Project Reinforcement Learning

Mohammad Taha Majlesi

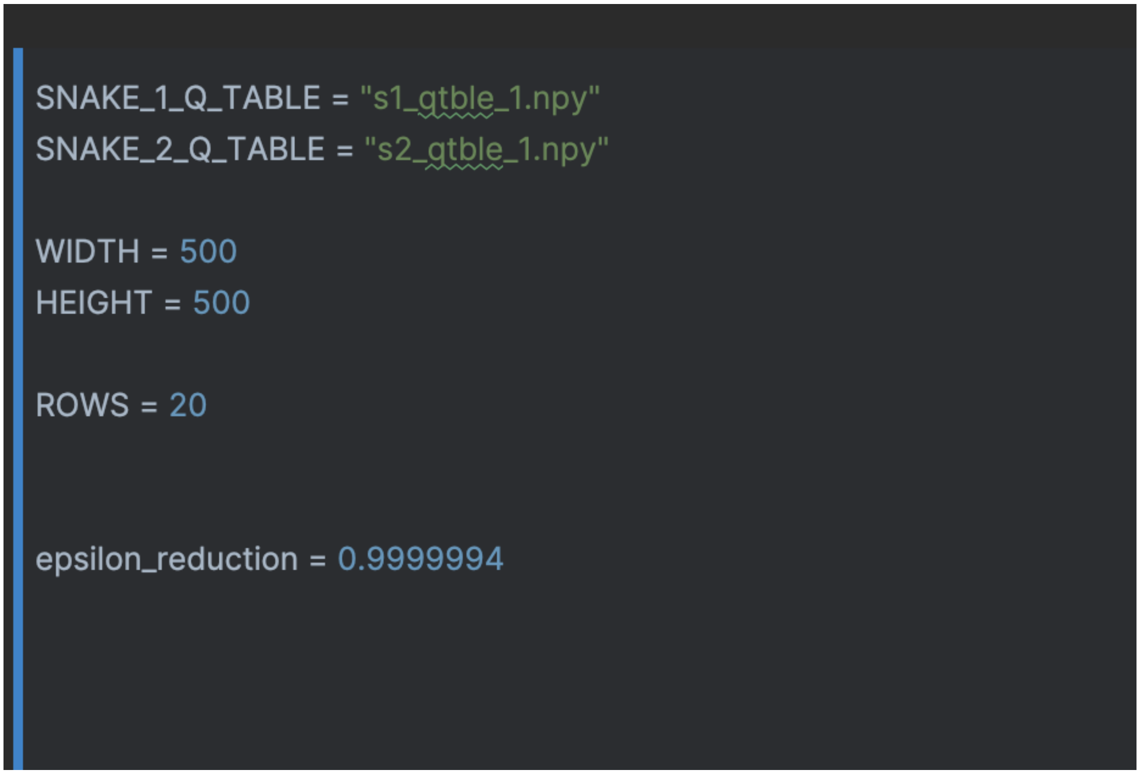
810101504

version 1 :

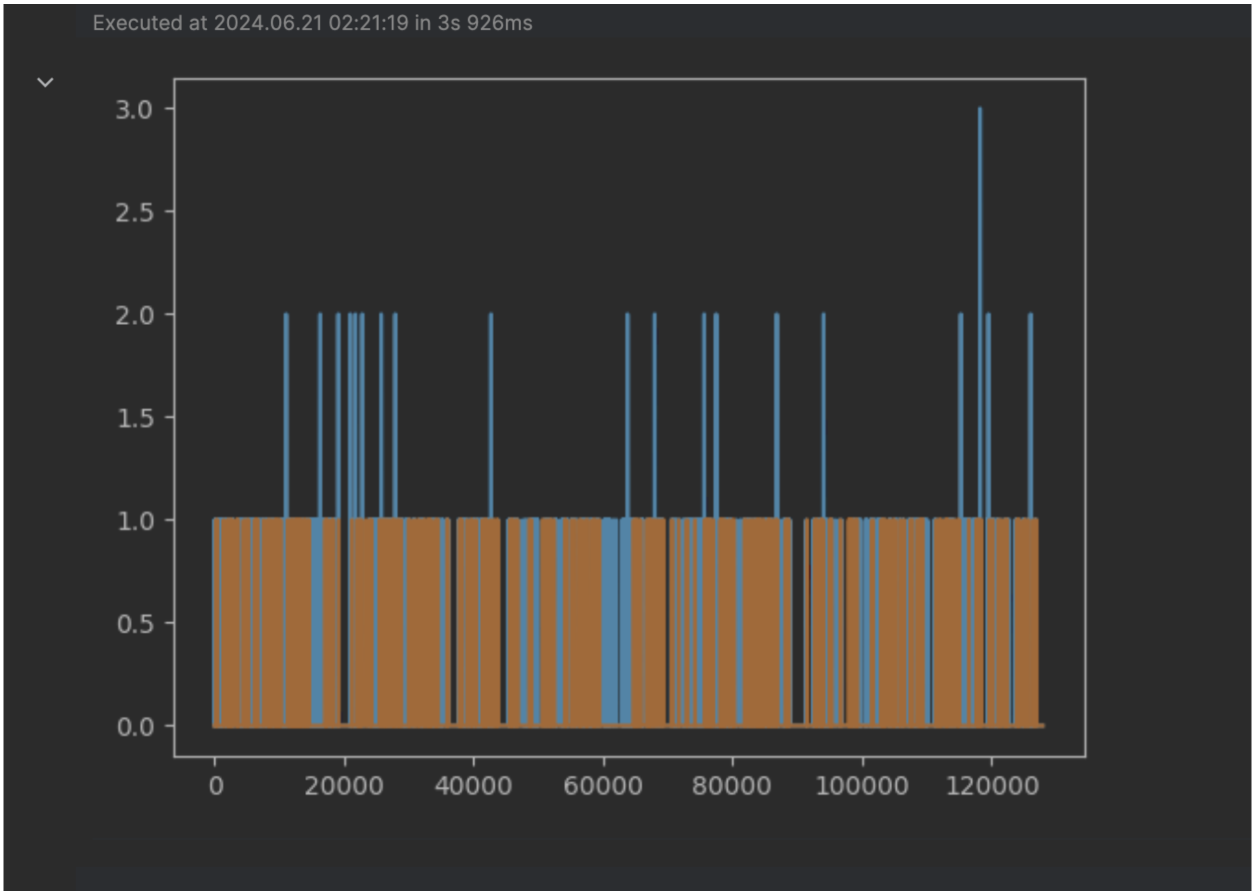
در ورشن اولیه با این مقادیر به این صورت خروجی را داریم که در پایان به مقادیر خوبی میل خواهد کرد

این کد اولیه ی مار است :

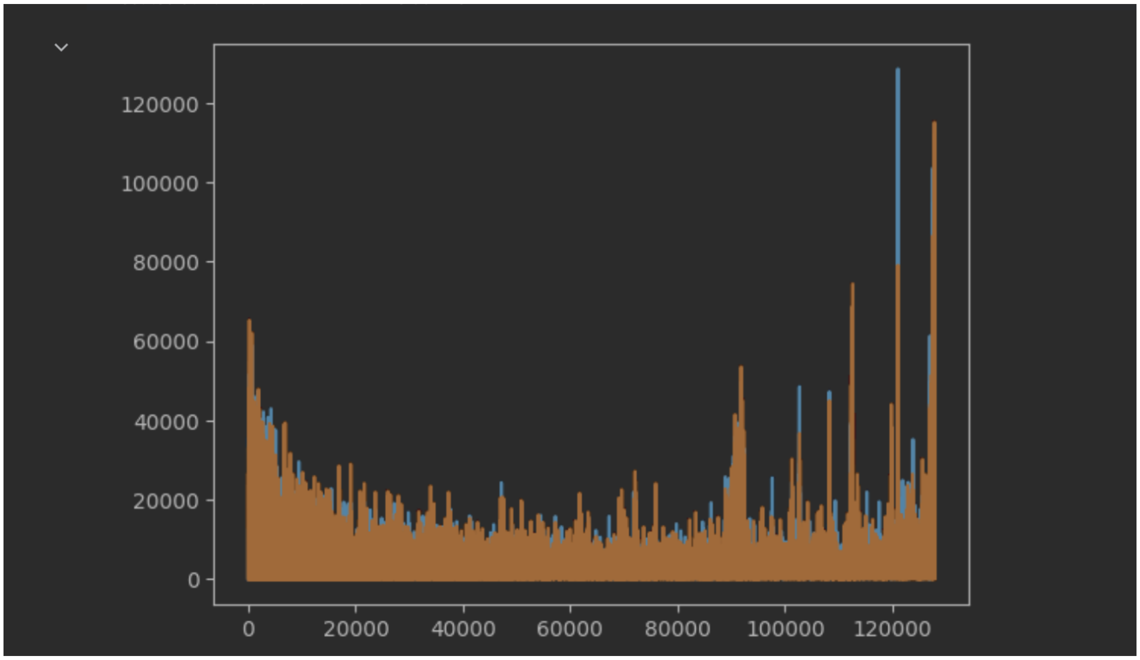
```
import random
import random
import numpy as np


class Snake:

    def __init__(self, color, pos, file_name=None):
        self.color = color
        self.head = Cube(pos, color=color)
        self.body.append(self.head)
        self.num_of_wins = 0
        self.total_reward = 0
        self.dirnx = 0
        self.dirny = 1
        self.q_table = np.load(file_name,allow_pickle=True).item() if os.path.ex

        self.lr = 0.4
        self.discount_factor = 0.9
        self.epsilon = 0.95

    def get_state(self , snake, other_snake):
        head_x, head_y = self.head.pos
        state = (head_x, head_y, other_snake.head.pos[0], other_snake.head.pos[1
        return state

    def get_optimal_policy(self, state):
        if state not in self.q_table:
            self.q_table[state] = np.random.uniform(0, 0, 4)
        return np.argmax(self.q_table[state])


    def make_action(self, state):
        chance = random.random()
        if chance < self.epsilon:
            action = random.randint(0, 3)
        else:
            action = self.get_optimal_policy(state)
        return action

    def update_q_table(self, state, action, next_state, reward):
        if state not in self.q_table:
            self.q_table[state] = np.random.uniform(0, 0, 4)
        if next_state not in self.q_table:
            self.q_table[next_state] = np.random.uniform(0, 0, 4)
        self.q_table[state][action] = self.q_table[state][action] + self.lr * (
            reward
            + self.discount_factor * np.max(self.q_table[next_state])
            - self.q_table[state][action]
        )

    def move(self, snack, other_snake):
        state = self.get_state(snack, other_snake)
        action = self.make_action(state)

        if action == 0:
            self.dirnx = -1
            self.dirny = 0
            self.turns[self.head.pos[:]] = [self.dirnx, self.dirny]
        elif action == 1:
            self.dirnx = 1
            self.dirny = 0
            self.turns[self.head.pos[:]] = [self.dirnx, self.dirny]
        elif action == 2:
            self.dirny = -1
            self.dirnx = 0
            self.turns[self.head.pos[:]] = [self.dirnx, self.dirny]
        elif action == 3:
            self.dirny = 1
            self.dirnx = 0
            self.turns[self.head.pos[:]] = [self.dirnx, self.dirny]

        for i, c in enumerate(self.body):
            p = c.pos[:]
            if p in self.turns:
                turn = self.turns[p]
                c.move(turn[0], turn[1])
                if i == len(self.body) - 1:
                    self.turns.pop(p)
            else:
                c.move(c.dirnx, c.dirny)

        next_state = self.get_state(snack, other_snake)


        return state, next_state, action


    def check_out_of_board(self):
        headPos = self.head.pos
        if headPos[0] >= ROWS - 1 or headPos[0] < 1 or headPos[1] >= ROWS - 1 or
            self.reset((random.randint(3, 18), random.randint(3, 18)))
            return True
        return False

    def calc_reward(self, snack, other_snake):
        reward = 0
        win_self, win_other = False, False

        current_distance = self.distance_to_snack(snack)

        if self.check_out_of_board():
            reward -= 100
            win_other = True
            self.reset((random.randint(3, 18), random.randint(3, 18)))

        if self.head.pos == snack.pos:
            self.addCube()
            snack = Cube(randomSnack(ROWS, self), color=(0, 255, 0))
            reward += 100
            self.num_of_wins += 1
            # print("snack")
        if self.head.pos in list(map(lambda z: z.pos, self.body[1:])):
            reward -= 100
            win_other = True
            self.reset((random.randint(3, 18), random.randint(3, 18)))

        if self.head.pos in list(map(lambda z: z.pos, other_snake.body)):
            if self.head.pos != other_snake.head.pos:
                reward -= 100
                win_other = True
            else:
                if len(self.body) > len(other_snake.body):
                    reward += 100
                    win_self = True
                elif len(self.body) == len(other_snake.body):
                    reward = 0
                else:
                    reward -= 100
                    win_other = True
            self.reset((random.randint(3, 18), random.randint(3, 18)))

        distance_to_snake = self.distance_to_snack(snack)
        distance_to_other_snake = self.avrage_distance_to_other_snake(other_snak
        if not distance_to_snake == 0 and not distance_to_other_snake == 0:
            reward += 1/( distance_to_snake)*1000
        # print("reward: ", reward)







        if self.epsilon > 0.1:
            self.epsilon = self.epsilon * epsilon_reduction


        return snack, reward, win_self, win_other

    def avrage_distance_to_other_snake(self, other_snake):
        sum = 0
        for cube in other_snake.body:
            sum += self.distance_to_snack(cube)
        return sum / len(other_snake.body)

    def distance_to_snack(self, snack):
        head_x, head_y = self.head.pos
        snack_x, snack_y = snack.pos

        return abs(head_x - snack_x) + abs(head_y - snack_y)


    def reset(self, pos):
        self.head = Cube(pos, color=self.color)
        self.body = []
        self.body.append(self.head)
        self.turns = {}
        self.dirnx = 0
        self.dirny = 1
        self.num_of_wins = 0
        self.total_reward = 0


    def addCube(self):
        tail = self.body[-1]
        dx, dy = tail.dirnx, tail.dirny

        if dx == 1 and dy == 0:
            self.body.append(Cube((tail.pos[0] - 1, tail.pos[1]), color=self.col
        elif dx == -1 and dy == 0:
            self.body.append(Cube((tail.pos[0] + 1, tail.pos[1]), color=self.col
        elif dx == 0 and dy == 1:
            self.body.append(Cube((tail.pos[0], tail.pos[1] - 1), color=self.col
        elif dx == 0 and dy == -1:
            self.body.append(Cube((tail.pos[0], tail.pos[1] + 1), color=self.col

        self.body[-1].dirnx = dx
        self.body[-1].dirny = dy

    def draw(self, surface):
        for i, c in enumerate(self.body):
            if i == 0:
                c.draw(surface, True)
            else:
                c.draw(surface)

    def save_q_table(self, file_name):
        np.save(file_name, self.q_table)
```

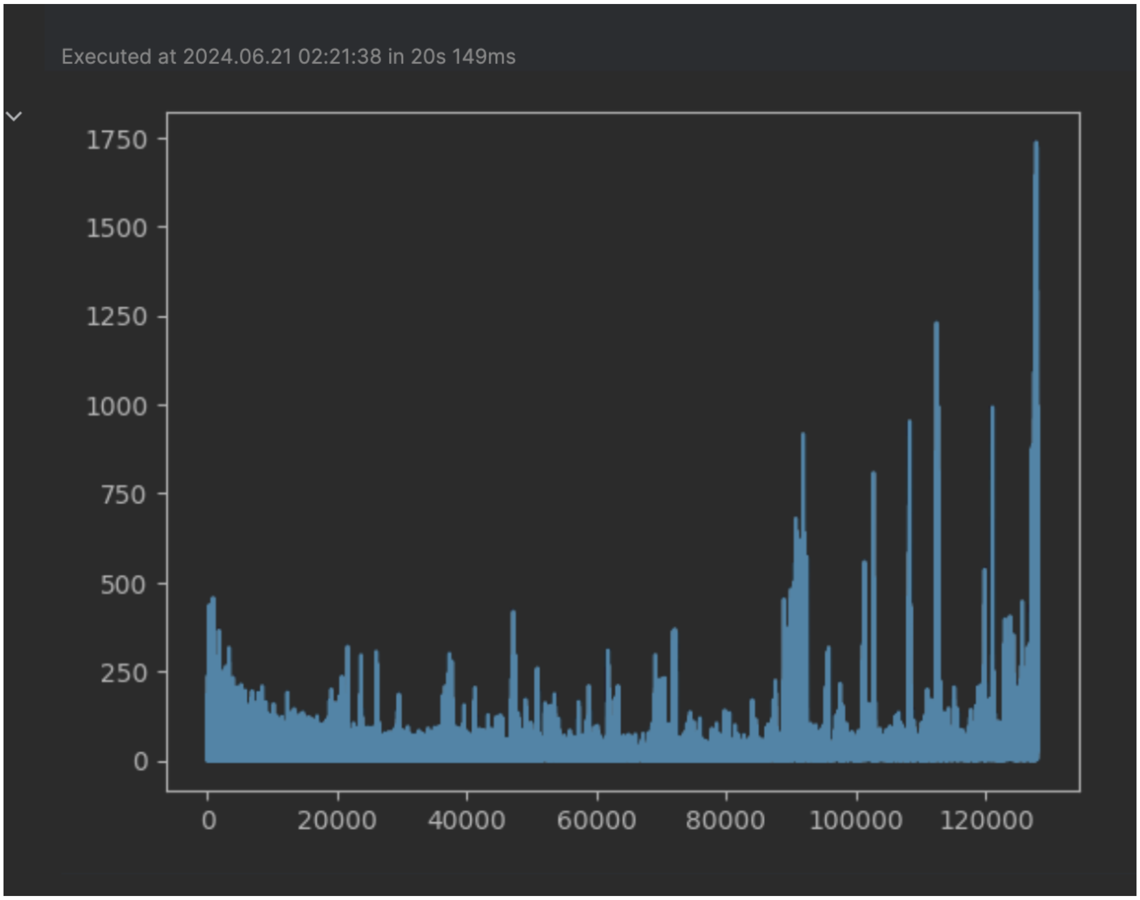


```
SNAKE_1_Q_TABLE = "s1_qtble_1.npy"
SNAKE_2_Q_TABLE = "s2_qtble_1.npy"

WIDTH = 500
HEIGHT = 500

ROWS = 20


epsilon_reduction = 0.9999994
```



Executed at 2024.06.21 02:21:19 in 3s 926ms

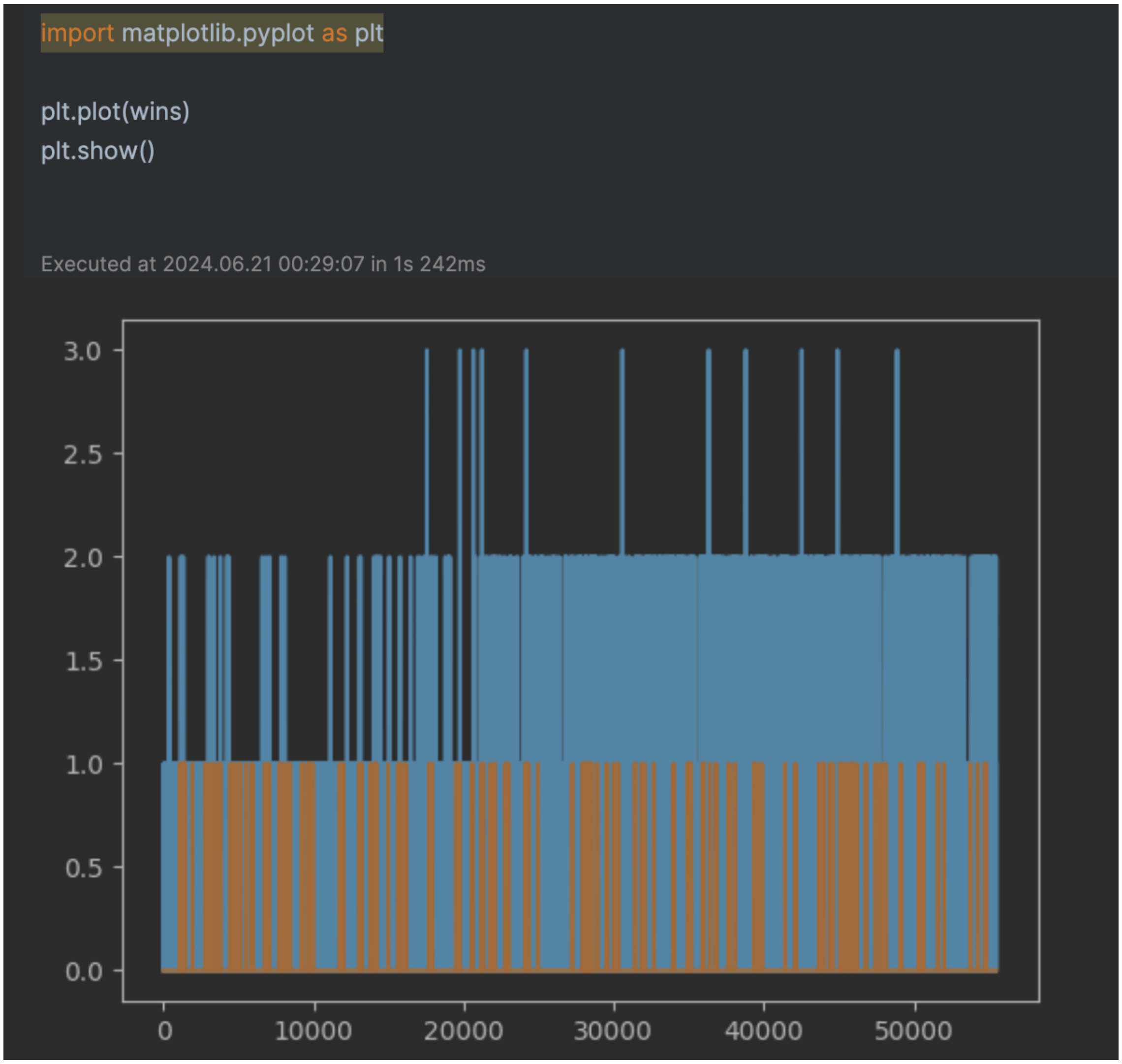



Executed at 2024.06.21 02:21:38 in 20s 149ms

در این قسمت حجم فایل اماده شده بسیار بالا بود و قابل ادلود کردن نبود

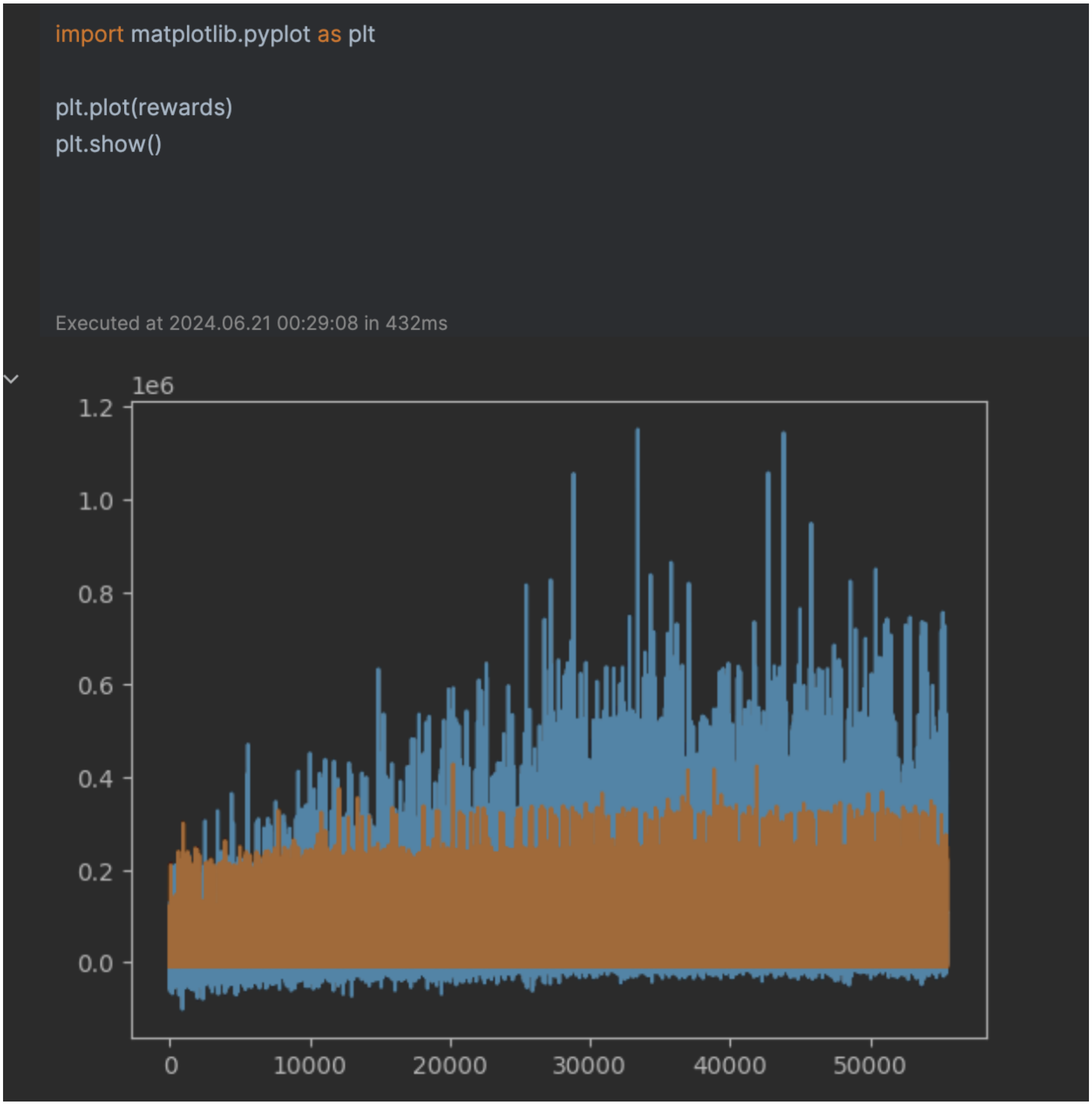
حدودا دارای ۵ ملیون پارامتر بود و چون حجم زیادی میخواست از روش های دیگر استفاده کردیم در پایین :

version 2 :



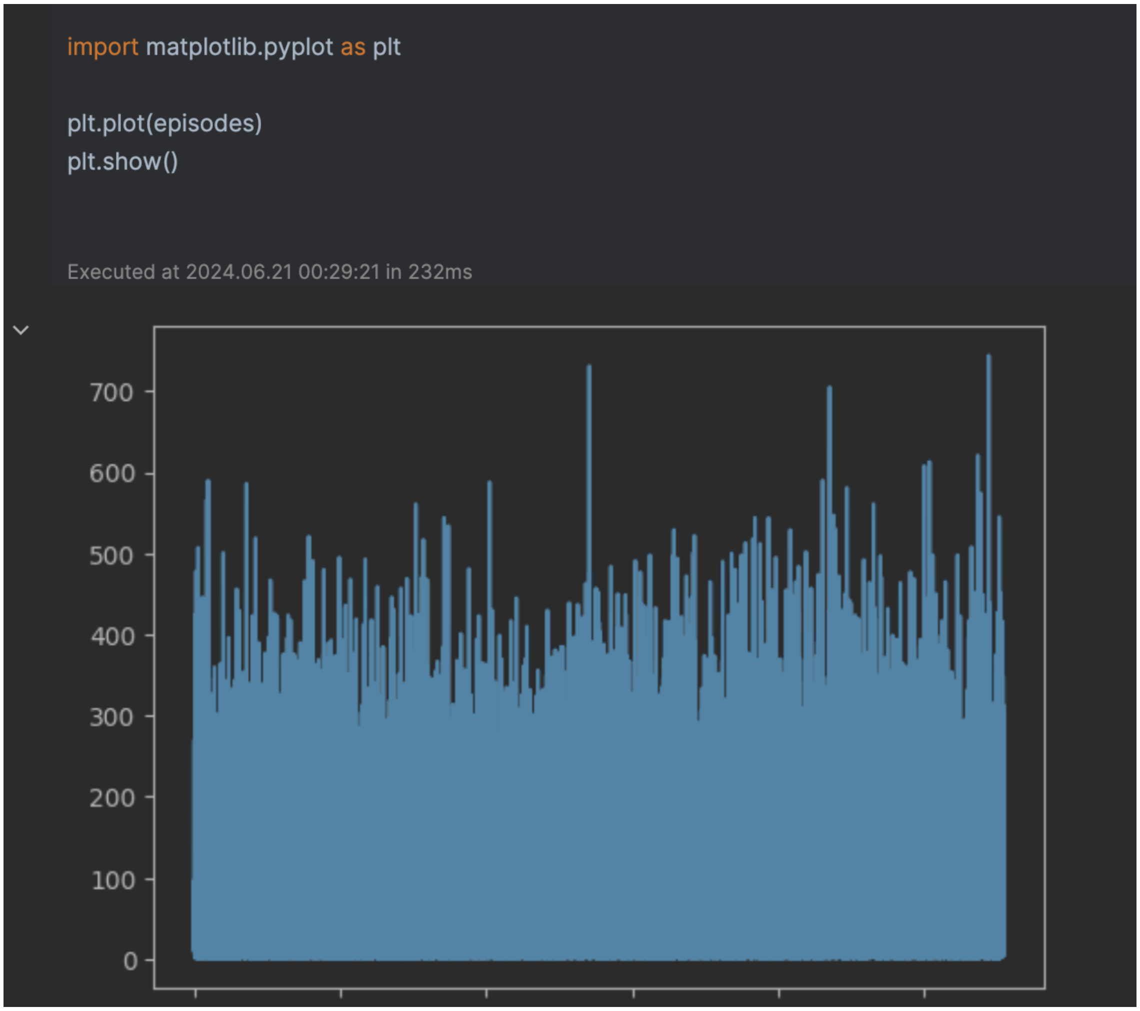
```
import matplotlib.pyplot as plt

plt.plot(wins)
plt.show()
```

Executed at 2024.06.21 00:29:07 in 1s 242ms



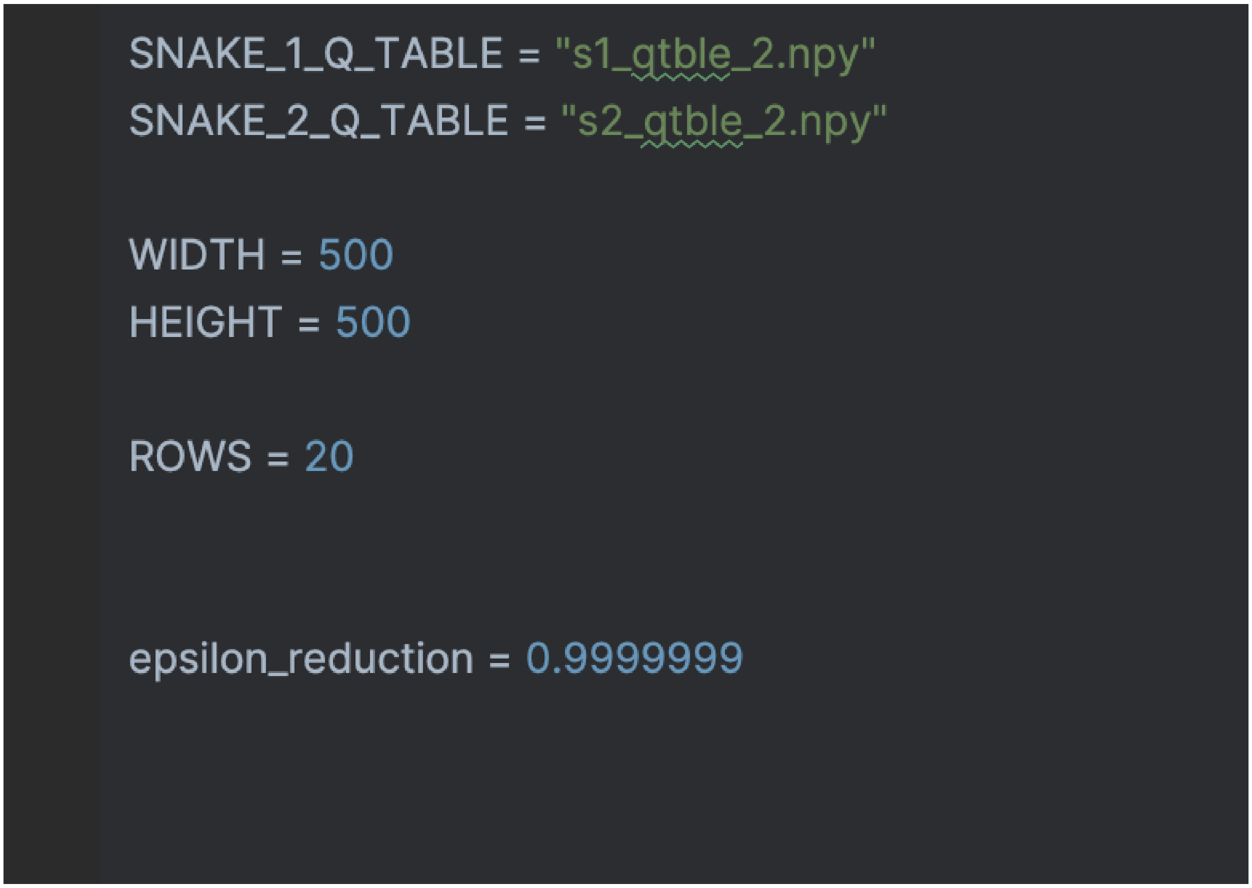
```
import matplotlib.pyplot as plt

plt.plot(rewards)
plt.show()
```

```
Executed at 2024.06.21 00:29:08 in 432ms
```

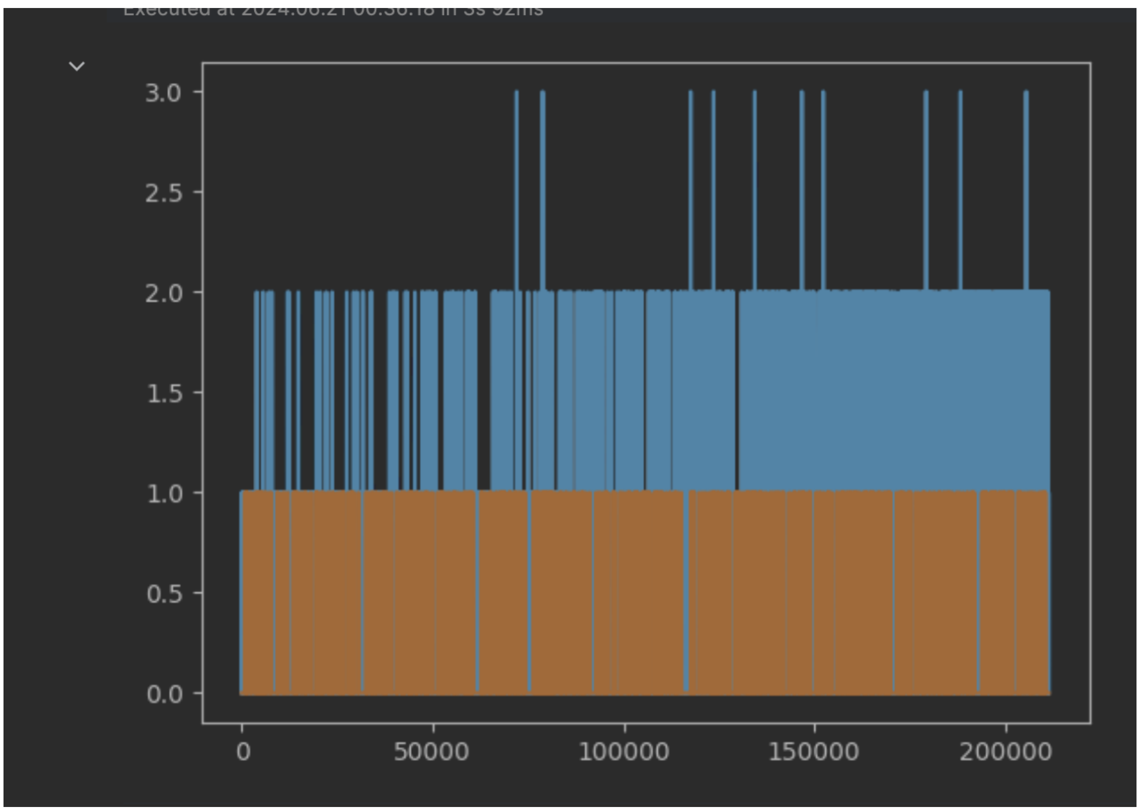


```python
import matplotlib.pyplot as plt

plt.plot(episodes)
plt.show()
```

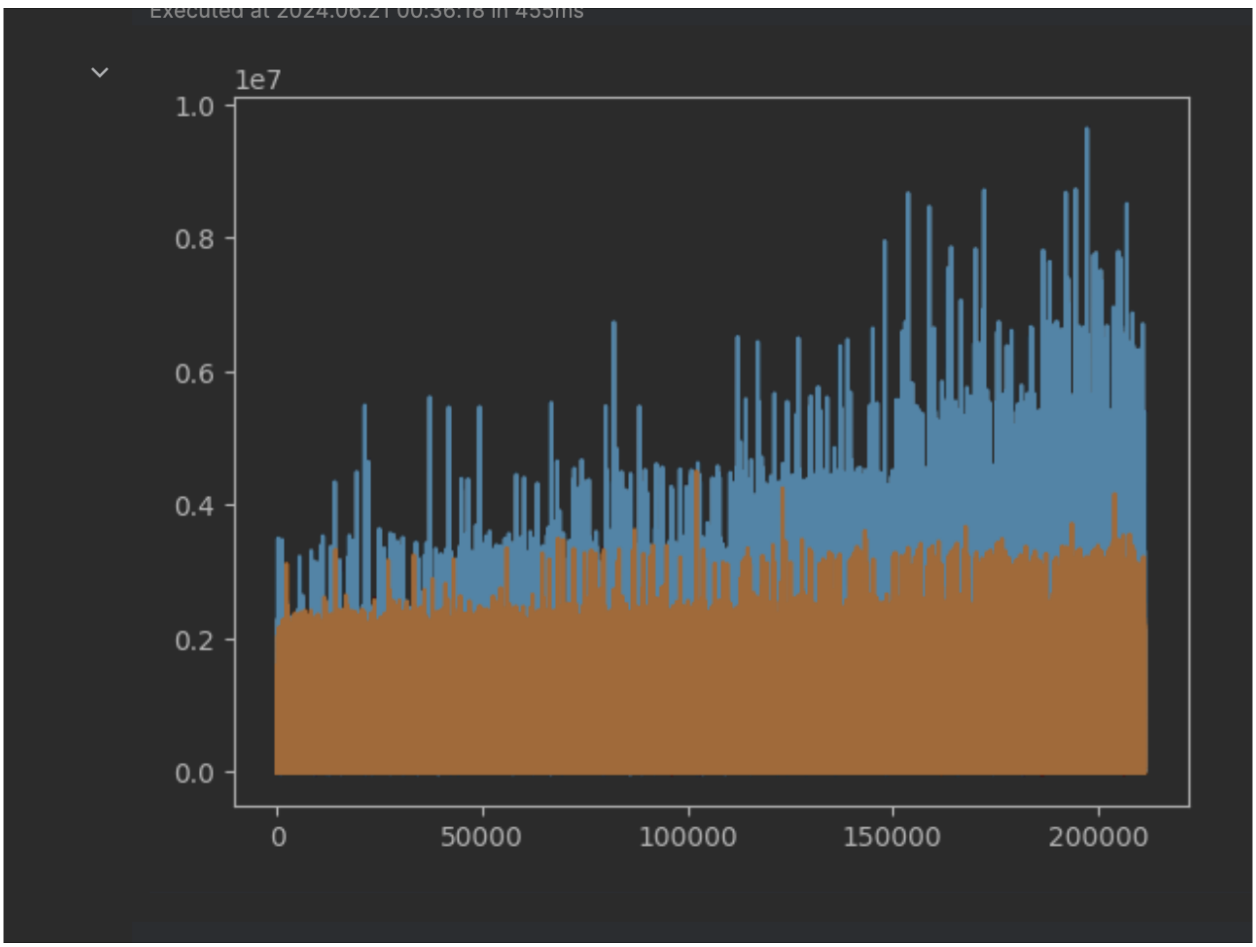
Executed at 2024.06.21 00:29:21 in 232ms



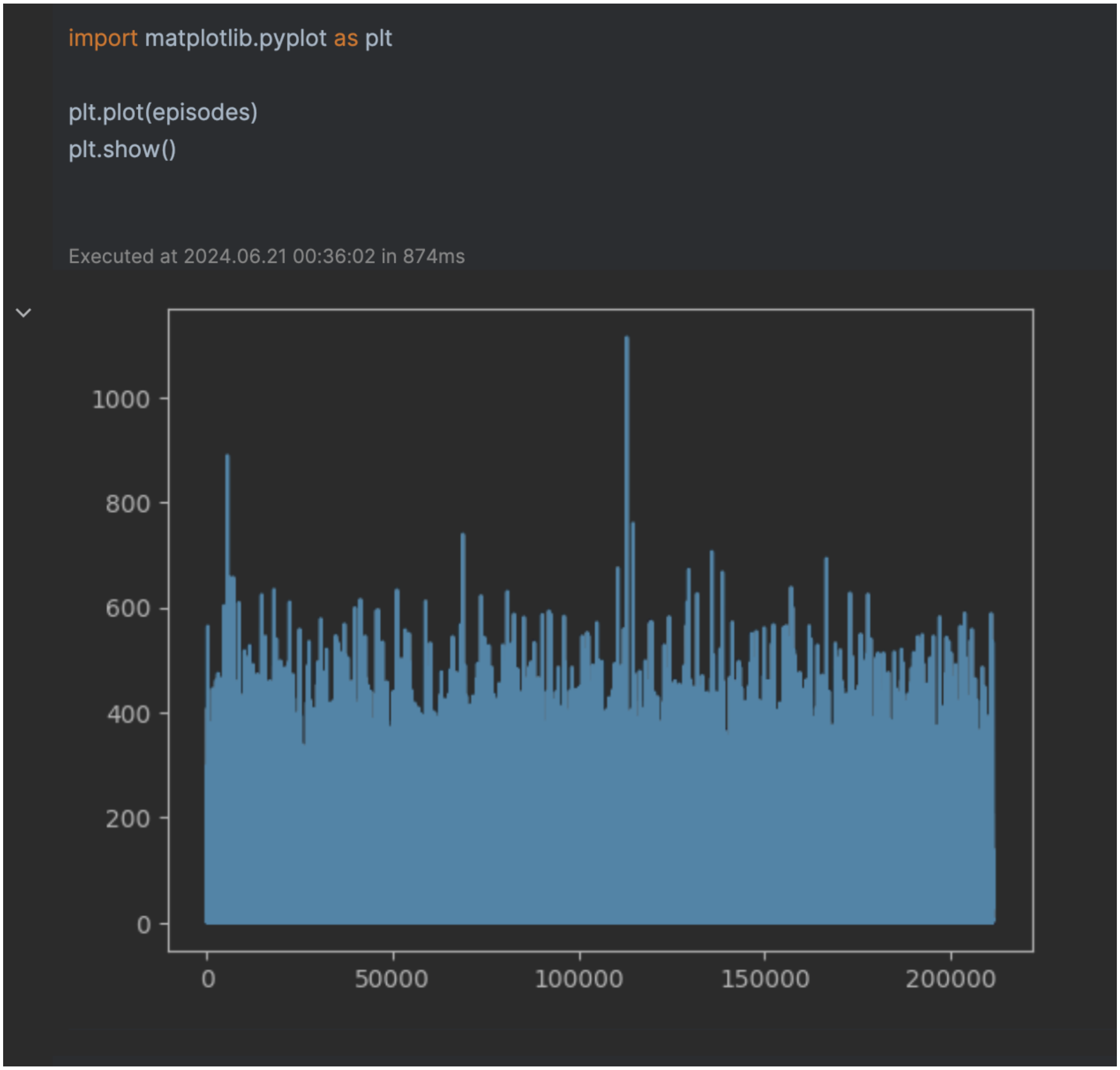
```
SNAKE_1_Q_TABLE = "s1_qtble_2.npy"
SNAKE_2_Q_TABLE = "s2_qtble_2.npy"

WIDTH = 500
HEIGHT = 500

ROWS = 20


epsilon_reduction = 0.9999999
```



Executed at 2024.06.21 00:36:18 in 3s 92ms



Executed at 2024.06.21 00:36:18 in 455ms

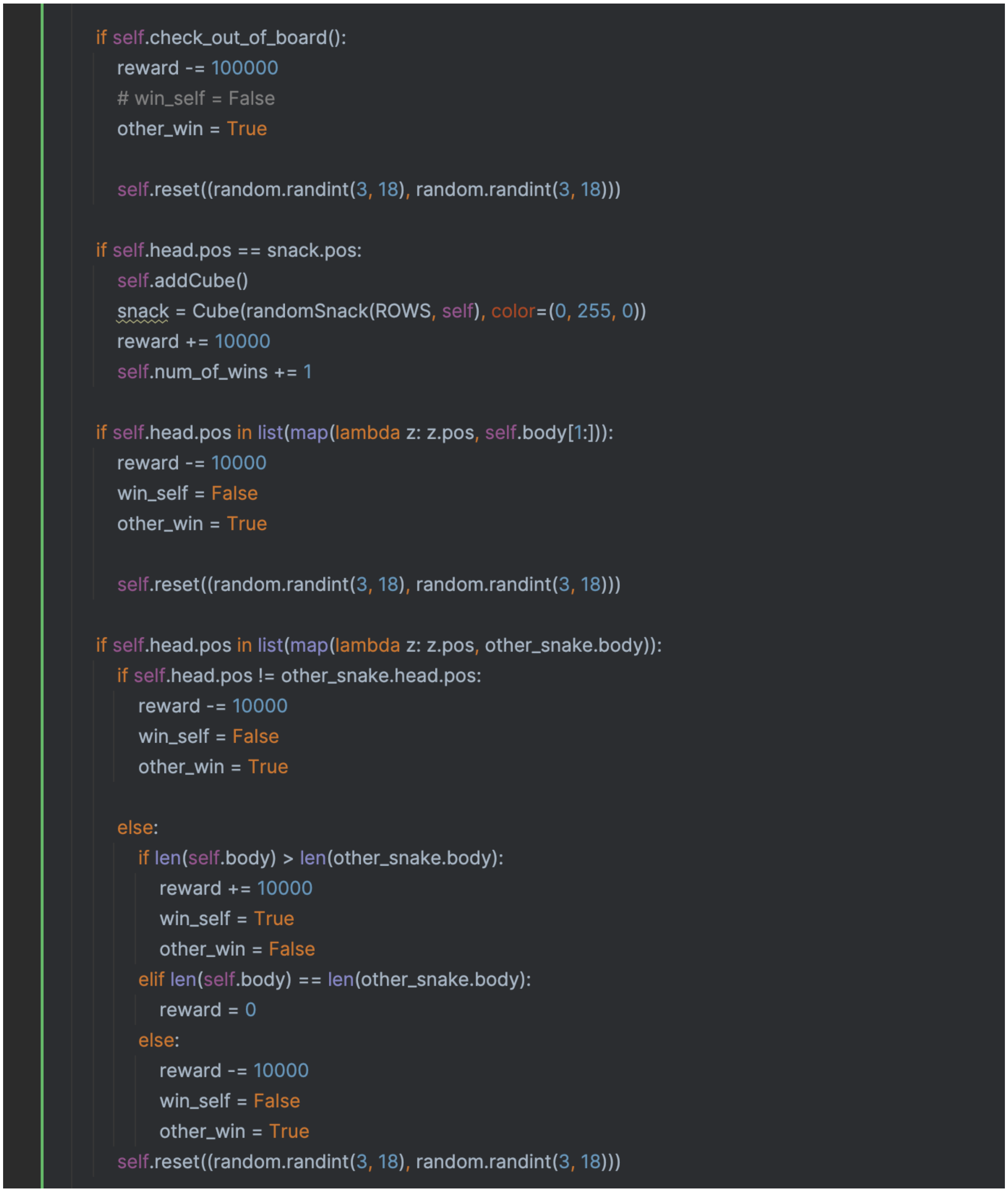


```
import matplotlib.pyplot as plt

plt.plot(episodes)
plt.show()
```

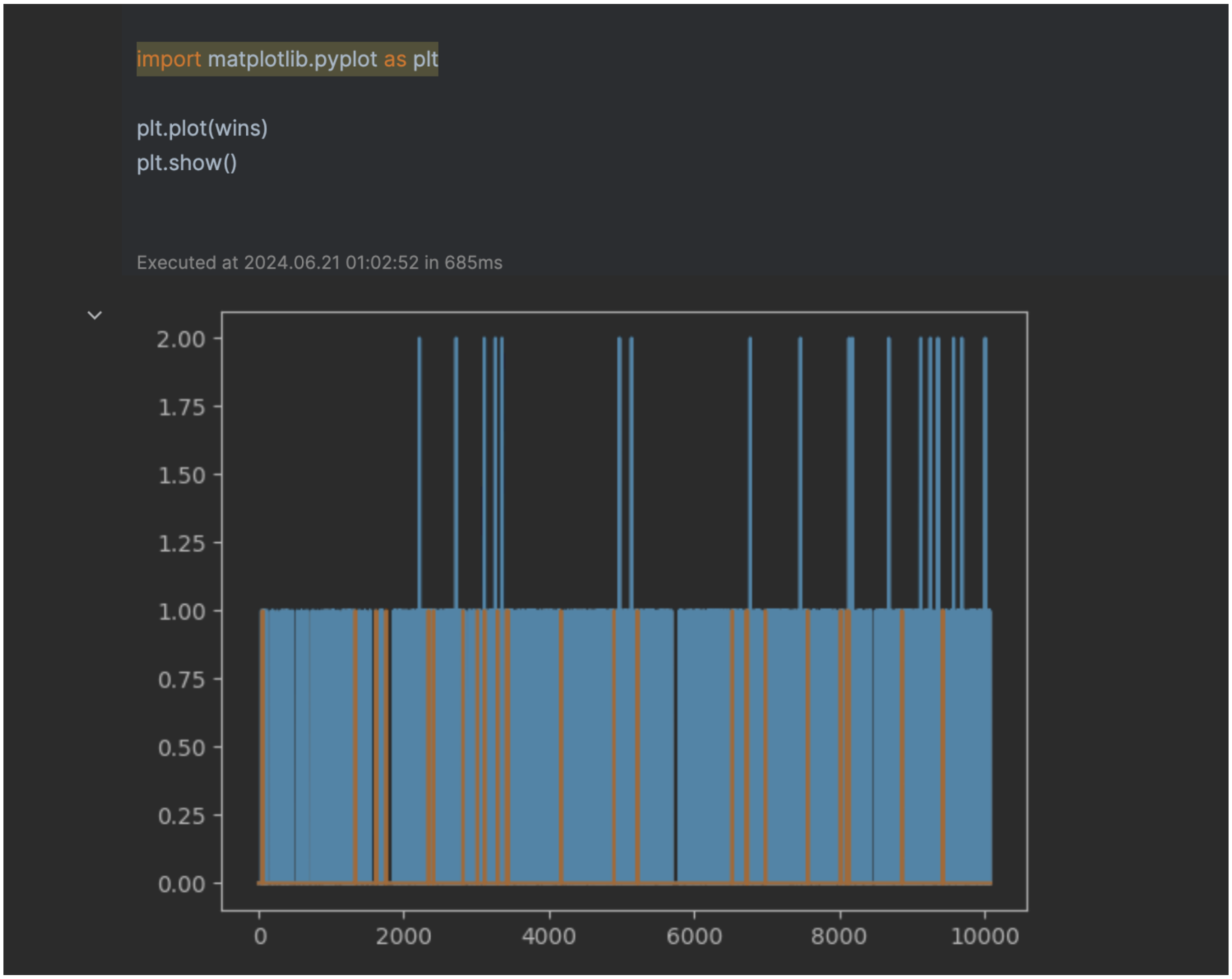
Executed at 2024.06.21 00:36:02 in 874ms

Version 3:

change on this part for rewards :

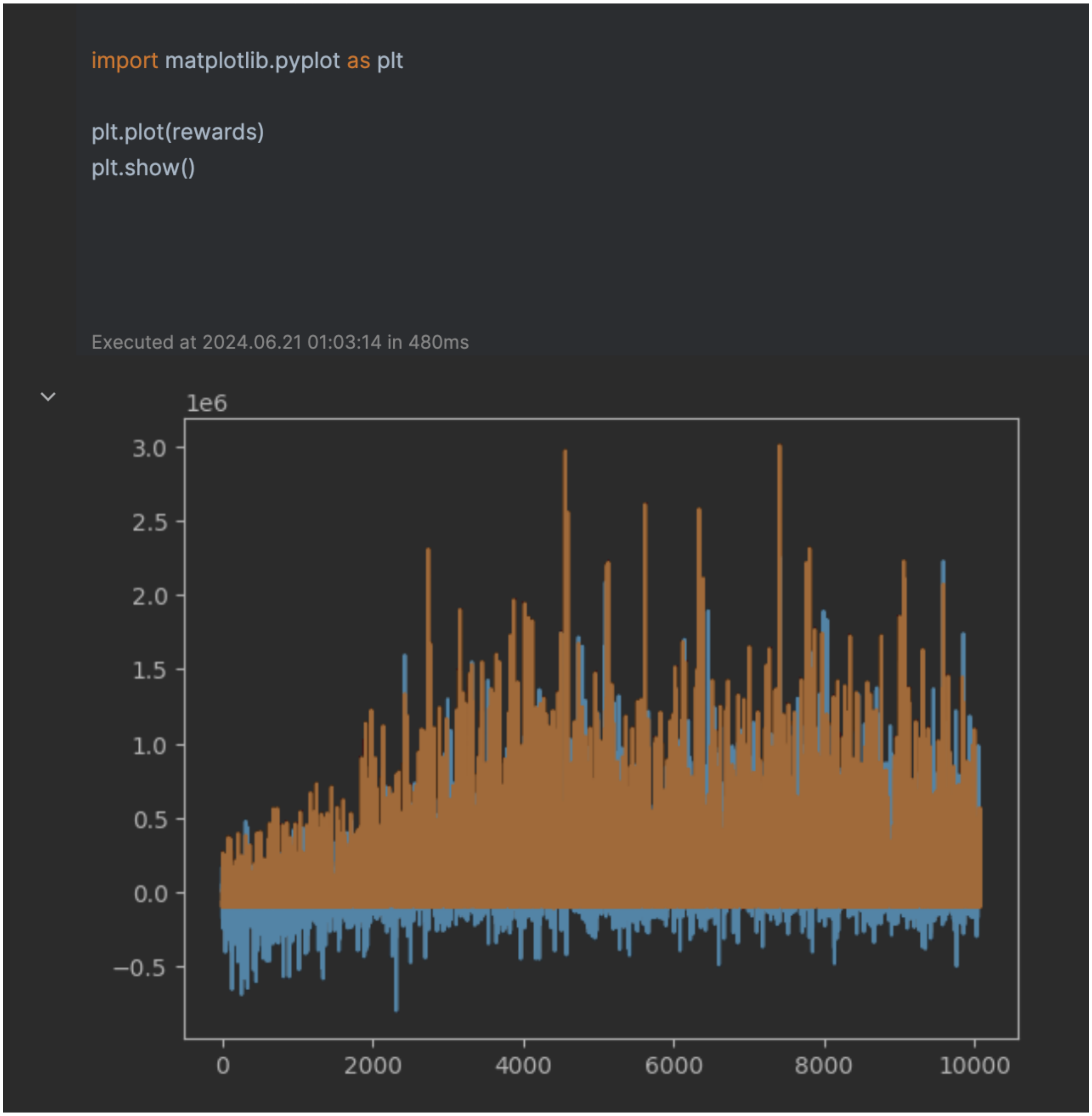


```
        if self.check_out_of_board():
            reward -= 100000
            # win_self = False
            other_win = True

            self.reset((random.randint(3, 18), random.randint(3, 18)))

        if self.head.pos == snack.pos:
            self.addCube()
            snack = Cube(randomSnack(ROWS, self), color=(0, 255, 0))
            reward += 10000
            self.num_of_wins += 1

        if self.head.pos in list(map(lambda z: z.pos, self.body[1:])):
            reward -= 10000
            win_self = False
            other_win = True

            self.reset((random.randint(3, 18), random.randint(3, 18)))

        if self.head.pos in list(map(lambda z: z.pos, other_snake.body)):
            if self.head.pos != other_snake.head.pos:
                reward -= 10000
                win_self = False
                other_win = True

            else:
                if len(self.body) > len(other_snake.body):
                    reward += 10000
                    win_self = True
                    other_win = False
                elif len(self.body) == len(other_snake.body):
                    reward = 0
                else:
                    reward -= 10000
                    win_self = False
                    other_win = True
            self.reset((random.randint(3, 18), random.randint(3, 18)))
```



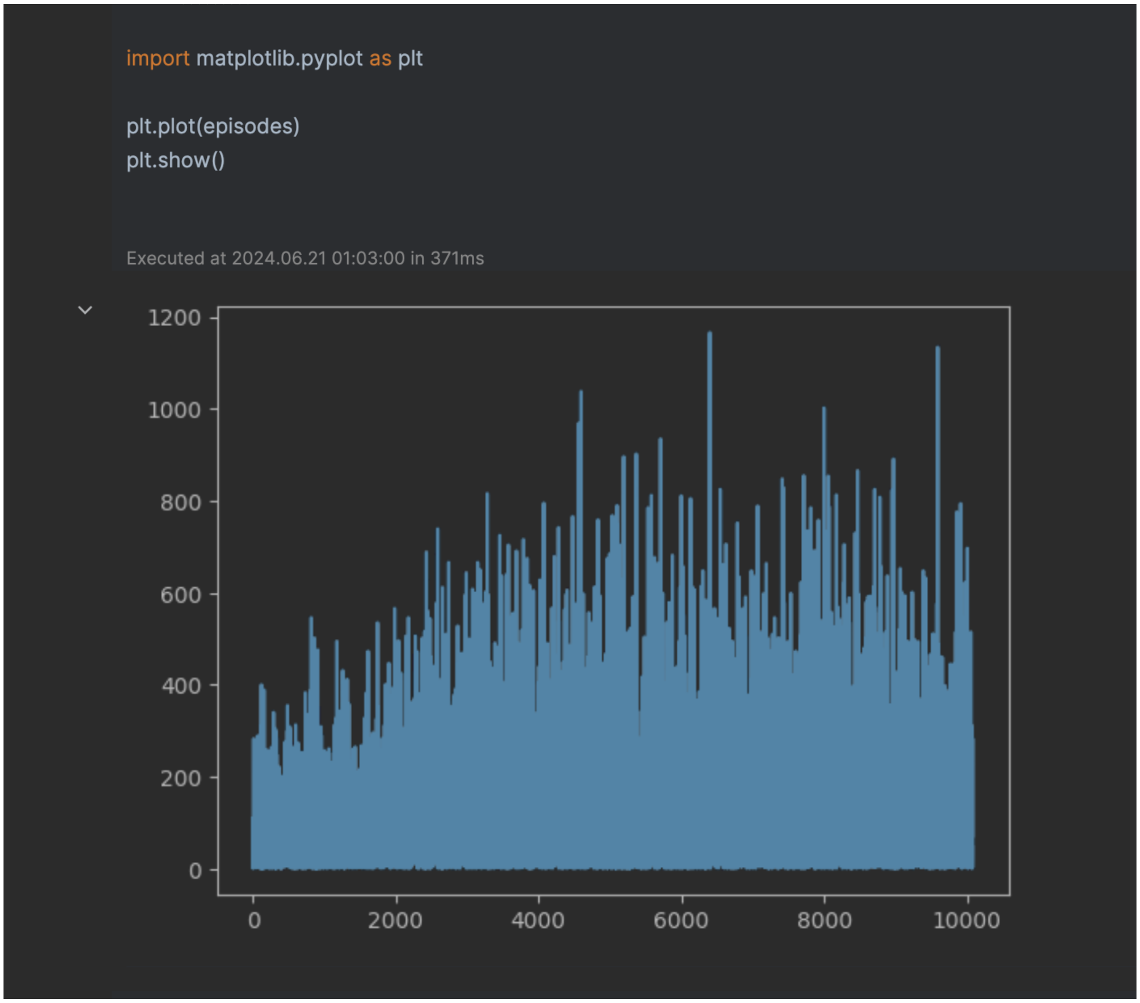
```
import matplotlib.pyplot as plt

plt.plot(wins)
plt.show()
```

Executed at 2024.06.21 01:02:52 in 685ms



```
import matplotlib.pyplot as plt

plt.plot(rewards)
plt.show()
```

Executed at 2024.06.21 01:03:14 in 480ms

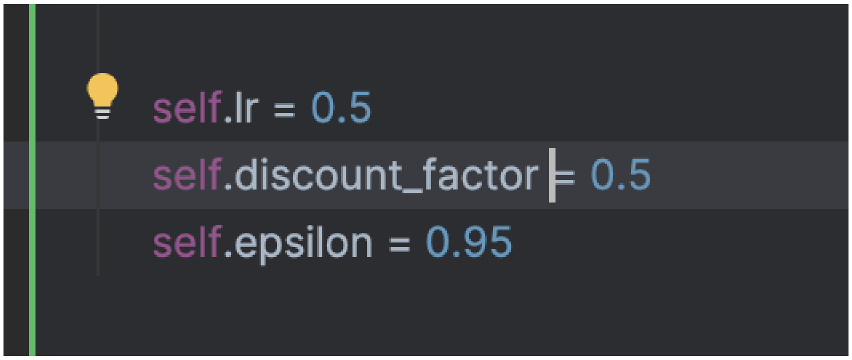


```
import matplotlib.pyplot as plt

plt.plot(episodes)
plt.show()
```

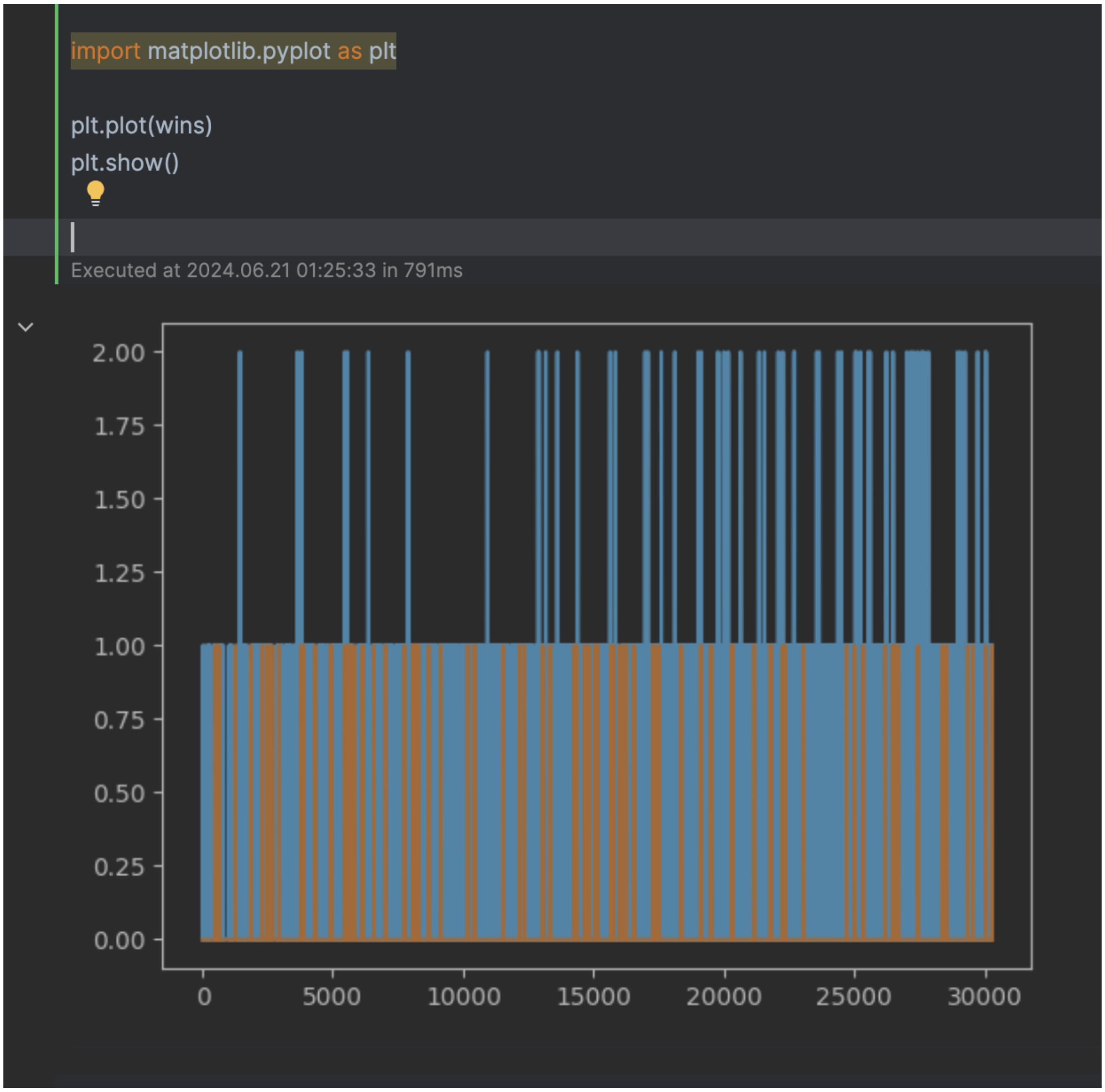
Executed at 2024.06.21 01:03:00 in 371ms

در این قسمت که مقدار نارنجی ها بیشتر از ابی ها است به خاطر ترتیب قرار گرفتن ان ها میباشد

در این قسمت مدل در حالت کلی مدل بهتری شد

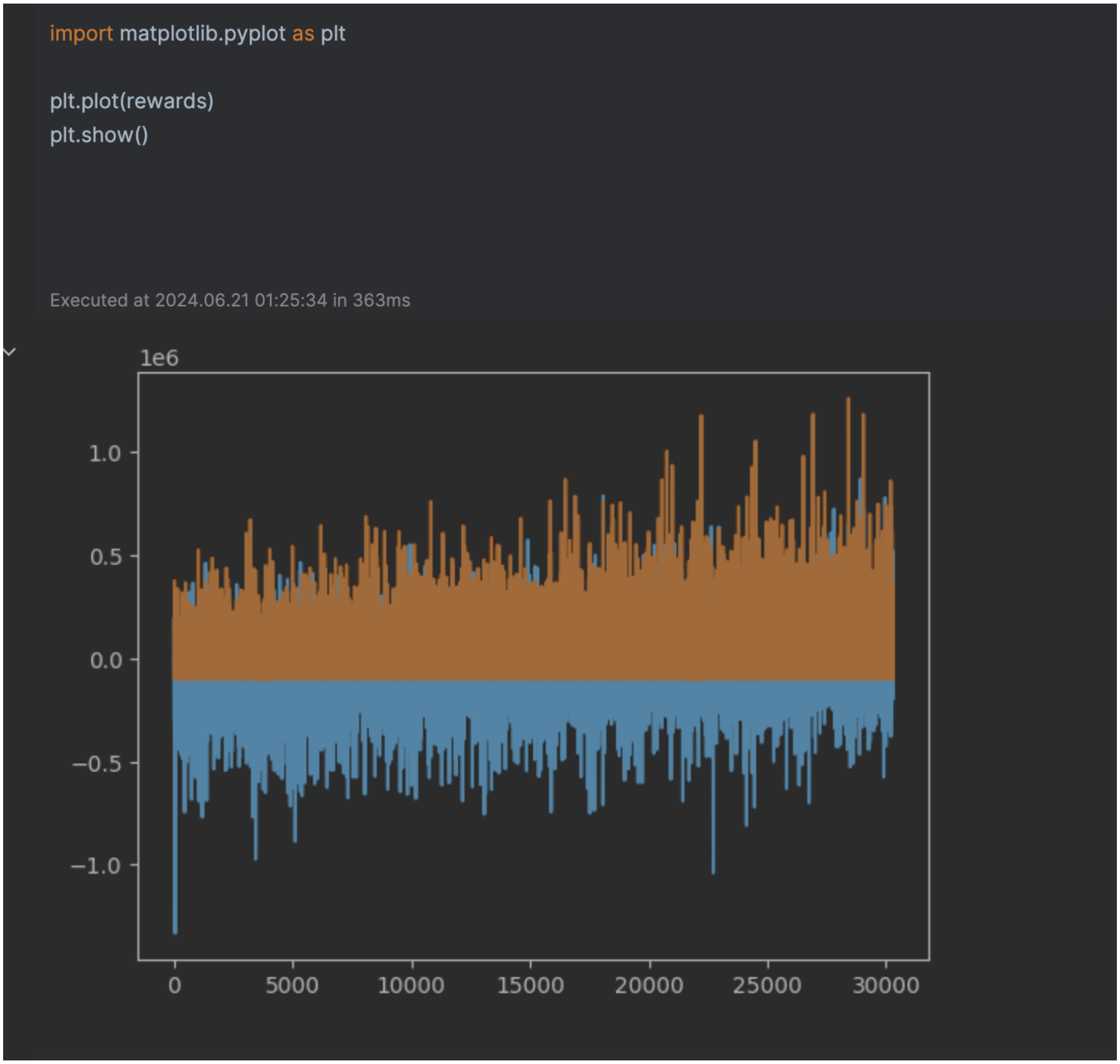


```
self.lr = 0.5
self.discount_factor = 0.5
self.epsilon = 0.95
```



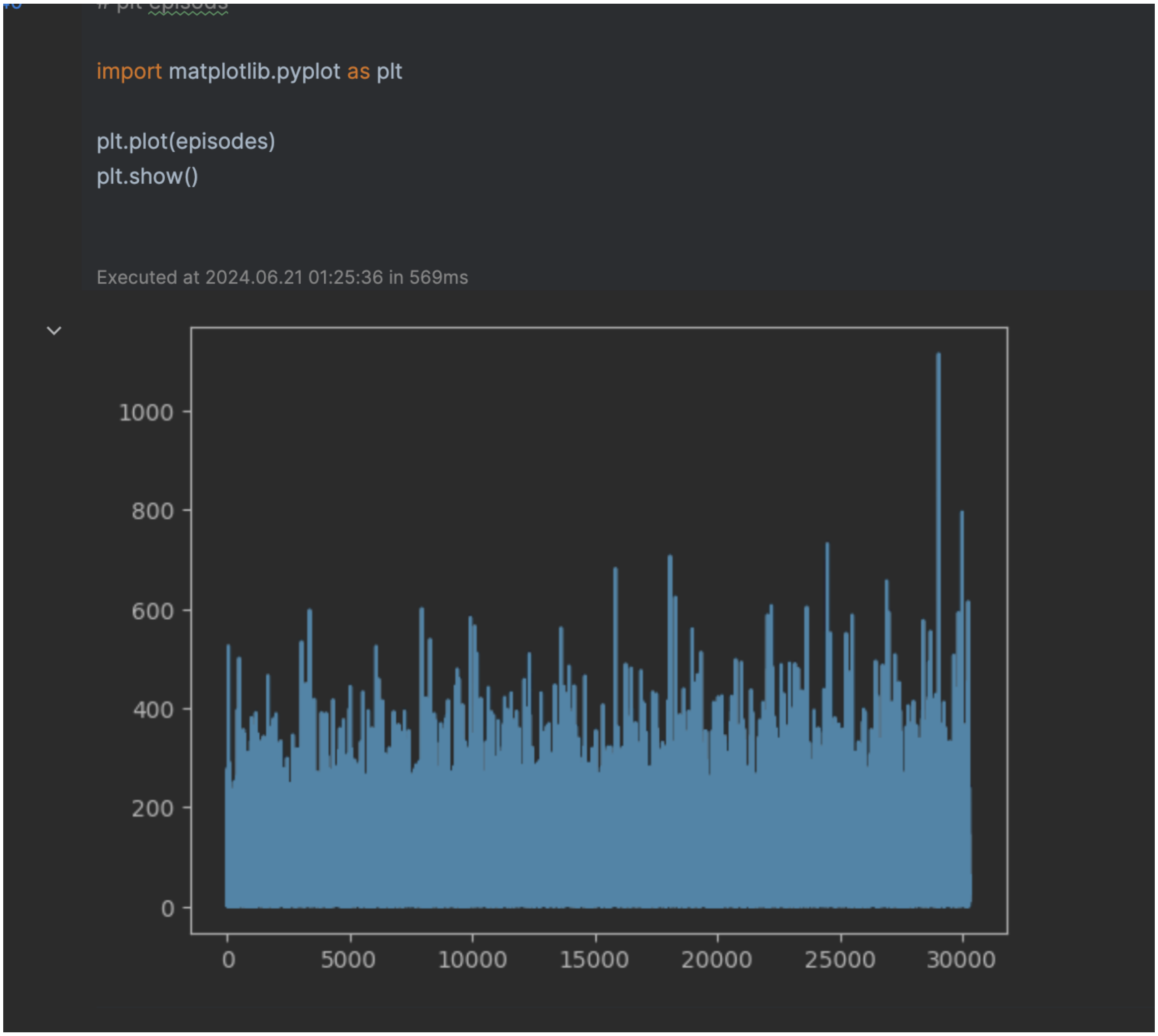
```
import matplotlib.pyplot as plt

plt.plot(wins)
plt.show()
```

Executed at 2024.06.21 01:25:33 in 791ms



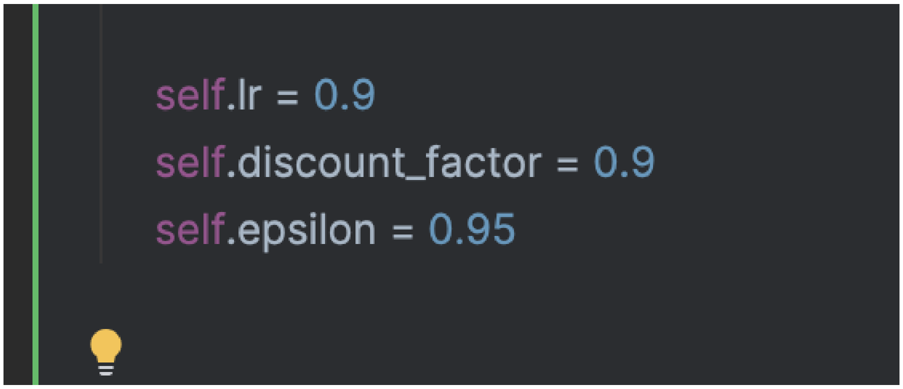
```
import matplotlib.pyplot as plt

plt.plot(rewards)
plt.show()
```

Executed at 2024.06.21 01:25:34 in 363ms



```
# plt episods

import matplotlib.pyplot as plt

plt.plot(episodes)
plt.show()
```

Executed at 2024.06.21 01:25:36 in 569ms



```
self.lr = 0.9
self.discount_factor = 0.9
self.epsilon = 0.95
```

version 4

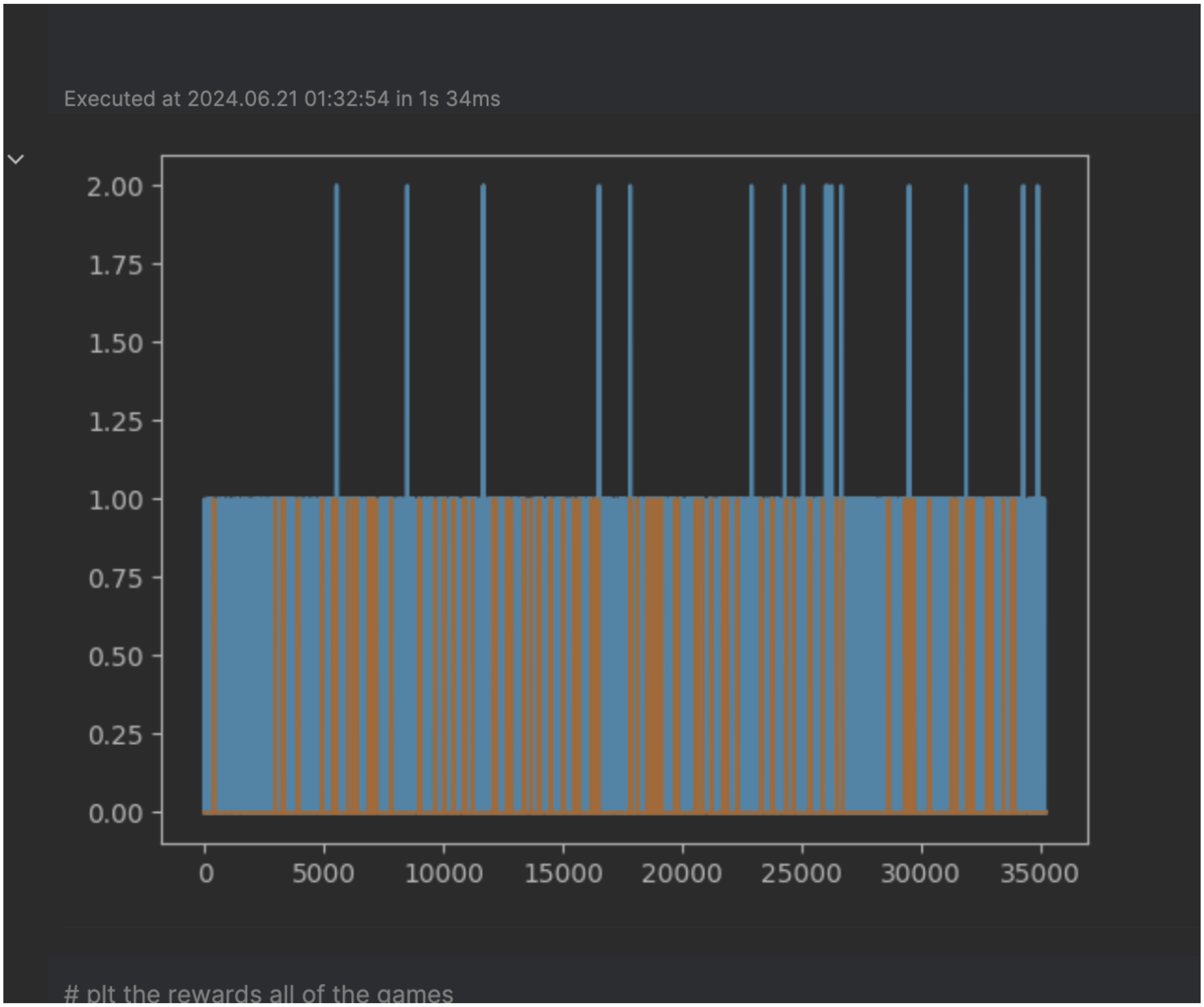
```
import random
import random
import numpy as np


class Snake:
    def __init__(self, color, pos, file_name=None):
        self.color = color
        self.head = Cube(pos, color=color)
        self.body = [self.head]
        self.dirnx = 0
        self.dirny = 1
        self.turns = {}
        self.num_of_wins = 0
        self.total_reward = 0

        self.q_table = np.load(file_name, allow_pickle=True).item() if file_name

        self.lr = 0.5
        self.discount_factor = 0.5
        self.epsilon = 0.95



    def get_state(self, snack, other_snake):
        head_x, head_y = self.head.pos
        snack_x, snack_y = snack.pos
        other_head_x, other_head_y = other_snake.head.pos

        diff_snack_x = abs(head_x - snack_x)
        diff_snack_y = abs(head_y - snack_y)
        diff_other_x = abs(head_x - other_head_x)
        diff_other_y = abs(head_y - other_head_y)


        state = (diff_snack_x, diff_snack_y, diff_other_x, diff_other_y)
        return state
```

```
def get_optimal_policy(self, state):
    if state not in self.q_table:
        self.q_table[state] = np.zeros(4)
    return np.argmax(self.q_table[state])


def make_action(self, state):
    if random.random() < self.epsilon:
        return random.randint(0, 3)
    else:
        return self.get_optimal_policy(state)

def update_q_table(self, state, action, next_state, reward):
    if next_state not in self.q_table:
        self.q_table[next_state] = np.zeros(4)
    if state not in self.q_table:
        self.q_table[state] = np.zeros(4)
    best_next_action = np.argmax(self.q_table[next_state])
    td_target = reward + self.discount_factor * self.q_table[next_state][bes
    td_error = td_target - self.q_table[state][action]
    self.q_table[state][action] += self.lr * td_error



def move(self, snack, other_snake):
    state = self.get_state(snack, other_snake)
    action = self.make_action(state)

    if action == 0:
        self.dirnx = -1
        self.dirny = 0
        self.turns[self.head.pos[:]] = [self.dirnx, self.dirny]
    elif action == 1:
        self.dirnx = 1
        self.dirny = 0
        self.turns[self.head.pos[:]] = [self.dirnx, self.dirny]
    elif action == 2:
        self.dirny = -1
        self.dirnx = 0
        self.turns[self.head.pos[:]] = [self.dirnx, self.dirny]
    elif action == 3:
        self.dirny = 1
        self.dirnx = 0
        self.turns[self.head.pos[:]] = [self.dirnx, self.dirny]
    for i, c in enumerate(self.body):
        p = c.pos[:]
        if p in self.turns:
            turn = self.turns[p]
            c.move(turn[0], turn[1])
            if i == len(self.body) - 1:
                self.turns.pop(p)
        else:
            c.move(c.dirnx, c.dirny)

    next_state = self.get_state(snack, other_snake)



    return state, next_state, action


def check_out_of_board(self):
    headPos = self.head.pos
    if headPos[0] >= ROWS - 1 or headPos[0] < 1 or headPos[1] >= ROWS - 1 or
        self.reset((random.randint(3, 18), random.randint(3, 18)))
        return True
    return False


def calc_reward(self, snack, other_snake):
    reward = 0
    win_self  = False
    other_win = False


    if self.check_out_of_board():
        reward -= 100000
        # win_self = False
        other_win = True

        self.reset((random.randint(3, 18), random.randint(3, 18)))

    if self.head.pos == snack.pos:
        self.addCube()
        snack = Cube(randomSnack(ROWS, self), color=(0, 255, 0))
        reward += 10000
        self.num_of_wins += 1
    if self.head.pos in list(map(lambda z: z.pos, self.body[1:])):
        reward -= 10000
        win_self = False
        other_win = True

        self.reset((random.randint(3, 18), random.randint(3, 18)))

    if self.head.pos in list(map(lambda z: z.pos, other_snake.body)):
        if self.head.pos != other_snake.head.pos:
            reward -= 10000
            win_self = False
            other_win = True

        else:
            if len(self.body) > len(other_snake.body):
                reward += 10000
                win_self = True
                other_win = False
            elif len(self.body) == len(other_snake.body):
                reward = 0
            else:
                reward -= 10000
                win_self = False
                other_win = True
        self.reset((random.randint(3, 18), random.randint(3, 18)))

    distance_to_snack = self.distance_to_snack(snack)
    if not distance_to_snack == 0:
        reward += 1 / distance_to_snack*10000

    if self.epsilon > 0.1:
        self.epsilon = self.epsilon * epsilon_reduction

    self.total_reward += reward


    return snack, reward, win_self , other_win





def avrage_distance_to_other_snake(self, other_snake):
    sum = 0
    for cube in other_snake.body:
        sum += self.distance_to_snack(cube)
    return sum / len(other_snake.body)

def distance_to_snack(self, snack):
    head_x, head_y = self.head.pos
    snack_x, snack_y = snack.pos

    return abs(head_x - snack_x) + abs(head_y - snack_y)


def reset(self, pos):
    self.head = Cube(pos, color=self.color)
    self.body = [self.head]
    self.turns = {}
    self.dirnx = 0
    self.dirny = 1
    self.num_of_wins = 0
    self.total_reward = 0


def addCube(self):
    tail = self.body[-1]
    dx, dy = tail.dirnx, tail.dirny


    if dx == 1 and dy == 0:
        self.body.append(Cube((tail.pos[0] - 1, tail.pos[1]), color=self.col
    elif dx == -1 and dy == 0:
        self.body.append(Cube((tail.pos[0] + 1, tail.pos[1]), color=self.col
    elif dx == 0 and dy == 1:
        self.body.append(Cube((tail.pos[0], tail.pos[1] - 1), color=self.col
    elif dx == 0 and dy == -1:
        self.body.append(Cube((tail.pos[0], tail.pos[1] + 1), color=self.col

    self.body[-1].dirnx = dx
    self.body[-1].dirny = dy

def draw(self, surface):
    for i, c in enumerate(self.body):
        if i == 0:
            c.draw(surface, True)
        else:
            c.draw(surface)

def save_q_table(self, file_name):
    np.save(file_name, self.q_table, allow_pickle=True)
```

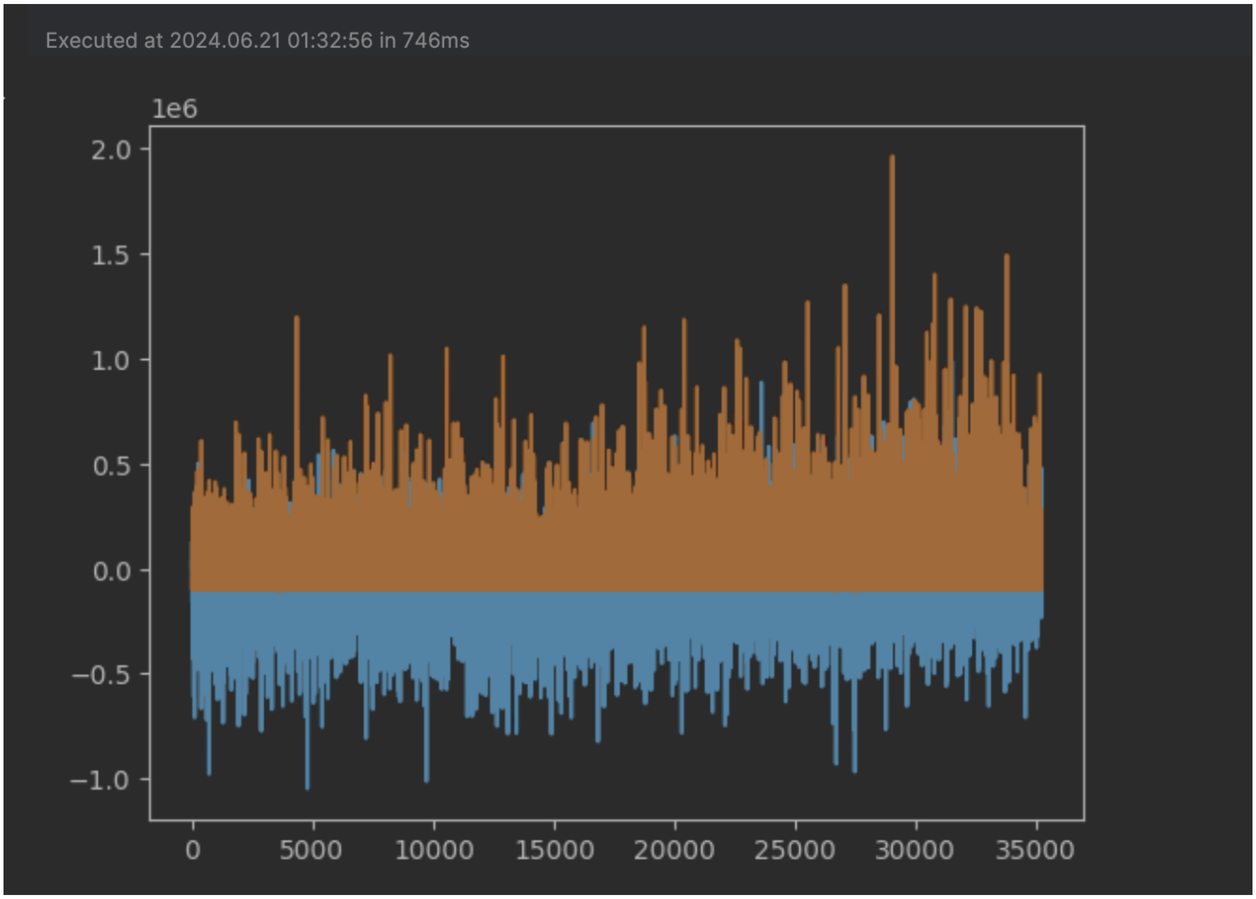
Version 5 :

```
import pygame
import numpy as np
from tkinter import messagebox

# pygame.init()
# win = pygame.display.set_mode((WIDTH, HEIGHT))

snake_1 = Snake((255, 0, 0), (15, 15), SNAKE_1_Q_TABLE)
snake_2 = Snake((255, 255, 0), (5, 5), SNAKE_2_Q_TABLE)
snake_1.addCube()
snake_2.addCube()

rewards = []
ss = 0
snack = Cube(randomSnack(ROWS, snake_1), color=(0, 255, 0))
wins = []
episodes = []
# clock = pygame.time.Clock()


rr1 = 0
rr2 = 0
while True:
    ss += 1
    reward_1 = 0
    reward_2 = 0
    # pygame.time.delay(1)
    # clock.tick(10)

    # for event in pygame.event.get():
    #     if event.type == pygame.QUIT:
    #         if messagebox.askokcancel("Quit", "Do you want to save the Q-table
    #             save(snake_1, snake_2)
    #         pygame.quit()
    #         exit()
    #     if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
    #         np.save(SNAKE_1_Q_TABLE, snake_1.q_table)
    #         np.save(SNAKE_2_Q_TABLE, snake_2.q_table)
    #         pygame.time.delay

    state_1, new_state_1, action_1 = snake_1.move(snack, snake_2)
    state_2, new_state_2, action_2 = snake_2.move(snack, snake_1)

    snack, reward_1, win_1 , win_2 = snake_1.calc_reward(snack, snake_2)
    snack, reward_2, win_2  , win_1= snake_2.calc_reward(snack, snake_1)

    rr1 += reward_1
    rr2 += reward_2


    if win_1 or win_2:
        wins.append([snake_1.num_of_wins, snake_2.num_of_wins])
        rewards.append([rr1, rr2])
        episodes.append(ss)
        print("Win state detected:", (win_1, win_2))
        print("snake_1 wins: ", snake_1.num_of_wins)
        print("snake_2 wins: ", snake_2.num_of_wins)
        print("epsilon: ", snake_1.epsilon)
        print("ss: ", ss)
        print(len(snake_1.q_table))
        print("rr1: ", rr1)
        print("rr2: ", rr2)
        print("total reward snake 1: ", snake_1.total_reward)
        print("total reward snake 2: ", snake_2.total_reward)
        ss = 0

        rr1 = 0
        rr2 = 0

    # Update Q-tables
    snake_1.update_q_table(state_1, action_1, new_state_1, reward_1)
    snake_2.update_q_table(state_2, action_2, new_state_2, reward_2)



    # rewards.append([reward_1, reward_2])



        # snack = Cube(randomSnack(ROWS, snake_1), color=(0, 255, 0))
```

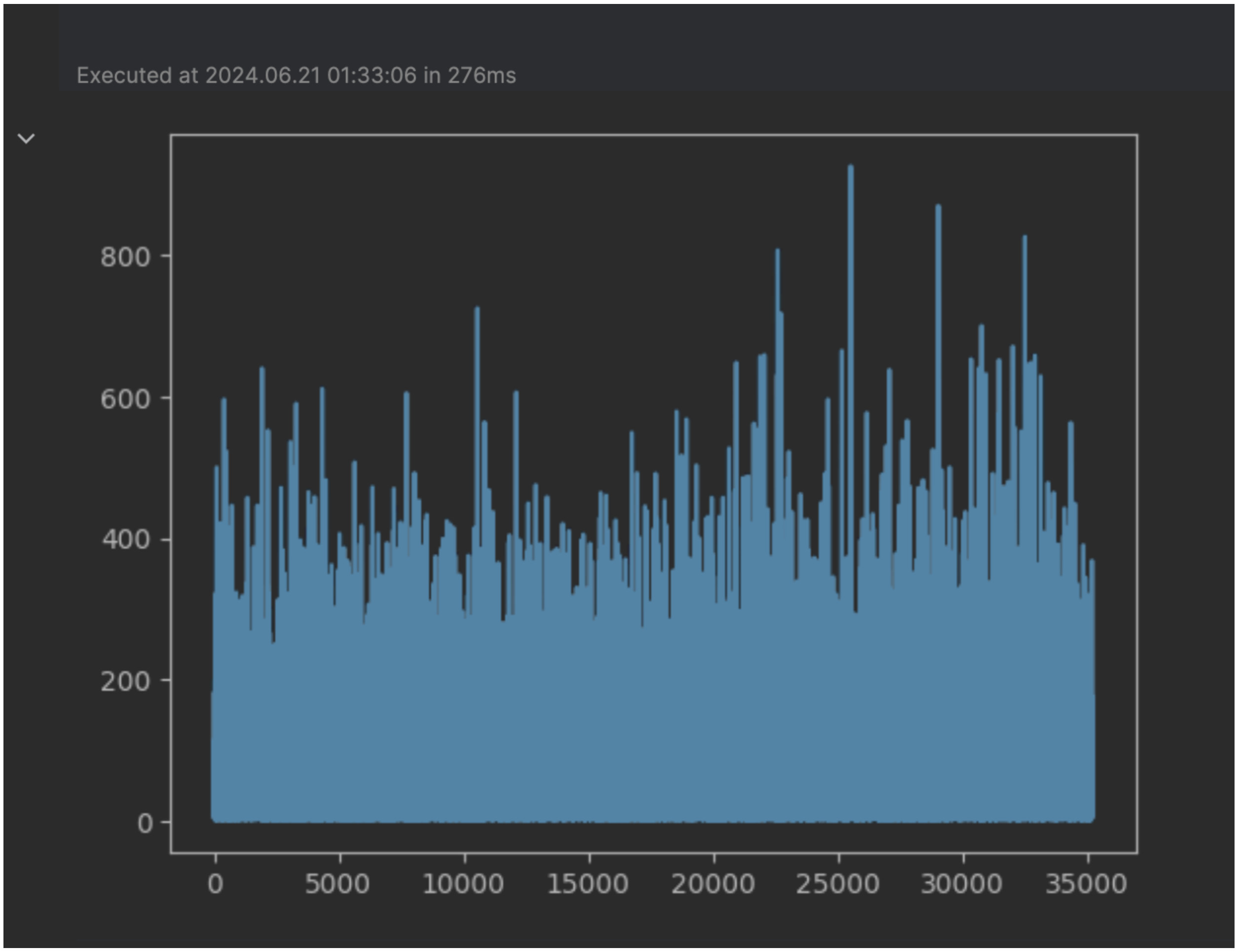
```
        # snake_1.reset((15, 15))
        # snake_2.reset((5, 5))

    # print()
    # print("reward_1: ", reward_1)
    # print("reward_2: ", reward_2)
    # print("ss", ss)
    # print("epsilon: ", snake_1.epsilon)

    # Redraw window (uncomment for visualizing the game)
    # redrawWindow(snake_1, snake_2, snack, win)
```


Executed at 2024.06.21 01:32:54 in 1s 34ms

```
# plt the rewards all of the games
```



Executed at 2024.06.21 01:32:56 in 746ms



Executed at 2024.06.21 01:33:06 in 276ms