KOLEJKA GÓRSKA - DOKUMENTACJA KOŃCOWA

Matuszewski Kamil, Matuszewski Maciej

POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH

UCZENIE MASZYNOWE

Spis treści

1. Wstęp

Dokument stanowi sprawozdanie z drugiego etapu realizacji projektu „Kolejka górska" polegającej na przygotowaniu implementacji Projektu Końcowego na rzecz przedmiotu Uczenie Maszynowe.

2. Krótkie streszczenie założeń z Projektu Wstępnego

Projekt „Kolejka górska" ma na celu naukę agenta sterującego wagonikiem w symulacji kolejki górskiej z wykorzystaniem techniki uczenia ze wzmocnieniem (Q-Learning). Celem jest nauczenie agenta osiągnięcia szczytu wzniesienia poprzez optymalne decyzje w środowisku dostarczonym przez bibliotekę Gymnasium. Agent jest umieszczony w dolinie, a celem jest dotarcie na górę, pokonując przeszkody, przy minimalnym koszcie nagrody.

Q-Learning jest techniką off-policy, w której agent uczy się na podstawie doświadczeń, aby podejmować najlepsze decyzje, a jego celem jest maksymalizacja skumulowanej nagrody. Środowisko jest modelowane poprzez parametry: pozycję i prędkość wagonika, a agent wykonuje trzy możliwe akcje: rusz w lewo, nie ruszaj się, rusz w prawo. Dla każdej pary stan-akcja, agent przechowuje wartość w tablicy Q, która jest aktualizowana na podstawie otrzymanych nagród.

Niezwykle istotnym aspektem realizacji tego projektu jest zrozumienie i odpowiednie wykorzystanie algorytmu Q-learning opierającego się na iteracyjnej aktualizacji wartości $Q(s, a)$ według wzoru (opisanego dokładniej w sekcji **4. Precyzyjny opis algorytmów oraz opis zbiorów danych**):

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

Projekt zakłada eksperymenty z różnymi parametrami algorytmu (współczynnik uczenia, dyskontowania, eksploracji) oraz porównanie wyników z losową strategią. W ramach eksperymentów sprawdzamy także wpływ losowej inicjalizacji stanu. Projekt opiera się na dynamicznie generowanych danych, które są wykorzystywane do aktualizacji Q-Tablicy.

Założenia obejmują także plan eksperymentów, który ma na celu przetestowanie wyuczonych Q-Tablic w przypadkach standardowych.

3. Pełen opis funkcjonalny

3.1. Kod programu

Realizacją naszego projektu jest program Python:

```
import random
import gymnasium as gym
import numpy as np
import pickle
import matplotlib.pyplot as plt
import seaborn as sns

def run(num_episodes, is_training = True, render = False, initial_p=None):
   # Initialize environment
   env = gym.make('MountainCar-v0', render_mode='human' if render else None)
   env = env.unwrapped
   env.metadata['render_fps']= 1
   position_states = np.linspace(-1.2, 0.6, 20)
   velocity_states = np.linspace(-0.07, 0.07, 20)
   if is_training:
        Q_table = np.zeros((len(position_states), len(velocity_states), 3))  # initialize Q-table
with zeros
   else:
       f = open("Q_table.pkl", "rb")
       Q_table = pickle.load(f)
       f.close()

   learning_rate = 0.4
   discount_rate = 0.9
   epsilon = 1  # Start with full exploration

   episode_rewards = []

   for episode in range(num_episodes):
       if initial_p:
            env.state = np.array([initial_p, 0])
       else:
            env.reset()

       state_p = np.digitize(env.state[0], position_states)
       state_v = np.digitize(env.state[1], velocity_states)

       total_reward = 0
       done = False

       while not done and total_reward > -1000:  # Run until the environment signals termination or
the car does too many actions
           # Epsilon-greedy action selection
           if random.uniform(0, 1) < epsilon and is_training:
               action = env.action_space.sample()
           else:
               action = np.argmax(Q_table[state_p, state_v, :])

           next_state, reward, done, _, _ = env.step(action)
           next_state_p = np.digitize(next_state[0], position_states)
           next_state_v = np.digitize(next_state[1], velocity_states)

           if is_training:
           # Update Q-table using Q-Learning formula
               Q_table[state_p, state_v, action] += learning_rate * (
                        reward + discount_rate * np.max(Q_table[next_state_p, next_state_v, :]) -
Q_table[state_p, state_v, action]
```

```
                )

            # Move to the next state
            state_p, state_v = next_state_p, next_state_v
            total_reward += reward  # Accumulate reward for the episode

        # Decay epsilon
        epsilon = max(epsilon - 2/num_episodes, 0)

        episode_rewards.append(total_reward)

        # Log results
        print(f"Episode: {episode + 1}, Total Reward: {total_reward:.2f}, Epsilon: {epsilon:.2f}")

    env.close()

    # Save Q-table and plot
    if is_training:
        f = open("Q_table.pkl", "wb")
        pickle.dump(Q_table, f)
        f.close()
        # Save Q-table visualization
        plot_q_table(Q_table)

    # Save rewards per episode plot
    plot_rewards_per_episode(episode_rewards)

    # Save moving average plot
    plot_moving_average(episode_rewards)

def plot_q_table(Q_table):
    fig, ax = plt.subplots(figsize=(10, 10))
    heatmap = sns.heatmap(np.max(Q_table, axis=2), cmap="viridis", ax=ax)
    ax.set_title("Q-table Visualization")
    ax.set_xlabel("Velocity State")
    ax.set_ylabel("Position State")
    fig = heatmap.get_figure()
    fig.savefig('mountain_car_q_table.png')
    plt.close(fig)

def plot_moving_average(episode_rewards):
    window_size = 100
    mean_rewards = []
    for i in range(len(episode_rewards)):
        start_idx = max(0, i - window_size + 1)
        mean_rewards.append(np.mean(episode_rewards[start_idx:i+1]))
    plt.plot(mean_rewards)
    plt.title('Mean Rewards per Episode (Moving Average)')
    plt.xlabel('Episode')
    plt.ylabel('Mean Reward')
    plt.savefig('mountain_car_moving_average.png')
    plt.close()

def plot_rewards_per_episode(episode_rewards):
    plt.figure()
    plt.title('Rewards per Episode')
    plt.plot(episode_rewards)
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.savefig('mountain_car_rewards_per_episode.png')
    plt.close()
```

```
if __name__ == '__main__':
    # For training
    run(5000, is_training=True, render=False, initial_p=-0.5)
    # For testing (with a trained Q-table)
    #run(10, is_training=False, render=False, initial_p= -0.5)
```

Program oparty jest na algorytmie Q-Learning, który jest metodą uczenia maszynowego wzmocnionego, stosowaną do problemów decyzyjnych. W tym przypadku model uczy się, jak sterować samochodem w MountainCar-v0, by osiągnąć jak najwyższą nagrodę. Na początku model jest w pełni eksploracyjny (wykorzystuje *epsilon-greedy*), a z biegiem treningu coraz bardziej polega na swojej Q-Tablicy, by podejmować optymalne decyzje.

3.2. Struktura programu

Program został podzielony na kilka głównych sekcji:

3.2.1. Inicjalizacja środowiska i Q-Tablicy

W kodzie, na początku każdej iteracji epizodu, tworzymy środowisko MountainCar-v0 za pomocą funkcji gym.make(). Ponadto, w przypadku treningu (parametr is_training=True), inicjalizujemy Q-Tablicę o wymiarach odpowiadających podzielonym stanom pozycji i prędkości, a także akcji (3 możliwe akcje w tym przypadku). Jeśli jesteśmy w trybie testowym, wczytujemy wcześniej wytrenowaną Q-Tablicę z pliku.

3.2.2. Dyskretyzacja stanów

Fragment:

```
position_states = np.linspace(-1.2, 0.6, 20)
velocity_states = np.linspace(-0.07, 0.07, 20)
```

Dyskretyzujemy przestrzeń stanów samochodu, dzieląc zakresy pozycji i prędkości na 20 równych podzbiorów. Decyzja ta wynika z faktu, że zarówno pozycja, jak i prędkość są liczbami ciągłymi, a ich dyskretyzacja pozwala na skuteczne uzupełnianie Q-Tablicy. Liczbę 20 podziałów dobraliśmy eksperymentalnie. W oparciu o doświadczenia wywnioskować można, że:

- Zbyt duża liczba podziałów powoduje nadmierną złożoność i wolniejszą naukę modelu.
- Zbyt mała liczba podziałów sprawia, że model staje się zbyt ogólny, przez co nie jest w stanie nauczyć się skutecznych decyzji.

3.2.3. Inicjalizacja parametrów algorytmu Q-Learning

```
learning_rate = 0.4
discount_rate = 0.9
epsilon = 1.0
```

Ustawiamy współczynniki odpowiedzialne za współczynnik uczenia (learning rate), współczynnik dyskontowania (discount rate) oraz poziom eksploracji (epsilon). Wartości te są wybierane doświadczalnie (więcej w sekcji **5. Raport z przeprowadzonych testów oraz wnioski**), a ich rolą jest kontrolowanie procesu nauki i eksploracji.

3.2.4. Główna pętla epizodów

Dla każdego epizodu, model resetuje środowisko, uzyskuje początkowy stan, a następnie przechodzi przez pętlę decyzyjną, aż osiągnie stan końcowy (flaga done lub zbyt duża liczba kroków):

```
for episode in range(num_episodes):
    state = env.reset()[0]
    state_p = np.digitize(state[0], position_states)
    state_v = np.digitize(state[1], velocity_states)
```

Wartości pozycji i prędkości są konwertowane na indeksy dyskretnych stanów, co pozwala na skuteczną pracę z Q-Tablicą.

3.2.5. Wybór akcji i aktualizacja Q-Tablicy

Pętla wewnętrzna (*while not done and total_reward* > -1000) odpowiada za eksplorację i aktualizację Q--Tablicy:

- Algorytm korzysta z *epsilon-greedy* dla wyboru akcji
- Po wykonaniu akcji, model otrzymuje nagrodę (w naszym przypadku bardziej karę, a program dąży do uzyskania jej jak najmniejszej) i aktualizuje Q-Tablicę zgodnie z formułą:

```
  Q_table[state_p, state_v, action] += learning_rate * (
    reward + discount_rate * np.max(Q_table[next_state_p, next_state_v, :]) - Q_table[state_p,
state_v, action])
```

- Zmniejszanie wartości epsilon w trakcie treningu pozwala na stopniowe zmniejszanie eksploracji na rzecz eksploatacji (z Q-Tablicą)

3.2.6. Zakończenie treningu

Po zakończeniu treningu zapisujemy wytrenowaną Q-Tablicę do pliku:

```
if is_training:
    f = open("Q_table.pkl", "wb")
    pickle.dump(Q_table, f)
    f.close()
```

3.2.7. Generowanie wykresów

W programie generujemy trzy wykresy:

- Heatmapa Q-Tablicy – wizualizuje najlepsze akcje w danej pozycji i prędkości. Obrazuje to, które stany zostały najlepiej wyuczone przez model.
- Nagrody per epizod – przedstawia zmiany w nagrodach osiąganych w kolejnych epizodach, co pozwala na ocenę postępu modelu.
- Średnia ruchoma nagród – wykres pokazujący średnią wartość nagród z ostatnich 100 epizodów, co daje lepszy obraz ogólnych trendów w procesie treningowym.

3.2.8. Testowanie modelu

Po wytrenowaniu modelu, Q-Tablica może być załadowana i używana w trybie testowym ustawiając flagę *is_training=False*.

3.2.9. Obsługa działania

Program może działać w trybach:

- *is_training* - tryb uczenia modelu i aktualizacji Q-Tablicy
- *render* - tryb renderowania wagonika (poprzez *env.metadata['render_fps']= 0* ustalamy „prędkość" renderowania).

Poprzez *num_episodes* definiujemy liczbę epizodów, a poprzez *initial_p* pozycję startową (jeśli puste pozycja startowa jest ustalana zgodnie z dokumentacją środowiska).

3.3. Testy jednostkowe

Poniższy program wykonuje 7 testów jednostkowych:

```
import unittest
import numpy as np
import gymnasium as gym
from unittest.mock import patch, MagicMock
import os
import pickle
import random

# Import functions from the main file
from mountain_car import run, plot_q_table, plot_rewards_per_episode, plot_moving_average

class TestMountainCar(unittest.TestCase):
    def setUp(self):
        """
        Setup method run before each test.
        Initializes the environment and defines state discretization parameters.
```

```
    """
    self.env = gym.make('MountainCar-v0')
    # Define state space discretization parameters - adding 1 to include upper bound
    self.position_states = np.linspace(-1.2, 0.6, 21)[:-1]  # Use 20 bins
    self.velocity_states = np.linspace(-0.07, 0.07, 21)[:-1]  # Use 20 bins

def tearDown(self):
    """
    Cleanup method run after each test.
    """
    self.env.close()
    # Clean up generated files
    files_to_remove = ['Q_table.pkl',
                       'mountain_car_q_table.png',
                       'mountain_car_moving_average.png',
                       'mountain_car_rewards_per_episode.png']
    for file in files_to_remove:
        if os.path.exists(file):
            os.remove(file)

def test_q_table_initialization(self):
    """
    Test if Q-table is properly initialized and saved.
    """
    # Run a short training session
    run(num_episodes=10, is_training=True, render=False)

    # Check if Q-table file was created
    self.assertTrue(os.path.exists('Q_table.pkl'))

    # Load and verify Q-table dimensions
    with open('Q_table.pkl', 'rb') as f:
        Q_table = pickle.load(f)
    self.assertEqual(Q_table.shape, (20, 20, 3))

def test_reward_calculation(self):
    """
    Test if rewards are calculated correctly.
    """
    rewards = []
    # Mock plotting functions to focus on reward calculation
    with patch('mountain_car.plot_rewards_per_episode'), \
         patch('mountain_car.plot_q_table'), \
         patch('mountain_car.plot_moving_average'):
        run(num_episodes=1, is_training=True, render=False, initial_p=-0.5)

    # Check if rewards are within reasonable bounds
    self.assertTrue(all(reward > -1000 for reward in rewards))

def test_state_discretization(self):
    """
    Test if continuous state space is correctly discretized.
    """
    # Test discretization for various positions within bounds
    test_positions = [-1.1, 0.5, 0.0]  # Safe values within bounds
    for pos in test_positions:
        state_p = np.digitize(pos, self.position_states)
        # Verify that discretized state is within bounds
        self.assertTrue(0 <= state_p < 20)  # Now explicitly checking against 20

@patch('matplotlib.pyplot.savefig')
def test_plotting_functions(self, mock_savefig):
```

```
        """
        Test if visualization functions work correctly.
        """
        # Prepare sample data for plotting
        Q_table = np.random.random((20, 20, 3))
        episode_rewards = [random.random() for _ in range(100)]

        # Test plotting functions
        plot_q_table(Q_table)
        plot_rewards_per_episode(episode_rewards)
        plot_moving_average(episode_rewards)

        # Verify that plots were saved
        self.assertEqual(mock_savefig.call_count, 2)

    def test_training_vs_testing_mode(self):
        """
        Test differences between training and testing modes.
        """
        # First run a short training session
        run(num_episodes=10, is_training=True, render=False)

        # Then test with deterministic behavior
        with patch('random.uniform', return_value=1.0):  # Force deterministic behavior
            run(num_episodes=1, is_training=False, render=False)

    def test_edge_cases(self):
        """
        Test edge cases and boundary conditions.
        """
        # Test with safe positions near the edges
        edge_positions = [-1.1, 0.5]  # Adjusted to be within bounds
        for pos in edge_positions:
            with patch('mountain_car.plot_rewards_per_episode'), \
                 patch('mountain_car.plot_q_table'), \
                 patch('mountain_car.plot_moving_average'):
                run(num_episodes=1, is_training=True, render=False, initial_p=pos)

    def test_learning_progress(self):
        """
        Test if the agent shows learning progress over episodes.
        """
        with patch('mountain_car.plot_rewards_per_episode'), \
             patch('mountain_car.plot_q_table'), \
             patch('mountain_car.plot_moving_average'):
            run(num_episodes=100, is_training=True, render=False)

        # Load Q-table and verify it's not all zeros
        with open('Q_table.pkl', 'rb') as f:
            Q_table = pickle.load(f)
        self.assertFalse(np.all(Q_table == 0))

if __name__ == '__main__':
    unittest.main()
```

Testy przebiegają pomyślnie:

```
Ran 7 tests in 6.376s

OK
```

4. Precyzyjny opis algorytmów oraz opis zbiorów danych

W naszym przypadku opis algorytmu był już stosunkowo precyzyjny w Projekcie Wstępnym, dlatego zdecydowaliśmy się na jego ponowne przywołanie dodając przy tym bezpośrednie odwołania do naszego programu.

4.1. Q-Learning

Q-Learning to jedna z metod uczenia ze wzmocnieniem, która umożliwia agentowi podejmowanie optymalnych decyzji w środowisku na podstawie doświadczeń. Jest to metoda off-policy, co oznacza, że uczy się optymalnej polityki działania niezależnie od bieżącej polityki, którą stosuje agent.

4.1.1. Q-Learning, a nasz projekt:

1. **Agent i środowisko**
   - Agent, który wykonuje akcje w środowisku to wagonik.
   - Środowisko reaguje, zmieniając swój stan i przekazując agentowi nagrodę (lub karę).
   - Środowisko generowane jest przez bibliotekę Gymnasium.

2. **Stany** (S)
   - Każda możliwa sytuacja, w jakiej może znaleźć się agent, jest określana jako stan. W *Observartion Space* wagonika znajdują się: pozycja oraz prędkość.
   - W naszym środowisku wagonik działa zgodnie z regułami:
     - $\text{velocity}_{t+1} = \text{velocity}_t + (\text{action} - 1) \cdot \text{force} - \cos(3 \cdot \text{position}_t) \cdot \text{gravity}$
     - $\text{position}_{t+1} = \text{position}_t + \text{velocity}_{t+1}$

     gdzie *force* $= 0.001$ i *gravity* $= 0.0025$. Zderzenia na obu końcach są nieelastyczne, a prędkość jest ustawiona na 0 po zderzeniu ze ścianą. Pozycja jest ograniczona do zakresu $[-1.2, 0.6]$, a prędkość jest ograniczona do zakresu $[-0.07, 0.07]$.
   - Stany w programie zmieniane są poprzez wykonanie *env.step(action)*.

3. **Akcje** (A)
   - Możliwe działania, które agent może podjąć w danym stanie. W przypadku w *Action Space* wagonika znajduja się: 0 = rusz w lewo, 1 = nie ruszaj się, 2 = rusz się w prawo.
   - Wybór akcji jest zdefiniowany poprzez *epsilon-greedy*.

4. **Nagrody** (R)
   - Liczby określające jakość wyniku danej akcji w danym stanie (może być pozytywna, negatywna lub zerowa). W przypadku wagonika każdy ruch kosztuje go -1 nagrody.
   - W celach optymalizacyjnych dodaliśmy warunek sprawdzający czy w danym epizodzie wagonik nie wykonał większej niż określonej ilości kroków.

5. **Q-Tablica**
   - Tablica przechowująca wartości Q dla każdej pary stan-akcja.
   - Wartość $Q(s, a)$ reprezentuje oczekiwaną skumulowaną nagrodę, jaką agent otrzyma w przyszłości, wykonując akcję a w stanie s, a następnie postępując optymalnie.
   - Q-Tablica jest serializowana i zapisywana po treningu. Tablica może posłużyć do działania wagonika bez przeprowadzania treningu.

6. **Epoki/epizody**
   - Kompletna sekwencja stanów, działań i nagród, która kończy się w momencie osiągnięcia stanu końcowego. W przypadku wagonika epizod kończy osiągnięcie określonej liczby ujemnych nagród lub osiągnięcie celu.

4.1.2. Algorytm Q-Learning

Proces polega na iteracyjnej aktualizacji wartości $Q(s, a)$ według następującego wzoru:

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

Gdzie:

- α: Współczynnik uczenia (learning rate) – kontroluje szybkość aktualizacji.
- γ: Współczynnik dyskontowania (discount factor) – określa znaczenie przyszłych nagród.

- $R(s, a)$: Natychmiastowa nagroda za wykonanie akcji a w stanie s.
- $\max_{a'} Q(s', a')$ Maksymalna oczekiwana nagroda w kolejnym stanie s'.

Działanie krok po kroku:
1. **Inicjalizacja Q-Tablicy**: Na początku wartości $Q(s, a)$ są zerowe.
2. **Wybór akcji**: Agent wybiera akcję w danym stanie, za pomocą polityki ε-greedy:
   - Jeśli losowa liczba jest mniejsza niż epsilon, wybierana jest losowa akcja (*env.action_space.sample()*), co jest formą eksploracji
   - W przeciwnym razie, wybierana jest akcja, która daje najwyższą wartość w Q-Tablicy (*np.argmax(Q_table[state_p, state_v, :])*), co jest formą eksploatacji.
3. **Wykonanie akcji**: Agent wykonuje akcję a, przechodząc do nowego stanu s' i otrzymuje nagrodę $R(s, a)$.
4. **Aktualizacja Q-Tablicy**: Wartość $Q(s, a)$ jest aktualizowana według wzoru.
5. **Powtarzanie procesu**: Pętla trwa do osiągnięcia określonego kryterium zakończenia (np. stabilne wartości Q lub osiągnięcie celu przez agenta).

4.2. Opis zbioru danych

W kontekście Q-Learingu, który jest algorytmem off-policy, nasz projekt nie opiera się na tradycyjnych zbiorach danych. Zamiast tego, model działa poprzez eksplorację i eksploatację środowiska, generując dane w trakcie procesu nauki. Zbiór danych w tym przypadku stanowi Q-Tablica, która jest stopniowo uzupełniana w miarę interakcji z otoczeniem. Dopiero po zakończeniu procesu uczenia, Q-Tablica staje się źródłem wiedzy, na podstawie której model podejmuje decyzje. W związku z tym, przed stworzeniem i wytrenowaniem tej tablicy, nie istnieje zdefiniowany, stały zbiór danych, z którego model by korzystał.

5. Raport z przeprowadzonych testów oraz wnioski

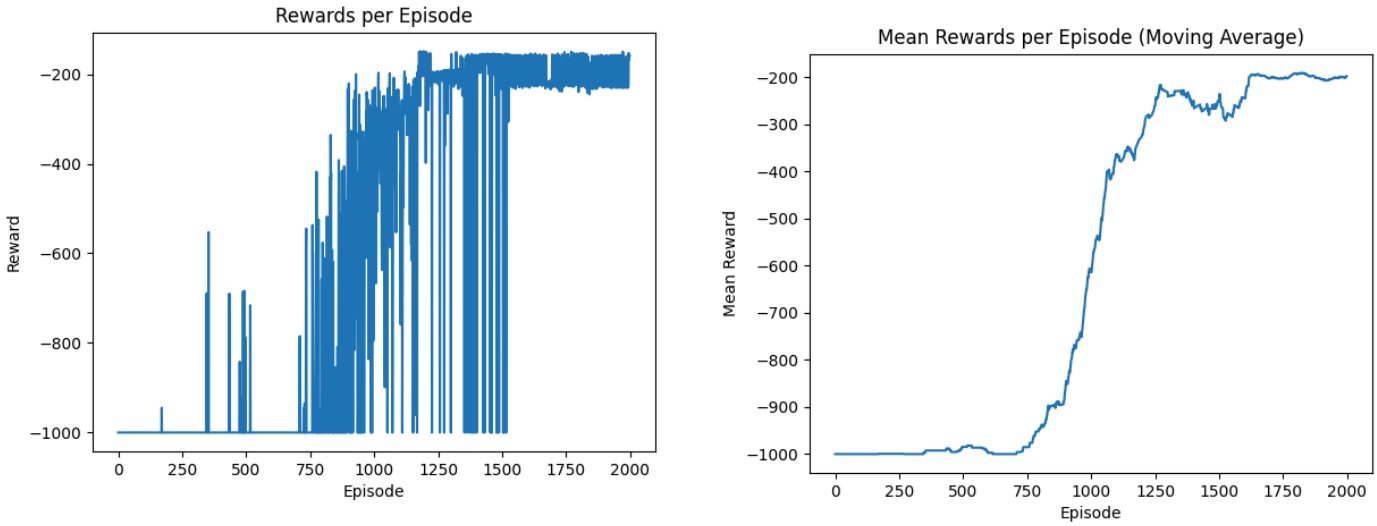
5.1. Eksperyment z konfiguracją parametrów algorytmu Q-Learning

Celem tego eksperymentu jest sprawdzenie wpływu parametrów na nasz program.

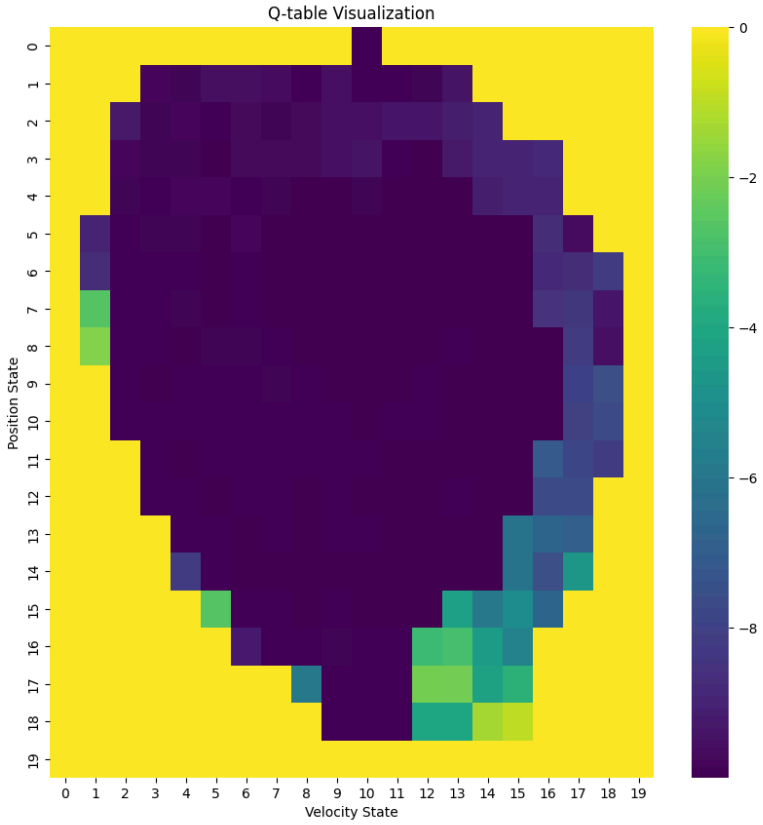
5.1.1. Współczynnik uczenia (α)

Przetestujemy dwie skrajne wartości α:
- $\alpha = 0.95$, $\gamma = 0.9$, $\varepsilon = 1$, $num\_episodes = 2000$, $total\_rewards < -1000$
- $\alpha = 0.05$, $\gamma = 0.9$, $\varepsilon = 1$, $num\_episodes = 2000$, $total\_rewards < -1000$

Dla $\alpha = 0.95$ uzyskujemy:

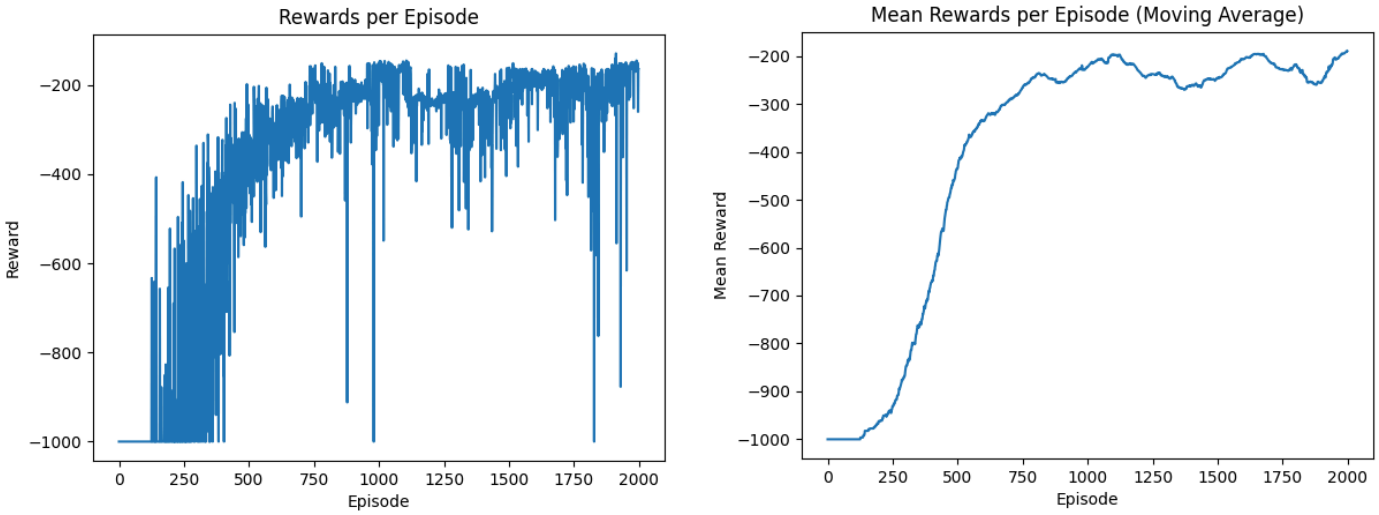


Rysunek 1: Wykresy dla $\alpha = 0.95$

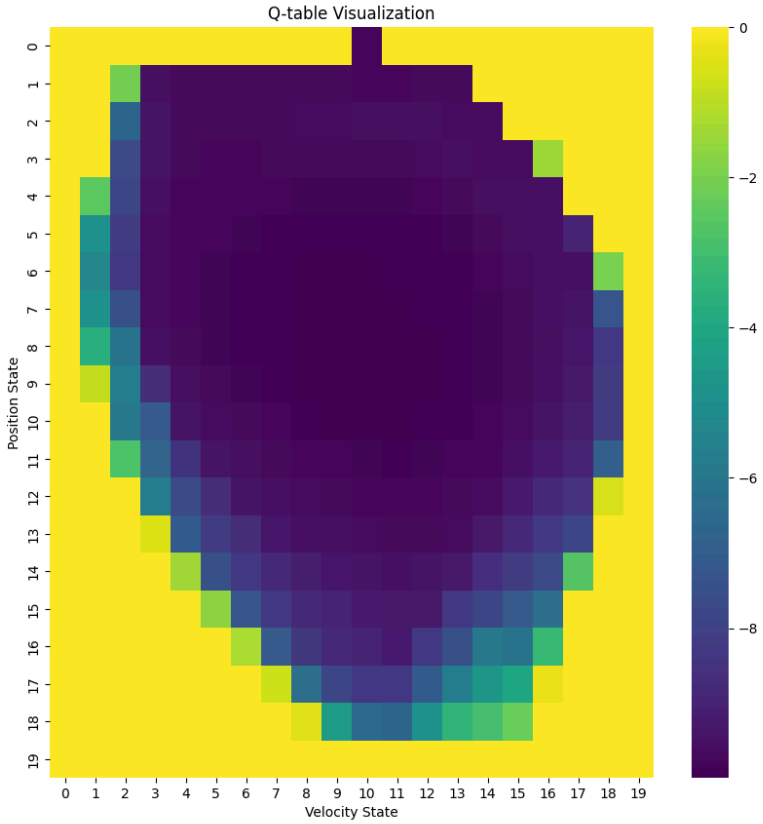


Rysunek 2: Heatmapa dla $\alpha = 0.95$

Dla $\alpha = 0.05$ uzyskujemy:



Rysunek 3: Wykresy dla $\alpha = 0.05$



Rysunek 4: Heatmapa dla $\alpha = 0.05$

Wnioski: Współczynnik uczenia kontroluje, w jakim stopniu nowe informacje (nagrody) powinny wpływać na aktualizację wartości Q dla danego stanu i akcji.

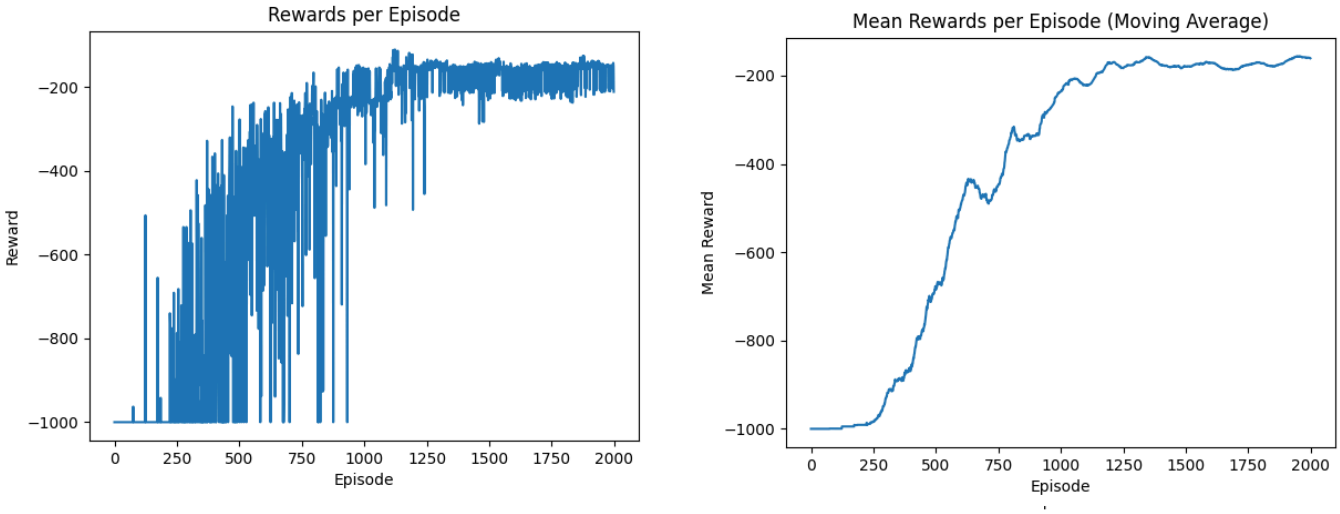
- Z wykresów jednoznacznie wynika, że szybciej osiąga cel model z $\alpha = 0.05$.
- Z heatmap Q-Tablicy widać, że dla $\alpha = 0.05$ model bardziej podejmuje decyzje na podstawie eksploatacji, a dla $\alpha = 0.95$ widać częstsze eksploracje.
- α bliskie 1:
  - Wartość α bliska 1 oznacza, że agent będzie szybko aktualizował swoje wartości Q, przyznając dużą wagę nowym informacjom, nawet jeśli są one sprzeczne z dotychczasową wiedzą. W naszym przypadku można zauważyć, że model do około 1500 epizodu ma trudności z nieprzekroczeniem 1000 kroków, aczkolwiek później model radzi sobie świetnie osiągając bardzo dobre wyniki.
  - Model może lepiej adaptować się do nowych sytuacji (być gotowym na więcej sytuacji), kosztem dłuższego oczekiwania na dobre rezultaty.
- α bliskie 0:
  - Jeśli α jest bliskie zeru, agent będzie częściej ignorować nowe doświadczenia i będzie polegać na dotychczasowej wiedzy. W naszym przypadku (dość prostego celu nauki) niski α umożliwia średnio szybciej uzyskanie zadowalających rezultatów (bo już od około 750 epizodu), aczkolwiek kosztem gorzej wypadających epizodów w drugiej połowie eksperymentu (widać, że czasem modelowi nawet nie udawało się dojść do celu, bo nie poświęcił dużo czasu na naukę z eksploracji).
  - Uczenie staje się bardziej stabilne, ale mniej dynamiczne. W przypadku bardziej złożonych problemów model mógłby mieć trudności z poprawnym dostosowywaniem się do nowych sytuacji (w drugiej połowie eksperymentu wciąż są sytuacje w których wagonik nie osiąga celu, bo gubi się w niektórych sytuacjach).

5.1.2. Współczynnik dyskontowania (γ)

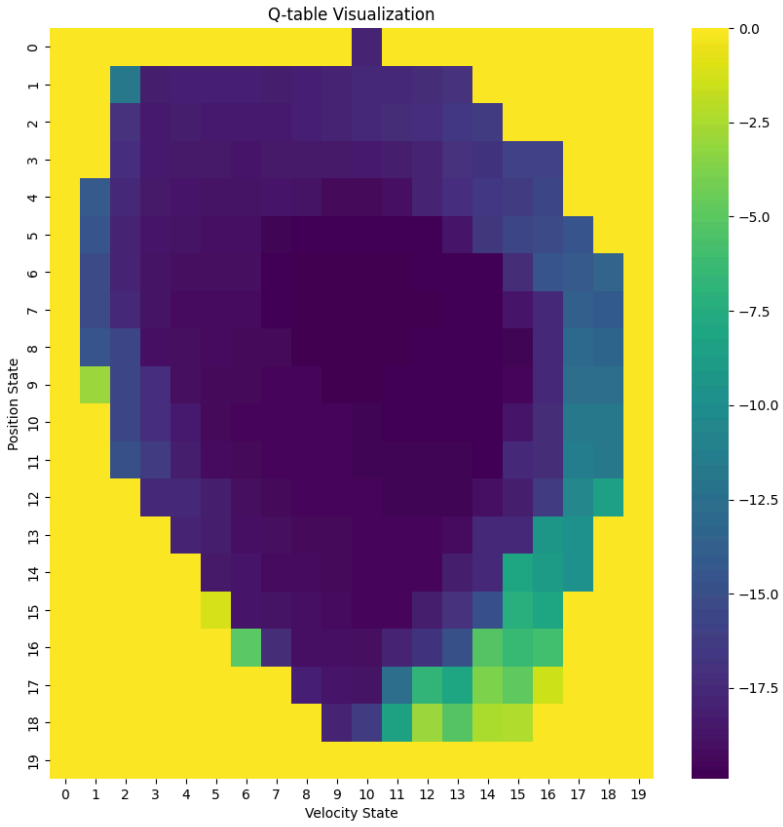
Przetestujemy dwie skrajne wartości γ:

- $\alpha = 0.3$, $\gamma = 0.95$, $\varepsilon = 1$, $num\_episodes = 2000$, $total\_rewards < -1000$
- $\alpha = 0.3$, $\gamma = 0.05$, $\varepsilon = 1$, $num\_episodes = 2000$, $total\_rewards < -1000$

Dla $\gamma = 0.95$ uzyskujemy:

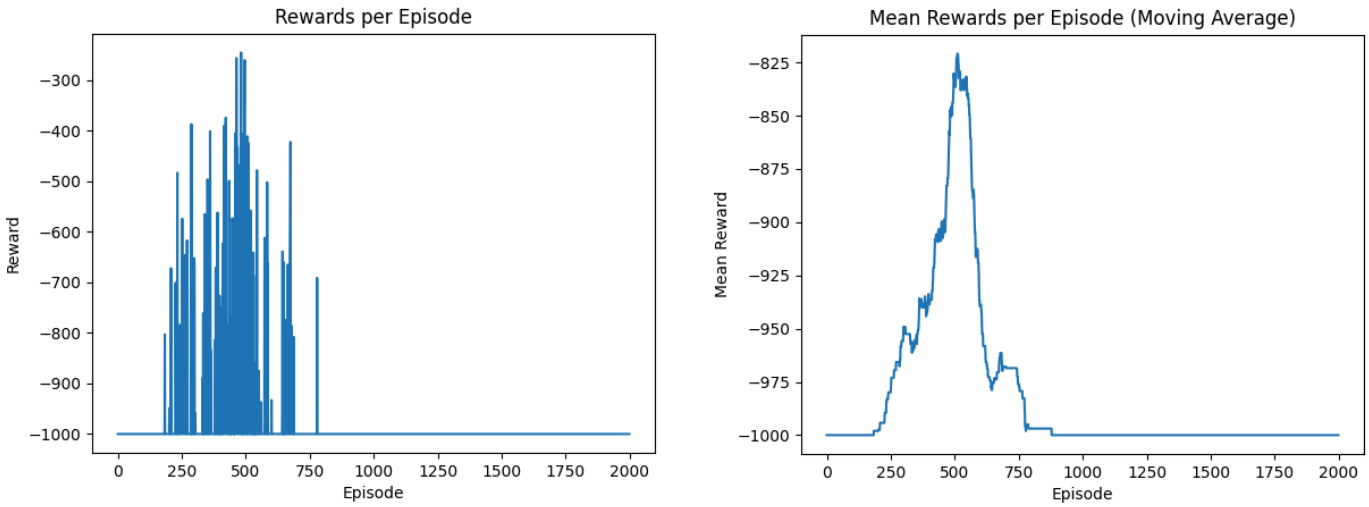


Rysunek 5: Wykresy dla $\alpha = 0.95$

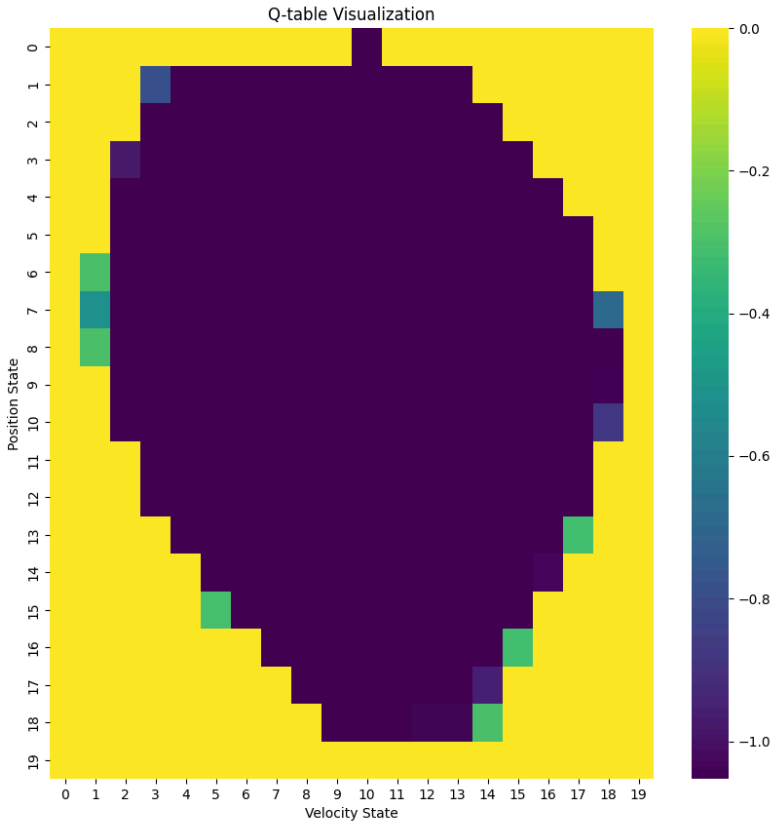


Rysunek 6: Heatmapa dla $\alpha = 0.95$

Dla $\gamma = 0.05$ uzyskujemy:



Rysunek 7: Wykresy dla $\gamma = 0.05$



Rysunek 8: Heatmapa dla $\gamma = 0.05$

Wnioski: Współczynnik dyskontowania kontroluje, jak bardzo agent uwzględnia przyszłe nagrody w swoich decyzjach.

- γ bliskie 1:
  - Gdy γ jest bliskie 1, agent przywiązuje dużą wagę do przyszłych nagród, biorąc pod uwagę długoterminowe konsekwencje swoich działań. Dzięki temu agent staje się bardziej „przewidujący” i podejmuje decyzje uwzględniające potencjalne nagrody w przyszłości.
- Może to prowadzić do lepszej optymalizacji w zadaniach, które wymagają długoterminowego planowania, ale także może sprawić, że agent będzie wolniej adaptował się do zmian w środowisku.
- γ bliskie 0:

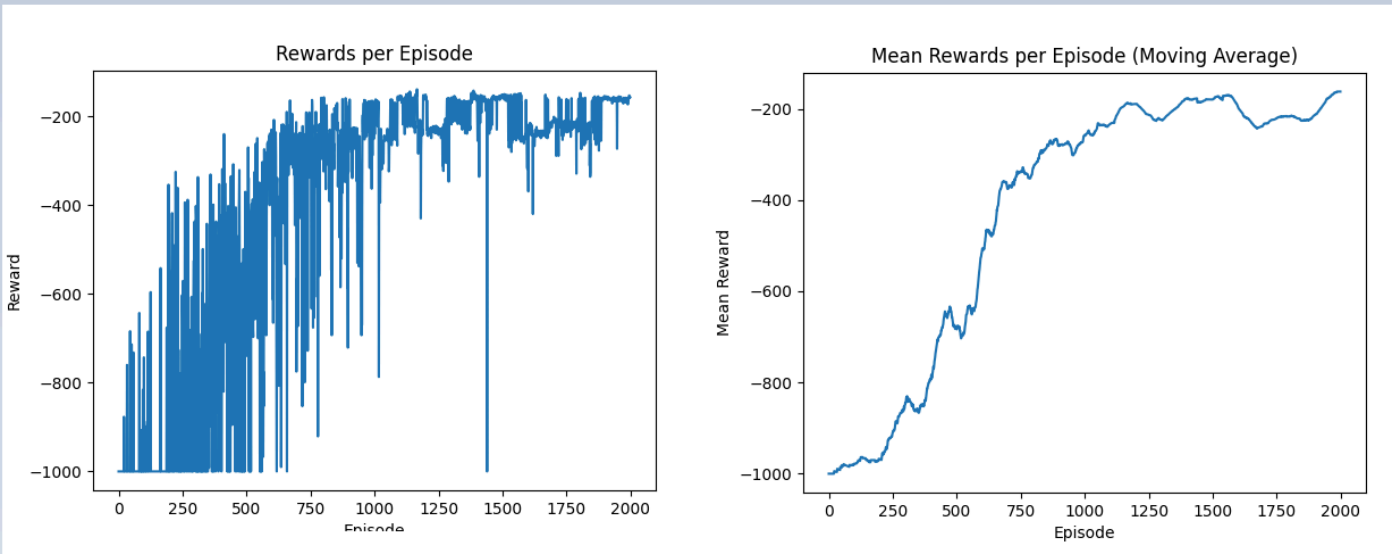
- Gdy γ jest bliskie zeru, agent skupia się głównie na krótkoterminowych nagrodach. W takim przypadku agent nie będzie się zbytnio martwić o długoterminowe konsekwencje swoich działań, co może być przydatne w prostszych zadaniach, gdzie celem jest szybka nagroda.
- Uczy się agresywnie dążyć do krótkoterminowych zysków, co może prowadzić do nieoptymalnych decyzji w dłuższej perspektywie (a w naszym przypadku do nieosiągania celu).

5.1.3. Współczynnik ekspolracji (ε)

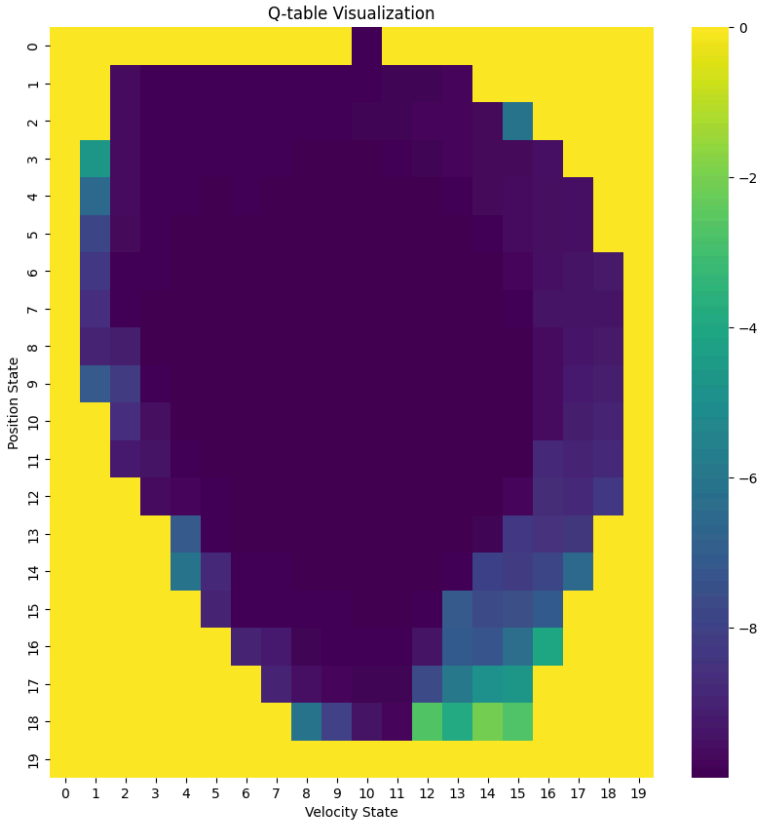
Przetestujemy dwie skrajne wartości ε:

- $\alpha = 0.3$, $\gamma = 0.9$, $\varepsilon = 0.9$, $num\_episodes = 2000$, $total\_rewards < -1000$
- $\alpha = 0.3$, $\gamma = 0.9$, $\varepsilon = 0.1$, $num\_episodes = 2000$, $total\_rewards < -1000$

Dla $\varepsilon = 0.9$:

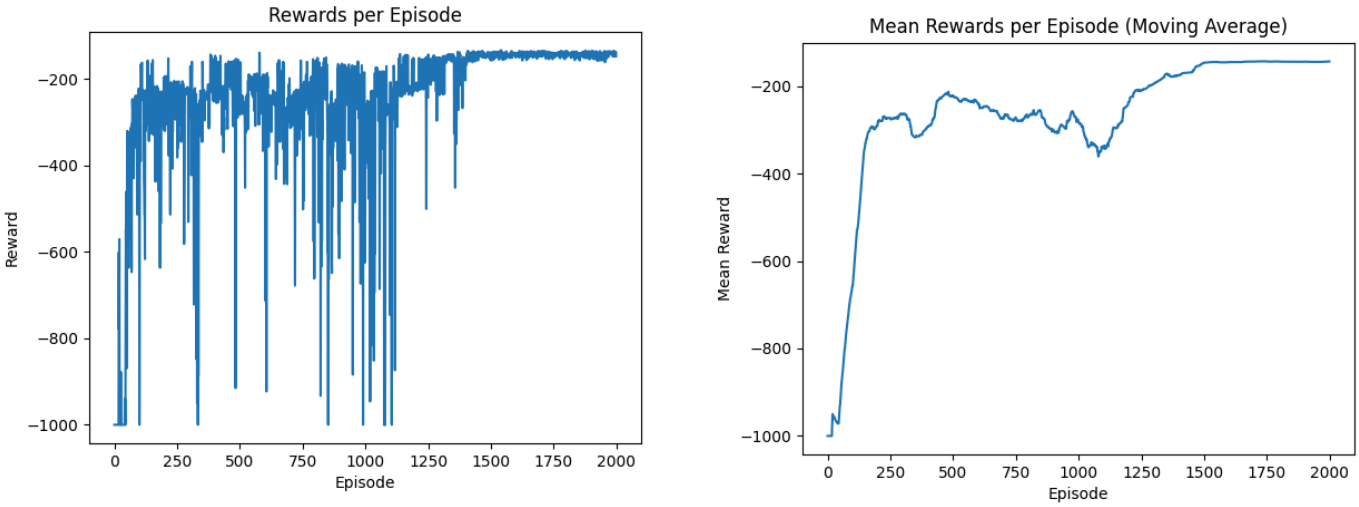


Rysunek 9: Wykresy dla $\varepsilon = 0.9$

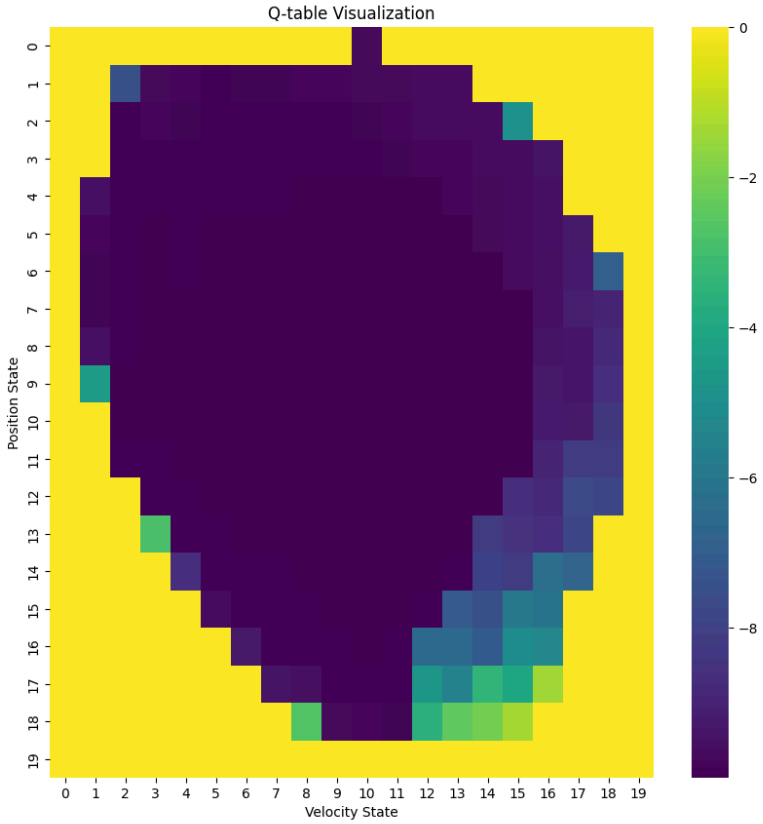


Rysunek 10: Heatmapa dla $\varepsilon = 0.9$

Dla $\varepsilon = 0.1$:



Rysunek 11: Wykresy dla $\varepsilon = 0.1$



Rysunek 12: Heatmapa dla $\varepsilon = 0.1$

Wnioski: Współczynnik ε w algorytmie Q-Learning jest kluczowy w procesie eksploracji i eksploatacji, ponieważ kontroluje, jak często agent będzie wybierał akcje losowo, a jak często będzie korzystał z dotychczasowej wiedzy zawartej w Q-Tablicy.

- ε bliskie 1(większa eksploracja)
  - Plusy:
    - Pomaga agentowi zbierać szeroką gamę doświadczeń, co może być pomocne w nieznanych lub trudnych środowiskach.
    - Unika utknięcia w lokalnych minimach, ponieważ agent nie jest skazany tylko na wybór najlepszej akcji z dotychczasowych doświadczeń.
  - Minusy:
    - Może powodować wolniejsze uczenie się, ponieważ agent częściej wybiera losowe akcje, które mogą nie przynosić dużych nagród.
- ε bliskie 0 (większa eksploatacja)
  - Plusy:
    - Agent szybciej podejmuje optymalne decyzje na podstawie dotychczasowej wiedzy (Q-Tablicy), co może prowadzić do szybszego osiągania dobrych wyników.
- Minusy:
  - Może prowadzić do utknięcia, jeśli agent nie zbada innych, potencjalnie lepszych możliwości (brak eksploracji).

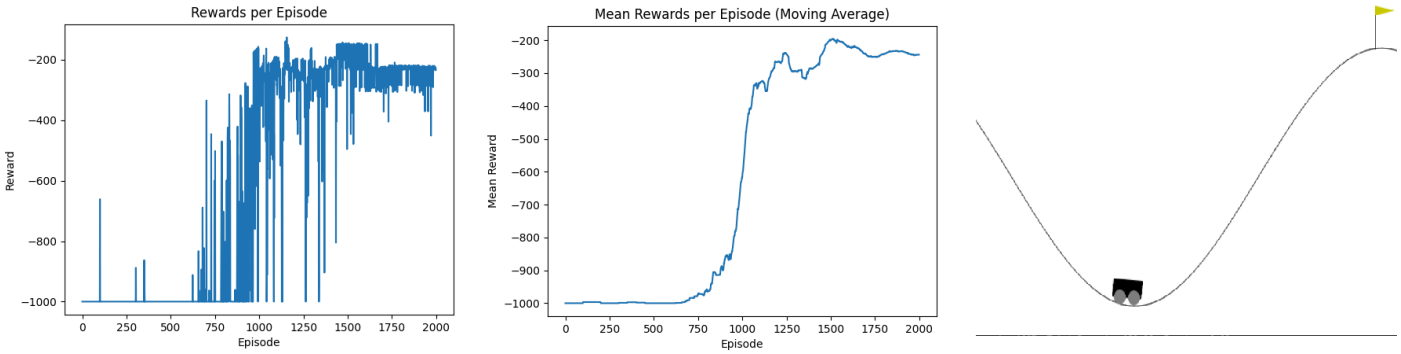
5.2. Wpływ początkowych stanów

Celem tego eksperymentu będzie zbadanie, czy inicjalizacja pozycji wagonika wpływa na czas nauki. Podczas tego badania inicjalna pozycja wagonika będzie wybierana. Wartościami mierzonymi podczas tego eksperymentu będą średnia liczba epizodów potrzebnych do osiągnięcia sukcesu oraz stabilność wyników w przeprowadzanych testach.

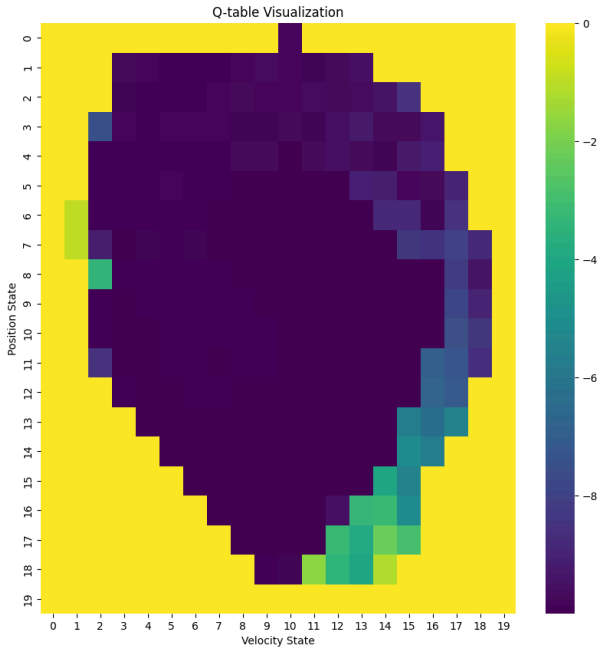
Wartości parametrów są następujące:

- $\alpha = 0.9$, $\gamma = 0.9$, $\varepsilon = 1$, $num\_episodes = 2000$, $total\_rewards < -1000$

Na sam początek pozycja startowa:

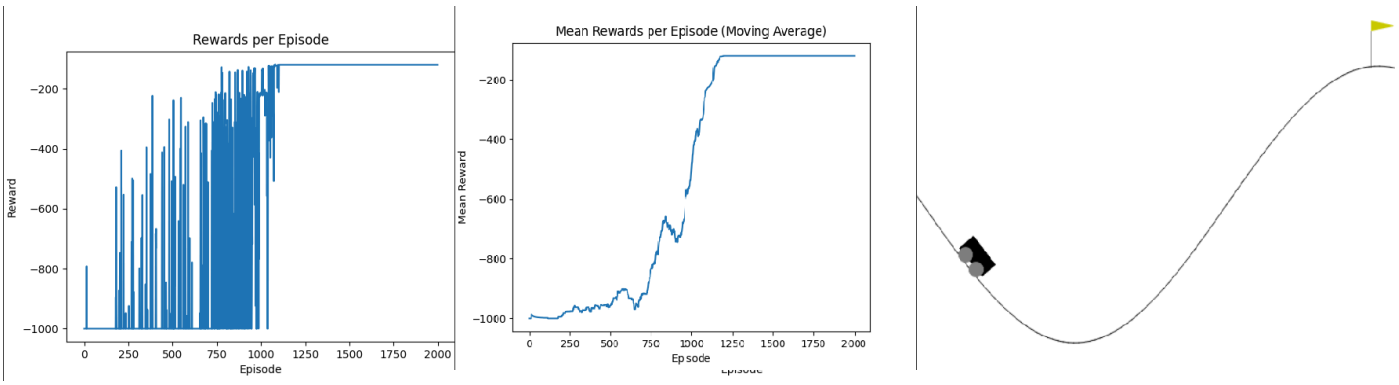


Rysunek 13: Grafiki dla pozycji startowej $= -0.53$

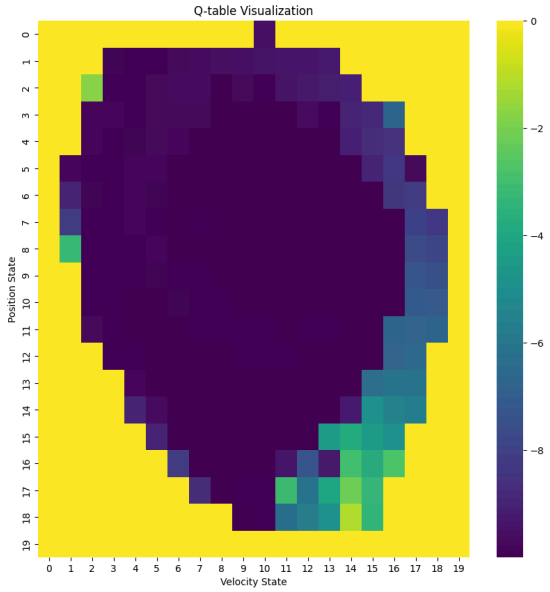


Rysunek 14: Heatmapa dla pozycji startowej $= -0.53$

Pozycja -0.9:

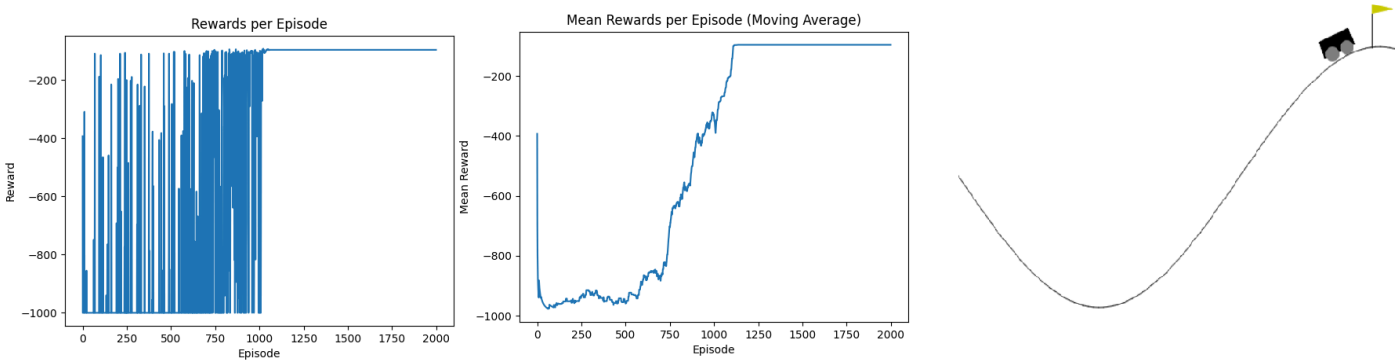


Rysunek 15: Grafiki dla pozycji startowej $= -0.9$

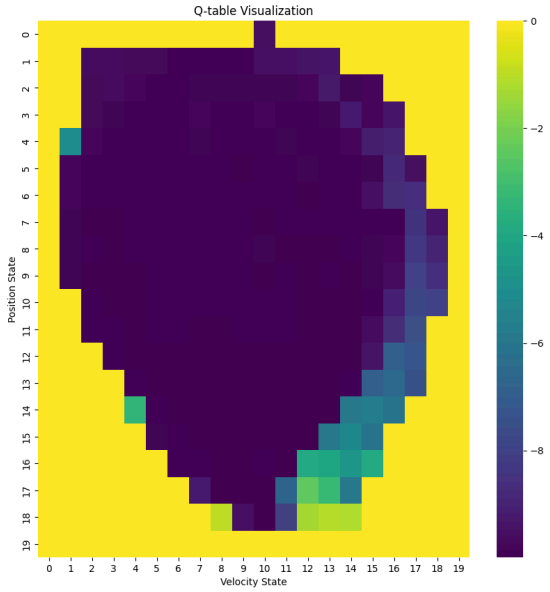


Rysunek 16: Heatmapa dla pozycji startowej = −0.9

Pozycja 0.4:



Rysunek 17: Grafiki dla pozycji startowej = 0.4



Rysunek 18: Heatmapa dla pozycji startowej = 0.4

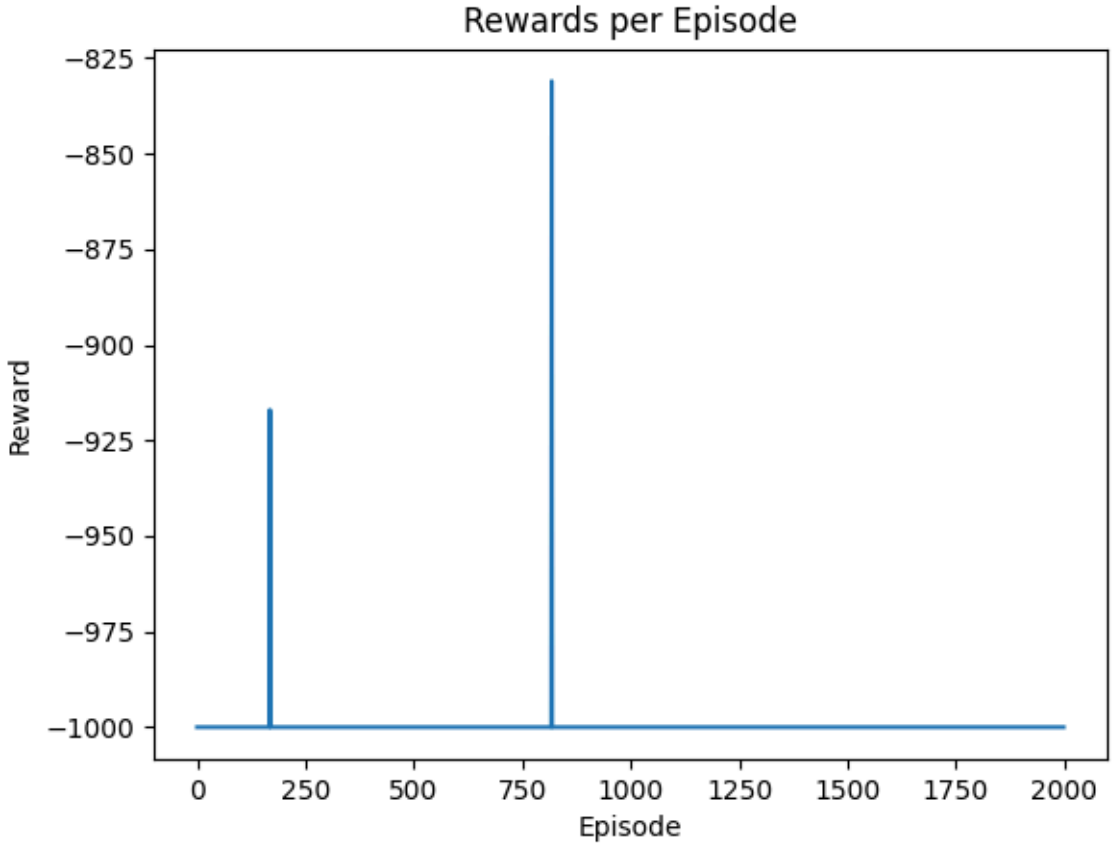
Wnioski: Można wprost zauważyć, że pozycja startowa ma znaczenie. Model, podobnie jak człowiek, jest w stanie doświadczalnie nauczyć się wyciągania korzyści z pozycji startowej. Można to zauważyć między innymi z wykorzystania wysokiego wychylenia (w przód) w przypadku trzecim i wysoki średni poziom nagród (mimo dużego „doświadczalnego chaosu", który widać w wykresach nagród per epizod). Tak samo po przypadku drugim widać, że nieco bardziej korzystna pozycja wychylenia (w tył) pozwoliła na szybsze uzyskiwanie rezultatów korzystnych. Dodatkowo, warto zauważyć, że w niestandardowych położeniach i przy tych parametrach model uczy się lepiej, co uwidacznia ustabilizowanie się wysokiej sumy nagród w wykresach średniej kroczącej w przypadkach drugim i trzecim.

5.3. Porównanie z losową strategią

Aby model wykonał działania w pełni losowe ustawiamy parametry:

- $\varepsilon = 1$, $num\_episodes = 2000$, $total\_rewards < -1000$, $initial\_p = -0.5$
- wyłączamy aktualizację Q-Tablicy
- Wyłączamy zmniejszanie ε

W wyniku tego działania tylko 2 razy udało się nie przekroczyć 1000 kroków osiągajac cel.



Rysunek 19: Nagrody per epizod w 2000 losowych próbach

Tym samym postanowiliśmy sprawdzić w 10 próbach, ile kroków zajmie losowa strategia wyłączając warunek maksymalnej ilości kroków = 1000:

```
Episode: 1, Total Reward: -36779.00, Epsilon: 1.00
Episode: 2, Total Reward: -12090.00, Epsilon: 1.00
Episode: 3, Total Reward: -59152.00, Epsilon: 1.00
Episode: 4, Total Reward: -15137.00, Epsilon: 1.00
Episode: 5, Total Reward: -18272.00, Epsilon: 1.00
Episode: 6, Total Reward: -73460.00, Epsilon: 1.00
Episode: 7, Total Reward: -24747.00, Epsilon: 1.00
Episode: 8, Total Reward: -38269.00, Epsilon: 1.00
Episode: 9, Total Reward: -5759.00, Epsilon: 1.00
Episode: 10, Total Reward: -32738.00, Epsilon: 1.00
```

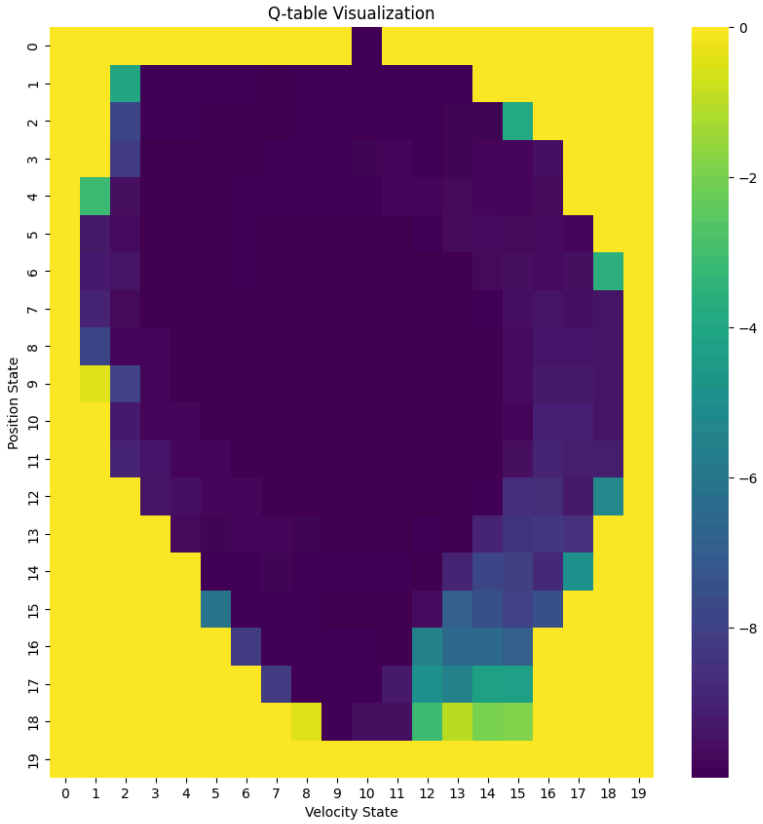
Wnioski: Rezultaty jednoznacznie wykazują, że nasze rozwiązanie jest efektywne i uczenie daje wymagane rezultaty, co podkreśla różnica liczby kroków wyuczonego agenta (około 135 kroków), a agenta robiącego kroki losowo.

5.4. Wykorzystanie wyuczonej Q-Tablicy

Ostatecznie pokażemy, że zapis zserializowwanej Q-Tablicy i późniejsze wykorzystywanie jej (tryb bez uczenia) daje oczekiwane rezultaty. Parametrami nauki są:

- $\alpha = 0.4$, $\gamma = 0.9$, $\varepsilon = 1$, $num\_episodes = 5000$, $total\_rewards < -1000$, $initial\_p = -0.5$

Q-Tablica po tym działaniu wygląda następująco:



Rysunek 20: Wyuczona Q-Tablica

Rezultatami działania modelu wykorzystującego utworzoną Q-Tablicę na 10 epizodach są:

```
Episode: 1, Total Reward: -135.00, Epsilon: 0.80
Episode: 2, Total Reward: -135.00, Epsilon: 0.60
Episode: 3, Total Reward: -135.00, Epsilon: 0.40
Episode: 4, Total Reward: -135.00, Epsilon: 0.20
Episode: 5, Total Reward: -135.00, Epsilon: 0.00
Episode: 6, Total Reward: -135.00, Epsilon: 0.00
Episode: 7, Total Reward: -135.00, Epsilon: 0.00
Episode: 8, Total Reward: -135.00, Epsilon: 0.00
Episode: 9, Total Reward: -135.00, Epsilon: 0.00
Episode: 10, Total Reward: -135.00, Epsilon: 0.00
```

Wnioski: Zgodnie z oczekiwaniami proces nauki przechodzi poprawnie. Przy dużej ilości epizodów i dużym treningu można utworzyć Q-Tabelę, dzięki której model rewelacyjnie będzie radził sobie z zadaniem.

6. Opis wykorzystanych bibliotek

6.1. random

Moduł random jest używany do generowania losowych wartości w procesie wyboru akcji przez agenta w strategii epsilon-greedy. Dzięki temu możliwe jest zachowanie elementu eksploracji w algorytmie uczenia Q--learning.

6.2. Gymnasium

Biblioteka Gymnasium zapewnia gotowe środowiska symulacyjne używane w uczeniu ze wzmocnieniem. W projekcie wykorzystano środowisko MountainCar-v0, które symuluje problem sterowania samochodem poruszającym się po wzgórzu. Środowisko dostarcza stan, nagrody, oraz umożliwia podejmowanie akcji przez agenta.

6.3. numpy

Biblioteka numpy jest używana do operacji matematycznych i zarządzania tablicami. W projekcie służy między innymi do:

- Tworzenia siatki stanów dla pozycji i prędkości.
- Aktualizacji i przetwarzania tablic Q.
- Obliczania średnich wartości nagród.

6.4. pickle

Moduł pickle jest używany do serializacji i deserializacji danych. W projekcie pozwala na zapisanie Q-table do pliku (Q_table.pkl) po treningu oraz jej odczyt podczas testowania.

6.5. matplotlib.pyplot

Biblioteka matplotlib jest używana do wizualizacji danych. W projekcie generuje wykresy:

- Nagrody uzyskane w każdym epizodzie.
- Średnia krocząca.
- Wizualizacja tablicy Q (maksymalnych wartości Q dla stanów).

6.6. seaborn

Biblioteka seaborn rozszerza możliwości wizualizacji oferowane przez matplotlib. W projekcie jest używana do wizualizacji Q-table jako heatmapa.

6.7. unittest

Biblioteka unittest została wykorzystana do testów jednostkowych.

7. Podsumowanie

Implementacja algorytmu Q-Learning, który nauczył wagonik samodzielnie wspinać się po wzniesieniu pozwolił nam poszerzyć wiedzę na temat ML oraz głębiej zaciekawił nas algorytmami uczącymi. Utworzenie modelu oraz przeprowadzone eksperymenty pozwoliły nam zapoznać się z podstawami tworzenia modeli oraz zrozumieć, jak kluczowe parametry algorytmu wpływają na proces uczenia. Szczególnie interesujące okazało analizowanie zachowań modelu nie tylko przy zmianach parametru, ale także jego zachowań w różnych sytuacjach (tj. zmiany stanów inicjalnych) oraz szybkości przyswajanej wiedzy.