Sztuczna inteligencja

Projekt

Temat:

“Uczenie ze wzmocnieniem z wykorzystaniem algorytmów Deep Q-learning oraz Q-learning dla środowiska MountainCar z OpenAI gym.”

Wykonali:

Rafał Paprocki

Piotr Ryzak

1. Wstęp

Projekt miał na celu wytrenowanie agenta dla środowiska z biblioteki OpenAI przy użyciu algorytmu q-learning oraz jego rozszerzonej wersji korzystającej z sieci neuronowych deep q-learning. Najpierw wytrenowaliśmy naszego agenta dla środowiska MountainCar za pomocą algorytmu deep q-learningu, oraz przetestowliśmy jego wyniki dla różnych parametrów hiperbolicznych.

Następnie ten sam problem rozwiązaliśmy przy użyciu algorytmu q-learning w celu sprawdzenia skuteczności obu modeli i wybrania lepszej metody dla tego problemu.

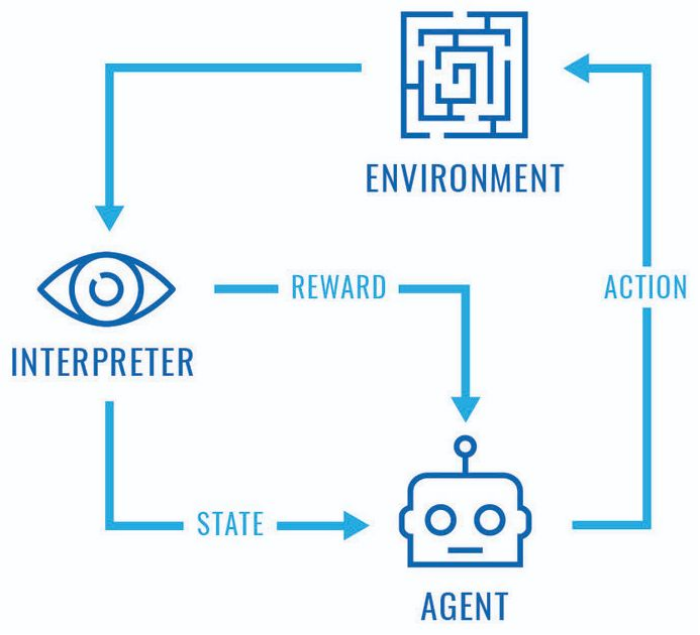
Użycie dwóch różnych algorytmów do tego problemu pomogło nam również zauważyć zalety i wady każdego z tych rozwiązań.

2. Opis używanych algorytmów

Opis ten rozpoczniemy od wprowadzenia pojęcia reinforcement learning, a następnie opiszemy algorytmy użyte w projekcie.

Reinforcement learning

Ogólny schemat działania obrazuje poniższy obrazek.



W procesie uczenia występuje środowisko - environment, w przypadku obrazka jest to labirynt. Środowisko jest miejscem w którym nasza sztuczna inteligencja będzie wykonywała różne akcje oraz chciało osiągnąć jak najlepszy wynik i na przykład pokonać labirynt, wychodząc z niego. Mamy też Agenta. Agent - jest to nasza sztuczna inteligencja - możemy nazwać go osobą, która będzie nawigować
w środowiskach i uczyć się wykonywać różne czynności poprzez otrzymywanie od

środowiska wyników jak wykonać daną czynność np. jazda samochodem - dostaję od środowiska (policjanta) mandat, że wykonałem daną czynność źle. Tak więc Agent wykonuje pewne czynności w środowisku, w wyniku czego stan w jakim się znajduje ulega zmianie, przez co może być np. bliżej prawej lub lewej części labiryntu. Agent otrzymuje także nagrodę za każdą zmianę stanu po wykonaniu jakiejś czynności. Przez co będzie eksplorować środowisko i rozumieć jaka czynność prowadzi do dostania nagrody i jaki stan jest pomyślny, a jaka czynność jest nieopłacalna.

W uczeniu maszynowym środowisko jest zazwyczaj formułowane jako proces decyzyjny Markowa (MDP), ponieważ wiele algorytmów uczenia wykorzystuje techniki programowania dynamicznego. MDP mówi nam o tym, że przyszłość nie jest zależna od przeszłości, pod warunkiem posiadania stanu obecnego. Oznacza to, że wszystkie potrzebne informacje, które powinniśmy wynieść z przeszłości są już zawarte w stanie obecnym. Nie musimy używać poprzednich stanów, możemy użyć jedynie stanu obecnego do modelowania przyszłych. Uczenie ze wzmocnieniem różni się od standardowego uczenia nadzorowanego, tym że pary wejścia/wyjścia nie muszą być prezentowane, a suboptymalne działanie nie musi być jawnie korygowane. Zamiast tego nacisk kładziony jest na wydajność, co wymaga znalezienia równowagi między eksploracją niezbadanego terytorium, a eksploatacją obecnie zdobytej wiedzy.

Q - Learning

Jest to algorytm stosowany w reinforcement learning do trenowania modelu.

W algorytmie tym musimy stworzyć tabelę zwaną Q-table zawierającą wszystkie możliwe stany jako wiersze oraz akcje jako kolumny. Dla każdej wartości [stan, akcja] określona jest nagroda jaką otrzymamy po wykonaniu danej akcji w tym stanie. Następnie będziemy ją aktualizować.

Algorytm wygląda następująco:

1. Inicjalizujemy tablicę losowymi wartościami.
2. W każdym kroku stosujemy strategię argmax - jeśli znajdujemy się w stanie st, to wybieramy akcję at=argmaxa′q(st,a′).
3. Po otrzymaniu nagrody rt+1 uaktualniamy tabelę w taki sposób, aby bardziej odpowiadała polityce, którą aktualnie stosujemy za pomocą wzoru:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

$Q^{new}(s_t, a_t)$ - zaktualizowana wartość Q dla stanu St i akcji At

$\underbrace{Q(s_t, a_t)}_{\text{old value}}$

- poprzednia wartość Q dla stanu St i akcji At

rt (inaczej R(St, At)) - nagroda za wykonanie akcji At w stanie St

$\underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}$ - maksymalna wartość Q, ze wszystkich możliwych akcji w następnym stanie S t+1

α - współczynnik uczenia - jest to rozmiar kroku jaki jest wykonywany od obecnego stanu podczas eksploracji, im większa wartość tym większe kroki, gdy alfa jest małe to eksploracja jest ograniczona

γ - tzw. discount factor (współczynnik zniżkowy) - służy do zmniejszenia wagi niepewnych przyszłych nagród, względem pewnych natychmiastowych nagród

W tym wzorze aktualizujemy wartość danego pola tabelki przez dodanie wartości nagrody po wykonaniu akcji w danym stanie oraz maksymalnej nagrody jaką możemy w przyszłości otrzymać z tego nowego stanu pomnożonej razy współczynnik gamma.

Własności:

1. Aktualna aproksymacja funkcji q nie jest zgodna z polityką, którą stosujemy - być może nawet nie opisuje ona żadnej istniejącej polityki, ale to nie jest problem.
2. Bieżąca polityka to strategia argmax na aproksymacji funkcji q. W związku z tym gdyby aproksymacja była zgodna z pewną polityką π, to strategia argmax będzie od niej lepsza.
3. Krok trzeci sprawia, że aproksymacja zbliża się do bieżącej polityki. W tym wypadku jest to polityka argmax.
4. Własności 2. i 3. sprawiają, że agent faktycznie się uczy - aproksymacja q próbuje "dogonić" bieżącą politykę, ale bieżąca polityka (argmax) zawsze jest o krok do przodu przed aproksymacją. W ten sposób agent uczy się być lepszy od samego siebie. To rozwiązuje problem braku nauczonego "eksperta", którego agent mógłby naśladować.
5. Algorytm stabilizuje się po osiągnięciu optymalnej polityki $\pi*$, ponieważ wtedy strategia argmax odpowiada bieżącej polityce.

Głównym problemem tego algorytmu jest potrzeba posiadania skończonej ilości stanów.

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	327	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	0	0	0	0	0	0

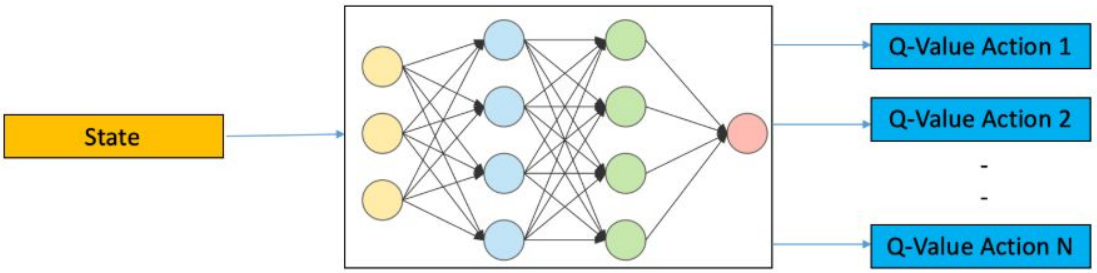
Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Deep Q-learning

Jest pewnego rodzaju inną wersją algorytmu q-learning która zamiast tabeli q-table używa sieci neuronowych. W tym algorytmie nie tworzymy tabeli przechowującej stany.

Stany są wejściami sieci neuronowej, natomiast wyjścia tej sieci to podejmowane akcje.



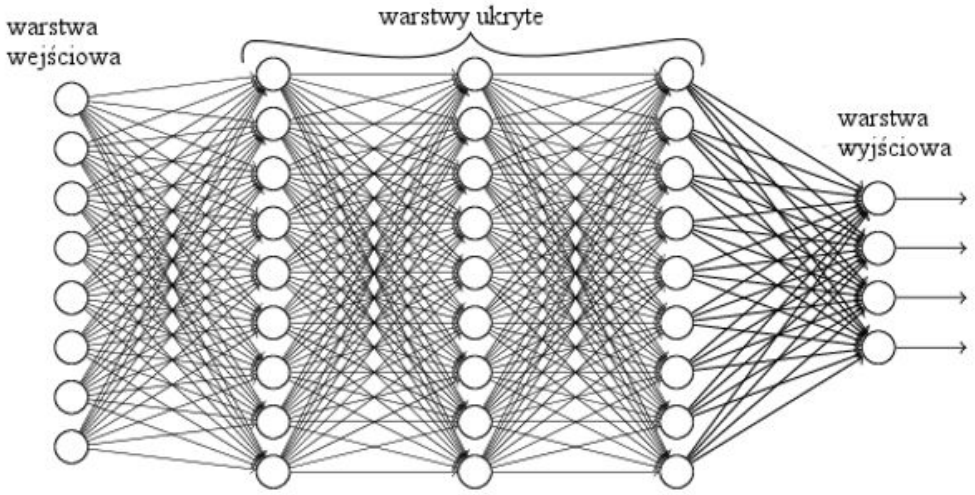
Deep Q Learning

Uczenie polega na tym, że wchodzące wartości do naszej sieci obliczają na wyjścia nowe wartości Q1, Q2 … Qn i zostają one porównywane do targetu - czyli wartości przewidzianej, dzięki temu lepiej się uczy jak ma się np. poruszać po mapie. Musimy teraz dostosować koncept Q-Learningu do zmiany wag, gdyż tak działają nasze sieci. Aby obliczyć Loss wykorzystamy wzór, który musi być jak najbliżej 0 jak jest to możliwe, i przy pomocy wstecznej propagacji błędu odświeżymy wagi.

$$L = \sum (Q\text{–}Target\ - Q)^2$$

Sieci neuronowe

Ogólna nazwa struktur matematycznych i ich programowych lub sprzętowych modeli, realizujących obliczenia lub przetwarzanie sygnałów poprzez rzędy elementów przetwarzających, zwanych sztucznymi neuronami, wykonujących pewną podstawową operację na swoim wejściu. Oryginalną inspiracją takiej struktury była budowa naturalnych neuronów, łączących je synaps, oraz układów nerwowych, w szczególności mózgu.



Najważniejsze algorytmy podczas uczenia sieci to algorytm wstecznej propagacji oraz propagacji w przód.

Jest to metoda umożliwiająca modyfikację wag w sieci o architekturze warstwowej (opisanej w poprzednim rozdziale), we wszystkich jej warstwach.

Ogólny schemat procesu trenowania sieci wygląda następująco:

1. Ustalamy topologię sieci, tzn. liczbę warstw, liczbę neuronów w warstwach.
2. Inicjujemy wagi losowo (na małe wartości).
3. Dla danego wektora uczącego obliczamy odpowiedź sieci (warstwa po warstwie).
4. Każdy neuron wyjściowy oblicza swój błąd, oparty na różnicy pomiędzy obliczoną odpowiedzią *y* oraz poprawną odpowiedzią *t*.
5. Błędy propagowane są do wcześniejszych warstw.
6. Każdy neuron (również w warstwach ukrytych) modyfikuje wagi na podstawie wartości błędu i wielkosci przetwarzanych w tym kroku sygnałów.
7. Powtarzamy od punktu 3. dla kolejnych wektorów uczących. Gdy wszystkie wektory zostaną użyte, losowo zmieniamy ich kolejność i zaczynamy wykorzystywać powtórnie.
8. Zatrzymujemy się, gdy średni błąd na danych treningowych przestanie maleć. Możemy też co jakiś czas testować sieć na specjalnej puli nieużywanych do treningu próbek testowych i kończyć trenowanie, gdy błąd przestanie maleć.

3. Opis środowiska.

Do zaimplementowania algorytmów uczenie ze wzmocnieniem skorzystaliśmy z gotowego projektu OpenAI które dostarcza wiele różnych środowisk, na których można testować i ulepszać własne algorytmy.

Skorzystaliśmy ze środowiska MountainCar-v0 znajdującego się znajdującego się w OpenAI gym.

MountainCar-v0 - w środowisku tym znajduje się samochodzik pomiędzy dwoma wzgórzami. Na jednym ze wzgórz znajduje się flaga. Naszym celem jest takie sterowanie pojazdem aby osiągnąc szczyt wzgórza na którym umieszczona jest flaga.

Jednak nasz samochodzik jest zbyt słaby aby osiągnąć szczyt zaczynając z punktu startowego i kierując się tylko w jednym kierunku. Zatem musimy najpierw odpowiednio rozpędzić samochodzik aby był w stanie osiągnąć szczyt.

Akcje.

W każdym kroku możemy wykonać jedną z trzech akcji:

a) Ruch w prawo

b) Ruch w lewo

c) Brak ruchu

Nagrody.

Za każdy ruch którym nie jest osiągnięcie szczytu nagroda wynosi -1.

Za ruch który osiąga szczyt otrzymujemy nagrodę 0.5.

Stany środowiska.

Stan jest opisany dwoma zmiennymi. Jedna z nich opisuje pozycje druga prędkość.

Dokładne przedziały dla tych wartości zawiera poniższa tabelka.

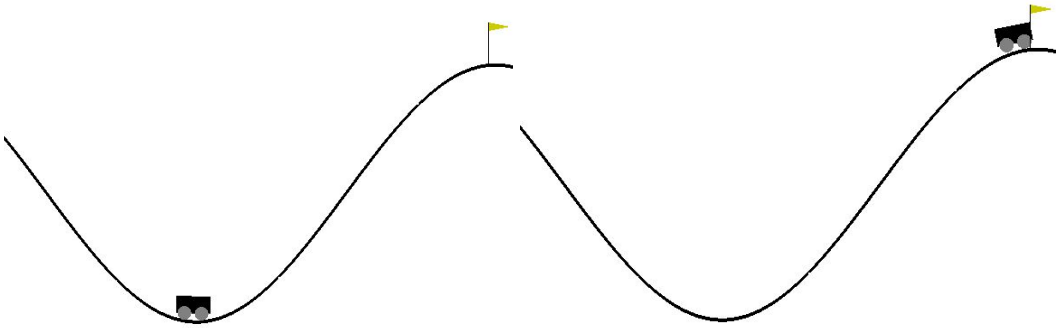
Num	Observation	Min	Max
0	position	-1.2	0.6
1	velocity	-0.07	0.07

Środowisko zostaje zamknięte gdy wystąpi jeden z dwóch podanych niżej warunków:

a) Osiągnięto szczyt

b) Wykonano więcej niż 200 operacji

Punktem startowym jest losowa pozycja z przedziału -0.6 do -0.4 oraz prędkość równa 0.



W obrazku po lewej środowisko znajduje się w stanie początkowym. Środowisko po prawej natomiast osiąga cel.

4. Opis kodu programu.

W tym punkcie został zawarty opis większości funkcji naszego programu. Cały kod programu znajduje się w załączniku.

Deep q-learning

```
1  import gym
2  from keras import models
3  from keras import layers
4  from keras.models import load_model
5  from keras.optimizers import Adam
6  from collections import deque
7  import random
8  import numpy as np
9  import matplotlib.pyplot as plt
```

Na początku importujemy wszystkie niezbędne pakiety. Będziemy wykorzystywać bibliotekę Keras w celu uczenia głębokich sieci neuronowych. W pierwszej linijce importujemy pakiet gym, który zawiera środowisko MountainCar z którego będziemy korzystać. Ponadto będziemy korzystać z biblioteki numpy ułatwiającej efektywne operacje na macierzach oraz biblioteki matplotlib aby graficznie prezentować uzyskane przez nasz program wyniki.

Teraz zaimplementujemy klasę która będzie zawierała implementacje algorytmu deep q-learning dla środowiska MountainCar oraz metodę do trenowania agenta przy użyciu tego algorytmu. Klasa ta wewnętrznie używa sieci neuronowych implementowanych przy użyciu biblioteki Keras.

```
class MountainCarTrain:
    def __init__(self, env, gamma = 0.99, learning_rate = 0.001, epsilon = 1, episodeNum = 200, epsilon_min = 0.01,
                 epsilon_decay = 0.05 ):
        self.env = env
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.learning_rate = learning_rate
        self.replayBuffer = deque(maxlen=20000)
        self.trainModel = self.createModel()
        self.episodeNum = episodeNum
        self.iterationNum = 201
        self.numPickFromBuffer = 32
        self.targetModel = self.createModel()
        self.targetModel.set_weights(self.trainModel.get_weights())
```

Powyższy kod zawiera inicjalizację zmiennych hiperparametrycznych za pomocą których będziemy sterować przebiegiem procesu uczenia.

Oto krótki opis tych parametrów:

a) Env - środowisko dla którego będziemy trenować naszego agenta

b) Gamma - określa w jakim stopniu skupiamy się na przyszłych nagrodach. Im większa wartość tego parametru tym algorytm w większym stopniu skupia się na nagrodach, które mogą nastąpić w dalszym okresie czasu.

c) Epsilon, epsilon_decay, epsilon_min - parametry te służą do określania kiedy wykonujemy eksploracje a kiedy eksploatację naszego środowiska.

Eksploracja polega na poszukiwaniu nowych rozwiązań natomiast eksploatacja polega na zgłębianiu, badaniu, polepszaniu już eksplorowanego środowiska. Ważne jest by w początkowym etapie uczenia nasz algorytm skupiał się na tym by jak najlepiej poznać środowisko, a dopiero w późniejszej fazie na jego eksploatacji.

d) Learning_rate - jest to współczynnik uczenia który wpływa na szybkość uczenia.

e) ReplayBuffer - jest to tablica zawierająca poprzednie stany środowiska. Z tej tablicy będziemy wybierać kolejny stan do uczenia naszej sieci.

f) EpisotNum - określa ilość epizodów podczas uczenia

g) IterationNum - określa maksymalną ilość iteracji w każdym epizodzie. Przy czym warto zauważyć że w środowisku ClimbingCar po wykonaniu więcej niż dwieście iteracji środowisko zostaje zamknięte.

h) TrainModel - model trenujący

i) TargetModel - model służący do śledzenia jaką akcję ma podjąć nasz model

```
def createModel(self):
    model = models.Sequential()
    state_shape = self.env.observation_space.shape
    model.add(layers.Dense(24, activation='relu', input_shape=state_shape))
    model.add(layers.Dense(48, activation='relu'))
    model.add(layers.Dense(self.env.action_space.n, activation='linear'))
    model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
    return model
```

Funkcja ta tworzy model posiadający jedną warstwę ukrytą. W dwóch pierwszych warstwach jako funkcja aktywacji została ustawiona funkcja relu w ostatniej zaś funkcja liniowa (linear). Do mierzenia błędów użyto błąd średnio-kwadratowy (mse), a jako optymalizator zastosowano klasę Adam().

Warstwa wejściowa składa się z 2 jednostek, ukryta z 48 natomiast wyjściowa z 3.

Każda z warstw wyjściowych opisuje jeden ze stanów 0, 1, 2.

```
def getBestAction(self, state):
    self.epsilon = max(self.epsilon_min, self.epsilon)
    if np.random.rand(1) < self.epsilon:
        action = np.random.randint(0, 3)
    else:
        action = np.argmax(self.trainModel.predict(state)[0])
    return action
```

Metoda wybiera odpowiednią akcję na podstawie obecnego stanu, jak i wartości parametrów epsilon, epsilon_min. Jeśli wartość losowa jest mniejsza niż epsilon (początkowo wynosi jeden) zostanie wykonana eksploracja, w przeciwnym wypadku akcja zostanie wybrana na podstawie trenowanego modelu, czyli będziemy eksploatować.

Jak widzimy początkowo duża wartość epsilon sprzyja eksploracji natomiast wraz z kolejnymi krokami jej wartość maleje, co przekłada się na częstszą eksploatację modelu.

```
def trainFromBuffer(self):
    if len(self.replayBuffer) < self.numPickFromBuffer:
        return 0

    samples = random.sample(self.replayBuffer, self.numPickFromBuffer)
    states = []
    newStates = []
    for sample in samples:
        state, action, reward, new_state, done = sample
        states.append(state)
        newStates.append(new_state)

    newArray = np.array(states)
    states = newArray.reshape(self.numPickFromBuffer, 2)

    newArray2 = np.array(newStates)
    newStates = newArray2.reshape(self.numPickFromBuffer, 2)

    targets = self.trainModel.predict(states)
    new_state_targets = self.targetModel.predict(newStates)

    i = 0
    for sample in samples:
        state, action, reward, new_state, done = sample
        target = targets[i]
        if done:
            target[action] = reward
        else:
            Q_future = max(new_state_targets[i])
            target[action] = reward + Q_future * self.gamma
        i += 1

    v = self.trainModel.fit(states, targets, epochs=1, verbose=0)
    return int(v.history['loss'][0])
```

Funkcja ta służy do trenowania sieci. Wybieramy z tablicy poprzednich stanów numPickFromBuffer stanów następnie przewidujemy wyniki dla każdego z nich. Za pomocą targetModel przewidujemy wyniki kolejnych stanów i zapisujemy je w macierzy. Następnie w pętli za pomocą algorytmu Q-learningu którego wzór został podany w części teoretycznej obliczmy target, który opisuje wartość nagrody która chcemy osiągnąć dla danego stanu. Na końcu uczymy tymi danymi nasz model trenujący za pomocą metody fit().

```
def orginalTry(self, currentState, eps):
    rewardSum = 0
    max_position = -99
    lossed = 0
    for i in range(self.iterationNum):
        bestAction = self.getBestAction(currentState)
        new_state, reward, done, _ = env.step(bestAction)
        new_state = new_state.reshape(1, 2)

        if new_state[0][0] > max_position:
            max_position = new_state[0][0]
        if new_state[0][0] >= 0.5:
            reward += 10

        self.replayBuffer.append([currentState, bestAction, reward, new_state, done])
        lossed += self.trainFromBuffer()

        rewardSum += reward

        currentState = new_state

        if done:
            break
    print(lossed)

    if i >= 199:
        print("Episode {} failed".format(eps))
    else:
        print("Success in episode {}, used {} iterations!".format(eps, i))
        self.trainModel.save('./trainModelInEPS{}.h5'.format(eps))

    self.targetModel.set_weights(self.trainModel.get_weights())
    print(self.targetModel.total_loss)
    print("now epsilon is {}, the reward is {} maxPosition is {}".format(max(self.epsilon_min, self.epsilon),
                                                                         rewardSum, max_position))
    self.epsilon -= self.epsilon_decay
    return rewardSum, lossed
```

Metoda ta służy do wykonania iterationNum iteracji w jednej epoce na modelu.
W każdej iteracji jest wykonywana funkcja step() dla środowiska z akcją pobraną z funkcji getBestAction (opisana powyżej). Funkcja step() zwraca nam następujące zmienne: stan po wykonaniu akcji, nagrodę, oraz czy środowisko zakończyło działanie po wykonaniu tego stanu . W każdym kroku dodajemy także nowy wpis do tablicy replayBuffer i wykonujemy trenowanie.

Jeśli zmienna done ma wartość True pętla zostaje przerwana następnie zostaje wykonany poniższy kod, w którym zapisujemy na dysku modele którym udało się osiągnąć szczyt.

```
if i >= 199:
    print("Episode {} failed".format(eps))
else:
    print("Success in episode {}, used {} iterations!".format(eps, i))
    self.trainModel.save('./trainModelInEPS{}.h5'.format(eps))

self.targetModel.set_weights(self.trainModel.get_weights())
print(self.targetModel.total_loss)
print("now epsilon is {}, the reward is {} maxPosition is {}".format(max(self.epsilon_min, self.epsilon),
                                                                     rewardSum, max_position))
self.epsilon -= self.epsilon_decay
return rewardSum, lossed
```

```
def start(self):
    lossed_tab = []
    reward_tab = []
    for eps in range(self.episodeNum):
        currentState = env.reset().reshape(1, 2)
        rew, los = self.orginalTry(currentState, eps)
        reward_tab.append(rew)
        lossed_tab.append(los)
        print(lossed_tab)
    plt.plot((np.arange(len(lossed_tab)) + 1), lossed_tab)
    plt.xlabel('Epizody')
    plt.ylabel('Nagroda')
    plt.title('Test modelu')
    plt.show()
    plt.close()
    return reward_tab
```

Funkcja wykonuje episodeNum epizodów oraz tworzy tablicę nagród dla każdego z nich. Tablica ta jest zwracana na końcu funkcji.

W projekcie istnieje także funkcja q_learning() służąca do trenowania agenta dla środowiska MountainCar za pomocą algorytmu q-learning z wykorzystaniem q-table.

Jest to inny sposób wytrenowania agenta dla danego środowiska. Przedstawiliśmy go w celu pokazania różnic pomiędzy tymi algorytmami oraz ich wady i zalety.

```
def q_learning(env, learning, discount, epsilon, min_eps, episodes):
    num_states = (env.observation_space.high - env.observation_space.low) * \
                 np.array([10, 100])
    num_states = np.round(num_states, 0).astype(int) + 1
```

W powyższym kodzie dokonujemy próbkowania aby otrzymać dyskretną tablicę stanów, jest to niezbędne dla tego algorytmu, aby posiadać skończoną liczbę stanów.

```
Q = np.random.uniform(low=-1, high=1,
                      size=(num_states[0], num_states[1],
                            env.action_space.n))
```

Tworzymy Q-table. W tym przypadku jest to tablica 3-wymiarowa. Pierwszy wymiar stanowią spróbkowane wartości położenia, drugi wymiar stanowią wartości przyspieszenia a trzeci to akcje.

```
if np.random.random() < 1 - epsilon:
    action = np.argmax(Q[state_adj[0], state_adj[1]])
else:
    action = np.random.randint(0, env.action_space.n)
```

Określa czy wykonać eksploracje czy eksploatacja a następnie zapisuje akcje do zmiennej action.

```
state2, reward, done, info = env.step(action)
state2_adj = (state2 - env.observation_space.low) * np.array([10, 100])
state2_adj = np.round(state2_adj, 0).astype(int)

if done and state2[0] >= 0.5:
    Q[state_adj[0], state_adj[1], action] = reward

else:
    delta = learning * (reward +
                        discount * np.max(Q[state2_adj[0],
                                            state2_adj[1]]) -
                        Q[state_adj[0], state_adj[1], action])
    Q[state_adj[0], state_adj[1], action] += delta

tot_reward += reward
state_adj = state2_adj
```

Powyższy kod wykonuje kolejny krok (step()) następnie konwertuje zwróconą obserwacje do liczby całkowitej znajdującej się w przedziale naszych stanów. Następnie za pomocą algorytmu q-learningu określa wartość o którą zostanie zwiększona Q-table dla danego stanu i akcji.

Cały kod tej metody znajduje się w załączonym pliku.

5. Dobierania parametrów i ocenianie agenta.

Uruchamiamy poniższy kod dla różnych parametrów i obserwujemy wyniki.

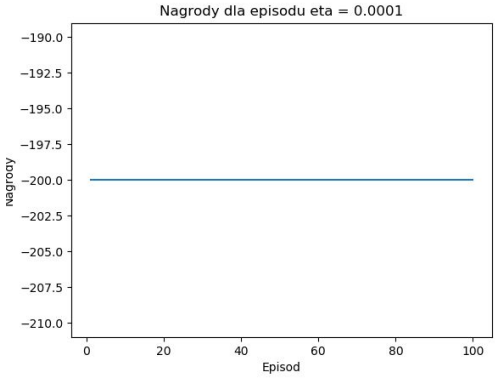
```
# initialization environment
env = gym.make('MountainCar-v0')
env.reset()

# train agent with deepQ_learning
dqn = MountainCarTrain(env=env, learning_rate=0.01, gamma=0.99, epsilon_min=0, episodeNum=200)
reward_tab = dqn.start()
env.reset()
```

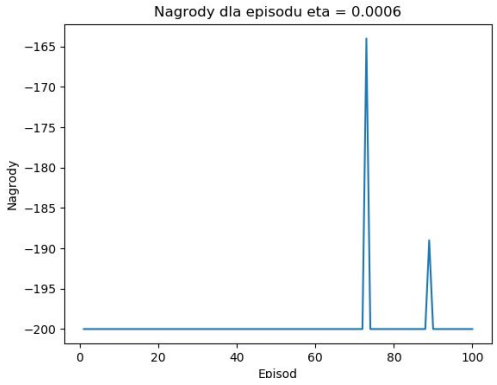
Podczas trenowania na konsoli wyświetlają nam się pomocne komunikaty mówiące czy w danym epizodzie nasz model osiągnął sukces czy nie. Ponadto jest wyświetlany numer epizodu, ogólna nagroda w danym epizodzie, wartość epsilon oraz

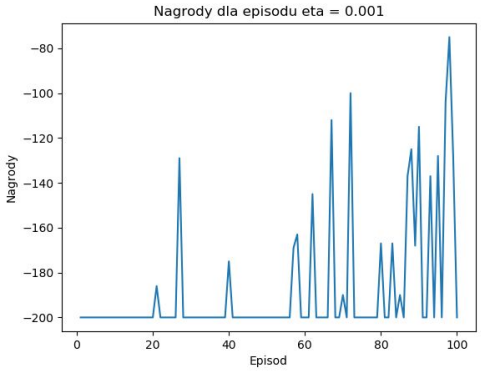
maksymalna osiągnięta pozycja. Pozycja 0.5 znajduje się na szczycie wzgórza i jest równoznaczna z osiągnięciem celu.

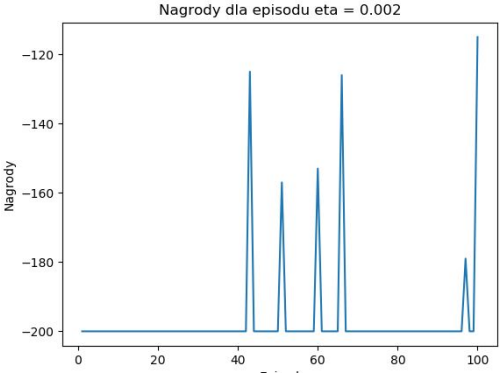
```
Episode 51 failed
Tensor("loss_27/mul:0", shape=(), dtype=float32)
now epsilon is 0, the reward is -200.0 maxPosition is 0.45776437865402914
[0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 20, 29, 30, 32, 35, 44, 40, 58, 54, 68, 57, 81, 92, 92, 87, 88, 76, 83, 92, 117, 145, 160,
187, 117, 208, 111, 172, 145, 149, 161, 167, 183, 268, 221, 279, 286, 342, 198, 272, 233, 243]
249
Episode 52 failed
Tensor("loss_27/mul:0", shape=(), dtype=float32)
now epsilon is 0, the reward is -200.0 maxPosition is -0.1495629249464006
[0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 20, 29, 30, 32, 35, 44, 40, 58, 54, 68, 57, 81, 92, 92, 87, 88, 76, 83, 92, 117, 145, 160,
187, 117, 208, 111, 172, 145, 149, 161, 167, 183, 268, 221, 279, 286, 342, 198, 272, 233, 243, 249]
200
Episode 53 failed
Tensor("loss_27/mul:0", shape=(), dtype=float32)
now epsilon is 0, the reward is -200.0 maxPosition is -0.3909647533383329
[0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 20, 29, 30, 32, 35, 44, 40, 58, 54, 68, 57, 81, 92, 92, 87, 88, 76, 83, 92, 117, 145, 160,
187, 117, 208, 111, 172, 145, 149, 161, 167, 183, 268, 221, 279, 286, 342, 198, 272, 233, 243, 249, 200]
180
```

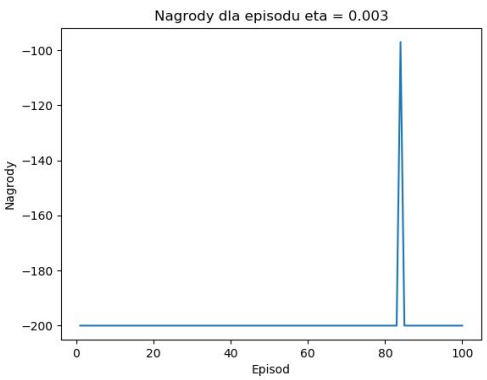
Zbadamy teraz nagrody dla poszczególnych epizodów przy różnych wartościach współczynnika uczenia. Wykresy te pokazują nam jakie wyniki (jaka była suma nagród) osiągał nasz model w poszczególnych epizodach,

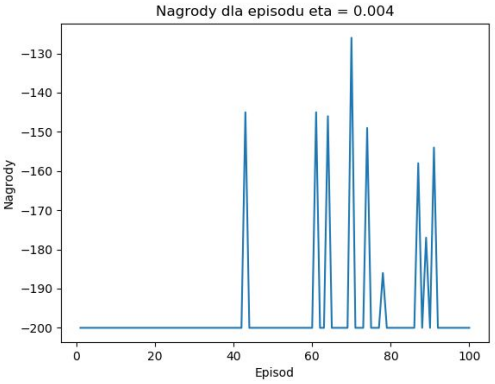


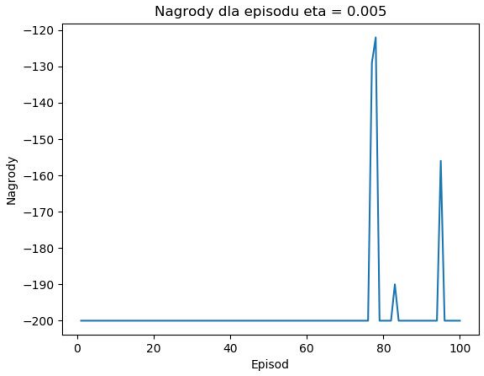


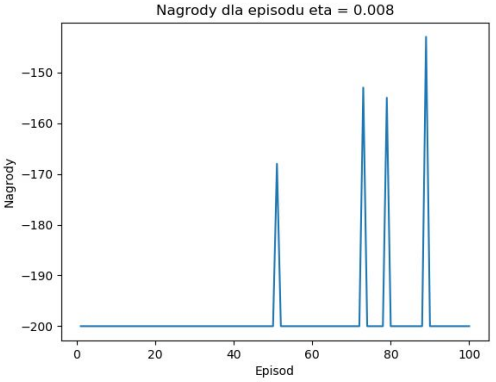


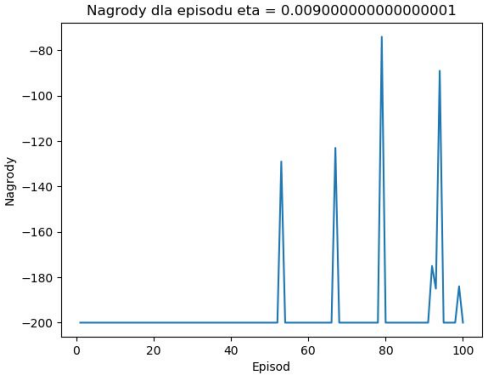


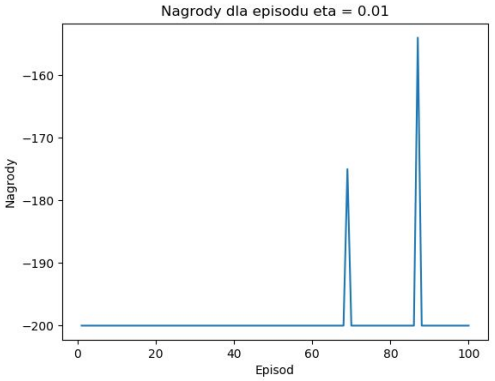

Nagrody dla episodu eta = 0.003
Nagrody
Episod

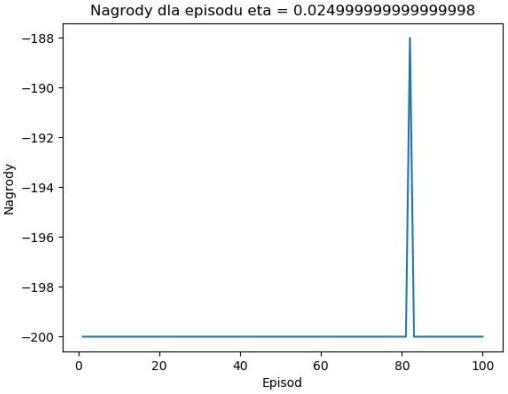

Nagrody dla episodu eta = 0.004
Nagrody
Episod

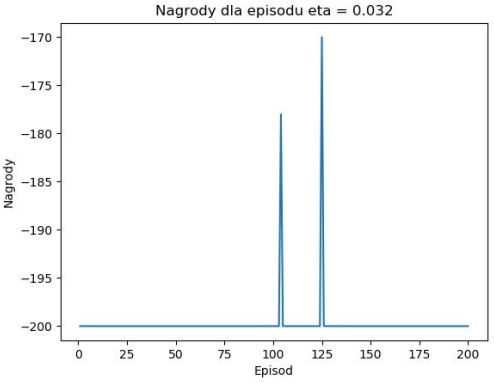

Nagrody dla episodu eta = 0.005
Nagrody
Episod

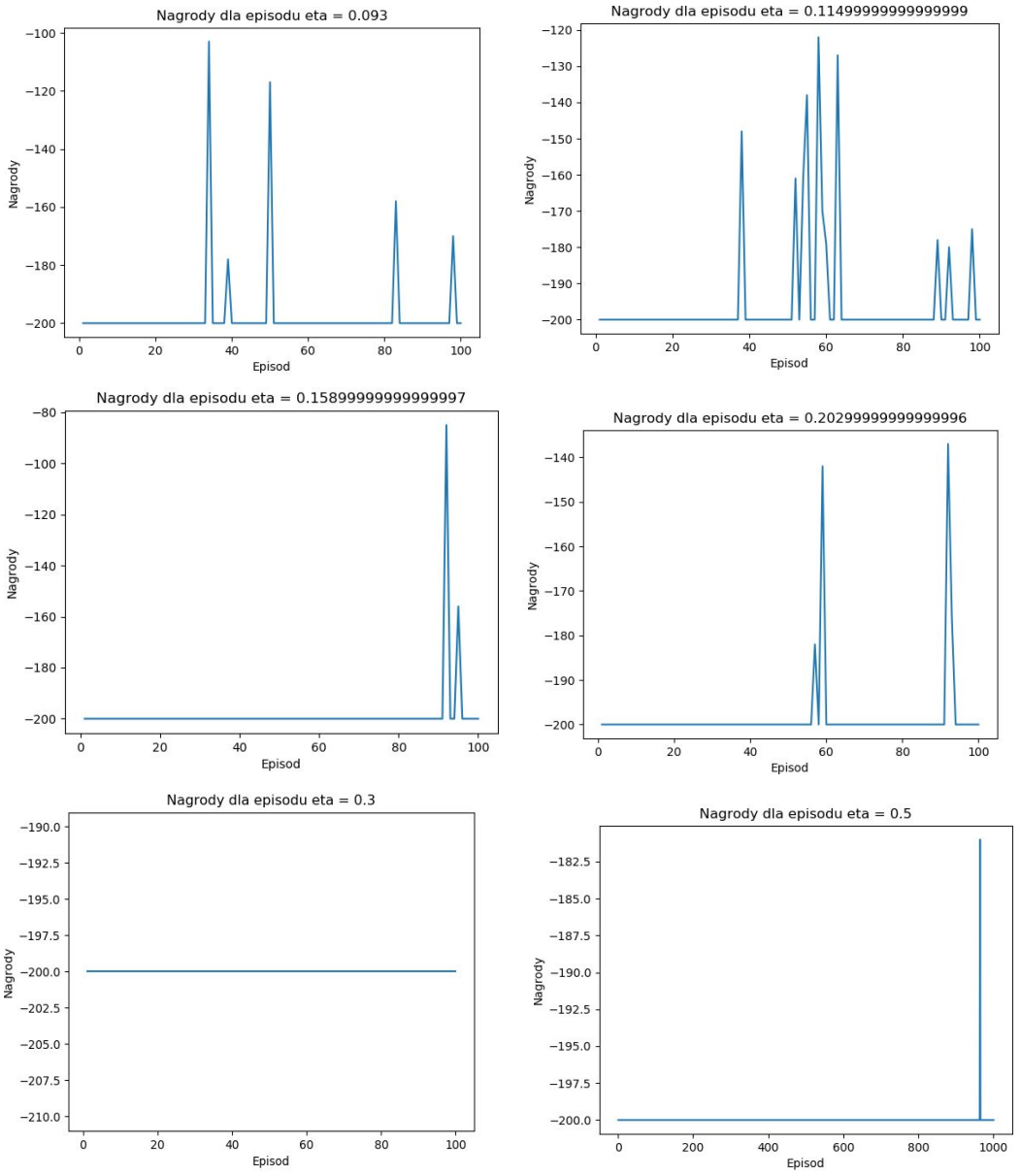

Nagrody dla episodu eta = 0.008
Nagrody
Episod


Nagrody dla episodu eta = 0.009000000000000001
Nagrody
Episod


Nagrody dla episodu eta = 0.01
Nagrody
Episod


Nagrody dla episodu eta = 0.024999999999999998
Nagrody
Episod


Nagrody dla episodu eta = 0.032
Nagrody
Episod



Możemy tutaj zaobserwować że dla różnych wartości parametru współczynnika uczenia oznaczonego na wykresach jako eta, możemy zaobserwować inne wyniki modeli. Wykresy te obrazują nagrody jakie udało się osiągnąć modelowi w danej epoce. Widzimy tutaj że dla niskich eta < 0.001 na 100 epizodów bardzo rzadko udawało się osiągnąć cel. Przy czym maksymalna nagroda nie była większa od -165. Dla parametrów eta o wartościach z przedziału od 0.001 do 0.2 możemy zaobserwować dość dobre wyniki osiągające w niektórych momentach model z nagrodami bliskimi -80. Jednak te wartości mają pewne wahania dla różnych wartości z tego przedziału.

Może wynikać to z faktu że trenując nasz model korzystamy z pewnej losowości podejmując akcje (zwłaszcza na etapie eksploatacji) oraz wybierając dane do trenowania. Wobec tego nawet kilkukrotne trenowanie agenta dla tych samych parametrów może dawać różniące się modele. Jednak próbując kilkukrotnie trenować model dla różnych parametrów możemy dojść do wniosku że istnieją parametry dla

których uda nam się wytrenować lepszy model pomimo pewnej dozy losowych danych.

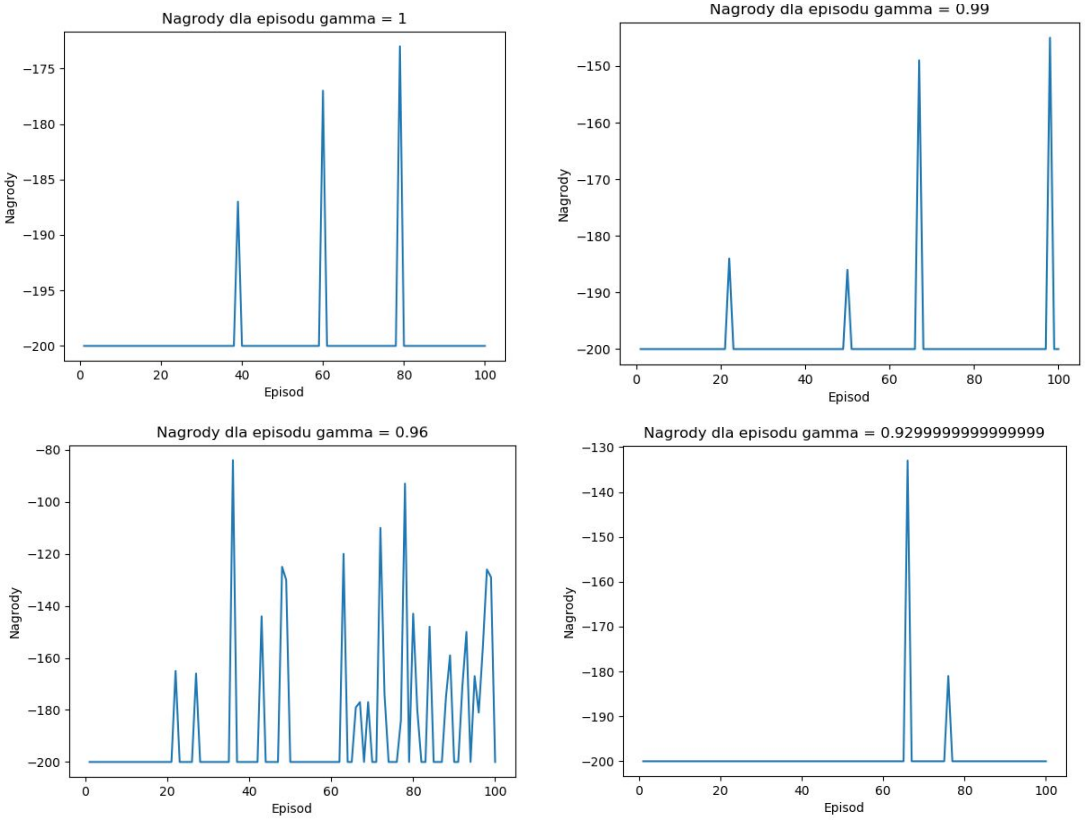
Na powyższych wykresach możemy także zaobserwować że dla parametru eta > 0.2 modele posiadają nagrodę -200 co jest równoważne z nieosiągnięciem celu dającego nagrodę (w środowisku MountainCar).

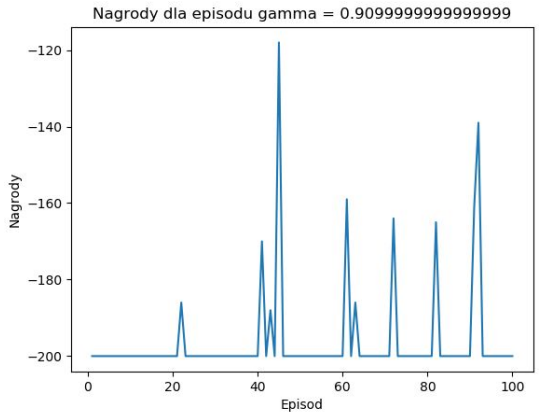
Zbyt niskie wartości parametru uczenia mogą powodować że nasz model utknie w lokalnym minimum.

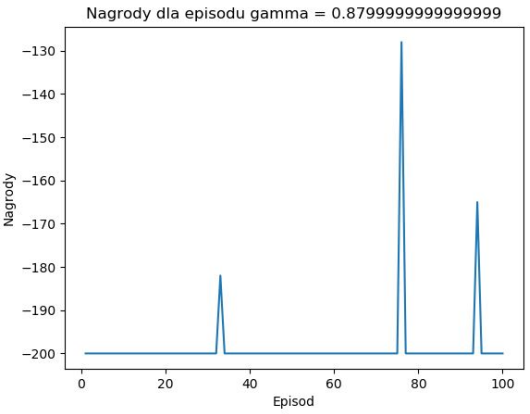
Natomiast zbyt wysokie mogą spowodować że pominiemy rozwiązanie optymalne.

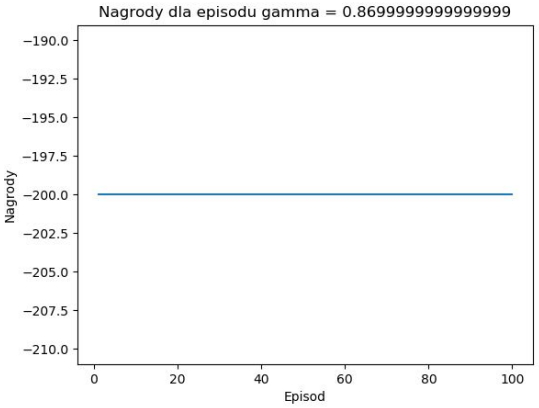
Zatem najlepsze wartości parametru eta znajdują się w przedziale pomiędzy 0.001 a 0.2.

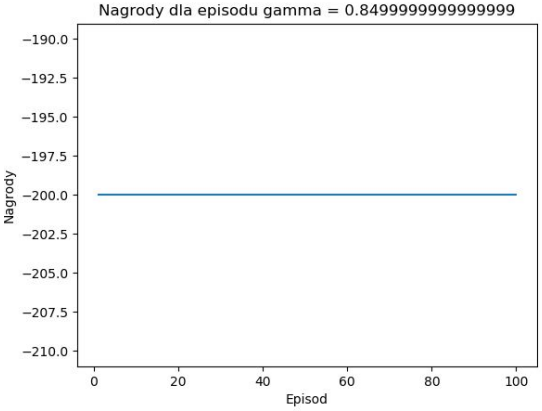
Teraz zbadamy w podobny sposób jakie powstaną modele dla różnych wartości parametru gamma.







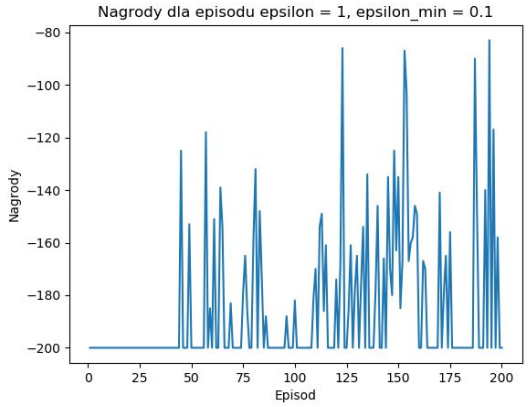


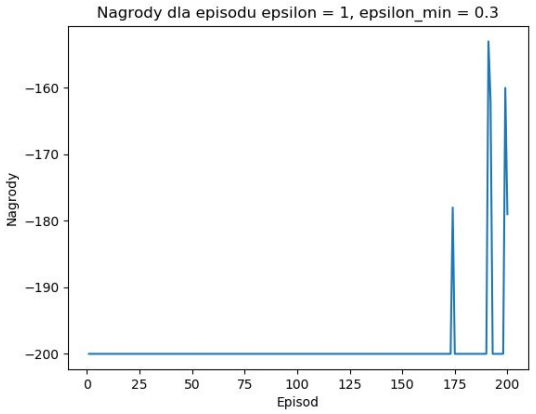


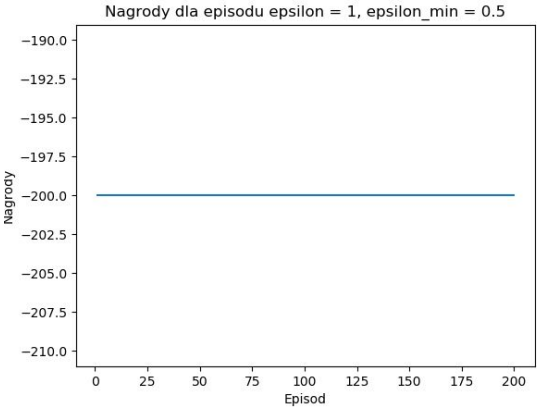
W przypadku parametru gamma widzimy że powinniśmy wybierać dość duże wartości bliskie 1. Widzimy że otrzymujemy modele które były w stanie w danych epizodach osiągnąć wysokie nagrody gdy parametr gamma jest > 86. Ten parametr mówi nam jakie znaczenie mają dla nas późniejsze nagrody im wyższa wartość gamma tym bardziej bierzemy pod uwagę późniejsze nagrody.

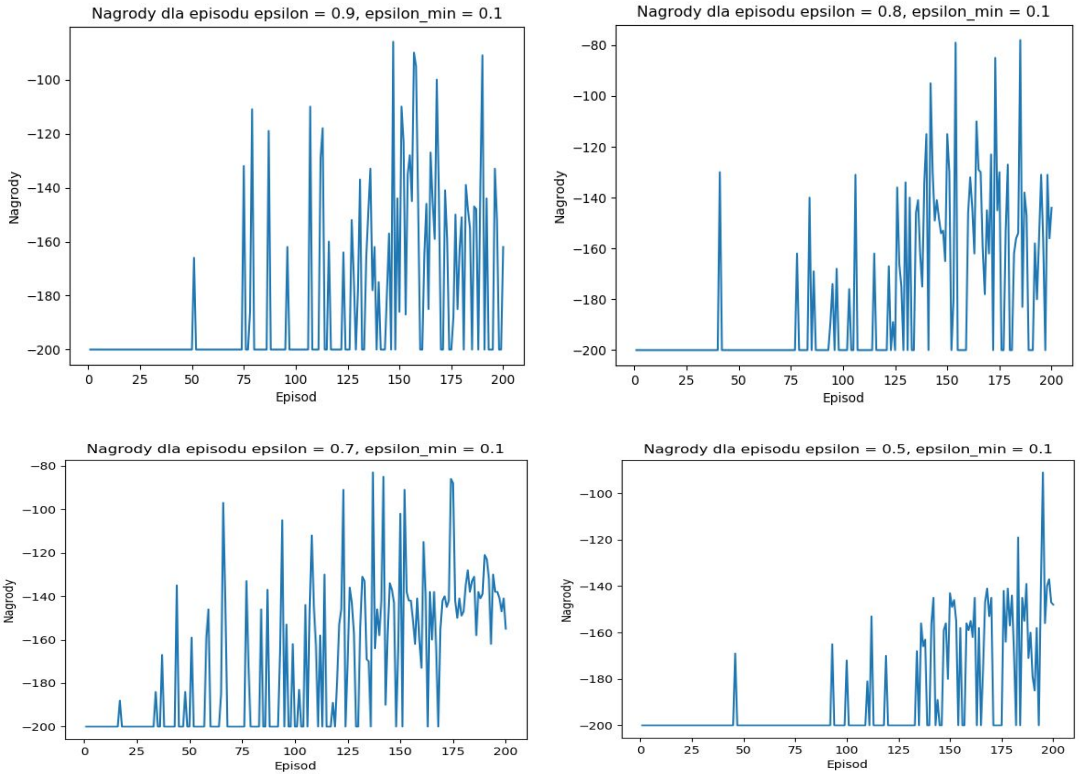
Zbadajmy teraz jak różne wartości epsilon, epsilon_min oraz epsilon_decay wpływają na nasz model. Parametry te służą do określania kiedy wykonywać eksploatację a kiedy eksploracje.

Epsilon początkowo ma wartość 1 jednak jego wartość w każdej iteracji jest zmniejszana o epsilon_decay. W programie losujemy losową liczbę w przedziale 0-1 i sprawdzamy czy jest ona mniejsza od większej z wartości epsilon oraz epsilon_min. Jeśli tak wykonujemy eksploatację, jeśli nie to wykonujemy eksploracje.





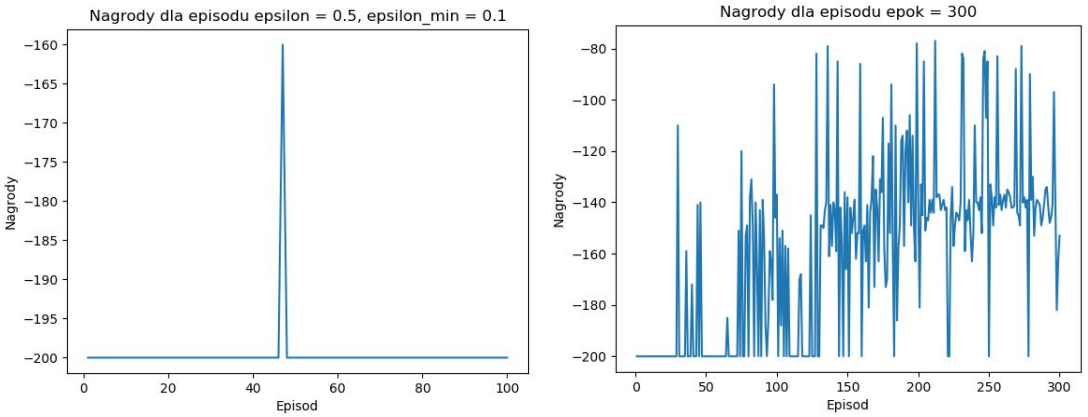


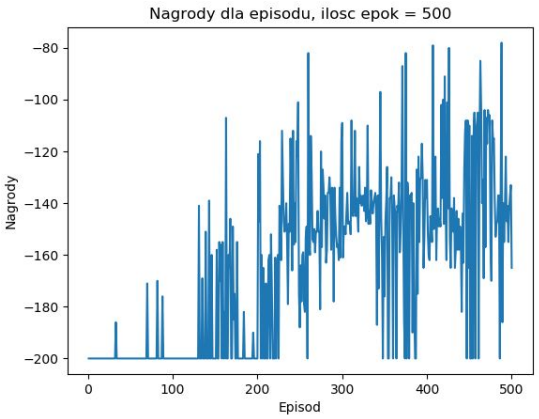


Możemy zaobserwować, że dla coraz większych wartości epsilon_min otrzymujemy modele z coraz mniejszymi nagrodami. Może to być spowodowane częstszą eksploatacją niż eksploracją, zatem aby ulepszać dane rozwiązanie, poszukujemy nowych.

Natomiast dla wartości epsilon od 1 do 0.6 dostajemy modele z dość wysokimi nagrodami, zatem te wartości tego parametru wydają się optymalne. Jednak warto dać tu wartość bliżej 1 aby na początku pozwolić modelowi dokonywać eksploatacji.

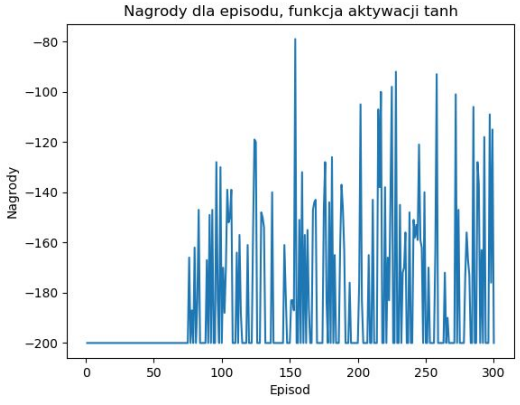
Teraz zbadajmy jakie wyniki otrzymamy zwiększając liczbę epizodów. Na podstawie poprzednich obserwacji dobierzmy następujące parametry: learning_rate = 0.05, gamma = 0.99, epsilon = 1, epsilon_min = 0.1.

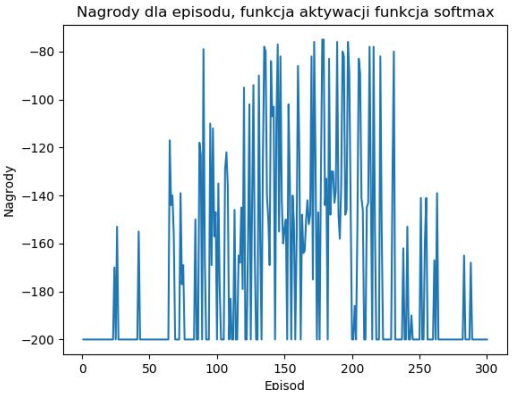


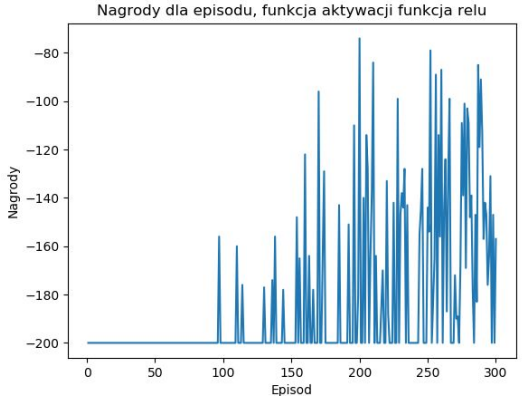


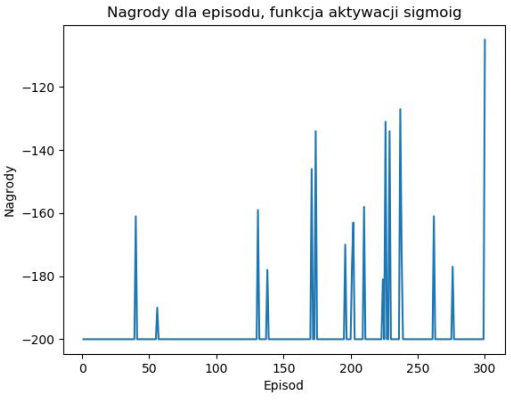
Widzimy, że wraz ze wzrostem epizodów otrzymujemy coraz lepszą nagrodę. Możemy zaobserwować, że ilość epizodów > 200 daje już dość dobre wyniki. Można określić przeprowadzając test na modelach, który opisaliśmy w dalszej części dokumentu, żę dla 300 episodów udaje nam się wytrenować model który bardzo dobrze sobie radzi w realnej grze, i osiąga szczyt wzgórza w każdej próbie. Po przeprowadzeniu kilku prób możemy zauważyć że modele wytrenowane w wyższych epizodach niż 100 sprawują się lepiej od tych w epizodach < 100.

Teraz sprawdzimy jak nasz model zachowuje się z dla różnych funkcji aktywacji dla danych warstw.









Funkcja tanh ma następujący wzór . Zaletami tej funkcji jest szerszy zakres wartości od funkcji logistycznej oraz obecność przedziału otwartego (-1,1).

Funkcja softmax - dzięki niej otrzymujemy prawdopodobieństwo przynależności do każdej z klas.

Funkcja relu (Rectifie linear unit) często jest wykorzystywana w głębokich sieciach neuronowych, jest to funkcja nieliniowa. Za jej pomocą możemy rozwiązać problem zanikającego gradientu.

Funkcja sigmoidalna to funkcja nieliniowa o wartościach wyjściowych w przedziale (0,1)

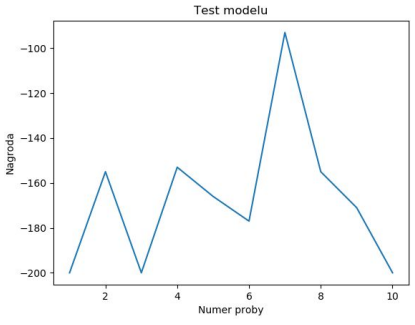
Najwięcej modeli otrzymaliśmy przy użyciu funkcji softmax, najmniej przy użyciu sigmoid. Po kilku próbach możemy określić że dla funkcji aktywacji ReLU nasz model daje najlepsze wyniki. Może to wynikać z faktu że funkcja ta pomaga
w rozwiązywaniu problemu znikającego gradientu który się pojawia dla funkcji tangensu hiperbolicznego i sigmoidalnej przy dużych wartościach całkowitego pobudzenia. Zbiór wartości funkcji relu znajduje się w przedziale
(0, +nieskończoności), dzięki temu problem znikającego gradientu nie występuje.

Sprawdźmy teraz jakie wyniki uzyska wytrenowany przez nas model. Aby to zrobić wytrenujmy agenta parametrami jak powyżej. Podczas trenowania, gdy otrzymamy nagrodę lepszą niż -199 zapisujemy model.

```
self.trainModel.save('./trainModelInEPS{}.h5'.format(eps))
```

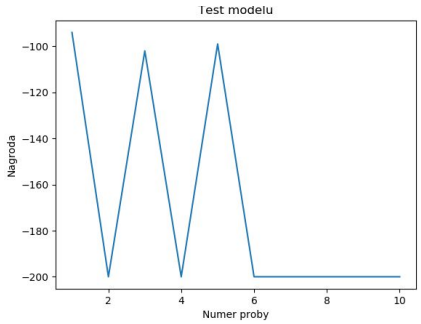
Kod testujący.

```
def test(env):
    sum_list = []
    for i in range(10):
        m = load_model("trainModelInEPS199.h5")
        c_s = env.reset().reshape(1,2)
        rew_sum = 0
        for i in range(200):
            env.render()
            act = np.argmax(m.predict(c_s)[0])
            o, r, d, z = env.step(act)
            new_state = o.reshape(1, 2)
            rew_sum += r
            c_s = new_state
            print(rew_sum)

            if d:
                print("Episode finished")
                break
        sum_list.append(rew_sum)
    plt.plot((np.arange(len(sum_list)) + 1), sum_list)
    plt.xlabel('Numer proby')
    plt.ylabel('Nagroda')
    plt.title('Test modelu' )
    plt.show()
    plt.close()
```

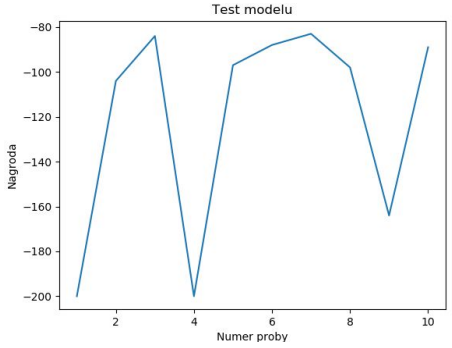


Następnie odczytajmy model wytrenowany w epizodzie 197 posiadającego nagrodę -144 i przetestujemy go kilkukrotnie. Jak widzimy nasz model sprawuje się całkiem dobrze. Na 10 prób udało mu się osiągnąć szczyt 9 razy. Najlepszy wynik wynosił około -90.

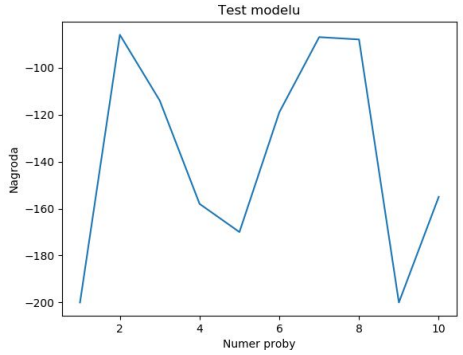
Teraz sprawdźmy wyniki innego modelu. Model ten został wytrenowany w epizodzie 176, a jego nagroda wynosiła -86.

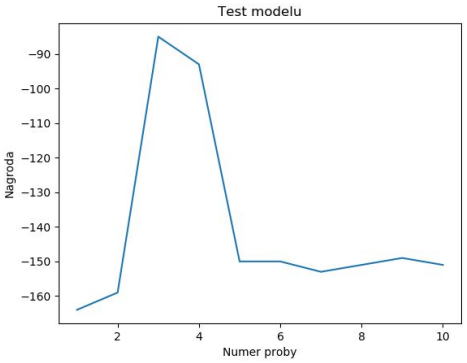


Jak widzimy ten model sprawuje się dużo gorzej od poprzedniego. Na 10 prób udało mu się osiągnąć szczyt wzgórza tylko 3 razy. Z najlepszym wynikiem około -90.



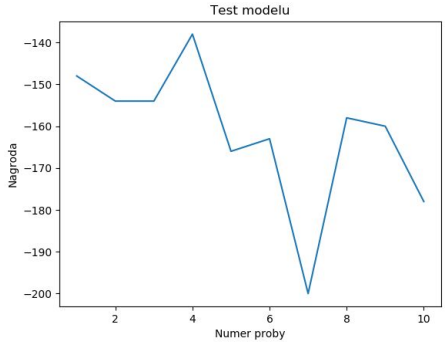
Powyższy model natomiast sprawuje się lepiej niż powyższe. Najwyższa nagroda dla tego modelu wynosi prawe -80. Model ten został wytrenowany w 187 epizodzie z nagrodą -137.





Wykres po lewej: dla epizodu 399. Nagroda -148. Widzimy że model nie radzi sobie najgorzej w 10 próbach poprawnie ukończył 8.

Wykres po prawej: dla epizodu 296. Nagroda -144. Radzi sobie bardzo dobrze. Ukończył poprawnie każdą próbę. W trzeciej próbie cel osiągnął w -90 krokach.



Powyższy wykres ukazuje wyniki dla modelu z epizodu 295 z nagrodą -135.

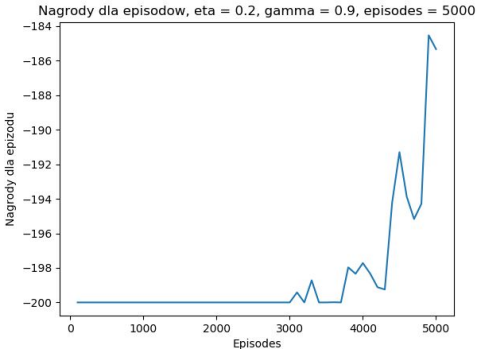
Z powyższych wykresów możemy wyciągnąć następujące wnioski:

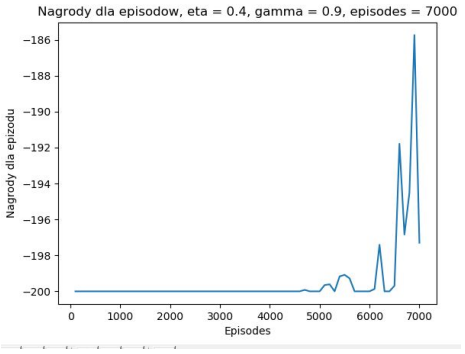
- to, że model na etapie uczenia osiągnął wysoką nagrodę, nie znaczy że będzie dobrze potrafił grać, w prawdziwej rozgrywce.

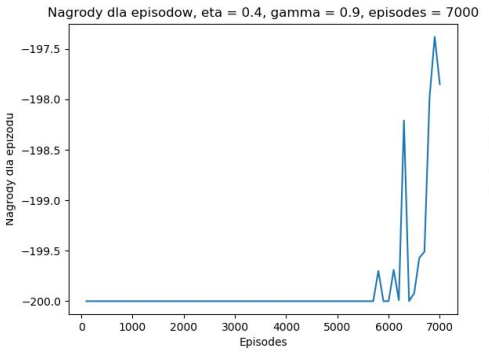
- jeśli wartość nagrody trenowanego agenta znajdowała się w przedziale -144 do 130, agent radził sobie najlepiej

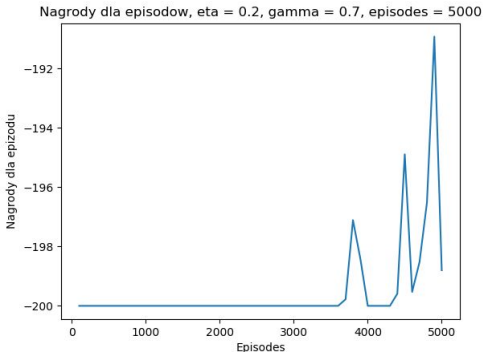
- modele wytrenowane w epizodach pomiędzy 150 a 300 radzą sobie lepiej od tych wytrenowanych wcześniej

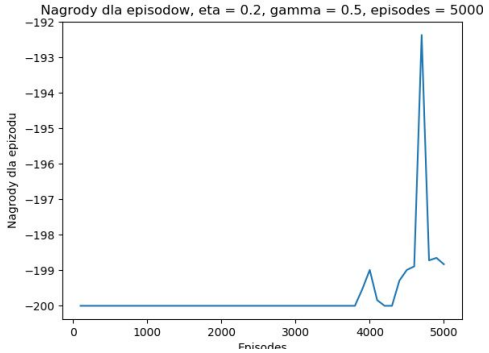
Na koniec przetestujmy jeszcze wyniki trenowania agenta przy użyciu tablicy q-table, dla różnych parametrów.

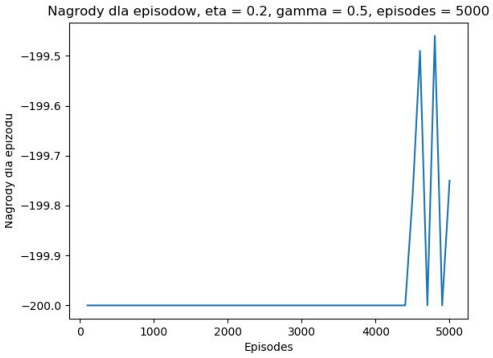












Obserwujemy że dla coraz większych wartości parametru eta otrzymujemy coraz mniejsze nagrody przy 5000 epizodach. Graniczną wartość możemy ustalić podobnie jak w przypadku DQ-learningu na 0.2.

Wraz z zmniejszaniem wartości parametru gamma maleją nagrody. Dobre wyniki są osiągane dla gamma z przedziału 0.8-0.99.

Różnice przy zastosowaniu q-learningu a deep q-learningu:

- w q-learningu potrzebujemy większą liczbę epizodów, a uczenie przebiega szybciej dla tej samej liczby epizodów.

- zmienne hiperparametryczne oddziałują podobnie na wynik końcowego agenta.

- w q-learningu potrzebujemy stany i akcje w postaci dyskretnej, możliwej do przedstawienia w tabeli, deep q-learning nie ma takiego ograniczenia. Ponadto ilość stanów może się zwiększać wraz z upływem czasu w deep q-learningu.

6. Wnioski

Wytrenowanie dobrego agenta w algorytmach uczenia ze wzmocnieniem, jest bardzo trudnym zadaniem. Wpływ na to ma m.in obecność pewnej losowości w danych trenujących, brak jasnych metod sprawdzania skuteczności modelu, długi czas trenowania.

Kolejnym problemem jest dobór odpowiedniego środowiska oraz wybór odpowiedniej funkcji nagrody za podjęte akcje. Jednak dzięki OpenAI możemy trenować naszych agentów na pokaźnej liczbie dobrze dobranych środowisk.