تمرین سری چهارم درس یادگیری ماشین

استاد:

دکتر مهدی علیاری شوره‌دلی

دانشجو:

نازنین بنداریان

۴۰۲۱۳۸۷۶

بهار ۱۴۰۳

https://github.com/nazaninbondarian/MachineLearning2024/tree/main/Chapter%204

فهرست مطالب

عنوان **صفحه**

فهرست شکل‌ها

عنوان | صفحه

۱- پرسش یک: حل دنیای Wumpus

۱-۱- بخش الف

در Q-learning در ابتدا ما یک جدول مربوط به state، action و value ایجاد و بروزرسانی می‌کنیم. با انجام دادن یکسری اقدامات به صورت رندم از حالت s به حالت $\acute{s}$ می‌رویم، که این اقدام امتیاز مشخصی دارد. مقدار Q پس از انجام هر اقدامی با استفاده از رابطه‌ی زیر، بروزرسانی می‌گردد.

$$Q(s,a) = Q(s,a) + \alpha\left(Q(s,a) - \gamma \max_{\acute{a}} Q(\acute{s},\acute{a})\right)$$

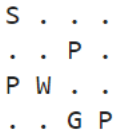
در این رابطه α نرخ یادگیری و γ ضریب تخفیف می‌باشد. به طور خلاصه می‌توان به موارد زیر برای ایجاد اشاره نماییم.

ایجاد محیط. ایجاد دیکشنری برای ذخیره state، action و value. انتخاب یکی از اقدامات به صورت رندم و اعمال آن به محیط. انتخاب بهترین مدل. بروزرسانی جدول Q.

در ابتدا کتابخانه‌های مورد نیاز افزوده شده است و محیط مورد نظر را به صورت یک شبکه ۴×۴ تعریف می‌کنیم. محل هر یک از موجودات درون بازی را تعریف می‌کنیم و آن را نمایش می‌دهیم. مقادیر امتیازات و اقدامات امکان‌پذیر تعریف شده و مقدار Q-Value را صفر در نظر می‌گیریم.

```
import numpy as np
import random
import matplotlib.pyplot as plt
#random.seed(76)  # Initialize the random number generator with a seed value of 42

# Define the grid size and initial positions
grid_size = 4
pits = [(1, 2), (2, 0), (3, 3)]  # Adjusted to 0-based indexing
wumpus = (2, 1)  # Adjusted to 0-based indexing
gold = (3, 2)  # Adjusted to 0-based indexing
start = (0, 0)  # Adjusted to 0-based indexing
other_points = [(0, 1), (1, 1), (1, 3), (2, 2), (2, 3), (3, 0)]  # Any other points

# Create an empty grid
grid = [["." for _ in range(grid_size)] for _ in range(grid_size)]

# Mark specific positions in the grid
grid[start[0]][start[1]] = "S"
for pit in pits:
    grid[pit[0]][pit[1]] = "P"
grid[wumpus[0]][wumpus[1]] = "W"
grid[gold[0]][gold[1]] = "G"

# Print the grid
for row in grid:
    print(" ".join(row))

# Rewards dictionary
rewards = {}
```

```
for x in range(grid_size):
    for y in range(grid_size):
        rewards[(x, y)] = -1
rewards[gold] = 100
for pit in pits:
    rewards[pit] = -1000
rewards[wumpus] = -1000

# Define actions
actions = ['up', 'down', 'left', 'right']

# Define the Q-table
Q = np.zeros((grid_size, grid_size, len(actions)))
```



شکل ۱-۱ زمین بازی

با استفاده از تابع زیر اقدامات ممکن را تعریف می‌کنیم.

```
def get_next_state(state, action):
    x, y = state
    if action == 'up':
        return (max(x - 1, 0), y)
    elif action == 'down':
        return (min(x + 1, grid_size - 1), y)
    elif action == 'left':
        return (x, max(y - 1, 0))
    elif action == 'right':
        return (x, min(y + 1, grid_size - 1))
```

درون یک تابع الگوریتم Q-learning را تعریف می‌کنیم. ورودی این تابع تعداد اپیزود و مقدار ماکزیمم اپیزود برای بخش ۱-۴-۱- می‌باشد. سپس مقادیر اولیه مورد نیاز الگوریتم مقدار دهی شده است.

```
def q_learning(episodes, consistency_threshold=10):
    Q = np.zeros((grid_size, grid_size, len(actions)))
    learning_rate = 0.1
    discount_factor = 0.9
    exploration_rate = 1.0
    exploration_decay = 0.995
    min_exploration_rate = 0.01
    cumulative_rewards = []
    consecutive_successes = 0
    episodes_to_consistency = None
```

درون این تابع یک حلقه for تعریف شده است که در ابتدای هر اپیزود عامل را در مکان شروع قرار می‌دهد. سپس در صورتی که عدد انتخابی به صورت رندم کوچکتر از مقدار ϵ باشد، یکسری اقدامات به صورت رندم انتخاب می‌گرددند. مقدار امتیاز این اقدام محاسبه می‌گردد و با توجه به آن مقدار Q-Value را بروزرسانی می‌کنیم. الگوریتم پس از یافتن طلا و یا افتادن داخل چاه و یا خورده شدن توسط wumpus پایان می‌یابد. تعداد اولین دفعات تکرار اپیزود که به صورت ۱۰ تکرار متوالی موفق بوده است نیز ذخیره شده است. مقدار ϵ نیز کاهش یافته است.

```
    for episode in range(episodes):
        state = start
        total_reward = 0
        done = False
        success = False

        while not done:
            if random.uniform(0, 1) < exploration_rate:
                action = random.choice(actions)
            else:
                action = actions[np.argmax(Q[state[0], state[1]])]

            next_state = get_next_state(state, action)
            reward = rewards[next_state]
            total_reward += reward

            # Update Q-value
            best_next_action = np.argmax(Q[next_state[0], next_state[1]])
            Q[state[0], state[1], actions.index(action)] += learning_rate * (
                reward + discount_factor * Q[next_state[0], next_state[1], best_next_action] -
                Q[state[0], state[1], actions.index(action)]
            )

            state = next_state

            if state == gold:
                done = True
                success = True
            elif reward == -1000:  # Pit or Wumpus
                done = True
                success = False

        if success:
            consecutive_successes += 1
            if consecutive_successes == consistency_threshold and episodes_to_consistency is None:
                episodes_to_consistency = episode + 1
        else:
            consecutive_successes = 0

        exploration_rate = max(min_exploration_rate, exploration_rate * exploration_decay)
        cumulative_rewards.append(total_reward)

    return cumulative_rewards, episodes_to_consistency
```

در Deep Q-Learning (DQN) از شبکه عصبی برای تقریب ststeهای دیده نشده براساس مقادیری که قبلا دیده شده می‌باشد. در واقع داریم به کمک شبکه عصبی مقدار زیر را تخمین می‌زنیم.

$$F(\bar{x}_t, \bar{w}, a) = \hat{Q}(s_t, a)$$

در واقع دوست داریم رابطه‌ی زیر برقرار باشد.

$$Q(s_t, a) = r_t + \gamma \max_{\acute{a}} \hat{Q}(s_{t+1}, \acute{a})$$

$$F(\bar{x}_t, \bar{w}, a) = r_t + \gamma \max_{\acute{a}} \hat{F}(\bar{x}_{t+1}, \bar{w}, \acute{a})$$

به این منظور تابع هزینه زیر ایجاد می‌گردد. شاید در ابتدا مقدار هدف خوب نباشد، اما با گذشت زمان بهبود می‌یابد و همگرا می‌گردد.

$$L_t = \left[\underbrace{r_t + \gamma \max_{\acute{a}} \hat{F}(\bar{x}_{t+1}, \bar{w}, \acute{a})}_{target(constant)} - F(\bar{x}_t, \bar{w}, a) \right]$$

روش‌های م.ورد استفاده در این بخش برای افزایش همگرایی و عدم گیر کردن در گرادیان محلی از Rep buffer استفاده می‌گردد، که در آن ارتباط داده‌ها را از بین می‌برد و shuffle می‌کند. در واقع ما داده‌ها را ذخیره می‌کنیم و با توجه به اندازه آن به صورت رندم به تعداد batch-size انتخاب می‌کنیم.

در ابتدا ساختار شبکه درون یک کلاس تعریف شده است که در بخش ۵-۱- توضیح داده شده است. سپس تابع برای انتقال به حالت بعدی تعریف شده است. درون تابع دوم نیز به تعداد حالت‌های ممکن ورودی و به تعداد اقدامات خروجی شبکه را تعریف می‌کنیم. سپس شبکه را ایجاد می‌کنیم که از بهین‌ساز Adam و تابع هزینه MSE استفاده شده است. سپس مقادیر ثابت به الگوریتم را تعریف می‌کنیم. سپس با مانند الگوریتم Q-learning از روش $\epsilon - geedy$ استفاده می‌گردد. سپس به آرایه memory افزوده می‌گردد. در صورتی که تعداد المان‌های memory بیشتر batch-size باشد به داخل حلقه‌ی if می‌رود و مقدار Q-value را بروزرسانی می‌کند و حالت بعدی را انتخاب می‌کند. در نهایت مقادیر وزن‌های شبکه و تابع هزینه بروزرسانی می‌گردد.

```
def get_state_vector(state):
    state_vector = np.zeros(grid_size * grid_size)
    state_vector[state[0] * grid_size + state[1]] = 1
    return state_vector

def dqn_learning(episodes, actions, consistency_threshold=10):
    input_dim = grid_size * grid_size
    output_dim = len(actions)
    dqn = DQN(input_dim, output_dim)
    optimizer = optim.Adam(dqn.parameters(), lr=0.001)
    criterion = nn.MSELoss()

    exploration_rate = 1.0
    exploration_decay = 0.995
    min_exploration_rate = 0.01
    discount_factor = 0.9
    batch_size = 32
    memory = []
    cumulative_rewards = []
    consecutive_successes = 0
    episodes_to_consistency = None

    for episode in range(episodes):
        state = start
        total_reward = 0
        done = False
        success = False
```

```
        while not done:
            if random.uniform(0, 1) < exploration_rate:
                action = random.choice(actions)
            else:
                state_vector = torch.FloatTensor(get_state_vector(state)).unsqueeze(0)
                q_values = dqn(state_vector)
                action = actions[torch.argmax(q_values).item()]

            next_state = get_next_state(state, action)
            reward = rewards[next_state]
            total_reward += reward

            memory.append((state, action, reward, next_state))

            if len(memory) > batch_size:
                batch = random.sample(memory, batch_size)
                states, actions_batch, rewards_batch, next_states = zip(*batch)

                state_vectors = torch.FloatTensor([get_state_vector(s) for s in states])
                next_state_vectors = torch.FloatTensor([get_state_vector(s) for s in next_states])

                q_values = dqn(state_vectors)
                next_q_values = dqn(next_state_vectors)

                targets = q_values.clone()
                for i in range(batch_size):
                        targets[i][actions.index(actions_batch[i])] = rewards_batch[i] + discount_factor *
torch.max(next_q_values[i]).item()

                optimizer.zero_grad()
                loss = criterion(q_values, targets)
                loss.backward()
                optimizer.step()

            state = next_state

            if state == gold:
                done = True
                success = True
            elif reward == -1000:  # Pit or Wumpus
                done = True
                success = False

        if success:
            consecutive_successes += 1
            if consecutive_successes == consistency_threshold and episodes_to_consistency is None:
                episodes_to_consistency = episode + 1
        else:
            consecutive_successes = 0
```

```
        exploration_rate = max(min_exploration_rate, exploration_rate * exploration_decay)
        cumulative_rewards.append(total_reward)

    return cumulative_rewards, episodes_to_consistency
```

حال یک شی از هر الگوریتم ایجاد می‌کنیم و برنامه را اجرا می‌کنیم.

```
# Run Q-learning
episodes = 1000
q_rewards, q_consistency_achieved = q_learning(episodes)
print("Training completed with Q-learning.")


# Run DQN
dqn_rewards, dqn_consistency_achieved = dqn_learning(episodes, actions)
print("Training completed with Deep Q-learning.")
```

به طور خلاصه می‌توان به موارد زیر اشاره نمود:

Q-Learning: از یک Q-Table برای ذخیره و بروزرسانی مقادیر Q برای هر جفت حالت-عمل استفاده می‌کند.

Deep Q-Learning (DQN): از یک شبکه عصبی برای تقریب مقادیر Q استفاده می‌کند و امکان نمایش وضعیت پیچیده‌تری را فراهم می‌کند.

هر دو روش شامل استراتژی‌های exploration و exploitation می‌شوند و نرخ اکتشاف در طول زمان کاهش می‌یابد تا عامل اجازه دهد تا Policyهای بهینه را بیاموزد. روش DQN برای فضاهای حالت بزرگتر که در آن جدول Q غیرعملی است، مناسبتر است.

۱-۲- بخش ب

۱-۲-۱- پاداش تجمعی

این کد هر دو الگوریتم Q-learning و DQN را برای ۱۰۰۰ قسمت اجرا می‌کند و پاداش‌های تجمعی آنها را در طول زمان ترسیم می‌کند. در اینجا تحلیلی از چگونگی بهبود عملکرد عامل در طول زمان ارائه شده است.

عملکرد اولیه:

هر دو الگوریتم معمولاً با پاداش‌های کم شروع می‌شوند زیرا عوامل به طور تصادفی کاوش می‌کنند و اغلب با جریمه-هایی (چاله‌ها یا Wumpus) مواجه می‌شوند.

مرحله یادگیری:

Q-Learning: معمولاً بهبود اولیه سریع‌تری را نشان می‌دهد، زیرا مستقیماً مقادیر Q را برای هر جفت حالت-عمل به‌روزرسانی می‌کند.

DQN: ممکن است به دلیل نیاز به آموزش شبکه عصبی شروع کندتری داشته باشد، اما به طور بالقوه می‌تواند عملکرد بهتری در دراز مدت داشته باشد.

همگرایی:

Q-Learning: در محیط‌های ساده‌ای مانند این شبکه ۴×۴ نسبتاً سریع به یک عملکرد پایدار همگرا می‌شود.

DQN: ممکن است برای همگرایی بیشتر طول بکشد، اما به طور بالقوه می‌تواند policy بهینه‌تری پیدا کند، به خصوص در محیط‌های پیچیده‌تر.

پایداری:

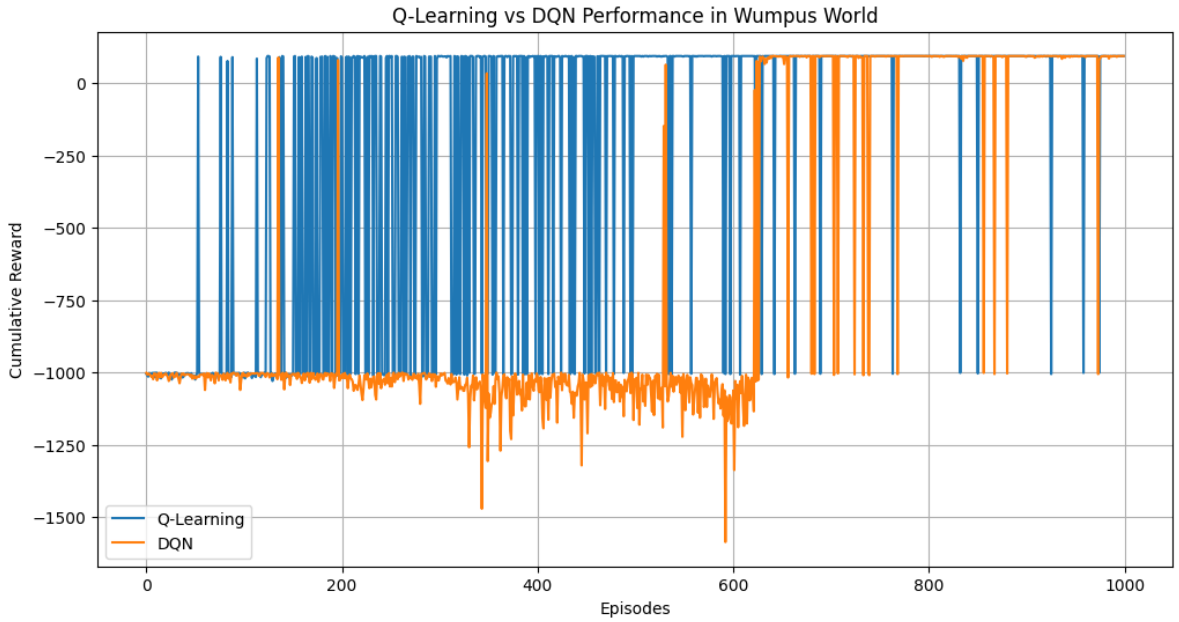
Q-Learning: عموماً پس از همگرا شدن، عملکرد پایدارتری را نشان می‌دهد.

DQN: ممکن است به دلیل ماهیت آموزش شبکه عصبی و نمونه‌برداری دسته‌ای، نوسانات بیشتری را نشان دهد.

عملکرد نهایی:

در این محیط ساده، هر دو روش باید در نهایت یاد بگیرند که به سمت طلا حرکت کنند و در عین حال از خطرات اجتناب کنند.

DQN ممکن است در درازمدت به عملکرد کمی بهتر دست یابد، زیرا توانایی آن در تعمیم بین حالت‌ها است.



شکل ۲-۱ نمایش تجمع پاداش‌ها

۱-۲-۲- میانگین پاداش

برای مقایسه میانگین پاداش در هر قسمت برای هر دو عامل یادگیری Q و Deep Q-learning (DQN) پس از ۱۰۰۰ قسمت، باید میانگین پاداش تجمعی را برای هر روش محاسبه کنیم و سپس نتایج را با هم مقایسه کنیم.

```
# Calculate average reward per episode
avg_q_reward = np.mean(q_rewards)
avg_dqn_reward = np.mean(dqn_rewards)

print(f"Average reward per episode for Q-Learning: {avg_q_reward}")
print(f"Average reward per episode for DQN: {avg_dqn_reward}")

# Determine which algorithm performed better
if avg_q_reward > avg_dqn_reward:
    print("Q-Learning performed better.")
else:
    print("DQN performed better.")
```

پس از اجرای کد، میانگین پاداش هر قسمت برای آموزش Q و DQN دریافت خواهید کرد. بیانیه‌های چاپی نشان می‌دهد که کدام الگوریتم براساس میانگین پاداش‌ها بهتر عمل کرده است. این رویکرد مقایسه واضحی از عملکرد Q-learning و DQN در محیط Wumpus World ارائه می‌دهد. همانطور که مشاهده می‌گردد میانگین امتیاز Q-learning بزرگتر است در نتیجه عملکرد بهتری دارد.

```
Average reward per episode for Q-Learning: -201.505
Average reward per episode for DQN: -626.173
Q-Learning performed better.
```

شکل ۳-۱ میانگین پاداش

۱-۳- بخش ج

در یادگیری تقویتی، پارامتر ϵ در الگوریتم $\epsilon - greedy$ برای کنترل میزان exploration و exploitation استفاده می‌شود. تاثیر مقادیر بالا و پایین ϵ به شرح زیر است:

۱. ϵ بالا:

-اکتشاف بیشتر: عامل (agent) به دفعات بیشتری انتخاب‌های تصادفی انجام می‌دهد و به جای انتخاب بهترین اقدام (action) شناخته شده، اقدامات جدید را امتحان می‌کند.

- خطر کاهش بهره‌وری: ممکن است به جای بهره‌برداری از اقدامات با بازدهی بالا، اقدامات تصادفی و ضعیف انتخاب شود.

-کشف بهتر فضای حالت: می‌تواند به شناسایی استراتژی‌ها و اقدامات بالقوه جدید و بهبود کلی سیاست (policy) منجر شود.

۲. ϵ پایین

- بهره‌برداری بیشتر: عامل بیشتر بر اساس اقدامات با بازدهی بالا عمل می‌کند و کمتر به اقدامات تصادفی می‌پردازد.

-خطر گیر افتادن در حد محلی: اگر عامل همیشه به بهره‌برداری از بهترین اقدامات شناخته شده بپردازد، ممکن است فرصت‌های بهبود را از دست بدهد و در یک استراتژی غیر بهینه گیر بیافتد.

-اکتشاف ناکافی: می‌تواند منجر به عدم کشف اقدامات و استراتژی‌های جدید و بالقوه بهینه‌تر شود.

به طور خلاصه، مقدار ϵ باید به گونه‌ای تنظیم شود که تعادل مناسبی بین اکتشاف و بهره‌برداری ایجاد شود. معمولاً از استراتژی‌هایی مانند $\epsilon - greedy$ کاهش‌یابنده استفاده می‌شود، که در آن ϵ به تدریج کاهش می‌یابد تا عامل در ابتدا به میزان بیشتری اکتشاف کند و به مرور زمان بیشتر به بهره‌برداری بپردازد.

از روش $\epsilon - greedy$ برای رفع عدم پایداری و عدم گیر کردن در بهینه محلی استفاده می‌گردد.

۱-٤- بخش د

۱-٤-۱- یافتن طلا توسط عامل بدون افتادن در گودال یا خورده شدن توسط wumpus

برای تعیین این موضوع، پیاده‌سازی باید شامل موارد زیر باشد:

۱. شمارنده‌ای برای اپیزودهای موفق (جایی که عامل بدون مواجهه با خطرات طلا پیدا می‌کند).

۲. راهی برای ردیابی قسمت‌های موفق متوالی

۳. یک آستانه از پیش تعریف شده برای آنچه که ما عملکرد سازگار می‌دانیم

برای مثال، می‌توانیم عملکرد عامل را در صورتی که کار (یافتن طلا بدون خطر) را در ۱۰ قسمت متوالی با موفقیت انجام دهد، سازگار در نظر بگیریم. در اینجا نحوه تغییر توابع برای ردیابی این است:

در این بخش اپیزودهای موفق متوالی را دنبال می‌کند و تعداد قسمت‌هایی را که برای دستیابی به عملکرد ثابت طول می‌کشد گزارش می‌کند (در این مورد به عنوان ۱۰ قسمت موفق متوالی تعریف می‌شود).

همانطور که نتایج نشان می‌دهد الگوریتم Q-learning زودتر توانسته است یک policy بهینه را بیابد.

```
Q-learning agent achieved consistent performance after 307 episodes
DQN agent achieved consistent performance after 637 episodes
```

شکل ۴-۱ دستیابی به عملکرد ثابت

۱-٤-۲- **مقایسه کارایی یادگیری** Q-learning **و** Deep Q-Learning (DQN)

برای مقایسه کارایی یادگیری Q-learning و Deep Q-Learning (DQN) و تعیین اینکه کدام یک policy بهینه را سریعتر آموخته‌اند، باید چندین عامل را در نظر بگیریم، از جمله سرعت همگرایی، توانایی مدیریت فضاهای حالت بزرگ، و عملکرد کلی از نظر پاداش‌های تجمعی.

تفاوت‌ها و مزایای کلیدی:

Q-Learning:

Q-Table: از یک جدول Q برای ذخیره و بروزرسانی مقدار هر جفت حالت-عمل استفاده می‌کند.

حافظه و محاسبات: با فضاهای حالت بزرگ به دلیل رشد تصاعدی جدول Q غیر ممکن می‌شود.

Exploration و Exploitation: برای ایجاد تعادل بین Exploration و Exploitation، بر policy حریصانه اپسیلون تکیه دارد.

همگرایی: می‌تواند به سرعت در فضاهای حالت کوچک و گسسته همگرا شود اما با فضاهای بزرگتر یا پیوسته مبارزه می‌کند.

Deep Q-Learning (DQN):

شبکه عصبی: از یک شبکه عصبی برای تقریب مقادیر Q استفاده می‌کند و به آن اجازه می‌دهد فضاهای حالت بزرگ و پیوسته را مدیریت کند.

Replay Memory: از یک بافر حافظه تکراری برای ذخیره تجربیات گذشته و شکستن همبستگی بین نمونه‌های متوالی استفاده می‌کند.

شبکه هدف: اغلب از یک شبکه هدف برای تثبیت آموزش با بروزرسانی مقادیر Q هدف با سرعت کمتر استفاده می‌کند.

اکتشاف و بهره برداری: همچنین از policy حریصانه اپسیلون استفاده می‌کند اما می‌تواند استراتژی‌های اکتشافی پیچیده‌تری را اعمال کند.

همگرایی: عموماً به دلیل پیچیدگی آموزش شبکه عصبی در ابتدا کندتر همگرا می‌شود اما می‌تواند در محیط‌های پیچیده به عملکرد بهتری دست یابد.

برای مقایسه تجربی کارایی یادگیری، می‌توانیم پاداش‌های انباشته و تعداد قسمت‌هایی را که هر عامل طول می‌کشد تا policy بهینه را پیدا کند، تجزیه و تحلیل کنیم.

Q-Learning: معمولاً بهبود اولیه سریع‌تری را در فضاهای کوچک نشان می‌دهد، اما ممکن است در محیط‌های بزرگ‌تر و پیچیده‌تر دچار مشکل شود.

DQN: ممکن است به دلیل پیچیدگی آموزش شبکه عصبی شروع کندتری داشته باشد، اما می تواند به پاداش‌های تجمعی بالاتر و تعمیم بهتر در محیط‌های پیچیده دست یابد.

به عنوان جمع‌بندی می‌توان بیان نمود که الگوریتم Q-learning زودتر توانسته است یک policy بهینه را بیابد.

۱-۵- ساختار شبکه عصبی

در این سوال ما از ساختار شبکه‌ای زیر استفاده نموده‌ایم. در اینجا از یک شبکه ۳ لایه استفاده شده که ورودی به تعداد حالت‌های ممکن و خروجی نیز به تعداد اقدامات ممکن است. سپس خروجی لایه اول و دوم از یک تابع relu عبور داده می‌شوند.

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

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

برای بهبود عملکرد می‌توانیم دو شبکه‌ی مجزا در نظر بگیریم برای تخمین دو بخش رابطه‌ی زیر تا target به صورت نویزی نباشد. این باعث می‌شود که target Network وزن‌های پایداری داشته باشد در نتیجه target پایدار داریم و باعث پایداری می‌گردد. رابطه‌ی بروزرسانی وزن‌ها چنین است.

$$L_t = \left[r_t + \gamma \max_{\acute{a}} \underbrace{\hat{F}(\bar{x}_{t+1}, \bar{w}, \acute{a})}_{target\ Network} - \underbrace{F(\bar{x}_t, \bar{w}, a)}_{Policy\ Network} \right]$$

$$\acute{\theta} = \tau\theta + (1-\tau)\theta$$

۲- بخش امتیازی

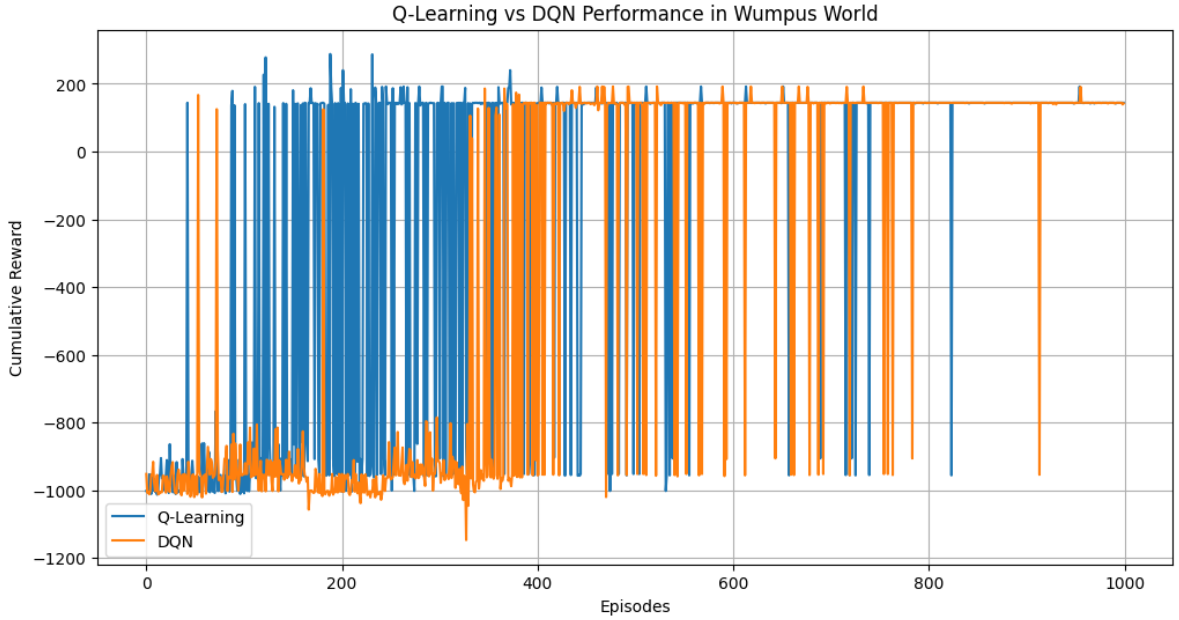
در این بخش با در نظر گرفتن شلیک به چهار جهت و گرفتن ۵۰ امتیاز در صورت کشته شدن wumpus نشان داده شده است. به این منظور یک تابع برای شلیک به اطراف تعریف شده است.

```
def shoot_arrow(state, direction):
    x, y = state
    wx, wy = wumpus
    if direction == 'up':
      if wx == x and wy < y:
        return 50 # Reward for killing the Wumpus

    elif direction == 'down':
      if wx == x and wy > y:
        return 50 # Reward for killing the Wumpus

    elif direction == 'left':
      if wy == y and wx < x:
        return 50 # Reward for killing the Wumpus

    elif direction == 'right':
      if wy == y and wx > x:
```

```
        return 50 # Reward for killing the Wumpus

    return 0  # No reward if the Wumpus is not hit
```

سپس درون توابع مربوط به الگوریتم‌ها آن تابع را فراخوانی و در امتیاز تاثیر می‌گذارد.

```
# Check if shooting the arrow is required and calculate additional reward
        shoot_reward = shoot_arrow(state, action)
        total_reward += shoot_reward
```



شکل ۱-۲ نمایش تجمع پاداش‌ها

```
Average reward per episode for Q-Learning: -124.02
Average reward per episode for DQN: -298.564
Q-Learning performed better.
```

شکل ۲-۲ میانگین پاداش

```
Q-learning agent achieved consistent performance after 264 episodes
DQN agent achieved consistent performance after 433 episodes
```

شکل ۳-۲ دستیابی به عملکرد ثابت