Zajęcia laboratoryjne 9: Implementacja algorytmów uczenia ze wzmocnieniem w Pythonie

Cel zajęć

Celem zajęć jest zapoznanie się z podstawowymi algorytmami uczenia ze wzmocnieniem (Reinforcement Learning, RL) oraz ich implementacja w języku Python. W szczególności skoncentrujemy się na algorytmie Q-learning oraz zastosowaniu środowiska `OpenAI Gym`.

Wymagania wstępne

- Znajomość podstaw programowania w Pythonie.
- Podstawowa wiedza z zakresu uczenia maszynowego.
- Zainstalowane biblioteki: `numpy`, `matplotlib`, `gym` (zalecana wersja: 0.26.2).

Teoria

Uczenie ze wzmocnieniem (ang. Reinforcement Learning, RL) to dziedzina uczenia maszynowego, w której agent uczy się podejmować decyzje poprzez interakcje ze środowiskiem. Agent wykonuje akcje, które wpływają na stan środowiska i otrzymuje nagrody, które służą jako informacja zwrotna o jakości jego działań. Celem agenta jest maksymalizacja sumy nagród w dłuższym okresie czasu.

Proces decyzyjny w RL jest formalizowany jako Proces Decyzyjny Markowa (Markov Decision Process, MDP), który definiuje się za pomocą czwórki:

- **S** - zbiór stanów,
- **A** - zbiór akcji,
- **P** - funkcja przejścia: $P(s'|s,a)$, czyli prawdopodobieństwo przejścia do stanu s' po wykonaniu akcji a w stanie s,
- **R** - funkcja nagrody: $R(s,a)$, czyli nagroda za wykonanie akcji a w stanie s.

Agent w każdym kroku obserwuje aktualny stan s, wybiera akcję a, otrzymuje nagrodę r i przechodzi do nowego stanu s'. Strategia działania agenta nazywana jest *polityką* (ang. policy), często oznaczaną jako $\pi(s)$.

Uczenie polega na znalezieniu optymalnej polityki π^*, która maksymalizuje oczekiwaną sumę skumulowanych nagród:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1}$$

Gdzie $\gamma \in [0,1]$ to współczynnik dyskontujący — określa, jak bardzo agent ceni przyszłe nagrody względem bieżących.

Funkcja wartości akcji

Funkcja wartości akcji (ang. *action-value function*), oznaczana jako $Q(s,a)$, określa oczekiwaną skumulowaną nagrodę, jaką agent otrzyma, zaczynając w stanie s, wykonując akcję a, a następnie postępując zgodnie z polityką π. Formalnie:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right] \tag{2}$$

gdzie:

- s — stan początkowy,
- a — akcja podjęta w stanie s,

- r_t — nagroda w chwili t,
- $\gamma \in [0, 1)$ — współczynnik dyskontujący,
- π — polityka, zgodnie z którą agent wybiera dalsze akcje.

Funkcja $Q^\pi(s, a)$ pomaga ocenić, które akcje są bardziej opłacalne w danym stanie. W algorytmach takich jak Q-learning celem jest wyuczenie optymalnej funkcji $Q^*(s, a)$, która spełnia równanie Bellmana:

$$Q^*(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right] \tag{3}$$

Q-learning to popularny, bezmodelowy algorytm uczenia ze wzmocnieniem, który uczy się funkcji wartości akcji (Q-funkcji):

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right] \tag{4}$$

Gdzie:

- $Q(s, a)$ to oszacowanie wartości wykonania akcji a w stanie s,
- α to współczynnik uczenia (learning rate),
- γ to współczynnik dyskontujący (discount factor),
- r to nagroda otrzymana po wykonaniu akcji a w stanie s,
- s' to nowy stan po przejściu.

Q-learning jest algorytmem *off-policy*, ponieważ aktualizuje wartość na podstawie maksymalnej przewidywanej nagrody niezależnie od aktualnej polityki działania.

Podstawy oraz możliwości biblioteki gym

Biblioteka `gym` od OpenAI to popularne narzędzie do budowania i testowania algorytmów uczenia ze wzmocnieniem. Udostępnia zestandaryzowany interfejs do środowisk symulacyjnych, co umożliwia szybkie prototypowanie i porównywanie różnych podejść.

Każde środowisko `gym` posiada metody:

- `reset()` – inicjuje nowy epizod i zwraca stan początkowy,

- `step(action)` – wykonuje akcję, zwraca: nowy stan, nagrodę, flagi zakończenia (`terminated`, `truncated`) oraz informacje diagnostyczne,
- `render()` – umożliwia wizualizację przebiegu symulacji,
- `action_space` oraz `observation_space` – definiują przestrzeń akcji i obserwacji.

Przykładowe dostępne środowiska to m.in.: `CartPole-v1`, `FrozenLake-v1`, `MountainCar-v0`, `Taxi-v3`. Możliwe jest także tworzenie własnych środowisk.

Od wersji 0.26 funkcja `step()` zwraca pięć wartości (dodano flagę `truncated`). Użycie `render_mode="human"` umożliwia tworzenie okien graficznych z animacją środowiska.

Minimalny działający przykład: FrozenLake-v1

Przykład w pełni deterministycznego środowiska, które działa bez graficznego interfejsu, idealny do testów i obserwacji działania agenta:

```
import gym

env = gym.make("FrozenLake-v1", is_slippery=False)
state, info = env.reset()

done = False
while not done:
    print("Stan:", state)
    action = env.action_space.sample()
    next_state, reward, terminated, truncated, info = env.
       ↪ step(action)

    print("␣->␣akcja:", action)
    print("␣->␣nowy␣stan:", next_state)
    print("␣->␣nagroda:", reward)
    print("␣->␣koniec?", terminated or truncated)
    print("-" * 20)

    done = terminated or truncated
    state = next_state
```

Co zobaczysz:

- numer stanu (np. $0 \ldots 15$),

- wykonane akcje,
- nagrody (tylko 1 przy sukcesie),
- jasną informację, czy epizod się zakończył.

Co to za środowisko?

- FrozenLake-v1
- plansza 4×4,
- agent ma dotrzeć z pola START do GOAL,
- bez is_slippery porusza się zawsze dokładnie w kierunku wybranej akcji.

Jak przetestować agenta?

- To środowisko jest idealne do nauki Q-learningu — a z is_slippery=False bardzo szybko zaczyna osiągać 100% skuteczności.

Przykład działania: Taxi-v3 (tekstowe środowisko logiczne)

Środowisko `Taxi-v3` to prosty przykład gry logicznej, w której agent steruje taksówką poruszającą się po siatce 5x5. Zadaniem agenta jest odebranie pasażera z jednego z czterech wyznaczonych punktów (oznaczonych literami R, G, B, Y) i przewiezienie go do miejsca docelowego.

Agent może wykonywać następujące akcje:

- 0 – przemieszczenie się na południe (dół),
- 1 – przemieszczenie się na północ (góra),
- 2 – przemieszczenie się na wschód (prawo),

- 3 – przemieszczenie się na zachód (lewo),
- 4 – podniesienie pasażera (pickup),
- 5 – odstawienie pasażera (dropoff).

Taksówka może zostać ukarana za niepoprawne akcje (np. próbę podniesienia pasażera w złym miejscu), a nagradzana za poprawne dostarczenie.

Poniższy kod demonstruje działanie tego środowiska z użyciem tekstowego trybu renderowania:

```
import gym

env = gym.make("Taxi-v3", render_mode="ansi")

state, info = env.reset()
done = False

while not done:
    print(env.render())
    action = env.action_space.sample()
    next_state, reward, terminated, truncated, info = env.
       ↪ step(action)

    print("Akcja:", action)
    print("Nagroda:", reward)
    print("-" * 20)

    done = terminated or truncated
    state = next_state
```

Opis wyjścia: Program wypisuje kolejne stany planszy ASCII, które pokazują lokalizację taksówki (litera T), pozycje punktów odbioru/odstawienia (R, G, B, Y), oraz wykonane akcje. Po każdej akcji zwracana jest nagroda i sprawdzany jest koniec epizodu. Środowisko kończy się sukcesem, gdy pasażer zostanie poprawnie dostarczony na miejsce docelowe.

To środowisko jest bardzo dobre do testowania Q-learningu z tablicową reprezentacją funkcji Q.

CartPole-v1

Środowisko `CartPole-v1` to jedno z najczęściej stosowanych środowisk testowych w Reinforcement Learning. Zadaniem agenta jest balansowanie słupkiem

zamocowanym na wózku poruszającym się wzdłuż poziomej osi. Agent może przesuwać wózek w lewo lub w prawo, a jego celem jest utrzymanie słupka w pionie jak najdłużej.

Szczegóły środowiska:

- **Stan (observation space):** czteroelementowy wektor ciągły: pozycja wózka, prędkość wózka, kąt słupka, prędkość kątowa.
- **Akcje (action space):** {0 – w lewo, 1 – w prawo}.
- **Nagroda:** +1 za każdy krok, w którym słupek nie przewróci się i wózek nie opuści przedziału.
- **Zakończenie epizodu:** gdy kąt słupka przekroczy dopuszczalny zakres lub wózek opuści pole działania.

Przykład kodu:

```
import gym
import time

env = gym.make("CartPole-v1", render_mode="human")
obs, info = env.reset()

for _ in range(200):
    action = env.action_space.sample()
    obs, reward, terminated, truncated, info = env.step(
       ↪ action)
    done = terminated or truncated
    time.sleep(0.02)
    if done:
        obs, info = env.reset()
```

Opis działania: Pętla wykonuje losowe akcje i pokazuje animację środowiska. Po zakończeniu epizodu następuje reset. Program działa w czasie rzeczywistym dzięki funkcji `time.sleep()`.

To środowisko może być używane do nauki strategii za pomocą Q-learningu, chociaż ze względu na ciągłą przestrzeń stanów, wymagane jest jej zdyskretyzowanie lub zastosowanie funkcji aproksymujących.

Jak tworzyć własne środowisko?

Tworzenie własnego środowiska w bibliotece `gym` jest przydatne, gdy chcemy przetestować algorytmy RL w niestandardowych warunkach. Własne środowisko

należy zaimplementować jako klasę dziedziczącą po `gym.Env`, implementującą cztery główne metody:

- `__init__()` – inicjalizacja środowiska, definiowanie przestrzeni akcji i obserwacji,
- `reset()` – resetuje środowisko i zwraca stan początkowy,
- `step(action)` – wykonuje akcję i zwraca: nowy stan, nagrodę, flagi zakończenia, informacje pomocnicze,
- `render()` – (opcjonalnie) wizualizacja stanu.

Przykład minimalnego środowiska:

```
import gym
from gym import spaces
import numpy as np

class MyEnv(gym.Env):
    def __init__(self):
        self.observation_space = spaces.Discrete(5)
        self.action_space = spaces.Discrete(2)
        self.state = 0

    def reset(self, seed=None, options=None):
        self.state = 0
        return self.state, {}

    def step(self, action):
        self.state = (self.state + 1) % 5
        reward = 1 if self.state == 0 else 0
        terminated = self.state == 0
        return self.state, reward, terminated, False, {}

    def render(self):
        print(f"Stan: {self.state}")
```

Aby użyć środowiska, można je zarejestrować:

```
from gym.envs.registration import register

register(
    id="MyEnv-v0",
    entry_point="__main__:MyEnv",
```

```
)

env = gym.make("MyEnv-v0")
```

Ten kod tworzy proste środowisko, w którym agent przechodzi przez 5 stanów cyklicznie, a nagroda przyznawana jest tylko za powrót do stanu 0.

Zadania

Zadanie 1: Instalacja środowiska

Zainstaluj bibliotekę `gym`:

```
pip install gym==0.26.2
pip install gym[classic_control]
```

Zadanie 2: Eksploracja `CartPole-v1`

Uruchom podstawowe środowisko i zaobserwuj zachowanie agenta.

```
import gym
import time

env = gym.make("CartPole-v1", render_mode="human")
obs, info = env.reset()
for _ in range(100):
    action = env.action_space.sample()
    obs, reward, terminated, truncated, info = env.step(
        ↪ action)
    done = terminated or truncated
    time.sleep(0.01)
    if done:
        obs, info = env.reset()
```

Zadanie 3: Implementacja Q-learningu na `FrozenLake-v1`

Zaimplementuj tablicowego agenta Q-learning dla środowiska `FrozenLake-v1`. Użyj tablicy Q o wymiarach [liczba stanów] x [liczba akcji]. Uwzględnij eksplorację $\epsilon - greedy$.

```
import numpy as np
import gym

env = gym.make("FrozenLake-v1", is_slippery=False)
n_states = env.observation_space.n
n_actions = env.action_space.n

Q = np.zeros((n_states, n_actions))

epsilon = 0.1
alpha = 0.1
gamma = 0.99
episodes = 1000

rewards = []

for episode in range(episodes):
    state, info = env.reset()
    done = False
    total_reward = 0
    while not done:
        if np.random.uniform(0, 1) < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(Q[state])

        next_state, reward, terminated, truncated, info = env
            ↪ .step(action)
        done = terminated or truncated

        Q[state, action] = Q[state, action] + alpha * (
            reward + gamma * np.max(Q[next_state]) - Q[state,
                ↪  action]
        )
        state = next_state
        total_reward += reward

    rewards.append(total_reward)
```

Zadanie 4: Ewaluacja

Oblicz średnią nagrodę co 100 epizodów i przedstaw wykres.

```
import matplotlib.pyplot as plt
```

```
avg_rewards = [np.mean(rewards[i:i+100]) for i in range(0,
    ↪ episodes, 100)]
plt.plot(avg_rewards)
plt.xlabel("Kolejne setki epizodów")
plt.ylabel("Średnia nagroda")
plt.title("Średnia nagroda agenta Q-learning")
plt.show()
```

Dodatkowe zadanie (dla chętnych)

Spróbuj zaimplementować algorytm SARSA i porównaj jego działanie z Q-learningiem. Możesz także eksperymentować z parametrem ϵ, $uczącymsię$

Spróbuj zastosować Q-learning w innym środowisku, np. Taxi-v3 lub MountainCar-v0. Wykonaj analizę ewaluacyjną analogicznie jak powyżej.

Zadania do realizacji: dotyczą opracowania własnego środowiska na podstawie gym, implementacji oraz ewaluacji

1. Zmodyfikuj środowisko tak, aby nagroda pojawiała się tylko co 3 kroki.
2. Wprowadź losowe przejścia między stanami (np. z 10% prawdopodobieństwem błędna akcja).
3. Dodaj stan końcowy po 10 krokach i oblicz średnią nagrodę agenta losowego.
4. Zmienna długość epizodu - epizod kończy się po losowej liczbie kroków z przedziału [5, 10].
5. Wprowadź akcję specjalną, która resetuje środowisko, ale kosztuje punkt kary.
6. Zwiększ przestrzeń stanów i sprawdź wpływ na czas nauki agenta Q-learning.
7. Wprowadź akcję penalizowaną - nagroda ujemna za wybranie niewłaściwej.
8. Dodaj mechanizm ,,bonusu'' - jeśli agent wykona poprawny ciąg trzech akcji, otrzymuje dodatkowe punkty.

9. Stwórz środowisko dwuwymiarowe (stan jako para liczb).
10. Rozszerz renderowanie o prostą reprezentację ASCII (np. pasek postępu).