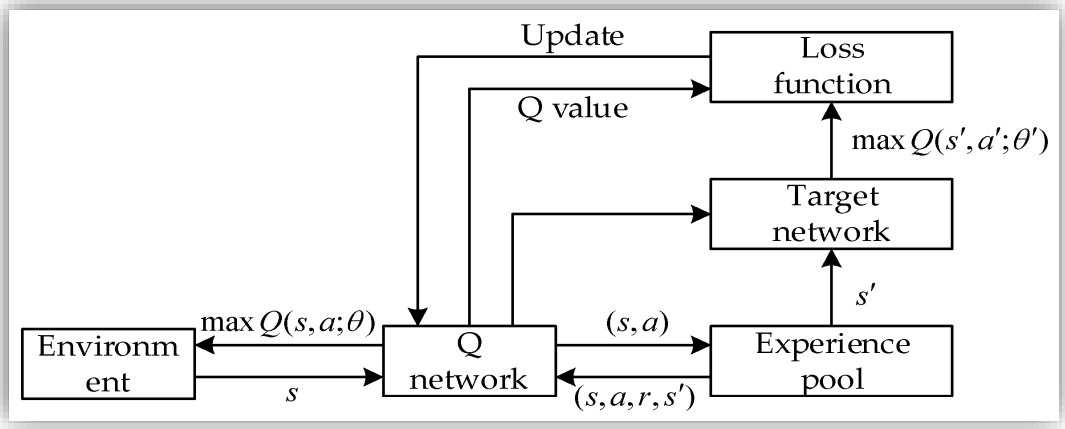
$$L_i(\theta_i) = E_{(s; a; r; s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s'; a'; \bar{\theta}_i) - Q(s; a; \theta_i)\right)^2\right]$$

در این معادله، θ_i وزن‌های فعلی شبکه عصبی و $\bar{\theta}_i$ وزن‌های شبکه هدف هستند که هر از چندی به‌روزرسانی می‌شوند. همچنین، U(D) نمایانگر نمونه‌ای از تجربه تکراری است.

به‌روزرسانی وزن‌های شبکه عصبی:

$$\theta_{i+1} = \theta_i + \alpha \nabla \theta_i L_i(\theta_i)$$

که در این معادله αنرخ یادگیری (learning rate) است و $\nabla \theta_i L_i(\theta_i)$ گرادیان تابع هدف نسبت به وزن‌های شبکه عصبی است



در مسئله Wumpus World، هدف عامل یادگیری، یافتن طلا و اجتناب از خطرات است. برای این کار، از DQN برای تخمین مقادیر Q-value استفاده می‌شود. تنظیم پارامترهای یادگیری، از جمله نرخ یادگیری و عامل تخفیف انجام می‌شود. شبکه عصبی عمیق برای تقریب Q-value طراحی شده و تجربیات عامل در حافظه تجربه تکراری ذخیره می‌شوند. تجربیات به‌طور تصادفی برای به‌روزرسانی وزن‌های شبکه استفاده می‌شوند و شبکه هدف به‌طور دوره‌ای به‌روزرسانی می‌شود.

DQN به دلیل کارایی بالا در محیط‌های پیچیده و با فضای حالت بزرگ، به‌ویژه در بازی‌های ویدیویی، به کار می‌رود و به عامل‌ها کمک می‌کند سیاست‌های پیچیده‌ای را یاد بگیرند. در کد DQN، از کتابخانه Keras برای

ایجاد شبکه عصبی و به‌کارگیری سیاست ϵ-greedy استفاده می‌شود. این الگوریتم نیاز به قدرت محاسباتی بالا و تنظیم دقیق هایپرپارامترها دارد و می‌تواند در فهم و پیاده‌سازی DQN در Wumpus World مفید باشد.

در الگوریتم DQN، حافظه بازپخش (Replay Memory)یکی از اجزای کلیدی است که به بهبود فرآیند یادگیری عامل کمک می‌کند. این حافظه تجربیات عامل را در طول تعاملاتش با محیط ذخیره می‌کند و با نمونه‌برداری تصادفی از این تجربیات، وابستگی زمانی بین نمونه‌ها را کاهش می‌دهد. این رویکرد به یادگیری پایدارتر و کارآمدتری منجر می‌شود، زیرا عامل می‌تواند از تجربیات قبلی خود بارها استفاده کند و نیاز به تعاملات جدید را کاهش دهد.

حافظه بازپخش شامل پنج مولفه اصلی است: وضعیت فعلی، عمل انجام شده، پاداش دریافتی، وضعیت بعدی و شاخص پایان اپیزود. این ساختار به عامل اجازه می‌دهد تا تجربیاتش را به‌طور مؤثرتری تحلیل کند و از نوسانات شدید در به‌روزرسانی وزن‌های شبکه عصبی جلوگیری کند. به طور کلی، استفاده از حافظه بازپخش در DQN به افزایش پایداری و کارایی یادگیری در محیط‌های پیچیده کمک می‌کند.

از کد پایتون زیر استفاده می کنیم و به تحلیل هر بخش آن می پردازیم :

تعاریف اولیه پیاده سازی مانند تعارف مجموعه های Reward Set و Action Set و ایجاد Environment و همچنین قابلیت های Agent عینا مانند قبل تعریف و ایجاد شده اند. در این قسمت تنها به بررسی بخش هایی که در این الگوریتم اضافه شده اند میپردازیم.

class ReplayMemory:

```
class ReplayMemory:
    def __init__(self, capacity, state_shape):
        self.capacity = capacity
        self.states = np.zeros((capacity,) + state_shape, dtype=np.float32)
        self.actions = np.zeros(capacity, dtype=np.int32)
        self.rewards = np.zeros(capacity, dtype=np.float32)
        self.next_states = np.zeros((capacity,) + state_shape, dtype=np.float32)
        self.dones = np.zeros(capacity, dtype=np.bool_)
        self.index = 0
        self.current_size = 0

    def store(self, state, action, reward, next_state, done):
        # Store a new memory of an experience
        self.states[self.index] = state
        self.actions[self.index] = action
        self.rewards[self.index] = reward
        self.next_states[self.index] = next_state
        self.dones[self.index] = done
        self.index = (self.index + 1) % self.capacity
        self.current_size = min(self.current_size + 1, self.capacity)

    def sample(self, batch_size):
        # Randomly sample a batch of experiences from memory
        indices = np.random.choice(self.current_size, batch_size, replace=False)
        return (self.states[indices], self.actions[indices], self.rewards[indices],
                self.next_states[indices], self.dones[indices])
```

با توجه به کد فوق کلاس ReplayMemory به منظور ذخیره و مدیریت تجربیات عامل در الگوریتم‌های یادگیری تقویتی مانند DQN طراحی شده است. این کلاس به عامل این امکان را می‌دهد که تجربیات گذشته را ذخیره کرده و به‌طور تصادفی از آن‌ها برای یادگیری استفاده کند. در سازنده ____init، ظرفیت حافظه و شکل وضعیت‌ها تعریف شده و آرایه‌هایی برای ذخیره وضعیت‌ها، اعمال، پاداش‌ها، وضعیت‌های بعدی و اطلاعات مربوط به پایان اپیزود ایجاد می‌شوند. همچنین، دو متغیر indexو current_size برای پیگیری موقعیت فعلی و تعداد تجربیات موجود در حافظه تعریف شده‌اند.

متد store برای ذخیره یک تجربه جدید در حافظه استفاده می‌شود. این متد تمام اطلاعات مربوط به وضعیت فعلی، عمل، پاداش، وضعیت بعدی و شاخص پایان اپیزود را در موقعیت مشخص‌شده ذخیره می‌کند. پس از ذخیره‌سازی، index به‌روزرسانی می‌شود تا به محل بعدی اشاره کند و در صورت رسیدن به ظرفیت حافظه، به ابتدای آرایه برمی‌گردد. همچنین، current_size به‌گونه‌ای تنظیم می‌شود که حداکثر به ظرفیت حافظه برسد.

متد sample به انتخاب تصادفی یک دسته (batch)از تجربیات از حافظه می‌پردازد. با استفاده از تابع numpy.random.choice، تعدادی ایندکس تصادفی انتخاب شده و تجربیات مربوط به آن‌ها برگشت داده می‌شوند. این روش به عامل اجازه می‌دهد تا از تجربیات متنوع و قدیمی استفاده کند و یادگیری خود را بهبود بخشد، که نتیجه‌اش پایداری بیشتر و کارایی بهتر در یادگیری سیاست‌های پیچیده است.

همچنین همانند الگوریتم قبلی، برای پیاده سازی DQN از کلاس استفاده میکنیم و قابلیت های لازم را برای train شدن مدل ایجاد میکنیم. (در کد به طور کامل موجود است)

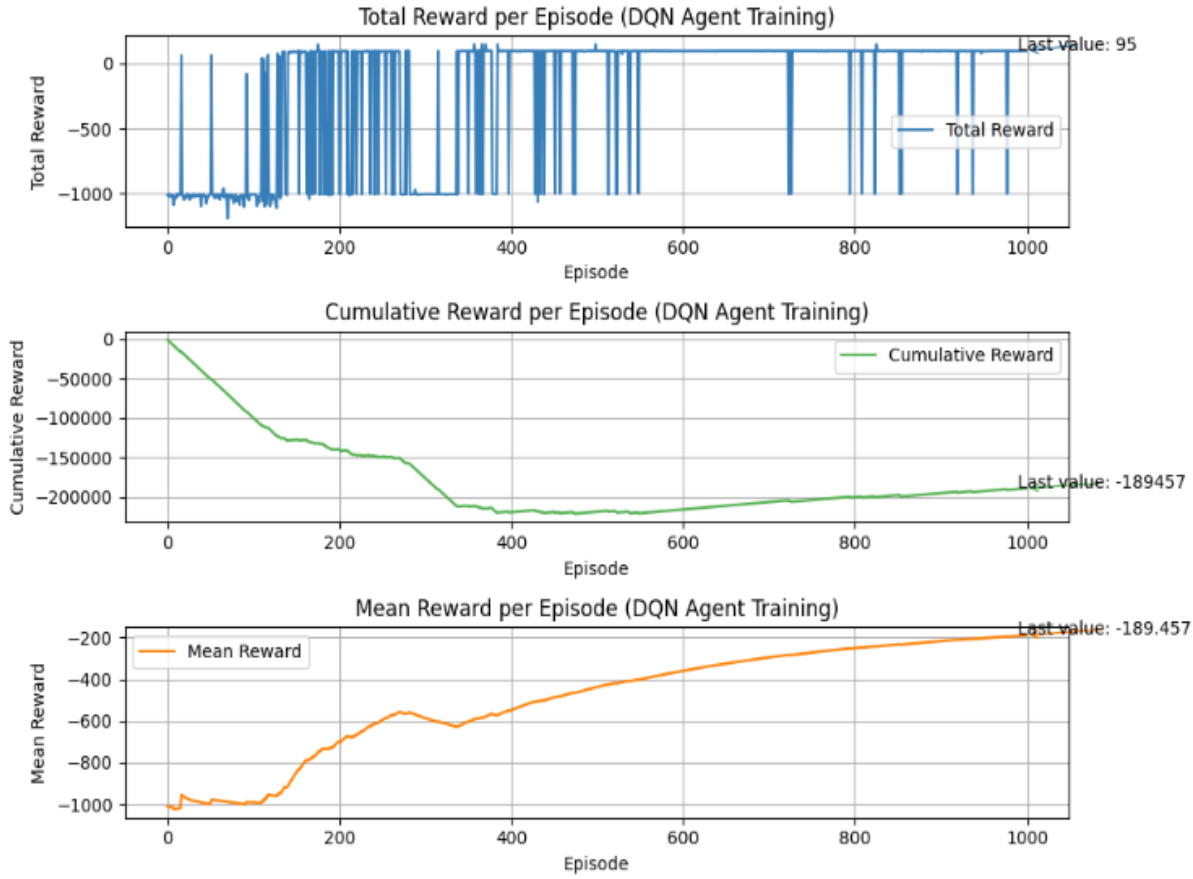
```python
class DQNAgent:
    def __init__(self, learning_rate, gamma, state_shape, num_actions, batch_size,
                 epsilon_initial=1.0, epsilon_decay=0.995, epsilon_final=0.05,
                 replay_buffer_capacity=1000):
        # Initialize DQN agent
        self.learning_rate = learning_rate  # Learning rate for optimization
        self.gamma = gamma  # Discount factor for future rewards
        self.num_actions = num_actions  # Number of possible actions
        self.batch_size = batch_size  # Batch size for training
        self.epsilon = epsilon_initial  # Initial probability for random actions
        self.epsilon_decay = epsilon_decay  # Decay rate for epsilon
        self.epsilon_final = epsilon_final  # Minimum value for epsilon
        self.buffer = ReplayMemory(replay_buffer_capacity, state_shape)
        self.q_network = self._build_model(state_shape, num_actions)
        self.target_network = self._build_model(state_shape, num_actions)
        self.update_target_network()

    def _build_model(self, state_shape, num_actions):
        # Build the neural network model
        model = keras.Sequential([
            keras.layers.Dense(128, activation='relu', input_shape=state_shape),
            keras.layers.Dense(128, activation='relu'),
            keras.layers.Dense(num_actions, activation=None)
        ])
        model.compile(optimizer=keras.optimizers.Adam(learning_rate=self.learning_rate),
                      loss=Huber())
        return model
```

بعد از انتخاب شبکه عصبی مورد استفاده که به طور مفصل در **بخش ه)** توضیح داده شد، با تعیین پارامترهای اولیه مدل، فرآیند آموزش آن را آغاز می‌کنیم. مقادیر هایپرپارامترها با استفاده از روش سعی و خطا به دست آمده‌اند تا به نتیجه مطلوب برسیم. با این حال، تنظیمات اولیه مقادیر پیشنهادی نتوانستند نتیجه مورد نظر ما را فراهم کنند. از کد زیر استفاده می کنیم :

```
# Initialize the environment
env = GridEnvironment()

# Define hyperparameters
learning_rate = 1e-4
gamma = 0.99
state_shape = (env.size * env.size * 5,)
actions = 8             # 4 for moving, 4 for shooting
batch_size = 64

# Create the agent
agent = DQNAgent(learning_rate, gamma, state_shape, actions, batch_size)

# Train the agent
episodes = 1000
total_rewards, cumulative_rewards, mean_rewards = agent.train(env, episodes)

# Plot the results
plot_rewards(total_rewards, cumulative_rewards, mean_rewards, "DQN Agent Training", "dqn_training_rewards.png")

# DQN results for comparison:
DQNtotal_rewards = total_rewards
DQNcumulative_rewards = cumulative_rewards
DQNmean_rewards = mean_rewards
```

کد فوق ابتدا محیطی به نام GridEnvironment را مقداردهی اولیه می‌کند. سپس هایپرپارامترها شامل نرخ یادگیری (learning_rate)، ضریب کاهش پاداش gamma، شکل حالت (state_shape)، تعداد اقدامات (actions)و اندازه بچ (batch_size)را تعریف می‌کند. یک عامل (agent) با استفاده از کلاس DQNAgent و هایپرپارامترهای مشخص‌شده ایجاد می‌شود. سپس، عامل برای ۱۰۰۰ اپیزود (که مقدار این می تواند تغییر کند.) آموزش داده می‌شود و پاداش‌های کل، پاداش‌های تجمعی و میانگین پاداش‌ها ذخیره می‌شوند. در نهایت، نتایج آموزش به صورت نمودار رسم شده و پاداش‌های به دست آمده برای مقایسه ذخیره می‌شوند.

پس از بررسی قسمت‌های کلیدی کد پیاده‌سازی الگوریتمDQN ، می‌توانیم با اجرای این الگوریتم به مدت ۱۰۰۰ اپیزود، نتایج حاصل را مشاهده کنیم. نتایج به دست آمده به شرح زیر هستند و در بخش بعدی به تحلیل آن‌ها خواهیم پرداخت.



و اطلاعات کلی زیر را هم داریم : (برای ۱۰۰۰ اپیزود)

```
=== Training Summary ===
Number of Deaths : 257
Total Rewards    : -189457
Highest Reward   : 145
Lowest Reward    : -1188
Accuracy         : 74.30%
========================
```

در طول آموزش ۱۰۰۰ اپیزود، عامل (Agent)۲۵۷ بار کشته شده است که این عملکرد اندکی بهتر از نتایج الگوریتم Q-learning است(در ان برای این تعداد اپیزود ۲۸۰ بود) . با اینکه معیار دقت به دلایل ذکر شده کاملاً قابل اعتماد نیست، اما بهبود حدود ۲ درصد در دقت مشاهده می‌شود. مشابه الگوریتم Q-learning، عامل توانسته است به بهینه‌ترین سیاست دست یابد و حداکثر امتیاز ممکن یعنی ۱۴۵ را کسب کند. همچنین، حداقل امتیاز دریافتی در اینجا کمی بالاتر از Q-learning است (در اینجا -۱۱۸۸ و در حالت قبل -۱۲۶۵) ، که نشان‌دهنده این است که عامل به طور مؤثری در محیط گرید حرکت کرده و سریع‌تر به نتیجه اپیزودها دست یافته است.

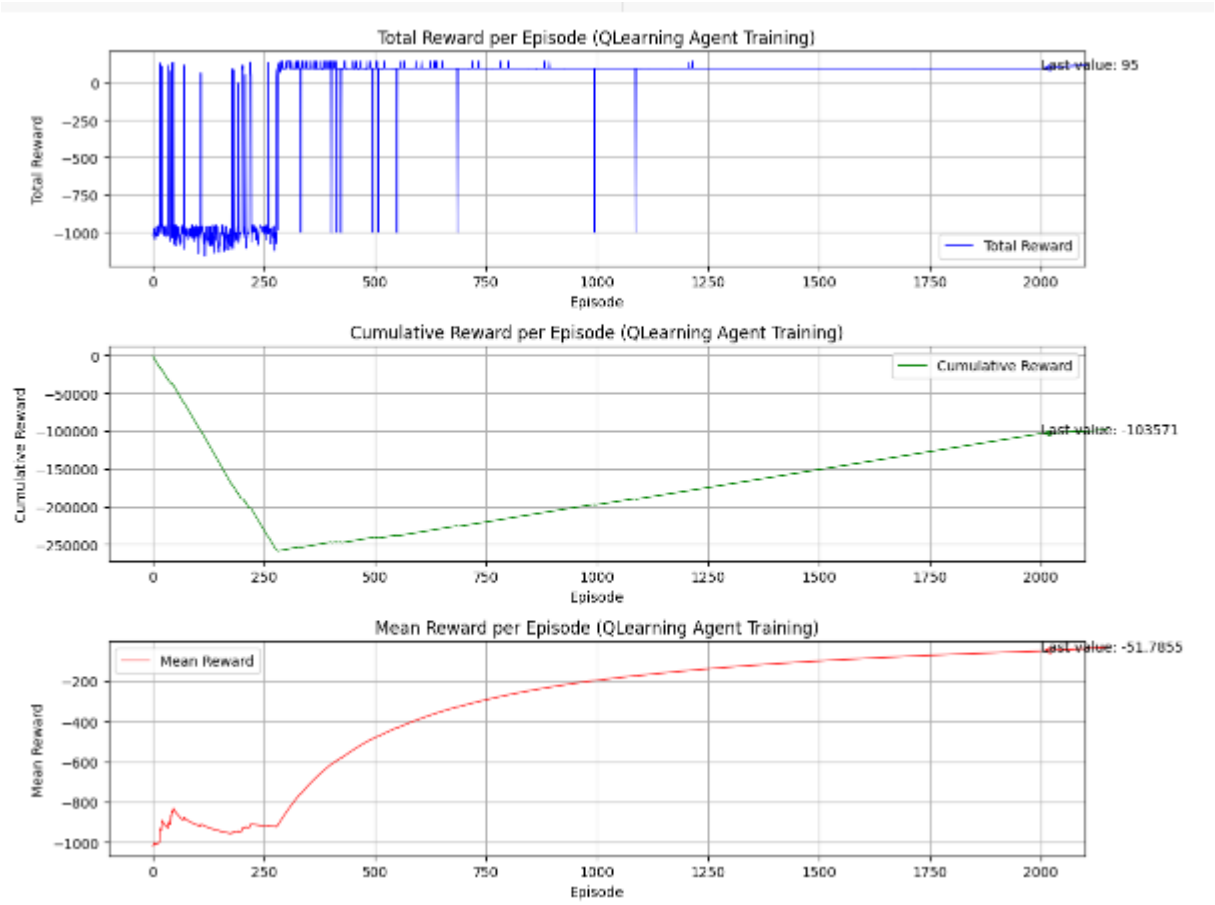
همان‌طور که در معرفی الگوریتم Q-learning اشاره کردیم، ممکن است در برخی موارد همگرایی در این الگوریتم حاصل نشود. در اینجا، حتی بعد از ۵۰۰ اپیزود، عامل همچنان پاداش‌های منفی دریافت می‌کند و کشته می‌شود.

برای بررسی همگرایی، این مدل را برای **۲۰۰۰ اپیزود** نیز اجرا کردیم و نتایج آن را در زیر ارائه خواهیم داد.

با هدف مقایسه دقیق‌تر، تعداد اپیزودها را به ۲۰۰۰ افزایش دادیم و مدل را با هر دو الگوریتم آموزش دادیم تا به نتایج دقیق‌تری دست یابیم.

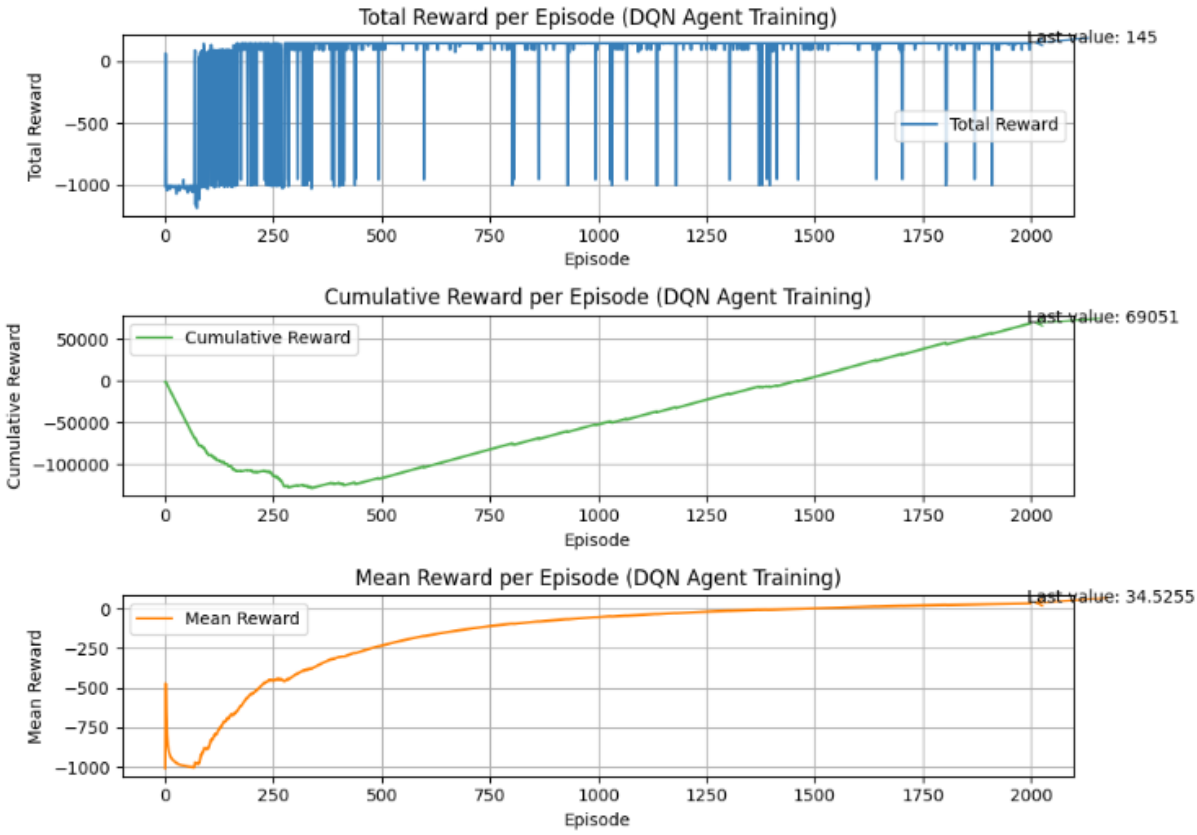
نتایج برای ۲۰۰۰ اپیزود به صورت زیر است :

نتایج برای الگوریتم Q learning :




```
=== Training Summary ===
Number of Deaths :        269
Total Rewards    :    -103571
Highest Reward   :        145
Lowest Reward    :      -1160
Accuracy         :     86.55%
========================
```

و نتایج برای الگوریتم DQN به صورت زیر است :




```
=== Training Summary ===
Number of Deaths : 181
Total Rewards    : 69051
Highest Reward   : 145
Lowest Reward    : -1183
Accuracy         : 90.95%
========================
```

نکته‌ای که باید به آن توجه کرد این است که عامل (Agent) به طور قطع کشته نشده و در بسیاری از موارد در محیط به شدت فعال بوده، اما به نتیجه‌ای نرسیده است. در برخی مواقع، عامل موفق به کشتن Wumpus و پیدا کردن گنج شده، اما به دلیل حرکات مکرر و دریافت امتیازهای منفی، نمی‌توانیم به وضوح تعیین کنیم که آیا واقعاً کشته شده یا خیر. ایده‌آل این است که نتیجه نهایی هر اپیزود را به‌طور جداگانه بررسی کنیم تا مشخص شود آیا عامل کشته شده است یا توانسته گنج را بیابد. همچنین، ضروری است که برای تعداد مراحل (step)در هر اپیزود محدودیت‌هایی تعیین کنیم و براساس آن نتیجه‌گیری کنیم.

به دلیل زمان طولانی اجرای این الگوریتم و محدودیت‌های زمانی، به این نتایج بسنده می‌کنیم. الگوریتم DQN همچنان پتانسیل بهبود دارد، به‌ویژه در زمینه کاهش نرخ کاوش (Exploration rate)و تنظیم نرخ یادگیری

(learning rate)برای جلوگیری از گیر افتادن در مینیمم‌های محلی، و همچنین تعیین تعداد حرکات معقول در هر اپیزود. از آنجا که هدف اصلی ما مقایسه این دو الگوریتم نبوده است، این مقایسه را در بخش بعدی به تفصیل خواهیم پرداخت. هدف از این آزمایش، بررسی این نکته بود که آیا در Q-learning بعد از تعدادی مشخص اپیزود، عامل می‌تواند بدون کشته شدن بازی را به خوبی انجام دهد یا خیر. در مقابل، دستیابی به همگرایی در DQN کار آسانی نیست.

با ادامه بررسی مدل به‌دست‌آمده پس از ۱۰۰۰ اپیزود با الگوریتم DQN، می‌توان دید که نمودارهای میانگین و مجموع امتیازها از حدود اپیزود ۴۰۰ به بعد روند صعودی را آغاز کرده‌اند. اگرچه پس از آن امتیازهای منفی نیز دریافت شده، اما به دلیل همگرایی الگوریتم به امتیاز ۱۴۵، روند صعودی میانگین پاداش‌ها به شکل قابل توجهی ادامه دارد. مانند الگوریتم Q-learning، روند میانگین امتیازها همچنان به صورت صعودی است و به عدد مشخصی همگرا نشده است. حتی در ۲۰۰۰ اپیزود نیز این روند صعودی ادامه یافته(از اپیزود ۲۸۰ به بعد) و با توجه به تجربیات گذشته، می‌دانیم که این نمودار در نهایت به عدد ۱۴۵ همگرا خواهد شد. همچنین، نتایج امتیازهای انباشته نشان می‌دهد که این الگوریتم نسبت به Q-learning نتایج بهتری ارائه داده و سریع‌تر به صعود دست یافته است. (در کل در حالت ۲۰۰۰ اپیزود دقت بالاتری هم بدست امده است)

ب)

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؟

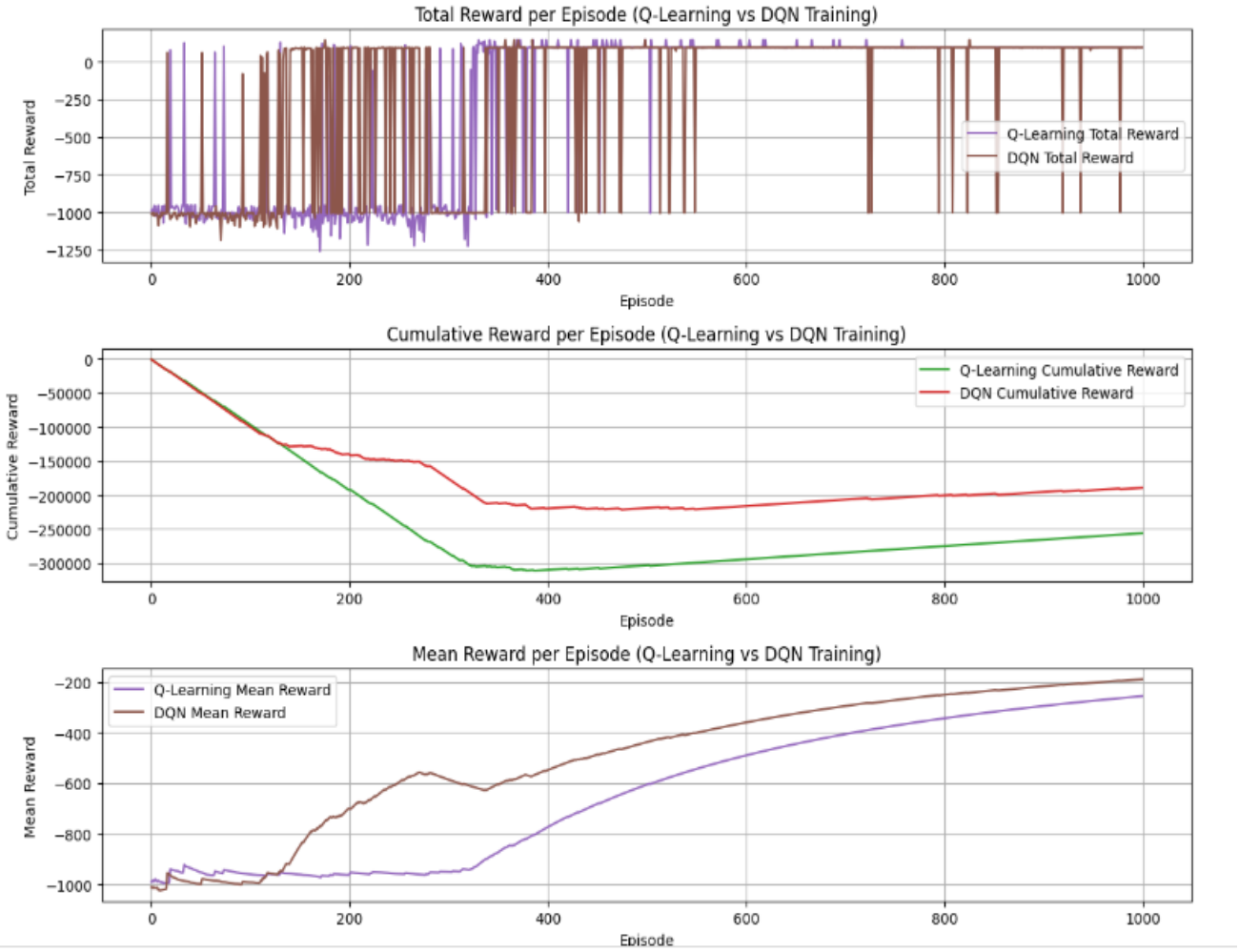
بعد از آموزش هر دو مدل، نتایج را توسط کد زیر کنار هم ترسیم می کنیم تا بهتر مقایسه کنیم.

یک بار برای حالت ۱۰۰۰ اپیزود و یک بار هم برای ۲۰۰۰ اپیزود مقایسه را انجام می دهیم.

```python
def plot_rewards(q_learning_rewards, q_learning_cumulative, q_learning_mean,
                 dqn_rewards, dqn_cumulative, dqn_mean, title, filename):
    # Create subplots with specified figure size
    fig, axs = plt.subplots(3, 1, figsize=(12, 9))

    # Plot Total Reward per Episode
    axs[0].plot(q_learning_rewards, label='Q-Learning Total Reward', color='tab:purple')
    axs[0].plot(dqn_rewards, label='DQN Total Reward', color='tab:brown')
    axs[0].set_xlabel('Episode')
    axs[0].set_ylabel('Total Reward')
    axs[0].set_title(f'Total Reward per Episode ({title})')
    axs[0].legend()
    axs[0].grid(True)

    # Plot Cumulative Reward per Episode
    axs[1].plot(q_learning_cumulative, label='Q-Learning Cumulative Reward', color='tab:green')
    axs[1].plot(dqn_cumulative, label='DQN Cumulative Reward', color='tab:red')
    axs[1].set_xlabel('Episode')
    axs[1].set_ylabel('Cumulative Reward')
    axs[1].set_title(f'Cumulative Reward per Episode ({title})')
    axs[1].legend()
    axs[1].grid(True)

    # Plot Mean Reward per Episode
    axs[2].plot(q_learning_mean, label='Q-Learning Mean Reward', color='tab:purple')
    axs[2].plot(dqn_mean, label='DQN Mean Reward', color='tab:brown')
    axs[2].set_xlabel('Episode')
    axs[2].set_ylabel('Mean Reward')
    axs[2].set_title(f'Mean Reward per Episode ({title})')
    axs[2].legend()
    axs[2].grid(True)

    # Adjust layout to prevent overlap of titles and labels
    plt.tight_layout()
```

کد فوق به منظور ترسیم و مقایسه نتایج الگوریتم‌های یادگیری تقویتی Q-Learning و DQN طراحی شده است. تابع plot_rewards چندین آرگومان ورودی می‌گیرد که شامل جوایز کلی، تجمعی و میانگین برای هر دو الگوریتم می‌باشد. هدف این است که با استفاده از کتابخانه matplotlib، سه نمودار جداگانه برای نمایش این داده‌ها ایجاد شود. نمودار اول نشان‌دهنده جوایز کلی در هر اپیزود است که به صورت خطوط جداگانه برای Q-Learning و DQN ترسیم شده است. این نمودار به وضوح روند تغییرات جوایز کلی را در طول زمان (اپیزودها) نمایش می‌دهد. نمودار دوم به نمایش جوایز تجمعی می‌پردازد و اطلاعات مشابهی را برای مجموع جوایز در طول اپیزودها ارائه می‌کند. در نهایت، نمودار سوم میانگین جوایز را نمایش می‌دهد که می‌تواند به ارزیابی کیفیت یادگیری و کارایی الگوریتم‌ها کمک کند.

مقایسه نتایج برای حالت ۱۰۰۰ اپیزود :



از نمودارهای ارائه شده، می‌توانیم اطلاعات قابل توجهی درباره عملکرد دو الگوریتم Q-Learning و DQN استخراج کنیم. در ابتدا، به بررسی نمودار پاداش کل در هر اپیزود می‌پردازیم. این نمودار نشان می‌دهد که Q-Learning در مراحل اولیه آموزش، به ویژه تا حدود ۳۵۰ اپیزود، با جوایز منفی و نوسانات زیادی مواجه است. این نوسانات به دلیل نرخ بالای اکتشاف و تلاش‌های بیشتر عامل برای کاوش در محیط رخ می‌دهد.

پس از گذشت حدود ۴۰۰ اپیزود، روند پاداش‌ها تغییر کرده و به تدریج مثبت می‌شوند. این تغییر نشان‌دهنده یادگیری بهتر سیاست‌های بهینه توسط عامل Q-Learning است و به وضوح کاهش نوسانات در این مرحله را نیز نشان می‌دهد. این بهبود در عملکرد نشان می‌دهد که عامل در حال نزدیک شدن به یک سیاست مؤثرتر است.

در مورد DQN، نمودار پاداش کل نیز نشان‌دهنده نوساناتی در اوایل آموزش است، اما این نوسانات به مراتب کمتر از Q-Learning هستند. از حدود اپیزود ۱۷۰ به بعد، پاداش‌های DQN به تدریج مثبت و پایدارتر می‌شوند. این

رفتار به این معناست که DQN توانسته سریع‌تر از Q-Learning به سیاست‌های بهینه دست یابد و به صورت مؤثرتری از تجربیات خود بهره‌برداری کند. این بررسی‌ها نشان می‌دهند که DQN در دستیابی به جوایز مثبت، به‌ویژه در مراحل اولیه، عملکرد بهتری نسبت به Q-Learning دارد. این امر می‌تواند به انتخاب بهینه‌تری برای مسائل یادگیری تقویتی منجر شود و نشان‌دهنده مزیت‌های DQN در یادگیری عمیق و اکتشاف محیط‌های پیچیده‌تر است.

در بررسی نمودار پاداش تجمعی در هر اپیزود، می‌توانیم نتایج جالبی از عملکرد الگوریتم‌ها استخراج کنیم. برای Q-Learning، مشاهده می‌شود که پاداش تجمعی در مراحل اولیه به سرعت کاهش می‌یابد، که این امر به دلیل پاداش‌های منفی و نوسانات زیاد است. با گذشت زمان و از حدود اپیزود ۳۵۰ به بعد، پاداش تجمعی به تدریج افزایش می‌یابد، اما همچنان نسبت به DQN در سطح پایین‌تری قرار دارد که این موضوع به وضوح در نمودارهای مقایسه‌ای قابل مشاهده است. در مورد DQN، نمودار پاداش تجمعی نیز در اوایل آموزش کاهش می‌یابد، اما این کاهش به مراتب کمتر از Q-Learning است. به عنوان مثال، Q-Learning تا حدود -۳۰ هزار کاهش می‌یابد، در حالی که DQN تنها به حدود -۱۰ هزار می‌رسد. این نشان‌دهنده این است که DQN در مراحل اولیه قادر به شناسایی سیاست‌هایی است که به پاداش‌های مثبت منجر می‌شوند، در حالی که Q-Learning برای رسیدن به این هدف حدود ۳۰۰ اپیزود زمان نیاز دارد و نتوانسته است سیاست بهینه‌ای که پاداش ۱۴۵ را به ارمغان می‌آورد، یاد بگیرد. از اپیزود ۱۵۰ به بعد، پاداش تجمعی DQN به سرعت افزایش یافته و به طور پیوسته مثبت می‌شود. این روند نشان‌دهنده عملکرد بهتر DQN در یادگیری سیاست‌های بهینه است، به طوری که نمودار این الگوریتم به وضوح بالاتر از Q-Learning قرار دارد و وضعیت بهتری را نمایش می‌دهد. این یافته‌ها برتری DQN را در یادگیری سریع‌تر و مؤثرتر در مقایسه با Q-Learning تأیید می‌کند.

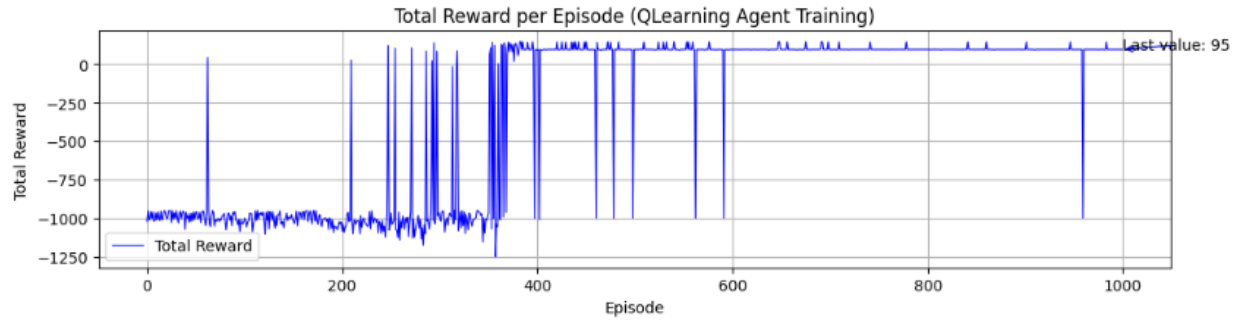
در نمودار تحلیل میانگین پاداش در هر اپیزود ؛ در ابتدا، در الگوریتم Q-Learning، میانگین پاداش در مراحل ابتدایی آموزش به شدت پایین و منفی است. این امر ناشی از کاوش‌های زیاد و دریافت پاداش‌های منفی از سوی عامل است. از حدود اپیزود ۳۰۰ به بعد، مشاهده می‌شود که میانگین پاداش به تدریج افزایش می‌یابد، اما هنوز از DQN پایین‌تر است و نتوانسته عملکرد بهتری ارائه دهد. به‌عبارت دیگر، این الگوریتم به مقدار پایین‌تری همگرا می‌شود و قادر به یادگیری سیاست بهینه‌ای که منجر به کسب پاداش ۱۴۵ می‌شود، نیست.

در مقابل، DQN در مراحل اولیه آموزش نیز میانگین پاداش پایینی دارد، اما به سرعت این مقدار افزایش می‌یابد. از حدود اپیزود ۱۰۰ به بعد، میانگین پاداش به طور مداوم به سمت مثبت حرکت می‌کند و این نشان‌دهنده یادگیری بهتر سیاست‌های بهینه است. این الگوریتم به تدریج عملکرد بهتری از خود نشان می‌دهد و به بهینه‌سازی پاداش نزدیک‌تر می‌شود. در نهایت، نمودارهای موجود نشان می‌دهند که در اپیزودهای بیشتر، DQN به مقدار

۱۴۵ نزدیک می‌شود در حالی که Q-Learning به عدد ۹۵ همگرا می‌شود. این مقایسه به وضوح نشان می‌دهد که DQN در این زمینه نسبت به Q-Learning عملکرد بهتری دارد و قادر است سیاست‌های بهینه را به شکل مؤثرتری یاد بگیرد.

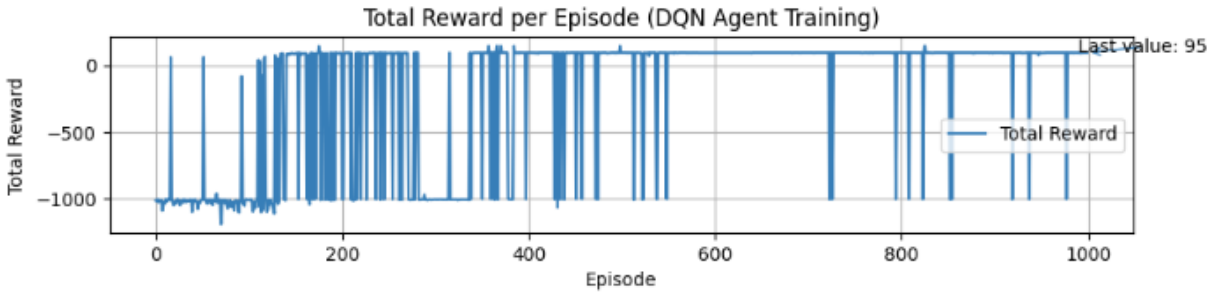
الگوریتم Q-Learning برای دستیابی به عملکرد پایدار و مثبت به حدود ۳۰۰ اپیزود نیاز دارد، که این زمان طولانی‌تر نشان‌دهنده ضرورت کاوش بیشتر و یادگیری سیاست‌های بهینه‌تر است. با وجود این، این مدل نتوانسته است سیاست بهینه‌ای برای مسئله خاص یاد بگیرد. پس از حدود ۵۲۰ اپیزود، Q-Learning به طور کامل بازی را فراگرفته و دیگر پاداش منفی دریافت نمی‌کند.

نمودار نتیجه پاداش در هر اپیزود با Q learning را مشاهده می کنیم:



در مقابل، الگوریتم DQN تنها به حدود ۱۰۰ اپیزود برای رسیدن به عملکرد پایدار و مثبت نیاز دارد که نشان‌دهنده سرعت بالاتر در یادگیری و شناسایی سیاست‌های بهینه است. این الگوریتم موفق به یادگیری سیاست بهینه مسئله شده است، اما همگرایی در آن به وضوح مشخص نیست. همچنین، نمی‌توان به طور دقیق گفت که پاداش‌های منفی به دلیل کشته شدن عامل یا حرکات بیش از حد آن ایجاد شده‌اند و راه‌حل‌های احتمالی برای این مشکلات نیز به تفصیل مورد بررسی قرار گرفته است.

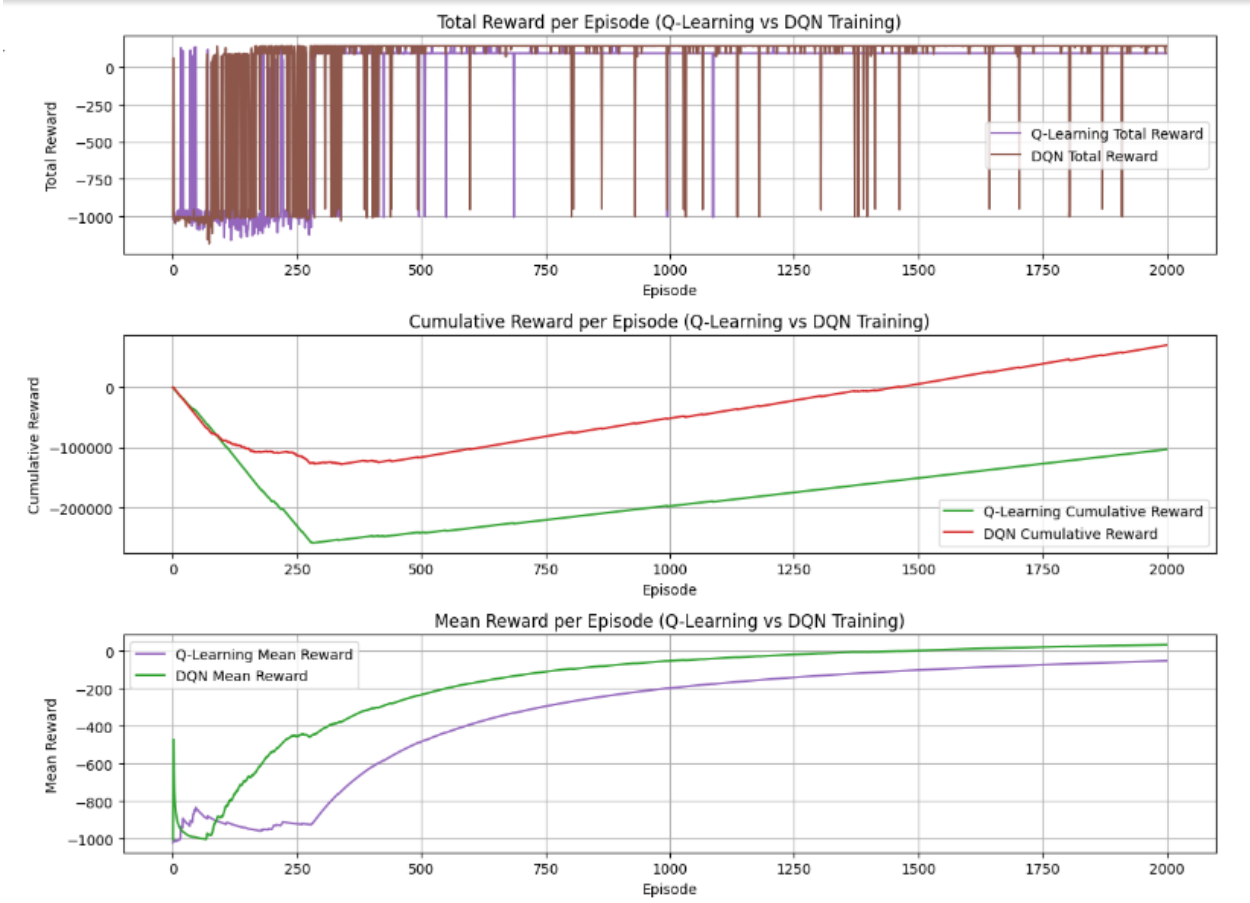
و نتیجه پاداش برای هر اپیزود در DQN را نیز در زیر می بینیم:



Q-Learning، با وجود نوسانات و پاداش‌های منفی در مراحل اولیه، به تدریج به سیاست‌های بهینه دست می‌یابد، اما این فرآیند ممکن است طولانی و ناپایدار باشد و نتواند بهترین سیاست را شناسایی کند. در مقابل، DQNبا استفاده از شبکه‌های عصبی عمیق و حافظه بازپخش، توانسته است سیاست‌های بهینه را سریع‌تر و پایدارتر یاد بگیرد و به نتایج بهتری در محیط Wumpus World دست یابد. به طور کلی، DQNبه مراتب سریع‌تر و موثرتر از Q-Learning عمل می‌کند و گزینه بهتری برای مسائل پیچیده با فضای حالت بزرگ است.

مقایسه نتایج برای حالت ۲۰۰۰ اپیزود :

در نمودار های زیر به مقایسه عملکرد دو الگوریتم در حالت ۲۰۰۰ اپیزود می پردازیم.



نتایج به دست آمده از الگوریتم‌ها مشابه بخش قبلی است. الگوریتم Q-Learning به تدریج تا رسیدن به یک اپیزود خاص همگرا شده و در نهایت به امتیاز ۹۵ که بهترین سیاستی است که قادر به شناسایی آن بوده، می‌رسد.

بعد از این همگرایی، عامل (Agent)هرگز نمی‌میرد. با توجه به آموزش ۲۰۰۰ اپیزود، این همگرایی در حدود اپیزود ۱۲۵۰ اتفاق می‌افتد. پس از این مرحله، عامل به بازی ادامه می‌دهد و تنها موفق به کشتن Wumpus می‌شود، بدون اینکه قادر به پیدا کردن گنج باشد، مگر در شرایط خاص... از طرف دیگر، الگوریتم DQN همچنان در طول آموزش پاداش‌های منفی دریافت می‌کند. نمودار پاداش تجمعی برای الگوریتم Q-Learning بالاتر از DQN قرار دارد و روند صعودی آن بسیار زودتر آغاز می‌شود. این الگوریتم پس از حدود ۲۰۰ اپیزود، تنها تا -۱۰ هزار کاهش می‌یابد، در حالی که Q-Learning به -۳۰ هزار سقوط کرده و روند صعودی آن پس از گذشت تقریباً ۴۵۰ اپیزود آغاز می‌شود. همچنین، بررسی نمودار میانگین پاداش نشان می‌دهد که نمودار DQN به سرعت به سمت بالا حرکت می‌کند. با وجود اینکه هر دو الگوریتم در نهایت صعودی می‌شوند، هیچ یک از آنها پس از ۲۰۰۰ اپیزود به همگرایی کامل نمی‌رسند. با این حال، DQN به میزان مثبت دست یافته است، در حالی که Q-Learning پس از ۲۰۰۰ اپیزود همچنان در وضعیت منفی قرار دارد و به‌طور کلی عملکرد بهتری نسبت به Q-Learning دارد.

ج)

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؟

➢ تاثیر نرخ اکتشاف ϵ بر فرآیند یادگیری مدل *Q-Learning*

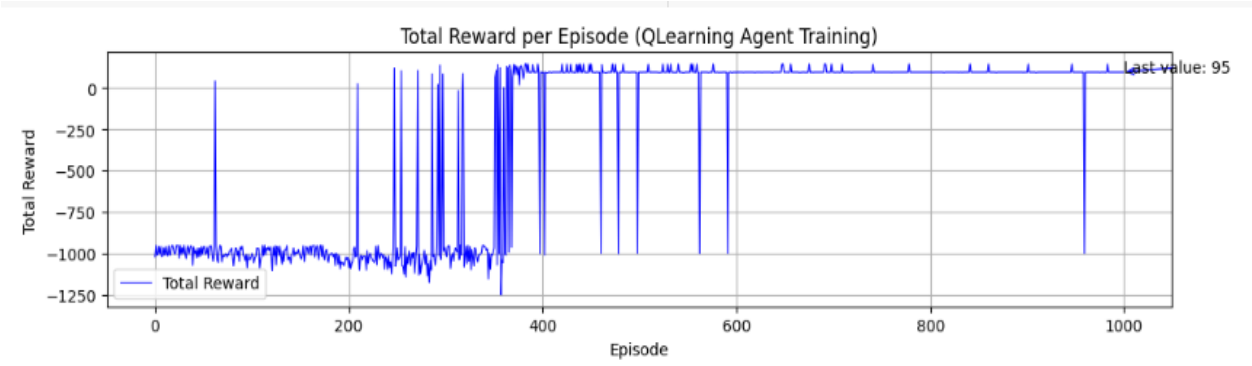
نرخ اکتشاف ϵ در الگوریتم Q-Learning به عنوان یکی از پارامترهای کلیدی در فرآیند یادگیری عامل شناخته می‌شود. این نرخ مشخص می‌کند که عامل چه مقدار از زمان خود را به کاوش محیط و چه مقدار را به بهره‌برداری از سیاست‌های آموخته‌شده اختصاص می‌دهد. در ابتدای یادگیری، مقدار ϵ برابر با ۱,۰ است که به عامل اجازه می‌دهد به طور کامل به کاوش بپردازد. با پایان هر اپیزود، مقدار ϵ با ضریب کاهش اکتشاف (۰,۹۹۵) ضرب می‌شود و به تدریج کاهش می‌یابد، به طوری که در پایان ۱۰۰۰ اپیزود به مقادیر بسیار کم می‌رسد.

نرخ اکتشاف بالا، به ویژه در مراحل اولیه یادگیری، به عامل این امکان را می‌دهد که از طریق انجام اعمال تصادفی، تنوع بیشتری از تجربیات را کسب کند. این تنوع در داده‌ها به عامل کمک می‌کند تا با شناسایی محیط به نحو بهتری، بهینه جهانی را پیدا کند. همچنین، نرخ اکتشاف بالا می‌تواند از گیر افتادن در نقاط بهینه محلی جلوگیری کند و به عامل این امکان را بدهد که به گزینه‌های بهتر دسترسی پیدا کند ؛ با این حال، نرخ اکتشاف بالا همچنین می‌تواند معایبی داشته باشد. به دلیل انجام اعمال تصادفی، عامل ممکن است پاداش‌های کمتری کسب کند و این می‌تواند منجر به ناپایداری در فرآیند یادگیری شود. این ناپایداری ممکن است باعث شود که همگرایی به سیاست بهینه مدت بیشتری طول بکشد و در برخی موارد، عامل نتواند به بهینه‌ترین نتیجه دست یابد.

در مقابل، کاهش نرخ اکتشاف به معنی افزایش بهره‌برداری از سیاست‌های آموخته‌شده است. در مراحل پایانی یادگیری، این وضعیت می‌تواند به عامل کمک کند تا از دانش موجود به نحو بهتری استفاده کند و در نتیجه پاداش‌های بیشتری کسب کند. این نوع بهره‌برداری معمولاً باعث می‌شود که فرآیند یادگیری پایدارتر شود و همگرایی به سیاست بهینه به شکل سریع‌تری انجام شود. با این حال، نرخ اکتشاف پایین نیز ممکن است معایبی به همراه داشته باشد. یکی از مشکلات این است که عامل ممکن است فرصت‌های کاوش محیط را از دست بدهد و از کشف سیاست‌های بهینه‌تر محروم بماند. این می‌تواند باعث شود که عامل در نقاط بهینه محلی گیر کند و نتواند به بهینه جهانی دست یابد.

به طور کلی، تنظیم بهینه نرخ اکتشاف ϵ از اهمیت بالایی برخوردار است. این تنظیم باید به گونه‌ای انجام شود که تعادل مناسبی بین کاوش و بهره‌برداری برقرار شود، به طوری که عامل بتواند بهترین عملکرد را در طول فرآیند یادگیری داشته باشد. مدیریت صحیح این پارامتر می‌تواند تأثیر چشمگیری بر کیفیت یادگیری و عملکرد نهایی عامل داشته باشد.

در شکل زیر امتیاز های مدل Q learning در هر اپیزود را مشاهده می کنیم:



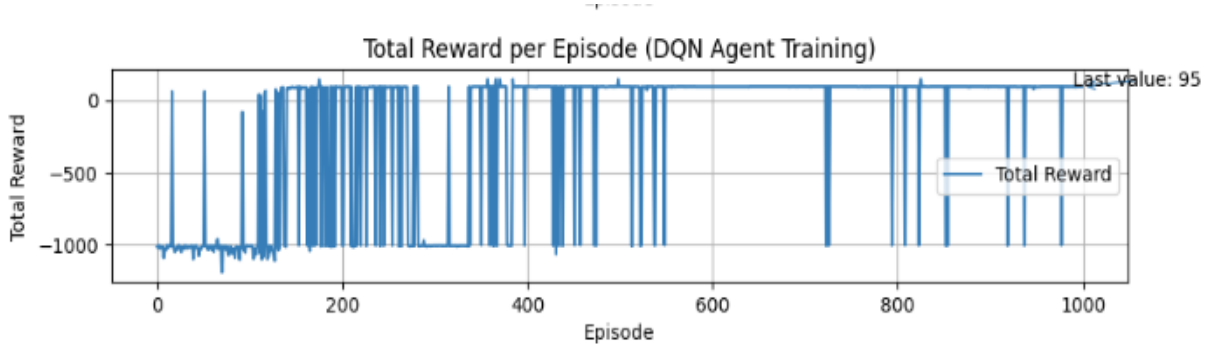
با توجه به نمودار ارائه‌شده، می‌توانیم استدلال کنیم که در مراحل ابتدایی یادگیری، وقتی نرخ ϵ بالا است، پاداش‌ها نوسان‌های زیادی دارند و معمولاً منفی هستند. این نوسانات به علت کاوش بیشتر عامل و انتخاب اعمال تصادفی

است. اما با گذشت زمان و کاهش نرخ ϵ، عامل به تدریج به بهره‌برداری از سیاست‌های آموخته‌شده روی می‌آورد و در نتیجه پاداش‌ها پایدارتر و مثبت‌تر می‌شوند. در نهایت، در حدود اپیزود ۵۵۰ ، عامل به پاداش‌های مثبت و پایدار می‌رسد که نشان‌دهنده همگرایی به سیاست بهینه است.

همان‌طور که قبلاً اشاره شد، انتخاب و تنظیم صحیح نرخ اکتشاف ϵ در الگوریتم Q-Learning بسیار مهم است. در مراحل اولیه یادگیری، نرخ بالای اکتشاف به عامل این امکان را می‌دهد که محیط را بهتر درک کند و تجربیات متنوع‌تری را جمع‌آوری کند. در مراحل بعدی، کاهش این نرخ به عامل کمک می‌کند که بیشتر از سیاست‌های یادگرفته‌شده بهره‌برداری کند و به سیاست بهینه نزدیک‌تر شود. تنظیم تدریجی ϵ و کاهش آن از طریق ضریب کاهش می‌تواند به بهبود عملکرد و پایداری فرآیند یادگیری منجر شود.

➢ تاثیر نرخ اکتشاف ϵ بر فرآیند یادگیری مدل *DQN*

اکنون موضوع را برای مدل DQN تحلیل می کنیم. با نگاهی به نمودار ارائه‌شده در زیر، مشاهده می‌شود که در مراحل ابتدایی یادگیری، زمانی که نرخ اکتشاف بالا است، پاداش‌ها نوسانات زیادی دارند و معمولاً منفی هستند. این نوسانات به دلیل تمرکز بیشتر عامل بر کاوش محیط و انجام اعمال تصادفی است که به او کمک می‌کند اطلاعات بیشتری درباره محیط به دست آورد و تجربیات متنوع‌تری کسب کند.



در میانه فرآیند یادگیری، با کاهش نرخ اکتشاف، عامل به تدریج به بهره‌برداری از سیاست‌های آموخته‌شده می‌پردازد. در این مرحله، نوسانات پاداش کاهش می‌یابد و پاداش‌های مثبت بیشتری مشاهده می‌شود، که نشان‌دهنده بهبود عملکرد عامل و یادگیری سیاست‌های بهینه‌تر است. با این حال، همچنان ممکن است امتیازهای منفی دریافت شود و نشان‌دهنده عدم همگرایی کامل است. این موضوع می‌تواند با محدود کردن تعداد مراحل (step)در هر اپیزود و مشخص کردن نتایج هر اپیزود مورد بررسی بیشتری قرار گیرد.

در مراحل پایانی یادگیری، وقتی نرخ اکتشاف به پایین‌ترین سطح خود می‌رسد، پاداش‌ها پایدارتر و مثبت‌تر می‌شوند. عامل در این مرحله بیشتر به بهره‌برداری از سیاست‌های آموخته‌شده می‌پردازد، اما هنوز به سیاست بهینه همگرا نشده است. با این حال، نمودار نشان می‌دهد که عامل به تدریج پاداش‌های بالاتری کسب می‌کند و عملکرد نسبتاً پایدارتری از خود نشان می‌دهد.

همان‌طور که قبلاً اشاره شد، تأثیر نرخ اکتشاف بر فرآیند یادگیری در مدل DQN نیز همانند الگوریتم Q-Learning حائز اهمیت است. نرخ اکتشاف بالا در مراحل اولیه به عامل کمک می‌کند تا محیط را به‌خوبی بشناسد و تجربیات متنوع‌تری را به‌دست آورد. این کاوش بیشتر از گیر افتادن در بهینه محلی جلوگیری می‌کند. با کاهش تدریجی نرخ اکتشاف، عامل به سمت بهره‌برداری از سیاست‌های آموخته‌شده حرکت می‌کند و به تدریج به سیاست بهینه نزدیک‌تر می‌شود.

کاهش نوسانات در پاداش‌ها و افزایش پاداش‌های مثبت نشان‌دهنده بهبود عملکرد عامل و یادگیری سیاست‌های بهینه‌تر است. به طور کلی، تنظیم مناسب نرخ اکتشاف و کاهش تدریجی آن نقش بسیار مهمی در بهبود فرآیند یادگیری و عملکرد نهایی عامل در الگوریتم DQN ایفا می‌کند.

```
Episode 858/1000, Total Reward: 95, Epsilon: 0.05
```

پس از تقریباً ۶۰۰ اپیزود، مقدار اپسیلون به حدود ۰,۰۵ همگرا می‌شود و این وضعیت به ما این امکان را می‌دهد که از مرحله کاوش (Explore)به مرحله بهره‌برداری (Exploit)منتقل شویم. این محاسبه به صورت زیر است:

$$۱ \times (۰,۹۹۵)^{۶۰۰} \approx ۰,۰۵.$$

البته این بدان معنا نیست که عامل حتماً کشته شده است؛ بلکه ممکن است زنده باشد، اما به دلیل حرکات بسیار زیاد، امتیازهای دریافتی او به طرز قابل توجهی منفی شده است. در این حالت، احتمالاً همگرایی نیز اتفاق افتاده است. بنابراین، برای بهبود عملکرد مدل، باید سعی کنیم تنظیمات بهتری انجام دهیم و محدودیت‌هایی برای تعداد مراحل مجاز برای عامل تعیین کنیم.

د)

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؟

پاسخ این بخش به طور کامل در بخش ب) بیان شد. در زیر خلاصه ای از آن اورده شده است و لطفا برای پاسخ دقیق به بخش ب) مراجعه بفرمایید.

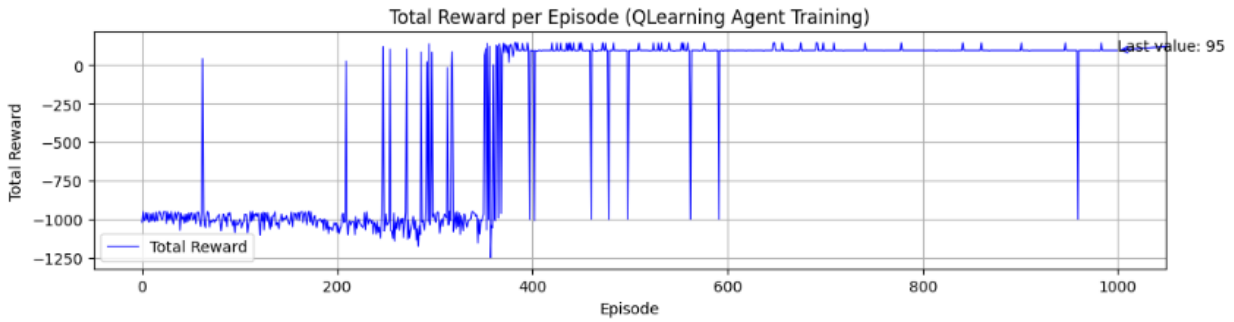
تحلیل تعداد اپیزودها برای دستیابی به عملکرد پایدار

الگوریتم Q-Learning برای دستیابی به عملکرد پایدار و مثبت به حدود ۳۰۰ اپیزود نیاز دارد، که این زمان طولانی‌تر نشان‌دهنده ضرورت کاوش بیشتر و یادگیری سیاست‌های بهینه‌تر است. با وجود این، این مدل نتوانسته است سیاست بهینه‌ای برای مسئله خاص یاد بگیرد. پس از حدود ۵۲۰ اپیزود، Q-Learning به طور کامل بازی را فراگرفته و دیگر پاداش منفی دریافت نمی‌کند.

در مقابل، الگوریتم DQN تنها به حدود ۱۰۰ اپیزود برای رسیدن به عملکرد پایدار و مثبت نیاز دارد که نشان‌دهنده سرعت بالاتر در یادگیری و شناسایی سیاست‌های بهینه است. این الگوریتم موفق به یادگیری سیاست بهینه مسئله شده است، اما همگرایی در آن به وضوح مشخص نیست. همچنین، نمی‌توان به طور دقیق گفت که پاداش‌های منفی به دلیل کشته شدن عامل یا حرکات بیش از حد آن ایجاد شده‌اند و راه‌حل‌های احتمالی برای این مشکلات نیز به تفصیل مورد بررسی قرار گرفته است.

Q-Learning، با وجود نوسانات و پاداش‌های منفی در مراحل اولیه، به تدریج به سیاست‌های بهینه دست می‌یابد، اما این فرآیند ممکن است طولانی و ناپایدار باشد و نتواند بهترین سیاست را شناسایی کند. در مقابل، DQNبا استفاده از شبکه‌های عصبی عمیق و حافظه بازپخش، توانسته است سیاست‌های بهینه را سریع‌تر و پایدارتر یاد بگیرد و به نتایج بهتری در محیط Wumpus World دست یابد. به طور کلی، DQNبه مراتب سریع‌تر و موثرتر از Q-Learning عمل می‌کند و گزینه بهتری برای مسائل پیچیده با فضای حالت بزرگ است.

نمودار نتیجه پاداش در هر اپیزود با Q learning را مشاهده می کنیم:



تحلیل کارایی مدل‌ها در دستیابی به سیاست بهینه

Q-Learning با وجود چالش‌هایی مانند نوسانات و پاداش‌های منفی در مراحل ابتدایی یادگیری، به تدریج به سیاست‌های بهینه نزدیک می‌شود. اما این روند معمولاً طولانی و ناپایدار است و ممکن است به بهترین سیاست دست نیابد. این موضوع در مقایسه و بررسی دو الگوریتم به تفصیل مورد بحث قرار گرفت.

DQN با بهره‌گیری از شبکه‌های عصبی عمیق و تکنیک حافظه بازپخش، توانسته است سیاست‌های بهینه را سریع‌تر و با ثبات بیشتری یاد بگیرد. نتایج نشان می‌دهد که نمودارهای پاداش تجمعی و میانگین پاداش حاکی از عملکرد برتر DQN در دستیابی به سیاست‌های بهینه است. این الگوریتم قادر بوده است سیاست بهینه را در زمان کمتری پیدا کند، در حالی که این دستاورد با الگوریتم دیگر امکان‌پذیر نبوده است.

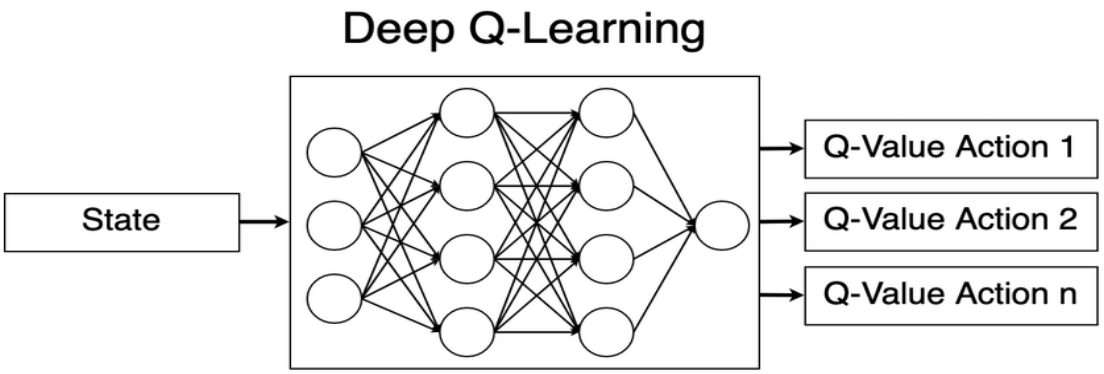
به‌طور کلی، نتایج نشان می‌دهند که DQN به مراتب سریع‌تر و پایدارتر از Q-Learning عمل می‌کند. با استفاده از شبکه‌های عصبی عمیق و روش‌های پیشرفته مانند حافظه بازپخش، DQN توانسته است سیاست‌های بهینه را در زمان کوتاه‌تری در محیط Wumpus World یاد بگیرد. این یافته‌ها حاکی از این است که در مسائل پیچیده با فضای حالت بزرگ، DQN گزینه بهتری نسبت به Q-Learning به شمار می‌رود.

---

۵)

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

در الگوریتم Deep Q-Networks (DQN)، شبکه عصبی به گونه‌ای طراحی شده است که بتواند توابع Q-value را برای محیط‌های پیچیده و دارای فضای حالت بزرگ تخمین بزند. شبکه عصبی به کار رفته در DQN، شامل لایه‌های مخفی و توابع فعال‌سازی است. در این بخش، به توضیح دقیق این ساختار و دلایل علمی انتخاب آن می‌پردازیم .ساختار کلی شبکه استفاده شده در الگوریتم DQN بصورت زیر است :



از کد پایتون زیر برای ساختن این معماری استفاده کرده ام :

```
def _build_model(self, state_shape, num_actions):
    # Build the neural network model
    model = keras.Sequential([
        keras.layers.Dense(128, activation='relu', input_shape=state_shape),
        keras.layers.Dense(128, activation='relu'),
        keras.layers.Dense(num_actions, activation=None)
    ])
    model.compile(optimizer=keras.optimizers.Adam(learning_rate=self.learning_rate),
                  loss=Huber())
    return model
```

در ادامه به توضیح این کد می پردازم :

کد فوق تابعی به نام _ build_modelرا تعریف می‌کند که وظیفه ساخت و کامپایل یک مدل شبکه عصبی با استفاده از کتابخانه Keras را بر عهده دارد. این تابع دو ورودی دارد: state_shape که شکل ورودی حالت‌ها در محیط را تعیین می‌کند و num_actions که تعداد اکشن‌های ممکن در محیط را مشخص می‌کند.

در ابتدا، مدل شبکه عصبی با استفاده از keras.Sequential ساخته می‌شود. این مدل شامل سه لایه است. لایه اول یک لایه Dense با ۱۲۸ نورون است که از تابع فعال‌سازی ReLU استفاده می‌کند و شکل ورودی آن با state_shape تعیین می‌شود. لایه دوم نیز یک لایه Dense با ۱۲۸ نورون و تابع فعال‌سازی ReLU است. لایه سوم و نهایی یک لایه Dense است که تعداد نورون‌های آن برابر با num_actionsاست و فاقد تابع فعال‌سازی می‌باشد، به این معنی که خروجی‌ها به صورت خطی ارائه می‌شوند. این لایه مقادیرQ-value را برای هر عمل ممکن در وضعیت داده شده تولید می‌کند.

پس از ساخت مدل، آن را با استفاده از متد compile کامپایل می‌کند. بهینه‌ساز مورد استفاده در اینجا Adam است که با نرخ یادگیری (learning_rate)تنظیم شده است. همچنین، از تابع هزینه هابر (Huber loss) برای ارزیابی خطا استفاده می‌شود. در نهایت، مدل ساخته شده برگردانده می‌شود.

این طراحی مدل به دلیل استفاده از لایه‌های Dense و توابع فعال‌سازی ReLU، قادر است تا ویژگی‌های پیچیده را از داده‌های ورودی استخراج کند و به خوبی با تعداد زیادی از اکشن‌های ممکن سازگار باشد. استفاده از بهینه‌ساز Adam نیز به بهبود و تسریع فرایند آموزش کمک می‌کند، در حالی که تابع هزینه هابر از تأثیر منفی نویزهای بزرگ در داده‌ها جلوگیری می‌کند.

حالا سوال اینه که دلایل انتخاب این معماری چیست ؟ سه دلیل عمده داشت که عبارت اند از : استفاده از لایه های مخفی و توابع فعال ساز ReLU ، تعداد نورون ها لایه ها و استفاده از تابع Huber به عنوان تابع هزینه.

در ادامه توضیح مختصری در این باره می دهم :

ReLUیکی از پرکاربردترین توابع فعال‌سازی در شبکه‌های عصبی عمیق است. این تابع دارای مزایای متعددی است که آن را برای استفاده در شبکه‌های عصبی مناسب می‌کند. نخست، سادگی محاسباتی این تابع باعث می‌شود که محاسبات آن نسبت به توابع پیچیده‌تری مانند سیگموئید (sigmoid) یا تانژانت هایپربولیک (tanh) سریع‌تر انجام شود. دوم، ReLUبه مشکل اشباع که در توابع سیگموئید و تانژانت هایپربولیک رخ می‌دهد، دچار نمی‌شود؛ این ویژگی می‌تواند به یادگیری سریع‌تر و بهتر منجر شود. علاوه بر این، ReLUاز مقیاس‌پذیری بهتری برخوردار است و می‌تواند اطلاعات را به طور مؤثرتری از لایه‌های پایین به لایه‌های بالاتر منتقل کند.

انتخاب ۱۲۸ نورون در هر لایه مخفی یک تعادل مناسب بین پیچیدگی مدل و قابلیت‌های محاسباتی است. این تعداد نورون به اندازه کافی بزرگ است تا ویژگی‌های پیچیده محیط را یاد بگیرد، اما نه آنقدر بزرگ که منجر به بیش‌برازش (overfitting) شود. همچنین، استفاده از دو لایه مخفی برای بسیاری از مسائل پیچیده در یادگیری تقویتی به عنوان یک نقطه شروع مناسب در نظر گرفته می‌شود. این معماری قادر است تا توابع Q-value پیچیده

را یاد بگیرد و تقریب بزند. همچنین تابع Huber یک ترکیب از خطای مطلق (mean absolute error) و خطای مربعی (mean squared error) است که به کاهش حساسیت به نقاط پرت کمک می‌کند. این ویژگی به ویژه در مسائل یادگیری تقویتی که در آن‌ها ممکن است مقادیر پاداش بسیار متفاوت باشند، مفید است. استفاده از این تابع به پایداری یادگیری کمک می‌کند و از نوسانات شدید در به‌روزرسانی وزن‌ها جلوگیری می‌کند.

پس به طور خلاصه ، استفاده از این معماری شبکه عصبی در الگوریتم DQN مزایای متعددی دارد. نخست، این شبکه عصبی قادر است ویژگی‌های پیچیده و غیرخطی محیط‌های بزرگ را یاد بگیرد و آن‌ها را به مقادیر Q-value تبدیل کند. دوم، با داشتن لایه‌های مخفی متعدد، شبکه می‌تواند روابط پیچیده بین وضعیت‌ها و اعمال را یاد بگیرد. سوم، استفاده از تابع Huber به عنوان تابع هزینه به پایداری یادگیری کمک می‌کند و از نوسانات شدید در به‌روزرسانی وزن‌ها جلوگیری می‌کند. در مجموع، معماری شبکه عصبی استفاده شده در DQN با استفاده از لایه‌های مخفی با تابع فعال‌سازی ReLU و تعداد مناسب نورون‌ها، قادر است ویژگی‌های پیچیده محیط را به خوبی یاد بگیرد. انتخاب تابع Huber نیز به پایداری و کارایی بهتر الگوریتم کمک می‌کند، و این معماری به دلیل سادگی و کارایی بالا، یکی از معماری‌های رایج در مسائل یادگیری تقویتی پیچیده مانند بازی‌های ویدیویی و شبیه‌سازی‌های پیچیده است.

```
def update_target_network(self):
    # Update target network with weights from the Q-network
    self.target_network.set_weights(self.q_network.get_weights())
```

در این الگوریتم نرخ کاوش (Exploration) نیز به تدریج کاهش می‌یابد، که در روش DQN نتایج بسیار خوبی به همراه داشته است. این رویکرد به الگوریتم کمک کرده است تا به سیاست بهینه‌ای که بیشترین امتیاز، یعنی ۱۴۵ امتیاز را دارد دست یابد. سایر قسمت‌های این الگوریتم مشابه با الگوریتم Q-learning تکرار شده‌اند.

پایان