دانشکده مهندسي برق و کامپیوتر

نام و نام خانوادگی :

امیر اسماعیل زاده نوبری

شماره دانشجویی

40101924

درس یادگیری ماشین

مینی پروژه چهارم

استاد درس:

دکتر علیاری

بهار 1403


بسم الله الرحمن الرحیم

Contents

لینک GIT

لینک google drive

سوال 1

۱ پرسش یک: حل دنیای Wumpus

شرح مساله و کد :

محیط دنیای وومپوس

مرور کلی

محیط دنیای وومپوس یک شبیه‌سازی مبتنی بر شبکه(grid) است که با استفاده از کتابخانه Gymnasium که قبلاً OpenAI Gym بود طراحی شده است. این محیط از مسئله کلاسیک هوش مصنوعی الهام گرفته است که در آن یک عامل در دنیایی حرکت می‌کند تا طلا پیدا کند و در عین حال از خطراتی مانند وومپوس و چاله‌ها اجتناب کند.

راه‌اندازی محیط

محیط با اجزای کلیدی زیر راه‌اندازی می‌شود:

- **اندازه شبکه** :اندازه پیش‌فرض شبکه 4 x4است.
- **اندازه پنجره** :اندازه پنجره PyGame به اندازه 512 *512 پیکسل تنظیم شده است.(در صورت استفاده)
- **اندازه سلول** :اندازه هر سلول در شبکه به صورت window_size // sizeمحاسبه می‌شود.
- **حالت نمایش** :می‌تواند به "human" تنظیم شود تا محیط با استفاده از PyGame به تصویر کشیده شود.

فضای مشاهده (Observation Space) :

agent: موقعیت عامل در شبکه.

wumpus: موقعیت وومپوس در شبکه.

gold: نشانگر دودویی برای حضور طلا در موقعیت عامل.

pit: نشانگر دودویی برای حضور یک چاله در موقعیت عامل.

```
# Observations include the agent's location, Wumpus's location, and binary indicators for gold and pit
self.observation_space = spaces.Dict({
    "agent": spaces.Box(0, size - 1, shape=(2,), dtype=int),
    "wumpus": spaces.Box(0, size - 1, shape=(2,), dtype=int),
    "gold": spaces.Discrete(2),  # 0 for not at agent's location, 1 for at agent's location
    "pit": spaces.Discrete(2)     # 0 for not at agent's location, 1 for at agent's location
})

```

‘observation_space’ به عنوان یک دیکشنری شامل اجزای مختلف حالت محیط تعریف شده است.

موقعیت عامل (agent) و وومپوس (Wumpus):

spaces.Box(0, size - 1, shape=(2,), dtype=int): این یک فضای باکس دو بعدی برای موقعیت agent و wumpus در شبکه تعریف می‌کند.

فضای Box پیوسته است و در اینجا برای نمایش مختصات (x, y) عامل استفاده می‌شود.

Shap=(2,) نشان می‌دهد که موقعیت یک مختصات دو بعدی است.

موقعیت طلا (gold) و چاله (pit):

spaces.Discrete(2) : این یک فضای گسسته با دو مقدار ممکن 0 یا 1 را تعریف می‌کند.

0 نشان می‌دهد که طلا یا چاله در موقعیت فعلی عامل نیست.

1 نشان می‌دهد که طلا یا چاله در موقعیت فعلی عامل است.

چون موقعیت چاله و طلا ثابت و دلخواه اند میتوان اینگونه نمایش داد .

فضای اقدام (Action space) :

اقدامات حرکتی: 0 = چپ، 1 = راست، 2 = بالا، 3 = پایین.

اقدامات شلیک: 4 = شلیک به چپ ، 5 = شلیک به راست، 6 = شلیک به بالا، 7 = شلیک به پایین.

```
# Action space: 0 = left, 1 = right, 2 = up, 3 = down, 4 = shoot left, 5 = shoot right, 6 = shoot up, 7 = shoot down
self.action_space = spaces.Discrete(8)

```

موقعیت اجزا به صورت دلخواه به صورت زیر اعمال شده :

```
# 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

```

: reset

موقعیت‌های agent و wumpus را راه‌اندازی می‌کند و موقعیت‌های طلا و چاله را تنظیم می‌کند. همچنین پاداش عامل را بازنشانی می‌کند.

```
def reset(self):
    self.agent_pos = self.agent_start_pos.copy()
    self.wumpus_pos = np.array([0, 3])
    self.wumpus_alive = True
    self.total_reward = 0
    return self._get_obs()

```

: Step

Step اقدامات Agent را پردازش می‌کند که شامل حرکت و شلیک است. عملکردهای کلیدی شامل:

- **حرکت** :عامل در صورت معتبر بودن حرکت در جهت مشخص حرکت می‌کند.

معتبر بودن حرکت به معنی این است که agent یا Wumpus نباید از چهارچوب تعریفی یا grid خارج شوند.

- **شلیک** :عامل می‌تواند در جهت مشخص شلیک کند تا وومپوس را در صورت قرار داشتن در خط شلیک بکشد.(امتیازی).
- **حرکت وومپوس** :در صورت زنده بودن، وومپوس به صورت تصادفی پس از اقدام عامل حرکت می‌کند.

```
def step(self, action):
    reward = 0  # Initial reward
    done = False

    if action < 4:  # Movement actions
        if self._valid_move(self.agent_pos, action):
            self._move(self.agent_pos, action)
            reward = -1  # Movement penalty
    elif self.wumpus_alive:  # Shoot actions
        if self._shoot(action):
            reward = 50
```

پاداش‌ها:(فعلا پاداش های خواسته سوال گذاشته شده است.)

- حرکت کردن شامل جریمه 1- است.
- پیدا کردن طلا پاداش 100+ دارد.
- افتادن در چاله یا خورده شدن توسط wumpus شامل جریمه 1000- است.
- شلیک موفق به وومپوس پاداش 50+ دارد(امتیازی).

```
# Check for terminal states
if np.array_equal(self.agent_pos, self.gold_pos):
    reward = +100  # Reward for finding gold
    done = True
elif np.array_equal(self.agent_pos, self.pit_pos):
    reward = -1000  # Penalty for falling into pit
    done = True
elif np.array_equal(self.agent_pos, self.wumpus_pos) and self.wumpus_alive:
    reward = -1000  # Penalty for getting eaten by the Wumpus
    done = True

# Move the Wumpus if it is still alive
if not done and self.wumpus_alive:
    wumpus_action = random.choice([0, 1, 2, 3])
    if self._valid_move(self.wumpus_pos, wumpus_action):
        self._move(self.wumpus_pos, wumpus_action)

    # Check if the Wumpus eats the agent after moving
    if np.array_equal(self.agent_pos, self.wumpus_pos) and self.wumpus_alive:
        reward = -1000  # Penalty for getting eaten by the Wumpus
        done = True
```

- **move_**: موقعیت عامل یا وومپوس را بر اساس اقدام به‌روزرسانی می‌کند.

```
def _move(self, pos, action):
    if action == 0 and pos[1] > 0:  # left
        pos[1] -= 1
    elif action == 1 and pos[1] < self.size - 1:  # right
        pos[1] += 1
    elif action == 2 and pos[0] > 0:  # up
        pos[0] -= 1
    elif action == 3 and pos[0] < self.size - 1:  # down
        pos[0] += 1
```

- **valid_move_**: بررسی می‌کند که آیا اقدام حرکتی معتبر است یا خیر.

```
def _valid_move(self, pos, action):
    if action == 0 and pos[1] == 0:  # left
        return False
    elif action == 1 and pos[1] == self.size - 1:  # right
        return False
    elif action == 2 and pos[0] == 0:  # up
        return False
    elif action == 3 and pos[0] == self.size - 1:  # down
        return False
    return True
```

- **shoot**_: تعیین می‌کند که آیا وومپوس توسط اقدام شلیک مورد اصابت قرار گرفته است یا خیر.

```
def _shoot(self, action):
    if not self.wumpus_alive:
        return False

    if action == 4:  # shoot left
        if self.agent_pos[0] == self.wumpus_pos[0] and self.agent_pos[1] > self.wumpus_pos[1]:
            self.wumpus_alive = False
            return True
    elif action == 5:  # shoot right
        if self.agent_pos[0] == self.wumpus_pos[0] and self.agent_pos[1] < self.wumpus_pos[1]:
            self.wumpus_alive = False
            return True
    elif action == 6:  # shoot up
        if self.agent_pos[1] == self.wumpus_pos[1] and self.agent_pos[0] > self.wumpus_pos[0]:
            self.wumpus_alive = False
            return True
    elif action == 7:  # shoot down
        if self.agent_pos[1] == self.wumpus_pos[1] and self.agent_pos[0] < self.wumpus_pos[0]:
            self.wumpus_alive = False
            return True
    return False
```

- **get_obs**_: مشاهده فعلی محیط را ایجاد می‌کند.

```
def _get_obs(self):
    return {
        "agent": self.agent_pos.copy(),
        "wumpus": self.wumpus_pos.copy(),
        "gold": int(np.array_equal(self.agent_pos, self.gold_pos)),
        "pit": int(np.array_equal(self.agent_pos, self.pit_pos))
    }
```

الگوریتم Q-learning

مرور کلی

Q-Learning یک الگوریتم یادگیری تقویتی بدون ناظر است که به عامل کمک می‌کند تا یاد بگیرد چگونه در یک محیط بهینه عمل کند. این الگوریتم به عامل این امکان را می‌دهد که سیاست بهینه‌ای را برای حداکثر کردن مجموع پاداش‌های آینده یاد بگیرد.

عناصر کلیدی

جدول Q (Q-Table):

یک ماتریس که مقادیر Q را برای هر جفت حالت و اقدام ذخیره می‌کند.

مقدار Q نشان‌دهنده ارزش مورد انتظار از انجام یک اقدام در یک حالت خاص است.

پاداش (Reward):

پاداشی که عامل پس از انجام یک اقدام در یک حالت خاص دریافت می‌کند.

هدف عامل حداکثر کردن مجموع پاداش‌های دریافتی است.

نرخ یادگیری (Learning Rate, α):

یک مقدار بین 0 و 1 که تعیین می‌کند چقدر عامل مقادیر Q را به روز رسانی می‌کند.

مقادیر بالاتر به‌روزرسانی‌های سریع‌تر ولی ناپایدارتر را منجر می‌شود.

فاکتور تخفیف (Discount Factor, γ):

یک مقدار بین 0 و 1 که اهمیت پاداش‌های آینده را تعیین می‌کند.

مقادیر نزدیک به 1 نشان می‌دهد که عامل پاداش‌های آینده را بیشتر در نظر می‌گیرد.

سیاست ε-greedy:

سیاستی که ترکیبی از اکتشاف و بهره‌برداری را فراهم می‌کند.

با احتمال ε یک اقدام تصادفی انتخاب می‌شود (اکتشاف) و با احتمال 1-ε بهترین اقدام بر اساس جدول Q انتخاب می‌شود (بهره‌برداری).

مراحل الگوریتم

مقداردهی اولیه جدول Q:

جدول Q با مقادیر صفر مقداردهی اولیه شده است که نمایانگر تمامی حالات و اقدامات ممکن است.

انتخاب اقدام:

اقدامی بر اساس سیاست epsilon-greedy انتخاب می‌شود. با احتمال epsilon، یک اقدام معتبر تصادفی برای اطمینان از اکتشاف انتخاب می‌شود. در غیر این صورت، بهترین اقدام بر اساس جدول Q انتخاب می‌شود.

```
if np.random.rand() < epsilon:
    # Sample a valid action
    action = np.random.choice(valid_actions)
else:
    # Choose the best valid action
    q_values = q_table[state]
    sorted_actions = np.argsort(q_values)[::-1]
    action = None
```

اجرای اقدام و به‌روزرسانی جدول Q:

اقدام انتخاب شده اجرا می‌شود و حالت نتیجه، پاداش و پرچم اتمام دریافت می‌شود.

جدول Q با استفاده از قانون به‌روزرسانی Q-learning به‌روزرسانی می‌شود:

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

```
best_next_action = np.argmax(q_table[next_state])
        q_table[state + (action,)] += alpha * (reward + gamma * q_table[next_state + (best_next_action,)] - q_table[state + (action,)])
```

حالت سپس به حالت بعدی به‌روزرسانی می‌شود.

```
obs, reward, done, _ = env.step(action)
next_state = get_state(obs)

total_reward += reward
```

کاهش نرخ اکتشاف:

نرخ اکتشاف epsilon به تدریج کاهش می‌یابد تا بهره‌برداری را نسبت به اکتشاف ترجیح دهد.

```
while not done and steps < max_steps:
    valid_actions = get_valid_actions(env)

    if np.random.rand() < epsilon:
        # Sample a valid action
        action = np.random.choice(valid_actions)
    else:
        # Choose the best valid action
        q_values = q_table[state]
        sorted_actions = np.argsort(q_values)[::-1]
        action = None
        for a in sorted_actions:
            if a in valid_actions:
                action = a
                break
```

الگوریتم Deep Q-Network (DQN)

DQN یک توسعه از الگوریتم Q-learning است که از شبکه‌های عصبی عمیق برای تقریب تابع Q-value استفاده می‌کند و به عامل اجازه می‌دهد تا در فضاهای حالت با ابعاد بالا یاد بگیرد و تصمیم‌گیری کند.

اجزای کلیدی

1. کلاس شبکه عصبی عمیق (DQN):

- یک مدل شبکه عصبی با سه لایه کامل (Fully Connected)با تابع فعال ساز relu, تعداد نرون 100، 50، وبعد اکشن تعریف می‌کنیم.

- ورودی: ابعاد حالت.

- خروجی: مقادیر Q برای هر اقدام.

```python
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_dim, 50)
        self.fc2 = nn.Linear(50, 20)
        self.fc3 = nn.Linear(20, action_dim)

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

2. کلاس Replay Buffer:

- داده های گذشته را برای آموزش ذخیره می‌کند.

- یک بافر دک با ظرفیت ثابت برای ذخیره تجارب `(state, action, reward, next_state, done)` است.

- به عامل اجازه می‌دهد تا همبستگی‌های بین داده های متوالی را بشکند و آموزش پایدارتری انجام دهد.

- روش‌هایی برای افزودن تجارب جدید و نمونه‌گیری مینی‌بچ‌ها برای آموزش فراهم می‌کند.

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        state, action, reward, next_state, done = zip(*random.sample(self.buffer, batch_size))
        return np.array(state), action, reward, np.array(next_state), done

    def __len__(self):
        return len(self.buffer)
```

3. کلاس DQNAgent:

- فرآیندهای آموزش و انتخاب اقدام را مدیریت می‌کند.
- شامل شبکه‌های policy و هدف، بهینه‌ساز و Replay Bufferاست.
- اقدامات را بر اساس سیاست epsilon-greedy انتخاب می‌کند.

```
class DQNAgent:
    def __init__(self, state_dim, action_dim, lr=1e-3, gamma=0.9, epsilon=1.0, epsilon_decay=0.995, epsilon_min=0.01):
```

4. تابع آموزش (train_dqn):

- حلقه آموزش را در طول چندین قسمت اجرا می‌کند.
- در هر گام، عامل یک اقدام را انتخاب می‌کند، پاداش دریافت می‌کند و به حالت بعدی منتقل می‌شود.
- تجربه در بافر حافظه ذخیره می‌شود و برای آموزش شبکه policy استفاده می‌شود.
- پاداش کل برای هر قسمت ثبت می‌شود.
- شبکه هدف را به صورت دوره‌ای به‌روزرسانی می‌کند.

5. آموزش و تجسم:

- محیط دنیای وومپوس و عامل DQN را مقداردهی اولیه می‌کند.
- عامل را در طول 1000 قسمت آموزش می‌دهد.
- تاریخچه پاداش و میانگین متحرک را برای تجسم بهبود عملکرد عامل رسم می‌کند.

شبیه سازی

حالت اول: خواسته مساله

ابتدا با خواسته ی مساله با REWARD های زیر Enviroment و agent را تعریف کرده ایم :

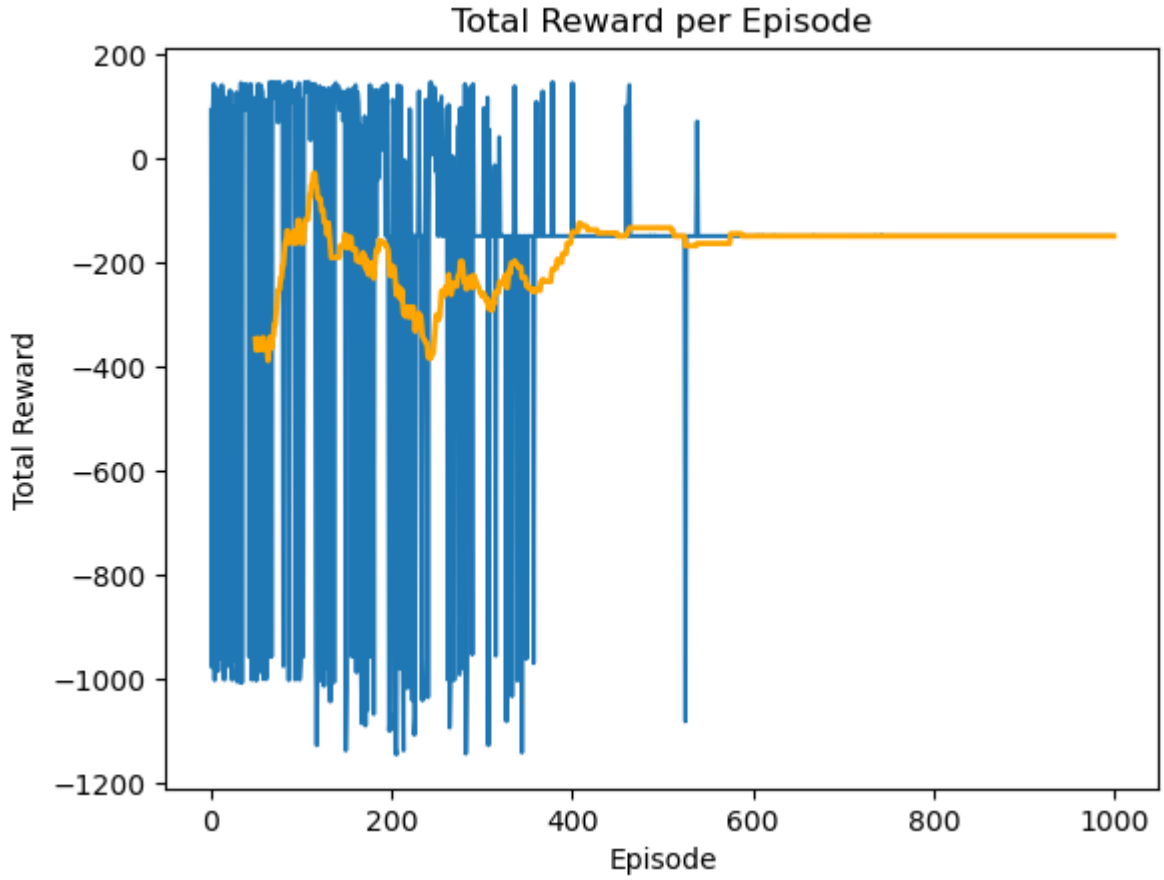
$$Rewards: \begin{cases} gold: & +100 \\ killing\ Wumpus: & +50 \\ wumpus\ and\ pit\ punishment: & -1000 \\ step\ punishment: & -1 \end{cases}$$

سپس با پارامتر های زیر برایq-learning و DQN مدل را آموزش می دهیم:

$$Q-learning\ \&\ DQN: parameters$$

$$: \begin{cases} \alpha = 0.1 & Q-learning\ learning\ rate \\ lr = 0.001 & Deep\ Q\ learning\ rate \\ \gamma = 0.9 & discount\ factor \\ \epsilon\ max = 1 & \max of\ epsilon \\ \epsilon\ min = 0.01 & \min of\ epsilon \\ \epsilon\ decay = 0.995 & epsilon\ decay \\ episode = 1000 & number\ of\ episodes \\ \max step = 200 & \max step\ for\ each\ episode \end{cases}$$

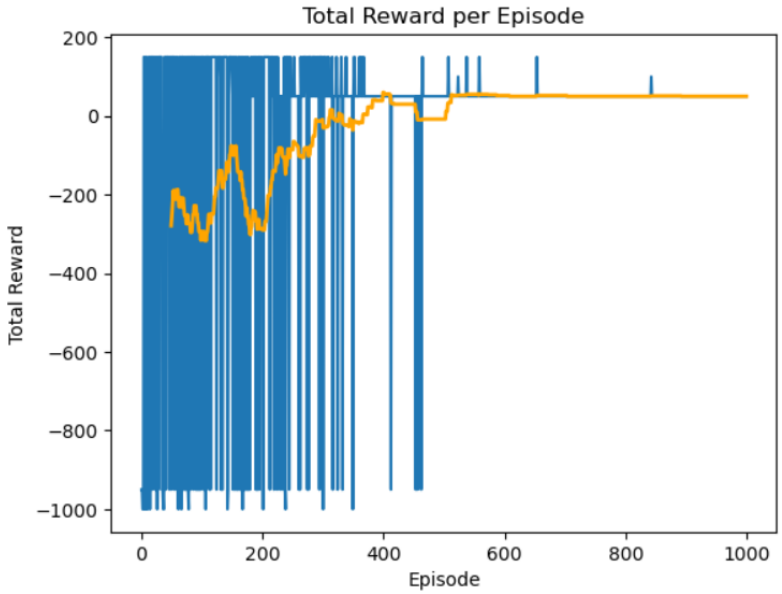
Q-learning




```
Episode 999, Step 186: Agent Position = [1 0], Wumpus Position = [0 3], Action = 3, Reward = -1
Episode 999, Step 187: Agent Position = [0 0], Wumpus Position = [0 3], Action = 2, Reward = -1
Episode 999, Step 188: Agent Position = [1 0], Wumpus Position = [0 3], Action = 3, Reward = -1
Episode 999, Step 189: Agent Position = [0 0], Wumpus Position = [0 3], Action = 2, Reward = -1
Episode 999, Step 190: Agent Position = [1 0], Wumpus Position = [0 3], Action = 3, Reward = -1
Episode 999, Step 191: Agent Position = [0 0], Wumpus Position = [0 3], Action = 2, Reward = -1
Episode 999, Step 192: Agent Position = [0 1], Wumpus Position = [0 3], Action = 1, Reward = -1
Episode 999, Step 193: Agent Position = [0 0], Wumpus Position = [0 3], Action = 0, Reward = -1
```

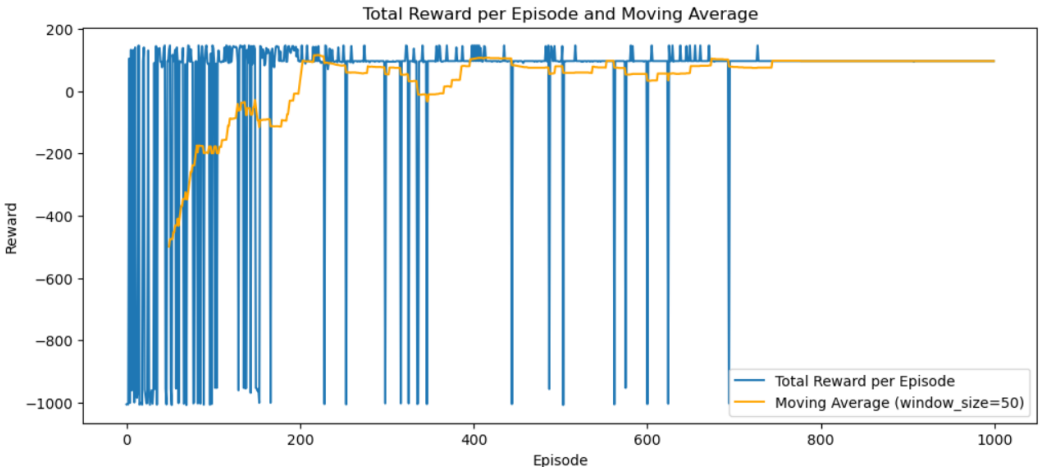
همانطور که دیده می شود agent میتواند یاد بگیرد که wumpus را بکشد و از افتادن در چاله و یا خرده شدن اجتناب کند ولی نمیتواند به خوبی طلا را پیدا کند. این میتواند به دلیل عدم تقسیم درست پاداش reward باشد .چون تعداد step های زده شده با جریمه1- ارجعیت به پاداش کمی به اندازه +100 دارند . برای رفع این مشکل در حال های بعد راه حل های متفاوتی اندیشیده ایم.

reward 1- هر قدم را صفر میگذاریم می بینیم:



مشاهده می شود که هنوز نمودار مطلوب نیست .

DQN

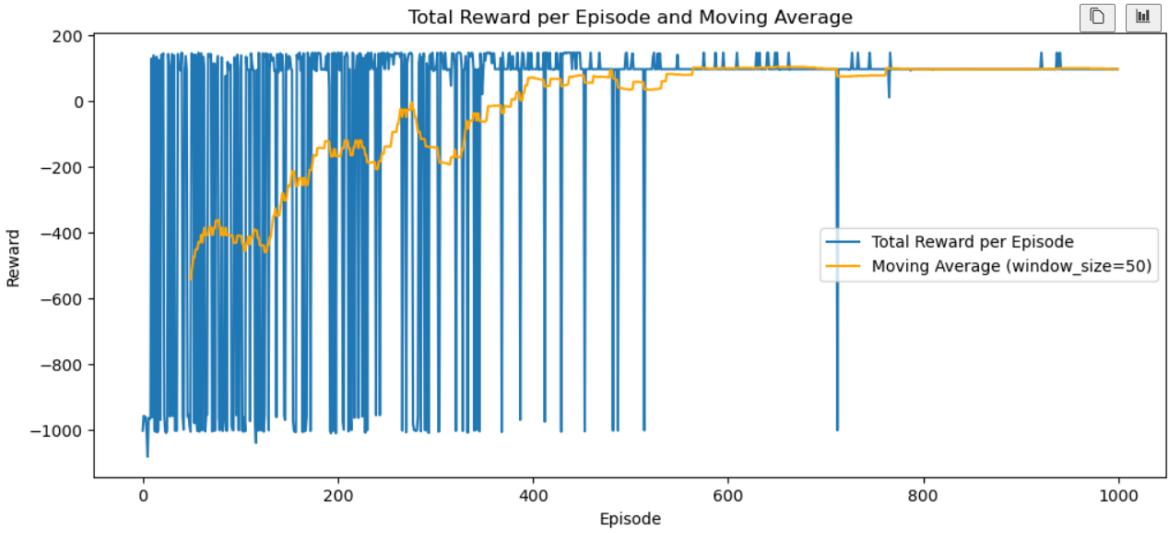



```
Episode 0, Total Reward: -1006, Epsilon: 0.995
Episode 1, Total Reward: -1004, Epsilon: 0.990025
Episode 2, Total Reward: -1006, Epsilon: 0.985074875
Episode 3, Total Reward: 105, Epsilon: 0.9801495006250001
Episode 4, Total Reward: -1001, Epsilon: 0.9752487531218751
Episode 5, Total Reward: 133, Epsilon: 0.9703725093562657
Episode 6, Total Reward: -961, Epsilon: 0.9655206468094844
Episode 7, Total Reward: -959, Epsilon: 0.960693043575437
Episode 8, Total Reward: 133, Epsilon: 0.9558895783575597
Episode 9, Total Reward: -1000, Epsilon: 0.9511101304657719
Episode 10, Total Reward: 141, Epsilon: 0.946354579813443
Episode 11, Total Reward: -955, Epsilon: 0.9416228069143757
Episode 12, Total Reward: -985, Epsilon: 0.9369146928798039
Episode 13, Total Reward: -963, Epsilon: 0.932230119415404
```

```
Episode 982, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 983, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 984, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 985, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 986, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 987, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 988, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 989, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 990, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 991, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 992, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 993, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 994, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 995, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 996, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 997, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 998, Total Reward: 97, Epsilon: 0.00998645168764533
Episode 999, Total Reward: 97, Epsilon: 0.00998645168764533
```

همانطور که مشاهده می شود deep q network به عدد 97+ همگرا می شود که عملکرد بسیار مطلوب ونشان از پیدا کردن طلا است .همچنان کم شدن نرخ اپسیلون در بالا نمایان است.

با تغییر مکان چاله به [2,3]دوباره تست را انجام می دهیم(برای تصویر کردن قسمت بعد):



ویدیویی از تست عملکرد مدل ترین شده با DQN با episod 10 تهیه شده:

LINK1 , LINK2

می توانید با تغییر مکان ها به انتخاب خود مشاهدات دیگری نیز بکنید.

Agent مهره آبی و وومپوس مهره قرمز می باشد

دلایل عملکرد بهتر DQN

استفاده از شبکه‌های عصبی در DQN:

DQN از شبکه‌های عصبی برای تقریب تابع Q استفاده می‌کند. این به عامل اجازه می‌دهد تا روابط پیچیده بین حالات و اقدامات را یاد بگیرد، که برای محیط‌های با ابعاد بالا و پیچیده مانند دنیای وومپوس ضروری است.

شبکه عصبی می‌تواند الگوهای پیچیده‌ای را از داده‌های ورودی بیاموزد و به عامل کمک کند تا تصمیمات بهتری بگیرد.

بازپخش تجربه (Experience Replay):

در DQN، عامل از یک بافر بازپخش تجربه برای ذخیره و نمونه‌گیری از تجارب گذشته استفاده می‌کند. این تکنیک به شکستن همبستگی‌های بین تجارب متوالی کمک می‌کند و آموزش پایدارتری فراهم می‌کند.

بازپخش تجربه به عامل اجازه می‌دهد تا چندین بار از تجارب قبلی بیاموزد، که بهره‌وری داده‌ها را افزایش می‌دهد و آموزش را بهبود می‌بخشد.

به‌روزرسانی شبکه هدف (Target Network):

DQN از دو شبکه جداگانه استفاده می‌کند: شبکه سیاست و شبکه هدف. شبکه هدف به صورت دوره‌ای با شبکه سیاست همگام‌سازی می‌شود، که باعث پایداری بیشتر در به‌روزرسانی‌های Q می‌شود.

این تکنیک از نوسانات شدید در به‌روزرسانی‌های Q جلوگیری می‌کند و آموزش پایدارتری فراهم می‌کند.

توانایی تعمیم:

DQN به دلیل استفاده از شبکه‌های عصبی، توانایی تعمیم‌دهی به حالات و اقدامات دیده نشده را دارد. این به عامل اجازه می‌دهد تا حتی در شرایطی که مستقیماً آموزش ندیده است، تصمیمات بهتری بگیرد.

در مقابل، Q-Learning از یک جدول Q استفاده می‌کند که به سختی می‌تواند به حالات دیده نشده تعمیم دهد، به ویژه در محیط‌های با فضای حالت بزرگ.

حالت دوم:افزایش پاداش یافتن طلا

برای بر طرف کردن مشکل Q-learning همانطور که گفته شد یکی از راه ها افزایش پاداش یافتن طلا است← پاداش گرفتن طلا رابه +1000 افزایش می دهیم

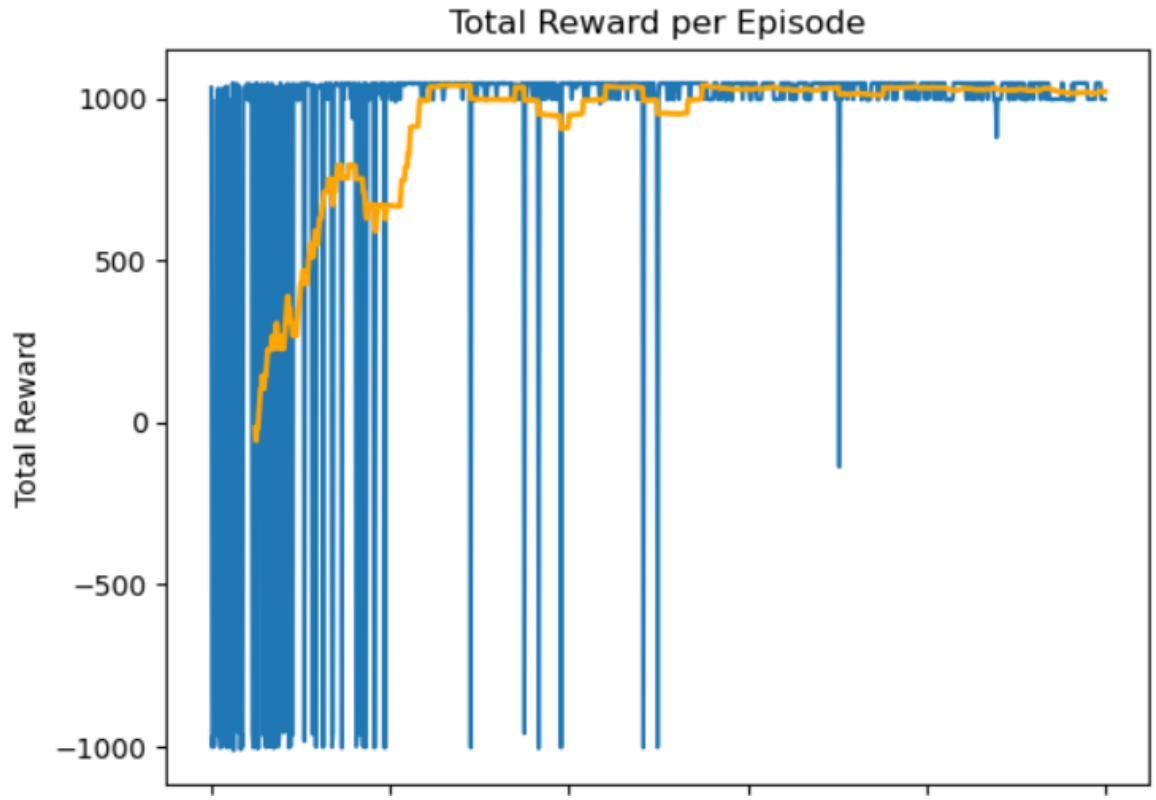
$$Rewards: \begin{cases} gold: & +1000 \\ killing\ Wumpus: & +50 \\ wumpus\ and\ pit\ punishment: & -1000 \\ step\ punishment: & -1 \end{cases}$$

سپس با پارامتر های زیر برایq-learning و DQN مدل را آموزش می دهیم:

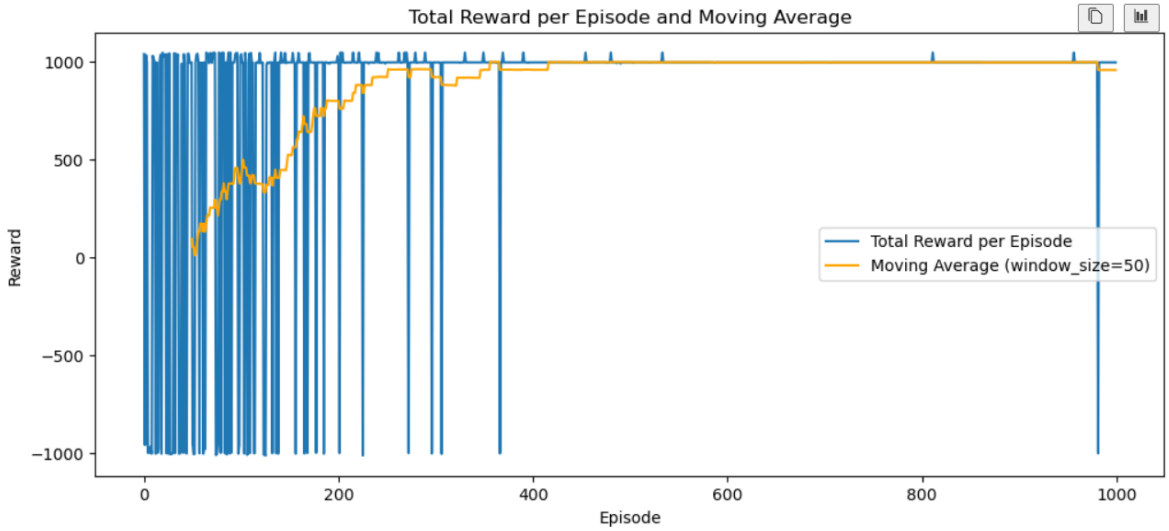
$$Q-learning\ \&\ DQN: parameters$$

$$: \begin{cases} \alpha = 0.1 & Q-learning\ learning\ rate \\ lr = 0.001 & Deep\ Q\ learning\ rate \\ \gamma = 0.5 & discount\ factor \\ \epsilon\ max = 1 & \max of\ epsilon \\ \epsilon\ min = 0.01 & \min of\ epsilon \\ \epsilon\ decay = 0.995 & epsilon\ decay \\ episode = 1000 & number\ of\ episodes \\ \max step = 200 & \max step\ for\ each\ episode \end{cases}$$

Q-learning



DQN



مشاهده می شود که حال مشکل پیدا کردن طلا و converge یا همگرا شدن الگوریتم-Q learning حل شد.لازم به ذکر هست که در این قسمت وقسمت های بعدی گاما(ضریب تخفیف) 0.5 در نظر گرفته شده (جواب داد) : این به این معنا است که agentبرای پاداش آنی ارزش

بیشتری (نسبت به 0.9) از پاداش لحظات بعد قاعل است که دلیل آن میتواند وجود عدم قطعیت در محیط (حرکت رندوم WUMPUS) باشد.

حالت سوم: Fixed Wumpus

Wumpus را ثابت درنظر میگیریم :

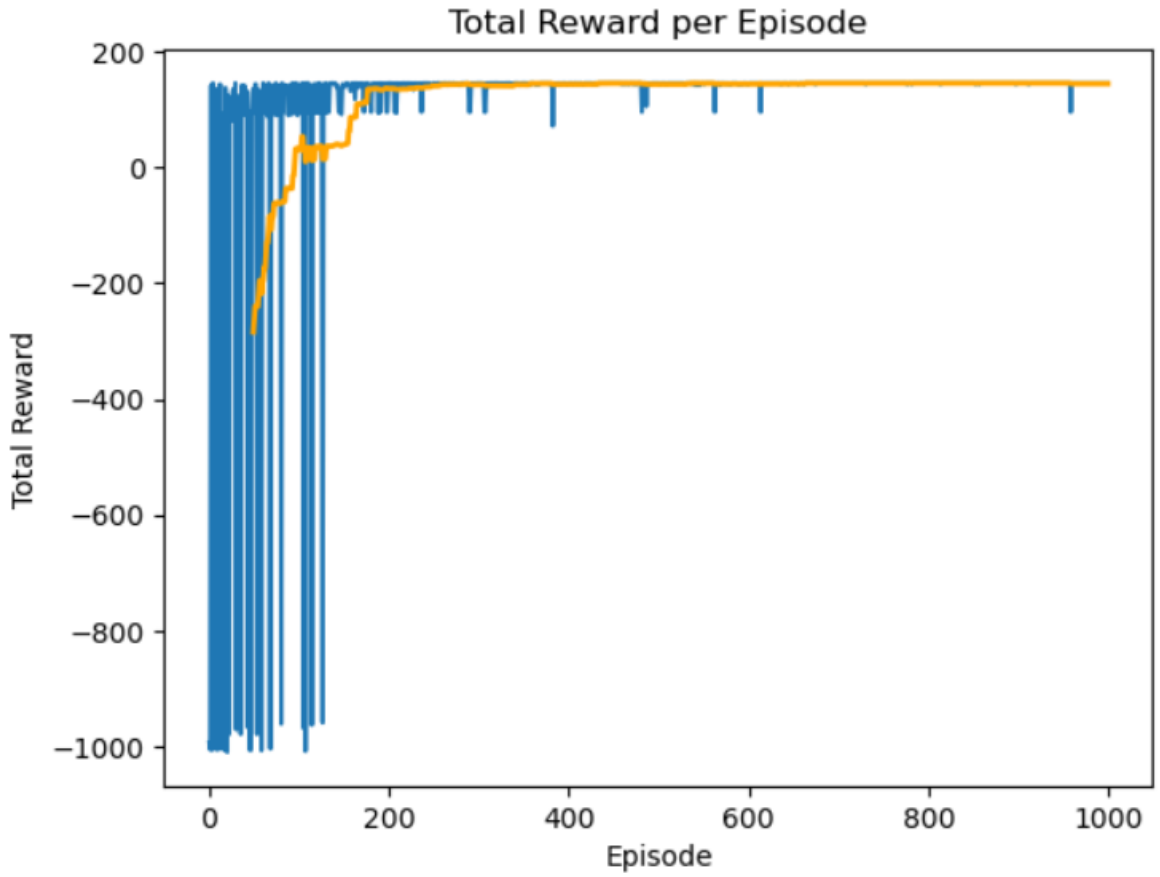
$$Rewards: \begin{cases} gold: & +100 \\ killing\ Wumpus: & +50 \\ wumpus\ and\ pit\ punishment: & -1000 \\ step\ punishment: & -1 \end{cases}$$

$$Q-learning\ \&\ DQN: parameters$$

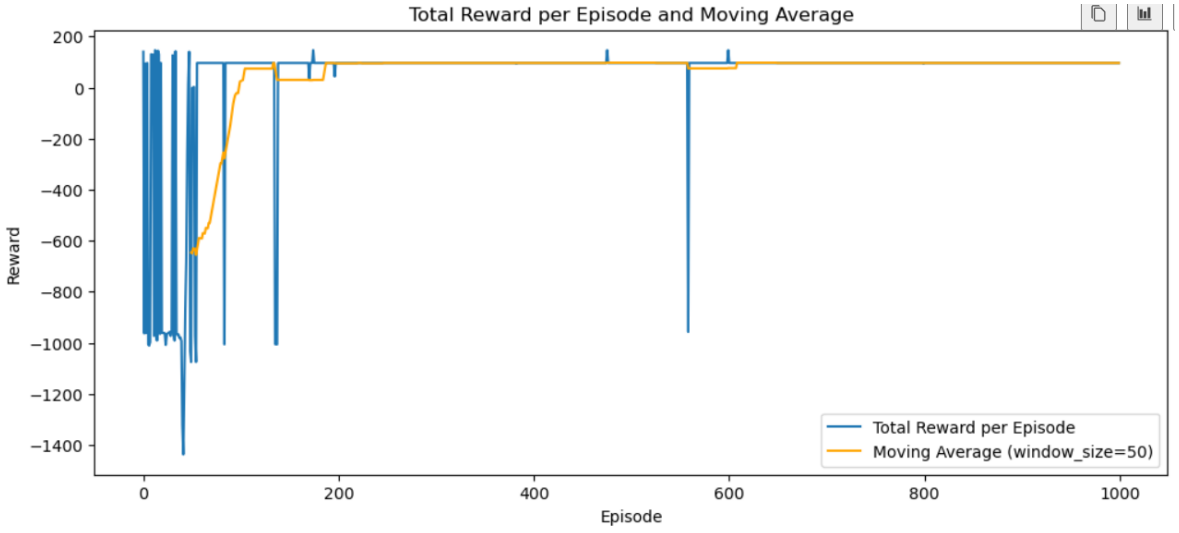
$$: \begin{cases} \alpha = 0.1 & Q-learning\ learning\ rate \\ lr = 0.001 & Deep\ Q\ learning\ rate \\ \gamma = 0.9 & discount\ factor \\ \epsilon\ max = 1 & \max of\ epsilon \\ \epsilon\ min = 0.01 & \min of\ epsilon \\ \epsilon\ decay = 0.995 & epsilon\ decay \\ episode = 1000 & number\ of\ episodes \\ \max step = 200 & \max step\ for\ each\ episode \end{cases}$$

یکی دیگر از روش های تصحیح Q-learning ثابت فرض کردن Wumpus است که فرض سوال بدون امتیازی بود. اینکار از پیچیدگی محیط کم می کند و میتوان زودتر و بهتر به نتیجه مطلوب رسید.

Q-learning



DQN



لینک ویدیو تست:

LINK

حالت چهارم

برای ردیابی بهتر طلا reward ای برای نزدیک شدن و دور شدن از طلا تعریف می کنیم:

$$Rewards: \begin{cases} stepping\ closer\ to\ gold\ : & +1 \\ stepping\ away\ from\ gold\ : & -1 \\ gold\ : & +100 \\ killing\ Wumpus\ : & +50 \\ wumpus\ and\ pit\ punishment\ : & -1000 \\ step\ punishment\ : & -1 \end{cases}$$

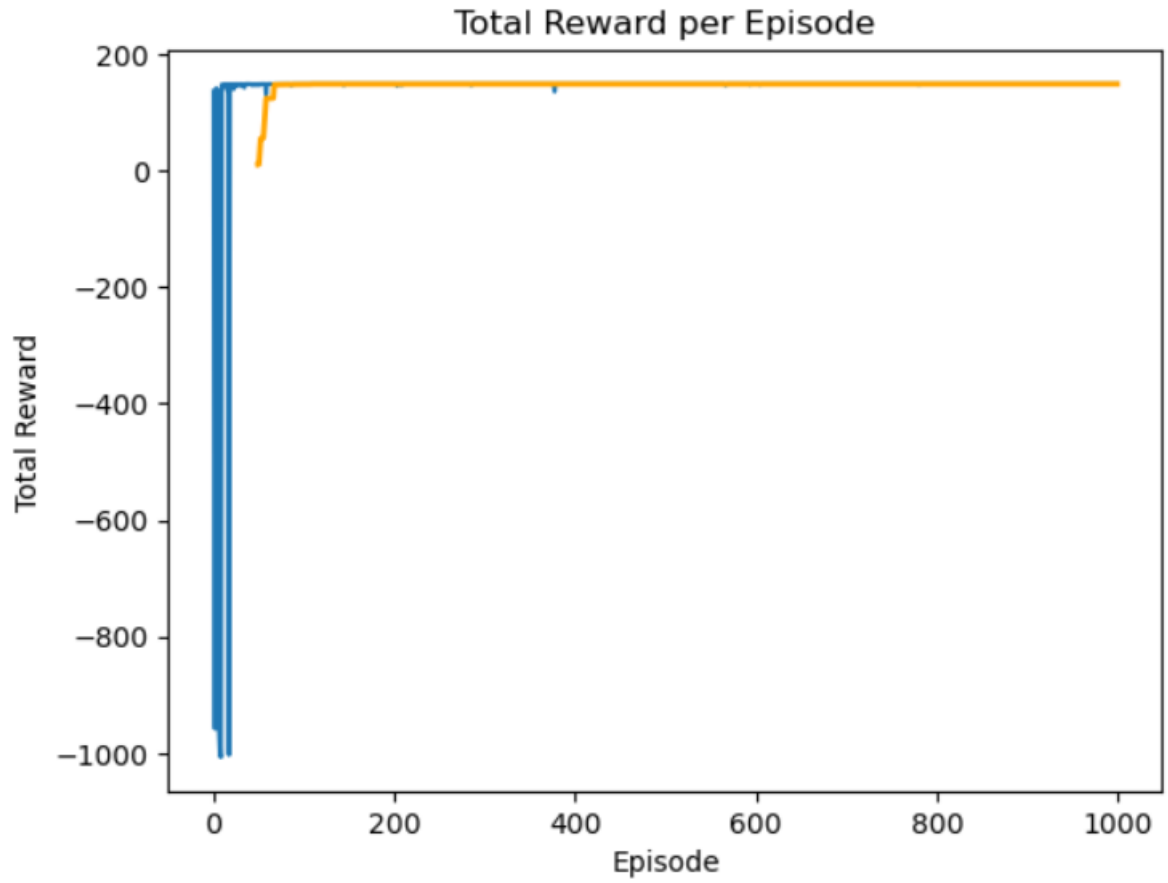
سپس با پارامتر های زیر برایq-learning و DQN مدل را آموزش می دهیم:

$$Q-learning\ \&\ DQN: parameters$$

$$: \begin{cases} \alpha = 0.1 & Q-learning\ learning\ rate \\ lr = 0.001 & Deep\ Q\ learning\ rate \\ \gamma = 0.5 & discount\ factor \\ \epsilon\ max = 1 & \max of\ epsilon \\ \epsilon\ min = 0.01 & \min of\ epsilon \\ \epsilon\ decay = 0.995 & epsilon\ decay \\ episode = 1000 & number\ of\ episodes \\ \max step = 200 & \max step\ for\ each\ episode \end{cases}$$

در این روش پاداشی (+1) برای نزدیک شدن به طلا و جریمه ای (-1) برای دور شدن از طلا در نظر گرفته شده تا agent را تشویق به پیدا کردن طلا بکند.

Q-learning



DQN

سوالات:

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.

در بالا آورده شد.

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؟

پاداش تجمعی در بالا تصویر شد:

بهبود عملکرد عامل در طول زمان در Q-Learning و DQN

Q-Learning

Q-Learning یک الگوریتم یادگیری تقویتی بدون مدل است که هدف آن یادگیری تابع Q است، که مقدار پاداش مورد انتظار را برای هر حالت و اقدام در آن حالت برآورد می‌کند. بهبود عملکرد عامل در Q-Learning به تدریج و از طریق تکرار زیر صورت می‌گیرد:

1. تکرار تجربه:

- عامل در محیط حرکت می‌کند و اقداماتی را انتخاب می‌کند.
- برای هر اقدام، پاداشی دریافت می‌کند و به حالت بعدی منتقل می‌شود.
- این تجربه به صورت `(state, action, reward, next_state)` ثبت می‌شود.

2. به‌روزرسانی تابع Q:

- مقدار Q برای حالت و اقدام فعلی با استفاده از معادله بلمن به‌روزرسانی Q-Learning به‌روزرسانی می‌شود

3. سیاست اکتشافی:

- برای بهبود سیاست، عامل از سیاست epsilon-greedy استفاده می‌کند که در آن با احتمال (epsilon) یک اقدام تصادفی (اکتشاف) و با احتمال (-epsilon1) بهترین اقدام شناخته‌شده (بهره‌برداری) را انتخاب می‌کند.

- با گذشت زمان، (epsilon) به تدریج کاهش می‌یابد تا عامل بیشتر به بهره‌برداری از دانش فعلی بپردازد.

4. حلقه آموزش:

- این فرآیند در طول چندین قسمت تکرار می‌شود، که هر قسمت شامل چندین مرحله است.

- با تکرار تجربه‌ها و به‌روزرسانی Q-جدول، عامل به تدریج سیاست بهینه را یاد می‌گیرد و عملکرد خود را بهبود می‌بخشد.

Deep Q-Network (DQN)

DQN یک توسعه از Q-Learning است که از شبکه‌های عصبی برای تقریب تابع Q استفاده می‌کند. بهبود عملکرد عامل در DQN به تدریج و از طریق تکرار زیر صورت می‌گیرد:

1. شبکه عصبی (Policy Network:

- یک شبکه عصبی برای تقریب تابع Q-جدول استفاده می‌شود.

- ورودی شبکه حالت محیط و خروجی آن مقادیر Q برای هر اقدام است.

2. بافر تجربه (Replay Buffer):

- تجارب عامل در طول زمان در یک بافر تجربه ذخیره می‌شود.

- این بافر تجارب را به صورت تصادفی نمونه‌برداری می‌کند تا همبستگی بین تجارب متوالی شکسته شود و آموزش پایدارتری صورت گیرد.

3. به‌روزرسانی شبکه عصبی:

- با استفاده از نمونه‌های تصادفی از بافر تجربه، شبکه عصبی با استفاده از الگوریتم‌های بهینه‌سازی مانند Adam به‌روزرسانی می‌شود.

- تابع هزینه معمولاً میانگین مربعات خطا (MSE) بین مقدار Q پیش‌بینی‌شده و مقدار Q هدف است

4. شبکه هدف (Target Network):

- یک شبکه عصبی دوم به عنوان شبکه هدف استفاده می‌شود.

- پارامترهای این شبکه به صورت دوره‌ای از شبکه سیاست کپی می‌شود تا پایداری آموزش افزایش یابد.

5. سیاست اکتشافی:

- عامل از سیاست epsilon-greedy استفاده می‌کند و مقدار (epsilon) به تدریج کاهش می‌یابد تا عامل بیشتر به بهره‌برداری از سیاست فعلی بپردازد.

6. حلقه آموزش:

- این فرآیند در طول چندین قسمت تکرار می‌شود و در هر قسمت عامل در محیط حرکت کرده و تجارب جدید جمع‌آوری می‌کند.

- با به‌روزرسانی مداوم شبکه عصبی و کاهش (epsilon) ، عامل به تدریج سیاست بهینه را یاد می‌گیرد و عملکرد خود را بهبود می‌بخشد.

جمع‌بندی

در هر دو الگوریتم Q-Learning و DQN، عملکرد عامل از طریق تکرار تجربه‌ها، به‌روزرسانی مداوم تابع Q و استفاده از سیاست‌های اکتشافی بهبود می‌یابد. در DQN، استفاده از شبکه‌های عصبی و بافر تجربه، آموزش پایدارتری فراهم می‌کند که به عملکرد بهتر عامل در محیط‌های پیچیده‌تر منجر می‌شود.

عملکرد دو الگوریتم در بالا مقایسه شدو دیده شد با طرح اصلی سوال الگوریتم DQN با دلایل ذگر شده جواب بهتری می دهد.

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؟

اینهم به همراه کد در بالا آورده شد ولی به طور خلاصه:

فرمول exploitation-exploration با استفاده از (Epsilon-Greedy):

در این روش، عامل با احتمال (epsilon) یک عمل تصادفی انتخاب می‌کند (اکتشاف) و با احتمال (epsilon-1) بهترین عمل را براساس Q-جدول فعلی خود انتخاب می‌کند (بهره‌برداری):

$$action = \begin{cases} if\ rand() < \epsilon \rightarrow\ Random\ Action \\ otherwise \quad\ \rightarrow\ argmax_a Q(s,a) \end{cases}$$

کاهش اپسیلون در طول زمان:

برای اطمینان از این که عامل به تدریج از اکتشاف به سمت بهره‌برداری حرکت می‌کند، اپسیلون به تدریج کاهش می‌یابد.

کاهش خطی اپسیلون

در این روش، اپسیلون به طور خطی کاهش می‌یابد:

$$\epsilon_{t+1} = decay\ rate \times \epsilon_t$$

تاثیرات مختلف مقادیر اپسیلون

- نرخ اکتشاف بالا $(\epsilon = 1)$:

- رفتار: عامل بیشتر کاوش می‌کند.

- نتیجه: ممکن است عملکرد ناپایدار و پاداش‌های متغیر بیشتری مشاهده شود. اما عامل اطلاعات بیشتری از محیط جمع‌آوری می‌کند.

- مزیت: جلوگیری از گیر افتادن در بهینه‌های محلی.

- نرخ اکتشاف پایین $(\epsilon = 0)$:

- رفتار: عامل بیشتر بهره‌برداری می‌کند.

- نتیجه: همگرایی سریع‌تر به سیاست بهینه اما احتمال گیر افتادن در بهینه‌های محلی.

- مزیت: پاداش‌های ثابت‌تر و بهره‌برداری بهتر از دانش فعلی.

استراتژی بهینه برای اپسیلون

استفاده از یک استراتژی کاهش تدریجی (نمایی یا خطی) به عامل اجازه می‌دهد که با کاوش زیاد شروع کرده و به تدریج به بهره‌برداری بیشتر بپردازد. این تعادل بین اکتشاف و بهره‌برداری کمک می‌کند تا عامل به یک سیاست بهینه برسد.

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؟

در محیط تعریف شده اول عامل نتوانست طلا را پیدا کند ولی پس از راه کار های داده شده توانست حتی بهتر و سریع تر از policy DQN بهینه را پیدا کند . این به دلایل زیر است:

مقایسه سرعت یادگیری

Q-learning: در مسائل ساده با فضای حالت کوچک، Q-learning می‌تواند به سرعت به سیاست بهینه برسد زیرا از محاسبات مستقیم و ذخیره‌سازی مقادیر Q در Q-table استفاده می‌کند.

DQN: در مسائل پیچیده‌تر با فضای حالت بزرگ، DQN با استفاده از شبکه‌های عصبی عمیق قادر است سریع‌تر به سیاست بهینه برسد، زیرا می‌تواند الگوهای پیچیده‌تری را در داده‌ها شناسایی کند و یادگیری موثرتری داشته باشد. در نتیجه:

برای مسائل ساده و فضای حالت کوچک، Q-learning سریع‌تر به سیاست بهینه میرسد.

برای مسائل پیچیده و فضای حالت بزرگ، DQN سریع‌تر به سیاست بهینه می‌رسد و عملکرد بهتری دارد.

با تغییر محیط با حالت های مختلف توانستیم به همین نتیجه برسیم .

۵.

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

معماری شبکه عصبی DQN

شبکه عصبی استفاده شده در عامل DQN (Deep Q-Network) معمولاً شامل چندین لایه‌ی پردازشی است که به منظور تقریب زدن تابع Q استفاده می‌شود. این معماری به عامل اجازه می‌دهد تا از ورودی‌های پیچیده مانند تصاویر، بردارهای ویژگی و دیگر انواع داده‌ها برای یادگیری سیاست بهینه استفاده کند.

جزئیات معماری:

ورودی:

بعد حالت (State Dimension): تعداد ویژگی‌هایی که وضعیت محیط را توصیف می‌کند. برای مثال، در مسئله‌ی "دنیای وومپوس" شامل موقعیت عامل، موقعیت وومپوس، وضعیت طلا و وضعیت چاله است.

لایه‌های مخفی (Hidden Layers):

لایه‌های کاملاً متصل (Fully Connected Layers): این لایه‌ها شامل نورون‌های متعددی هستند که به نورون‌های لایه‌های قبلی و بعدی متصل‌اند.

در معماری نمونه ما، دو لایه مخفی وجود دارد:

لایه اول: 100 نورون

لایه دوم: 50 نورون

لایه خروجی:

بعد عمل (Action Dimension): تعداد اکشن‌های ممکن که عامل می‌تواند انتخاب کند. هر نورون در این لایه نشان‌دهنده‌ی ارزش Q برای یک اکشن خاص است.

دلایل انتخاب این معماری

سادگی و کارایی:

معماری‌های با لایه‌های کاملاً متصل برای مسائل با داده‌های برداری مانند مسئله‌ی "دنیای وومپوس" که ورودی‌های آن شامل ویژگی‌های مختلف (موقعیت‌ها و وضعیت‌ها) است، ساده و کارا هستند.

این معماری‌ها به راحتی قابل پیاده‌سازی و آموزش هستند و می‌توانند به خوبی با داده‌های برداری کار کنند.

یادگیری ویژگی‌های پیچیده:

لایه‌های مخفی با تعداد نورون‌های مناسب به شبکه اجازه می‌دهند تا ترکیبات پیچیده‌تری از ویژگی‌ها را یاد بگیرد و به تصمیم‌گیری‌های دقیق‌تر برسد.

تنظیم‌پذیری:

معماری‌های کاملاً متصل به راحتی قابل تنظیم و تغییر هستند. می‌توان با تغییر تعداد نورون‌ها و لایه‌ها، عملکرد شبکه را بهینه کرد.

مناسب برای محیط‌های مختلف:

این معماری به راحتی قابل تطبیق برای محیط‌های مختلف با ویژگی‌های برداری است و می‌تواند در مسائل مختلف تقویت یادگیری به کار رود.