Projekt - Podstawy Gymnasium

Krzysztof Ferda, Krzysztof Czerenko

13 Maja 2025

1 Część I - Problem dyskretny

1.1 Wybrany problem

Wybrany przez nas problem z dyskretną przestrzenią obserwacji to `Cliff Walking v0` ze zbioru `Toy Text`.

1.2 Implementacja

Kluczową rolę w naszej implementacji odgrywa klasa `CliffWalkingAgent`:

```
class CliffWalkingAgent:

    def __init__(
            self,
            env: gym.Env,
            learning_rate : float,
            initial_epsilon: float,
            epsilon_decay: float,
            final_epsilon: float,
            discount_factor: float = 0.9
            ):

        self.env = env
        self.q_values = defaultdict(lambda: np.zeros(env.action_space.n))

        self.lr = learning_rate
        self.discount_factor = discount_factor

        self.epsilon = initial_epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon

        self.training_error = []
```

Przyjmuje ona `env` - środowisko, `learning_rate` - tempo uczenia,
`discount_factor` - współczynnik dyskontowy, oraz zmienne związane z epsilonem: `initial_epsilon` - startowy epsilon, `epsilon_decay` - formuła określająca zmianę epsilona w kolejnych epokach, oraz `final_epsilon` - najmniejszy możliwy epsilon.

Klasa ta ma następujące metody:

`get_action` - wybiera i zwraca akcję:

```
    def get_action(self, obs: tuple[int, int, bool]) -> int:

            if np.random.random() < self.epsilon:
                return self.env.action_space.sample()
            else:
                return int(np.argmax(self.q_values[obs]))
```

W zależności od tego czy wylosowana liczba jest mniejsza czy większa od epsilon, wybór ten może zostać dokonany losowo albo greedy - czyli najlepsza akcja.

`update` - uaktualnia słownik q wartości:

```
    def update(
        self,
        obs: tuple[int, int, bool],
        action: int,
        reward: float,
        terminated: bool,
        next_obs: tuple[int, int, bool],
    ):
        future_q_value = (not terminated) * np.max(self.q_values[next_obs])
        temporal_difference = (
            reward + self.discount_factor * future_q_value - self.q_values[obs][action]
        )

        self.q_values[obs][action] = (
            self.q_values[obs][action] + self.lr * temporal_difference
        )
        self.training_error.append(temporal_difference)
```

Więcej o jej dokładnym działaniu jest w sekcji Algorytm.

`decay_epsilon` - zmiejsza epsilon z kolejnymi epokami:

```
    def decay_epsilon(self):
        self.epsilon = max(self.final_epsilon, self.epsilon - self.epsilon_decay)
```

Samo uczenie odbywa się w pętli for:

```
for episode in tqdm(range(n_episodes)):
    obs, info = env.reset()
    done = False

    while not done:
        action = agent.get_action(obs)
        next_obs, reward, terminated, truncated, info = env.step(action)
```

```
        agent.update(obs, action, reward, terminated, next_obs)

        done = terminated or truncated
        obs = next_obs

    agent.decay_epsilon()
```

1.3 Algorytm - Q-learning

Algorytm uczy się zapamiętując tzw. **Q-wartości**- Q(s, a), czyli oczekiwaną nagrodę, jeżeli w stanie s wykona akcję a. Wartości te zapisywane są w słowniku.

Aby algorytm nie był liniowy, a agent też eksplorował stosuje się dodatkowo strategie **epsilon-greedy**. Za każdym razem podczas podejmowania decyzji losowana jest liczba z zakresu [0,1] i jeżeli jest ona mniejsza niż epsilon to agent wykonuje losową akcję, a inaczej najlepszą ze słownika Q-wartości. Z czasem epsilon się zmiejsza, co sprawia, że na początku agent głównie eskploruje a w końcowej fazie preferuje znane najlepsze rozwiązania.

Jednym z najważniejszych kroków algorytmu jest aktualizacja Q-wartości.
Najpierw trzeba obliczyć klasyczna **różnicę czasową (temporal difference, TD)**:

$$TD = r + \gamma \cdot \max_a Q(s', a) - Q(s, a)$$

Potem aktualizujemy wartość Q:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot TD$$

Gdzie:

- r – nagroda
- γ – współczynnik dyskontowy
- α – tempo uczenia się
- s, a, s' – stan, akcja, nowy stan
- $\max_a Q(s', a)$ – przewidywana wartość przyszłego najlepszego działania

Współczynnik dyskontowy wpływa na to na ile agent ceni przyszłość ponad natychmiastowe zyski. Im większy, tym bardziej agent wybiera przyszłe nagrody.

1.4 Przeprowadzone eksperymenty

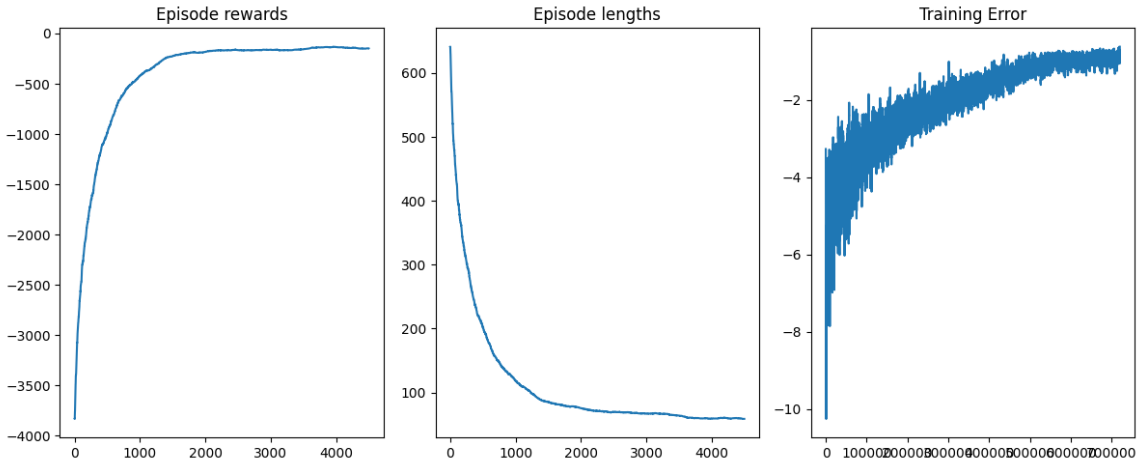
Uczenie zostało przeprowadzone trzykrotnie z różnymi parametrami.

1.4.1 Test 1

Współczynniki:

```
learning_rate = 0.001
n_episodes = 5_000
start_epsilon = 0.7
epsilon_decay = start_epsilon / (n_episodes / 2)
final_epsilon = 0.2
```

Wynik:

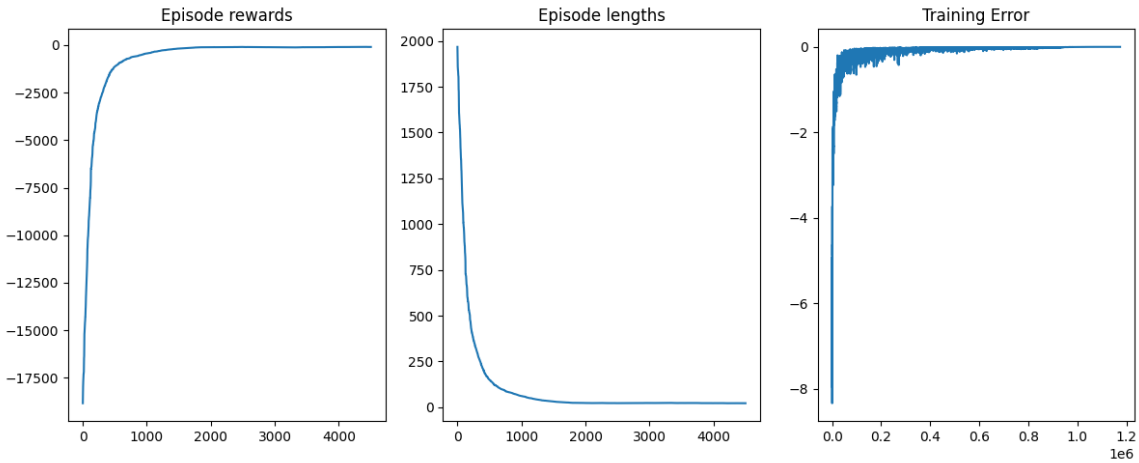


1.4.2 Test 2

Współczynniki:

```
learning_rate = 0.02
n_episodes = 5_000
start_epsilon = 1
epsilon_decay = start_epsilon / (n_episodes / 2)
final_epsilon = 0.2
```

Wynik:

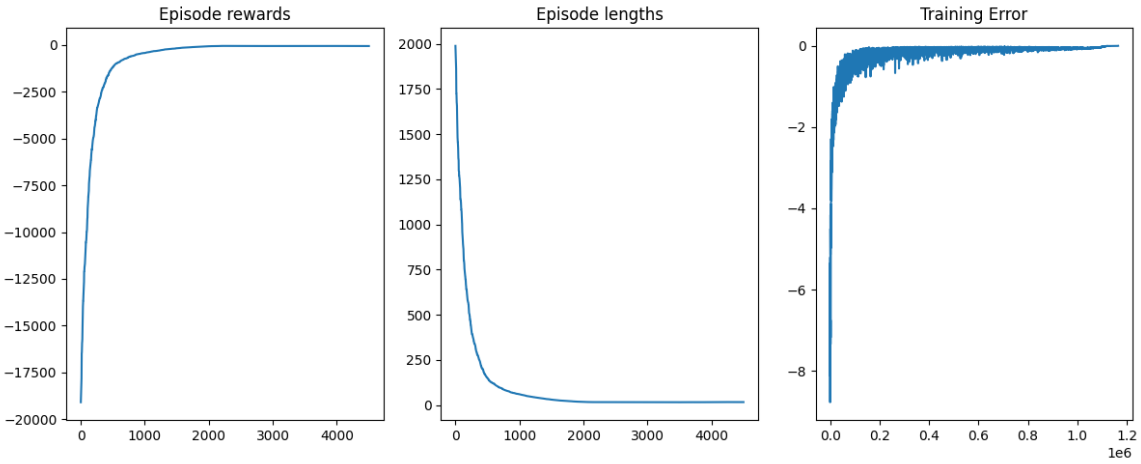


1.4.3 Test 3

Współczynniki:

```
learning_rate = 0.01
n_episodes = 5_000
start_epsilon = 1.0
epsilon_decay = start_epsilon / (n_episodes / 2)
final_epsilon = 0.1
```

Wynik:



1.5 Wyniki uczenia

Agent z testu nr 3 poradził sobie najlepiej i już około epoki 2000 zaczął zdobywać praktycznie maksymalne wyniki, cały czas zmiejszając swój błąd.

2 Cześć II - Problem ciągły

2.1 Opis problemu

W projekcie wykorzystano klasyczny algorytm Q-learning zaadaptowany do środowiska MountainCarContinuous-v0, które posiada ciągłą przestrzeń obserwacji. Aby zastosować Q-learning (który działa w przestrzeniach dyskretnych), zarówno przestrzeń stanów (pozycja i prędkość), jak i akcji została zdyskretyzowana na odpowiednią liczbę przedziałów (bins).

Agent zapamiętuje swoje doświadczenia w postaci tablicy Q, mapującej zdyskretyzowane stany do wartości Q dla wszystkich możliwych działań. W każdej iteracji uczy się poprzez aktualizację tej tablicy, zgodnie z klasycznym wzorem Q-learningu. Wybór akcji odbywa się według strategii -greedy – z prawdopodobieństwem wybierana jest akcja losowa, a w przeciwnym wypadku najlepsza znana.

Wartość stopniowo maleje z każdą epizodą, co pozwala agentowi przejść od eksploracji do eksploatacji. W każdej turze agent:

- Ocenia aktualny stan,
- Wybiera akcję,
- Otrzymuje nagrodę i obserwuje nowy stan,
- Aktualizuje tablicę Q,
- Powtarza proces aż do zakończenia epizodu.

Agent uczy się na podstawie wielu epizodów, a po zakończeniu treningu może rekomendować optymalne działania dla dowolnego stanu na podstawie wyuczonych wartości Q.

2.2 Agent z discount_factor=1.0

2.3 Porównanie 3 agentów o różnych parametrach