به نام خدا

MP4

دانشجو : مصطفی نبی پور

شماره دانشجویی : ۴۰۱۱۲۸۶۴

نشانی github :

https://github.com/mostafanb77/ML_4022_MP4

نشانی google drive :

https://drive.google.com/drive/folders/1XbRTelv7H89gzgOneYiqfahZuOEStQlz?usp=sharing

سوال ۱) گزارش کد ها در آخر سوال آمده است.

الف)

کد هایی که برای ۲ روش زدیم را در اینجا توضیح میدهیم :(موارد امتیازی به جز حرکت Wumpus رعایت شده است.)

```
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]
        self.pits = [[1, 1], [2, 2]]
        self.wumpus_position = [1, 3]
        self.wumpus_alive = True
        self.arrow_available = True

        # Setting up 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):
        self.agent_position = [0, 0]
        self.wumpus_position = [1, 3]
        self.wumpus_alive = True
        self.arrow_available = True
        return tuple(self.agent_position)

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

```
        done = False

        # Move the agent
        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:
            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:
            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:
            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:
            if self.wumpus_position[1] > self.agent_position[1]:
                reward = 50
                self.wumpus_alive = False
            self.arrow_available = False

        # Check if the agent has found the gold
        if self.agent_position == self.gold_position:
            reward = 100
            done = True
        elif self.agent_position in self.pits:
            reward = -1000
            done = True
        elif self.agent_position == self.wumpus_position and
self.wumpus_alive:
            reward = -1000
            done = True
```

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


    def get_possible_actions(self):
        return ['up', 'down', 'left', 'right', 'shoot_up', 'shoot_down',
'shoot_left', 'shoot_right']
```

همینجور که میبنید در اینجا ما در کل 8 action داریم که ۴ تای آنها حرکت کردن و ۴ تای دیگر تیرزدن به Wumpus میباشد.(توضیح کد این بخش مانند محیط DQN میباشد و گزارش آنرا در بخش بعدی آورده ام.)

Reward ها نیز به صورت کامل رعایت شده است.

کد Q-Learning :

```
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 = {}
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.exploration_decay = exploration_decay

    def get_q_value(self, state, action):
        return self.q_table.get((state, action), 0.0)

    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):
        if random.uniform(0, 1) < self.exploration_rate:
            return random.choice(self.env.get_possible_actions())
        else:
            return max(self.env.get_possible_actions(), key=lambda a:
self.get_q_value(state, a))

    def train(self, episodes):
        rewards_per_episode = []
        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
            rewards_per_episode.append(total_reward)
            self.exploration_rate *= (self.exploration_decay)
        return rewards_per_episode
```

در اینجا ضرایب همانطور که سوال گفته شده معلوم شده است.نرخ اکتشاف از ۱ شروع شده و با اندازه 995. کم میشود.

روش get_q_value

این روش مقدار Q را برای یک جفت حالت-عمل معین از جدول Q بازیابی می کند. اگر جفت state-action پیدا نشد، 0.0 برمی گرداند.

روش update_q_value

این روش، Q-value را برای یک جفت حالت-عمل (state-action) معین بر اساس پاداش دریافتی و پاداش های تخمینی آتی به روز می کند.

دریافت بهترین اقدام بعدی: این روش ابتدا بهترین اقدام بعدی را بر اساس مقادیر Q فعلی برای حالت بعدی مشخص می کند.

محاسبه Target Q-value: Q-value هدف را با استفاده از پاداش و Q-value تخفیف یافته(discount-factor) بهترین اقدام بعدی محاسبه می کند.

محاسبه خطای تفاوت زمانی: تفاوت بین Q-value هدف و Q-value فعلی را محاسبه می کند.

به روز رسانی Q-value: با استفاده از نرخ یادگیری و خطای تفاوت زمانی، Q-value را برای جفت حالت-عمل به روز می کند.

روش select_action

این روش یک عمل را بر اساس خط مشی epsilon-greedy انتخاب می کند:

با احتمالی برابر با نرخ اکتشاف، یک عمل تصادفی را انتخاب می کند.(exploration)

در غیر این صورت، عمل با بالاترین مقدار Q را برای وضعیت فعلی انتخاب می کند.(exploitation)

روش train

این روش عامل را برای تعداد معین قسمت آموزش می دهد:

این یک لیست خالی را برای ذخیره rewards برای هر قسمت مقداردهی می کند.

برای هر قسمت، محیط را بازنشانی می کند و کل پاداش را مقداردهی اولیه می کند.

این یک حلقه را اجرا می کند که در آن عامل یک عمل را انتخاب می کند، عمل را انجام می دهد، Q-value را به روز می کند، و پاداش را تا پایان قسمت جمع آوری می کند.

پاداش کل قسمت را به rewards اضافه می کند.

نرخ اکتشاف را بعد از هر قسمت کاهش می دهد.

: DQN

خوب در اینجا محیط دقیقا مانند قسمت قبل درست شده است ولی یک سری تفاوت ها دارند:

محیط DQN با استفاده از یک آرایه numpy چند کاناله، که شامل موقعیت ها و ویژگی های اضافی مانند در دسترس بودن پیکان است، نمایش وضعیت دقیق تری را ارائه می دهد.

محیط Q-learning از یک نمایش حالت ساده‌تر به عنوان چندگانه موقعیت عامل استفاده می‌کند، که مدیریت آن را آسان‌تر می‌کند اما به طور بالقوه اطلاعات کمتری دارد.

```
class GridEnvironment:
    def __init__(self):
        self.size = 4
        self.grid = np.zeros((self.size, self.size))
        self.agent_pos = (0, 0)
        self.gold_pos = (3, 3)
        self.wumpus_pos = (1, 3)
        self.pits = [(1, 1), (2, 2)]
        self.arrow_available = True
        self.wumpus_alive = True
        self.reset()

    def reset(self):
        self.agent_pos = (0, 0)
        self.arrow_available = True
        self.wumpus_alive = True
        return self.get_state()

    def step(self, action):
        x, y = self.agent_pos
        reward = -1
```

```
        done = False

        if action == 0:  # up
            x = max(0, x - 1)
        elif action == 1:  # down
            x = min(self.size - 1, x + 1)
        elif action == 2:  # left
            y = max(0, y - 1)
        elif action == 3:  # right
            y = min(self.size - 1, y + 1)
        elif action == 4 and self.arrow_available:  # shoot up
            self.arrow_available = False
            if any(self.wumpus_pos == (i, y) for i in range(x)):
                reward = 50
                self.wumpus_alive = False
        elif action == 5 and self.arrow_available:  # shoot down
            self.arrow_available = False
            if any(self.wumpus_pos == (i, y) for i in range(x+1,
self.size)):
                reward = 50
                self.wumpus_alive = False
        elif action == 6 and self.arrow_available:  # shoot left
            self.arrow_available = False
            if any(self.wumpus_pos == (x, j) for j in range(y)):
                reward = 50
                self.wumpus_alive = False
        elif action == 7 and self.arrow_available:  # shoot right
            self.arrow_available = False
            if any(self.wumpus_pos == (x, j) for j in range(y+1,
self.size)):
                reward = 50
                self.wumpus_alive = False

        self.agent_pos = (x, y)

        if self.agent_pos == self.gold_pos:
            reward = 100
            done = True
        elif self.agent_pos in self.pits:
            reward = -1000
            done = True
        elif self.agent_pos == self.wumpus_pos and self.wumpus_alive:
            reward = -1000
            done = True
```

```
        return self.get_state(), reward, done

    def get_state(self):
        state = np.zeros((self.size, self.size, 5))
        state[self.agent_pos][0] = 1
        state[self.gold_pos][1] = 1
        for pit in self.pits:
            state[pit][3] = 1
        state[:,:,4] = int(self.arrow_available)
        return state.flatten()
```

اجزاء

مقداردهی اولیه (روش __init__):

محیط روی شبکه 4×4 تنظیم شده است.

عامل از گوشه بالا سمت چپ شروع می شود (0، 0).

طلا در گوشه پایین سمت راست قرار می گیرد (3، 3).

Wumpus در (1، 3) قرار دارد.

گودال ها در (1، 1) و (2، 2) قرار دارند.

عامل یک تیر در دسترس دارد و Wumpus در ابتدا زنده است.

متد reset برای مقداردهی اولیه وضعیت محیط فراخوانی می شود.

Reset (reset method):

موقعیت عامل را به (0، 0) بازنشانی می کند.

در دسترس بودن پیکان و وضعیت Wumpus را بازنشانی می کند.

وضعیت اولیه محیط را برمی گرداند.

State Representation (روش get_state):

حالت به عنوان یک آرایه 3 بعدی NumPy با شکل (4,4,5) نشان داده می شود.

پنج کانال در بعد سوم نشان دهنده:

موقعیت نماینده

موقعیت طلا

موقعیت وومپوس (در صورت زنده بودن)

موقعیت های پیت ها(گودال ها)

در دسترس بودن پیکان (مقدار باینری در کل شبکه)

Step Function (step method) :

اقدامی را به عنوان ورودی انجام می دهد و موقعیت عامل را متناسب با آن به روز می کند.

اقدامات شامل حرکت به سمت بالا، پایین، چپ، راست و تیراندازی در این جهت ها (در صورت وجود فلش) است.

پاداش را بر اساس موقعیت و تعاملات جدید به روز می کند:

1- برای جابجایی

50 برای تیراندازی به Wumpus.

100 برای پیدا کردن طلا.

1000- برای افتادن در گودال یا برخورد با وومپوس زنده.

بررسی می‌کند که آیا اپیزود انجام شده است (طلا جمع‌آوری شده، مامور در یک گودال یا مامور با وومپوس زنده مواجه شده است).

وضعیت جدید، پاداش و وضعیت انجام شده را برمی‌گرداند.

فضای اکشن محیط از 8 عمل پشتیبانی می کند:

حرکت به بالا

حرکت به پایین

حرکت به سمت چپ

حرکت به راست

شلیک به بالا

شلیک به پایین

شلیک به چپ

شلیک به راست

کد DQN :

```
class ReplayMemory:
    def __init__(self, capacity, state_shape):
        self.capacity = capacity
        self.state_shape = state_shape
        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=bool)
        self.current_size = 0
        self.index = 0

    def store(self, state, action, reward, next_state, done):
        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):
        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])

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):
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.num_actions = num_actions
        self.batch_size = batch_size
        self.epsilon = epsilon_initial
        self.epsilon_decay = epsilon_decay
        self.epsilon_final = epsilon_final
        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):
        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.l
earning_rate),
                      loss=Huber())
        return model

    def update_target_network(self):
        self.target_network.set_weights(self.q_network.get_weights())


    def select_action(self, state):
        if np.random.rand() < self.epsilon:
            action = np.random.randint(self.num_actions)
        else:
            q_values = self.q_network.predict(state[np.newaxis])
            action = np.argmax(q_values[0])
        return action

    def train(self, env, episodes):
        cumulative_rewards, rewards_per_episode = [], []
        for episode in range(episodes):
            state = env.reset()
```

```
            done, total_reward = False, 0
            while not done:
                action = self.select_action(state)
                next_state, reward, done = env.step(action)
                self.buffer.store(state, action, reward, next_state, done)
                state = next_state
                total_reward += reward

                if self.buffer.current_size >= self.batch_size:
                    self.replay()

            self.update_epsilon()
            self.update_target_network()
            cumulative_rewards.append(total_reward)
            rewards_per_episode.append(total_reward)
            print(f'Episode {episode+1}/{episodes}, Total Reward:
{total_reward}, Epsilon: {self.epsilon}')

        return cumulative_rewards, rewards_per_episode

    def replay(self):
        if self.buffer.current_size < self.batch_size:
            return

        states, actions, rewards, next_states, dones =
self.buffer.sample(self.batch_size)
        q_values_current = self.q_network.predict(states)
        q_values_next = self.target_network.predict(next_states)

        targets = q_values_current.copy()
        batch_indices = np.arange(self.batch_size, dtype=np.int32)
        targets[batch_indices, actions] = rewards + self.gamma *
np.amax(q_values_next, axis=1) * (1 - dones)

        self.q_network.train_on_batch(states, targets)

    def update_epsilon(self):
        self.epsilon = max(self.epsilon_final, self.epsilon *
self.epsilon_decay)
    def save_model(self, model_path):
        self.q_network.save(model_path)

    def load_model(self, model_path):
        self.q_network = keras.models.load_model(model_path)
        self.update_target_network()
```

بخش اول از یک memory تشکیل شده است . کلاس ReplayMemory یک بافر با اندازه ثابتی از تجربیات را مدیریت می کند که عامل در طول تعامل خود با محیط با آن مواجه می شود. این بافر به عامل اجازه می دهد تا از تجربیات گذشته بیاموزد، همبستگی بین تجربیات متوالی را شکسته و منجر به یادگیری پایدارتر می شود.

در واقع کلاس ReplayMemory مسئول ذخیره سازی و نمونه برداری از تجربیات است.

کلاس ReplayMemory

کلاس ReplayMemory یک بافر با اندازه ثابتی از تجربیات را مدیریت می کند که عامل در طول تعامل خود با محیط با آن مواجه می شود. این بافر به عامل اجازه می دهد تا از تجربیات گذشته بیاموزد، همبستگی بین تجربیات متوالی را شکسته و منجر به یادگیری پایدارتر می شود.

اجزاء:

مقداردهی اولیه (روش __init__):

Capacity : حداکثر تعداد تجربه هایی که بافر می تواند داشته باشد.

state_shape(Shape of the state representation):

بافرهایی برای ذخیره وضعیت ها، اقدامات، پاداش ها، وضعیت های بعدی و پرچم های انجام شده(done flage در واقع به معنی این که آیا episodeتمام شده است یا نه).

current_size: تعداد تجربیاتی که در حال حاضر ذخیره شده است.

index: اشاره گر به موقعیت فعلی در بافر برای ذخیره تجربیات جدید.

Store Method:

یک تجربه جدید را در بافر ذخیره می کند.

شاخص بافر را به صورت دایره ای به روز می کند.

Current_size را تا ظرفیت بافر افزایش می دهد.

Sample Method :

به طور تصادفی دسته ای از تجربیات را از بافر نمونه برداری می کند.

حالت های نمونه برداری شده، اقدامات، پاداش ها، وضعیت های بعدی و پرچم های انجام شده را برمی گرداند.

کلاس DQNAgent

کلاس DQNAgent عملکردهای اصلی یک عامل DQN، از جمله پخش مجدد تجربه، انتخاب اکشن و به روز رسانی شبکه را پیاده سازی می کند.

اجزاء:

مقداردهی اولیه (روش __init__):

فراپارامترهایی مانند نرخ یادگیری، ضریب تخفیف (گاما)، اندازه دسته ای(batch size) و اپسیلون را برای خط مشی epsilon-greedy راه اندازی می کند.

یک نمونه از ReplayMemory ایجاد می کند.

شبکه Q و شبکه هدف را با استفاده از روش _build_model می سازد.

وزن های شبکه هدف را با شبکه Q با استفاده از روش update_target_network همگام می کند.

Build Model (روش _build_model):

با استفاده از فعال سازی ReLU یک شبکه عصبی با دو لایه مخفی 128 واحدی می سازد.

لایه خروجی دارای واحدهای num_actions است که مقادیر Q را برای هر عمل نشان می دهد.

مدل را با بهینه ساز Adam و تابع ضرر هوبر کامپایل می کند.

به روز رسانی شبکه هدف (روش به روز رسانی_تارگت_شبکه):

وزن ها را از شبکه Q به شبکه هدف کپی می کند.

Action را انتخاب کنید (روش Select_action):

اقدامی را بر اساس خط مشی epsilon greedy انتخاب می کند.

با اپسیلون احتمال، یک عمل تصادفی را انتخاب می کند.

در غیر این صورت، اقدام با بالاترین مقدار Q را همانطور که توسط شبکه Q پیش بینی شده است انتخاب می کند.

Train Method:

برای تعداد معینی از قسمت ها با محیط تعامل دارد.

در هر مرحله زمانی، یک عمل را انتخاب می کند، وضعیت و پاداش بعدی را مشاهده می کند و تجربه را در حافظه پخش ذخیره می کند.

اگر حافظه بازپخش تجربیات کافی داشته باشد، یک مرحله آموزشی را با استفاده از روش پخش مجدد انجام میدهد.

ارزش اپسیلون و وزن شبکه هدف را در پایان هر قسمت به روز می کند.

پاداش‌ها و پاداش‌های انباشته را در هر قسمت برای پیگیری پیشرفت یادگیری برمی‌گرداند.

Replay Method:

مجموعه ای از تجربیات را از حافظه پخش نمونه می گیرد.

مقادیر Q هدف را با استفاده از شبکه هدف محاسبه می کند.

وزن های شبکه Q را بر اساس تفاوت بین مقادیر Q فعلی و مقادیر Q هدف به روز می کند.

به روز رسانی اپسیلون (روش update_epsilon):

اپسیلون را بر اساس میزان decay epsilon کاهش می دهد تا زمانی که به حداقل مقدار (epsilon_final) برسد.

Save Model (روش save_model):

شبکه Q را در یک مسیر فایل مشخص ذخیره می کند.

مدل بارگذاری (روش load_model):

یک شبکه Q ذخیره شده را از یک مسیر فایل مشخص بارگیری می کند.

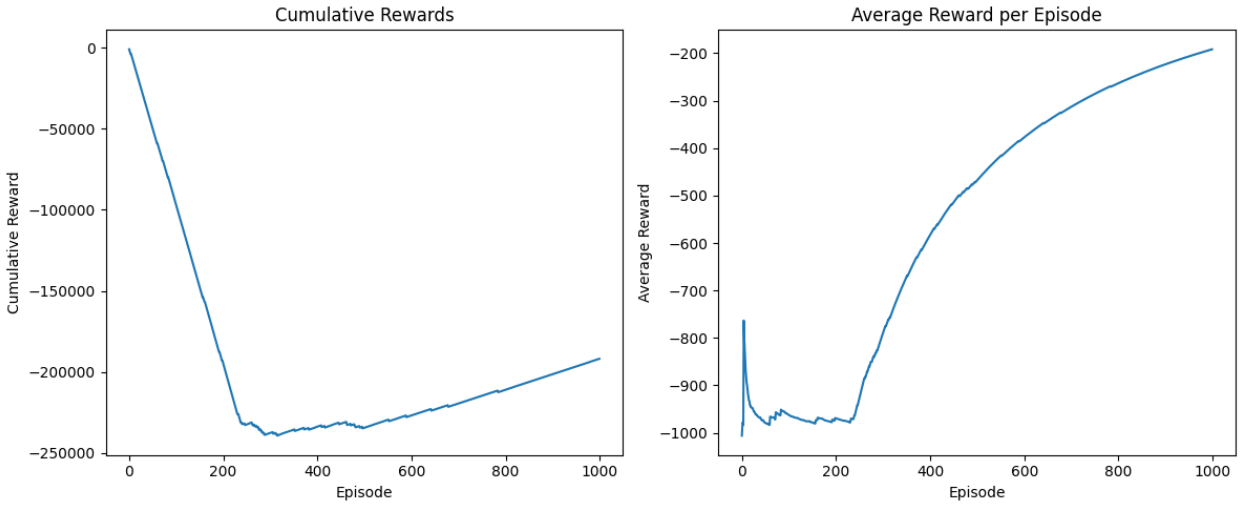
شبکه هدف را با وزن های بارگذاری شده شبکه Q به روز می کند.

و بدین شکل اجرا میکنیم :

```
# 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
cumulative_rewards, rewards_per_episode = agent.train(env, episodes)
```

ب)

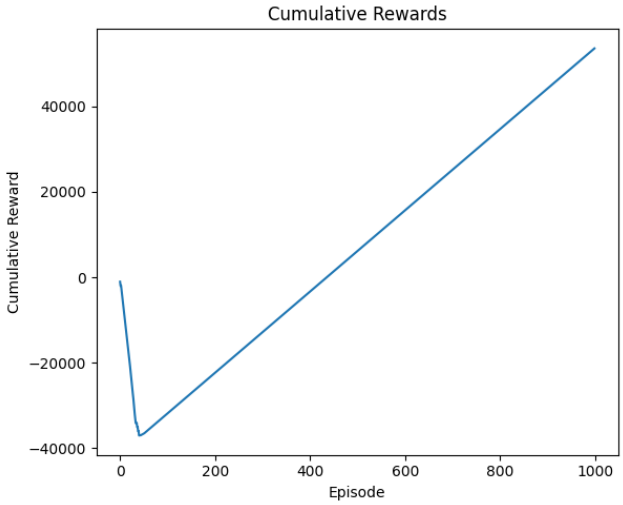
نتیجه Q-Learning بدین شکل میشود :

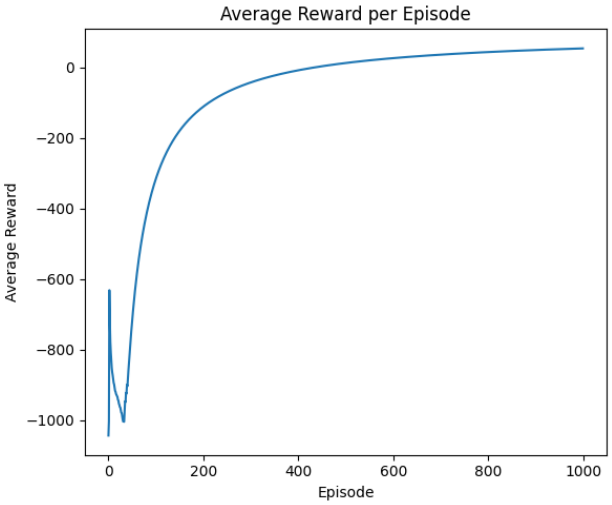


اگر ضریب اکشتاف با سرعت بیشتری کم شود :

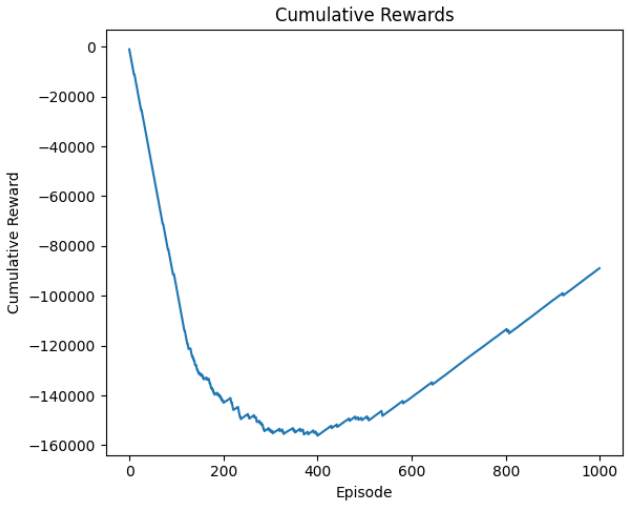
```
self.exploration_rate *= (self.exploration_decay**episode)
```

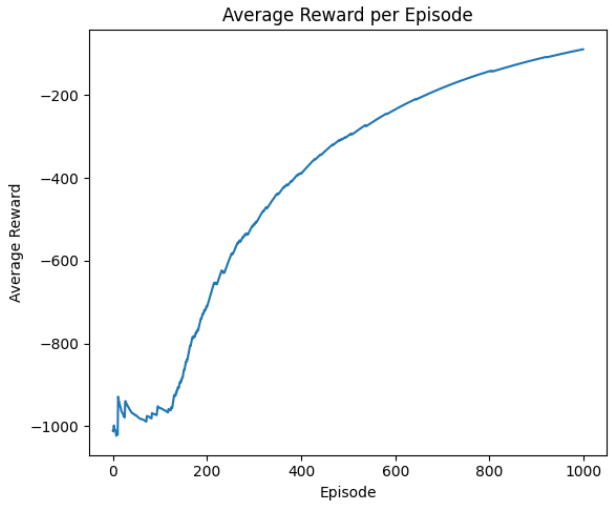
نتیجه بدین شکل میشود :





و نیز برای DQN :





توضیح چگونگی بهتر شدن عملکرد عامل :

توضیح روند Q-Learning:

ایجاد Q-table :

جدول Q، یک آرایه دو بعدی که در آن ردیف ها state و ستون ها action را نشان می دهند، با صفر یا مقادیر تصادفی مقداردهی اولیه می شود.

Action Selection: عامل از یک خط مشی epsilon-greedy برای انتخاب اقدامات استفاده می کند. با احتمال ε، یک عمل تصادفی (اکتشاف) و با احتمال1- ε ، عمل با بالاترین مقدار Q را برای حالت فعلی (exploitation) انتخاب می کند.

به روز رسانی Q-Value: پس از انجام یک اقدام، عامل پاداش و حالت بعدی را مشاهده می کند. Q-value برای جفت حالت-عمل با استفاده از معادله بلمن به روز می شود:

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

که آلفا در واقع ضریب یادگیری و گاما ضریب تخفیف و Q(s,a) نیز در واقع میزان rewardمورد انتظار در صورت انجام action در state s میباشد و R(s,a) نیز پاداش برای این است که در آن action a , state برداشته شود. Max Q(s',a') نیز در واقع بیشترین reward در صورت رفتن به state s' برای تمام action ها موجود (a') میباشد.

Policy improvement: با گذشت زمان، مقادیر Q همگرا می شوند و agent policy بهبود می یابد و اقداماتی را که پاداش بلندمدت بالاتری را به همراه دارند، ترجیح می دهد.

: DQN

مقداردهی اولیه: مقادیر Q با استفاده از یک شبکه عصبی تقریبی می شوند، که حالت را به عنوان ورودی می گیرد و مقادیر Q را برای تمام اقدامات ممکن خروجی می دهد.

Action Selection: مشابه Q-learning، یک خط مشی epsilon-greedy برای انتخاب کنش استفاده می شود.

Experience Replay: تجربیات (وضعیت، اقدام، پاداش، وضعیت بعدی، انجام شده) در بافر پخش مجدد ذخیره می شوند. این به شکستن همبستگی بین تجربیات متوالی کمک می کند و فرآیند یادگیری پایدارتری را فراهم می کند.

آموزش مینی دسته ای(mini batch): در هر مرحله، یک دسته کوچک تصادفی از تجربیات از بافر پخش مجدد برای آموزش شبکه نمونه برداری می شود. این باعث کاهش واریانس و بهبود همگرایی می شود.

به روز رسانی Q-Value: Q-value هدف با استفاده از حالت بعدی و پاداش محاسبه می شود:

$$y = r + \gamma \max Q_{target}(s', a')$$

که در واقع Q_{target} مقدار Q از شبکه هدف است.تابع loss نیز بدین شکل است :

$$Huber = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(y_i - \hat{y}_i)^2 \qquad |y_i - \hat{y}_i| \le \delta$$

$$Huber = \frac{1}{n}\sum_{i=1}^{n}\delta\left(|y_i - \hat{y}_i| - \frac{1}{2}\delta\right) \qquad |y_i - \hat{y}_i| > \delta$$

که در اینجا $\hat{y_i}$ در واقع مقدار $Q(s_i, a_i)$ میباشد.در واقع اختلاف این ۲ اگر از یک میزان (دلتا)بیشتر باشد از MAE میباشد و اگر کم تر باشد MSE میباشد. n نیز در واقع تعداد داده های mini batch است.

بهبود در طول زمان

Exploration to Exploitation: هر دو عامل با اکتشاف بالا (اقدامات تصادفی) برای کشف محیط شروع می کنند. با گذشت زمان، همانطور که Q-value را یاد می‌گیرند، به سمت بهره‌برداری می‌روند و اقداماتی را انتخاب می‌کنند که پاداش مورد انتظار را به حداکثر می‌رسانند.

همگرایی: در یادگیری Q، مقادیر Q-table به مقادیر Q واقعی برای هر جفت حالت-عمل همگرا می شوند. در DQN، شبکه عصبی این مقادیر Q را تقریب می‌زند و با دقیق‌تر شدن پیش‌بینی‌های شبکه، این خط‌مشی بهبود می‌یابد.

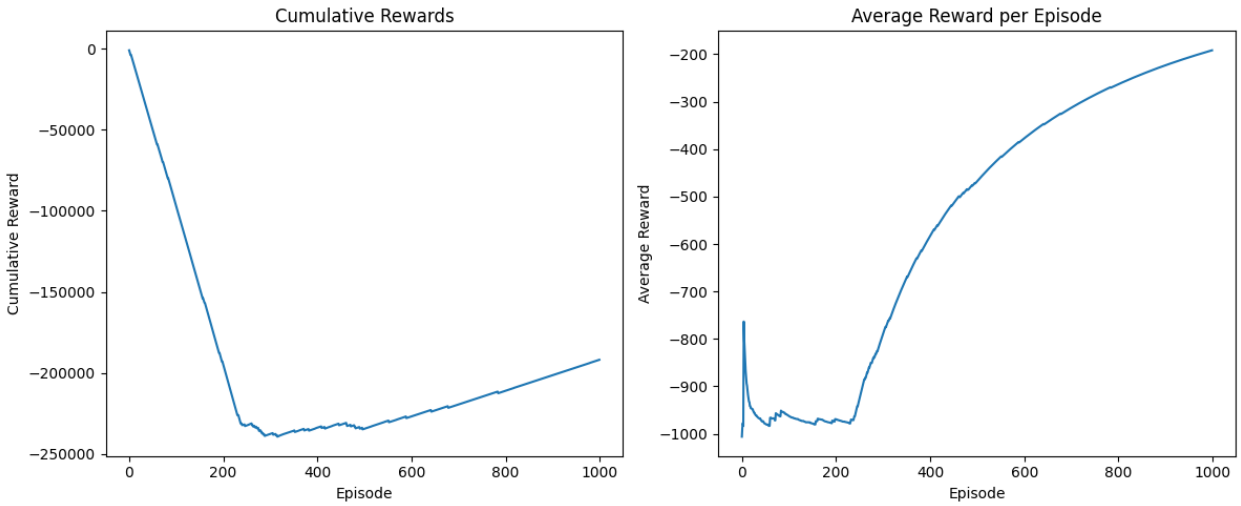
Experience Replay (DQN): با استفاده از Experience Replay، عوامل DQN می توانند به طور موثرتری از تجربیات گذشته بیاموزند و منجر به همگرایی سریعتر و پایدارتر شوند.

Target network (DQN): استفاده از شبکه هدف به تثبیت آموزش با ارائه اهداف Q-value ثابت کمک می کند.

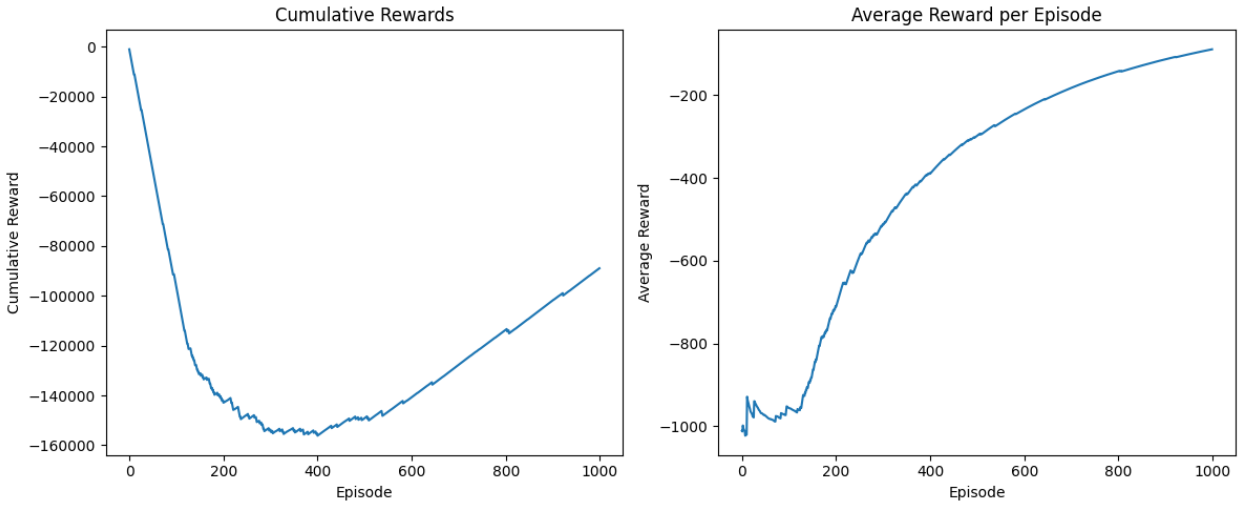
مقایسه میانگین پاداش :

یک بار دیگر نتایج را نشان میدهیم :

Q-learning :



DQN :



خوب همینجور که میبینید هم میانگین پاداش و هم تجمیع پاداش ها در DQN بهتر است.

ج)

Epsilon decay، یک پارامتر حیاتی در یادگیری تقویتی است که بر تعادل بین اکتشاف و بهره برداری تأثیر می گذارد.

Epsilon Discovery Rate

اپسیلون بالا (فاز اولیه):

Exploration: هنگامی که اپسیلون بالا باشد (نزدیک به 1)، عامل به طور تصادفی اقدامات را با احتمال زیاد انتخاب می کند. این بدان معنی است که عامل محیط را به طور کامل تری بررسی می کند و اقدامات مختلف را در حالت های مختلف آزمایش می کند.

reward: در طول این مرحله، عامل احتمالاً پاداش های کم یا حتی منفی دریافت می کند، زیرا اطلاعات کافی برای تصمیم گیری بهینه را ندارد. اغلب ممکن است اقدامات غیربهینه یا مخاطره آمیزی انجام شود که منجر به پیامدهای منفی شود، مانند افتادن در چاله ها یا مواجهه با wumpus.

یادگیری: این مرحله اکتشاف برای جمع آوری تجربیات متنوعی که عامل می تواند از آنها بیاموزد ضروری است. اگرچه پاداش کل در ابتدا کم است، اما عامل در حال ایجاد درک جامعی از محیط است.

کاهش اپسیلون (فاز میانی):

انتقال کاوش به بهره برداری(exploitation): با تحلیل رفتن اپسیلون، احتمال انتخاب کنش های تصادفی کاهش می یابد و عامل شروع به بهره برداری از دانشی می کند که تاکنون به دست آورده است. هنوز هم گهگاه کاوش می کند، اما با احتمال کمتر.

پاداش ها: اقدامات عامل آگاه تر می شود و شروع به اجتناب از نتایج منفی می کند. در نتیجه، کل پاداش ها شروع به افزایش می کند. عامل بین اکتشاف (برای اصلاح خط مشی خود) و بهره برداری (برای به حداکثر رساندن پاداش) تعادل برقرار می کند.

اپسیلون پایین (مرحله بعدی):

بهره برداری: وقتی اپسیلون کم است (نزدیک به 0 که در اینجا ما 05. در نظر گرفته ایم)، عامل در درجه اول از مقادیر Q آموخته شده برای تصمیم گیری استفاده می کند. اقداماتی را انتخاب می کند که معتقد است بر اساس تجربه اش بالاترین پاداش را به همراه خواهد داشت.

پاداش ها: مجموع پاداش های عامل در این مرحله بیشتر است زیرا تصمیمات تقریباً بهینه ای می گیرد. یاد گرفته است که به طور موثر در محیط حرکت کند، از خطرات اجتناب کند و پاداش را به حداکثر برساند.

ثبات: رفتار عامل با اتکا به خط مشی آموخته شده پایدارتر و قابل پیش بینی تر می شود.

تاثیر بر پاداش کل

اپسیلون اولیه و پاداش کم:

اکتشاف بالا به این معنی است که عامل بسیاری از اقدامات تصادفی را انجام می دهد که منجر به عملکرد کمتر از حد مطلوب و مجموع پاداش های پایین (به عنوان مثال، -1000) می شود.

این مرحله برای جمع آوری داده ها و جلوگیری از گیرکردن عامل در بهینه محلی بسیار مهم است.

کاهش اپسیلون و افزایش پاداش:

با کاهش اپسیلون، عامل شروع به استفاده از دانش آموخته شده خود می کند که منجر به تصمیم گیری بهتر و مجموع پاداش بالاتر می شود (به عنوان مثال، 145).

تغییر تدریجی از اکتشاف به بهره برداری به عامل اجازه می دهد تا سیاست خود را بهبود بخشد و در عین حال آن را از طریق اکتشاف گاه به گاه اصلاح کند.

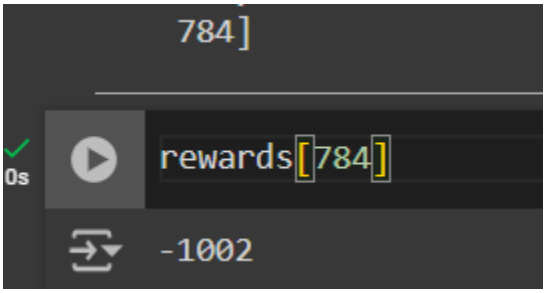
د)

در واقع در اینجا ما باید ببینیم که از کجا به بعد مجموع reward هایی که در یک episodeداشته ایم کوچک تر از 0 نشده این نشان دهنده این است که از آنجا به بعد دیگر در pit یا Wumpus نیفتاده ایم.

که اینکار را با این دستور انجام میدهیم :

```
import numpy as np

rewardql = np.array(rewards)
list(np.where(rewardql < 0)[0])
```

در واقع در این کد ما rewards که به صورت لیست هست را به صورت numpy در آورده ایم سپس index اعدادی که کوچک تر از ۰ هستند را نشان میدهد که بعد از اجرای کد ایندکس 784 میباشد.



عملکرد:

یکی از معیار های ارزیابی میتواند cumulative reward باشد همانطور که میبینید DQN نسبت به Q-learning توانسته در این زمینه بهتر عمل کند در واقع در Q learning توانسته در ۲۵۰ episode , policy را به ردستی یاد بگیرد و نمودار همانطور که معلوم است رشد کند ولی در DQN در تقریبا episode ۵۰۰ توانسته Policy را درست یادبگیرد ولی این مقدار بیشتر از Q learning است.

خلاصه DQN policy بهتر از Q learning میباشد ولی Q learning سریع تر یاد گرفته است.(شیب نمودار وقتی مثبت شود نشاندهنده یادگرفتن policy میباشد.)

همینطور که میبینید average reward در واقع همان cumulative reward/episode میباشد که در DQN نیز بهتر از دیگری است که نشان دهنده بهتر بودن polic در DQN میباشد.

ه)

کد معماری بدین صورت است :

```
        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)
        ])
```

این شبکه MLP میباشد که از۲ لایه پنهان تشکیل شده است.لایه های پنهان هر کدام از 128 نورون تشکیل شده اند و از تابع فعال سازی ReLU استفاده می کنند. ReLU انتخاب شده است زیرا به شبکه کمک می کند تا الگوهای پیچیده را با معرفی غیرخطی بیاموزد و به کاهش مشکل ناپدید شدن گرادیان(vanishing gradiant) را کمک می کند.

لایه خروجی دارای تعدادی واحد برابر با تعداد اقدامات ممکن (num_actions) است. هر واحد در این لایه با مقدار Q یک عمل خاص با توجه به وضعیت ورودی مطابقت دارد.

استدلال پشت انتخاب معماری:

سادگی و کارایی:

این معماری نسبتا ساده با تنها دو لایه پنهان است. این سادگی به کاهش پیچیدگی محاسباتی و زمان آموزش کمک می‌کند و در عین حال قادر به یادگیری بازنمایی‌های معنی‌دار از فضای عمل حالت است.

ظرفیت کافی:

دو لایه پنهان با 128 نورون هر کدام ظرفیت کافی برای یادگیری توابع پیچیده Q-value را فراهم می کنند. این تعداد نورون ها و لایه ها اغلب نقطه شروع خوبی برای بسیاری از مشکلات RL است و می تواند طیف گسترده ای از الگوها و ویژگی ها را از حالت ورودی ثبت کند.

فعال سازی ReLU:

توابع فعال سازی ReLU به طور گسترده در یادگیری عمیق مورد استفاده قرار می گیرند زیرا با کاهش مشکل گرادیان ناپدید شدن به آموزش شبکه های عمیق کمک می کنند. ReLU همچنین غیرخطی بودن را معرفی می کند که برای یادگیری الگوهای پیچیده بسیار مهم است.(یکی دیگر اینکه مقدار منفی برای action معرفی نشده است.)

لایه خروجی بدون فعال سازی:

لایه خروجی از یک تابع فعال سازی استفاده نمی کند ، زیرا ما مقادیر Q را پیش بینی می کنیم که می تواند هر عدد واقعی باشد. عدم وجود یک تابع فعال سازی تضمین می کند که شبکه می تواند طیف وسیعی از مقادیر Q را پیش بینی کند.

انعطاف پذیری:

این معماری را می توان به راحتی بر اساس پیچیدگی مشکل تنظیم کرد. برای مثال، اگر فضای حالت یا فضای عمل پیچیده‌تر بود، می‌توان لایه‌ها یا نورون‌های اضافی را برای افزایش ظرفیت شبکه اضافه کرد.

نکات اضافی کد :

```
# 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
cumulative_rewards, rewards_per_episode = agent.train(env, episodes)
```

این کد برای اجرای محیط میباشد.

سپس پارامتر های DQN را معلوم کرده ایم سپس همانطور که Function DQN را تعریف کرده بودیم ورودی های مورد نیاز را به آن داده سپس تعداد episode را ۱۰۰۰ تا قرار دادیم.

سپس خروجی آن را مجموع reward و خود آن در هر episodeگزاشتیم.

```
# Initialize environment and agent
env = GridEnvironment()
agent = QLearningAgent(env)

# Train agent
rewards = agent.train(1000)
```

کد بالا هم برای runکردن Q-Learning میباشد.