Sieci samouczące się

Laboratorium 4, 5

Przemysław Rośleń 180150

Problem parkowania samochodu rozwiązałem stosując algorytm aproksymacji liniowej z kodowaniem, stosując przy tym 2 różne warianty kodowania:

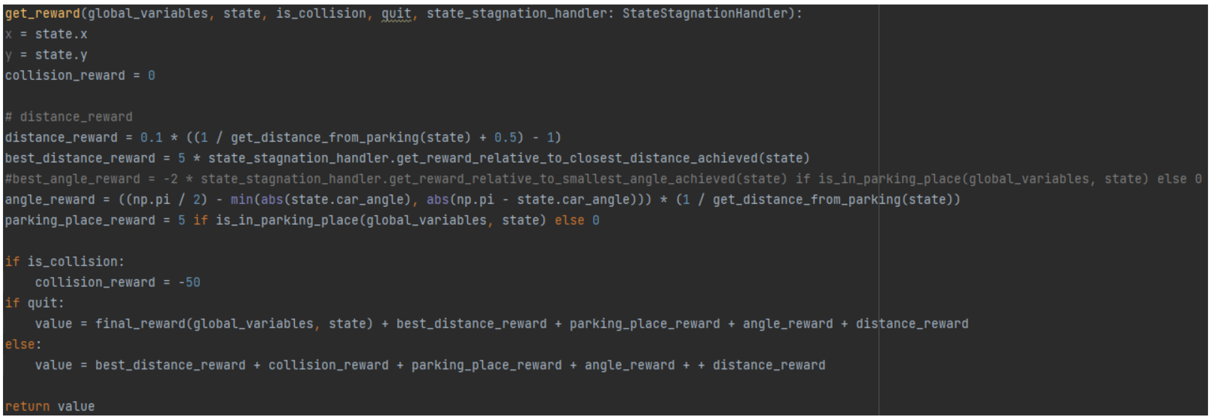
- kodowanie metodą pokryć
- kodowanie metodą prototypową

Poniżej przedstawiłem:

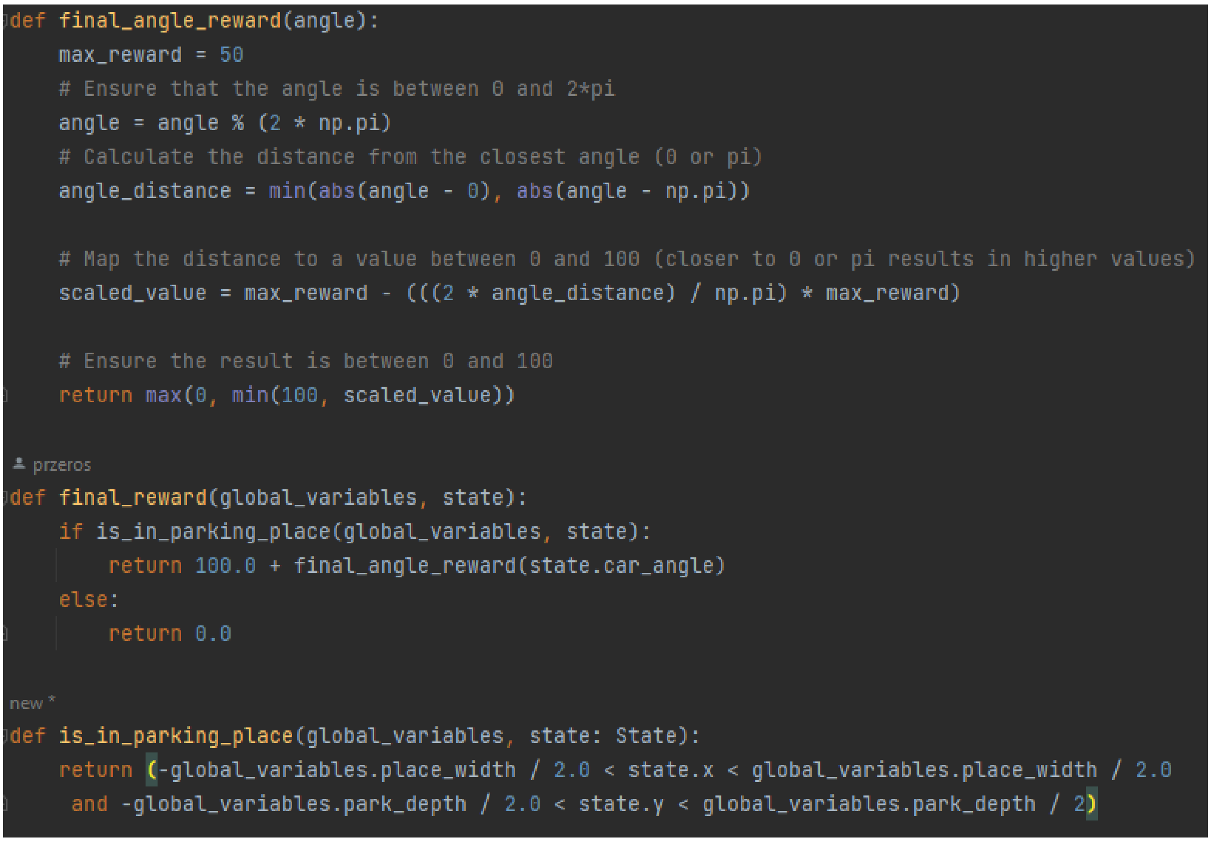
- przedstawienie funkcji nagrody
- istotne fragmenty kodu
- najskuteczniejsze wartości hiperparametrów
- wyniki w postaci wizualizacji zachowania samochodu

Kodowanie metodą pokryć:

Funkcja nagrody:

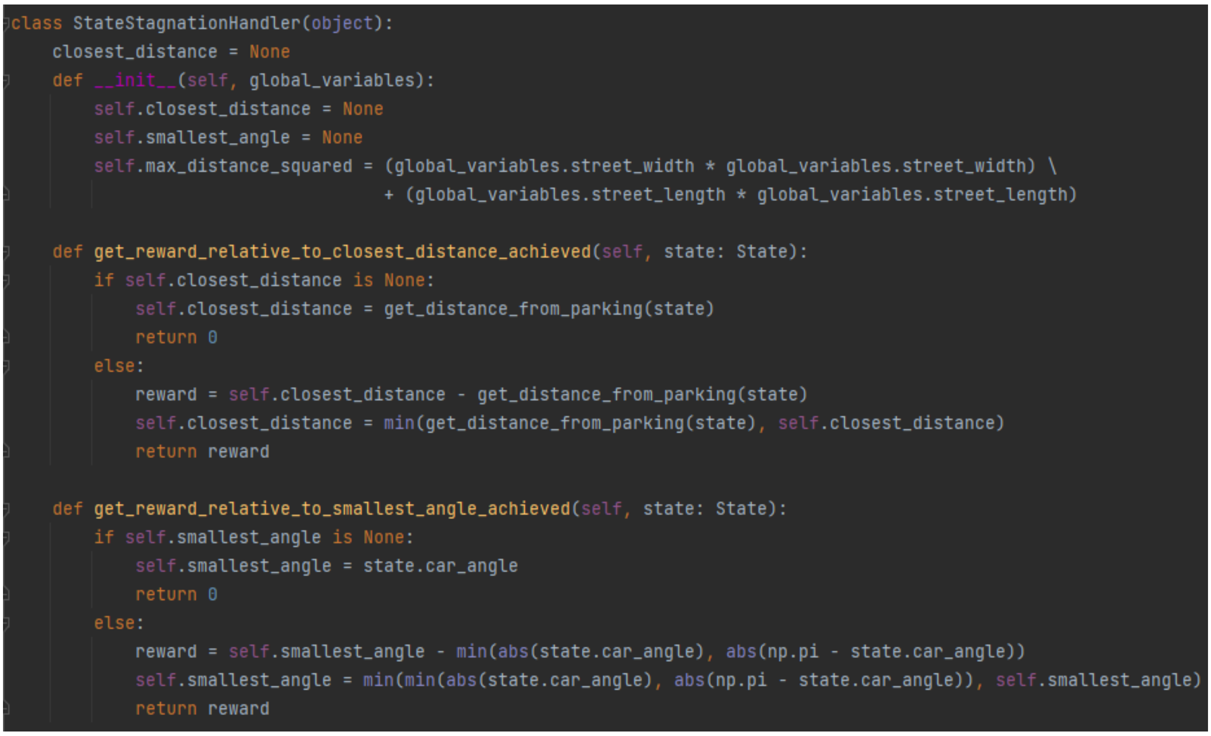


```
get_reward(global_variables, state, is_collision, quit, state_stagnation_handler: StateStagnationHandler):
x = state.x
y = state.y
collision_reward = 0

# distance_reward
distance_reward = 0.1 * ((1 / get_distance_from_parking(state) + 0.5) - 1)
best_distance_reward = 5 * state_stagnation_handler.get_reward_relative_to_closest_distance_achieved(state)
#best_angle_reward = -2 * state_stagnation_handler.get_reward_relative_to_smallest_angle_achieved(state) if is_in_parking_place(global_variables, state) else 0
angle_reward = ((np.pi / 2) - min(abs(state.car_angle), abs(np.pi - state.car_angle))) * (1 / get_distance_from_parking(state))
parking_place_reward = 5 if is_in_parking_place(global_variables, state) else 0

if is_collision:
    collision_reward = -50
if quit:
    value = final_reward(global_variables, state) + best_distance_reward + parking_place_reward + angle_reward + distance_reward
else:
    value = best_distance_reward + collision_reward + parking_place_reward + angle_reward + + distance_reward

return value
```



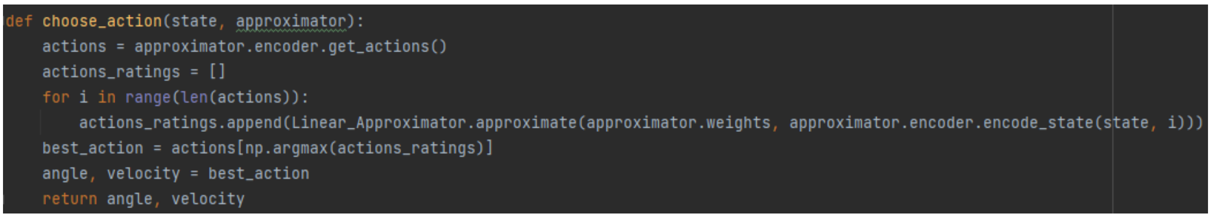
```
def final_angle_reward(angle):
    max_reward = 50
    # Ensure that the angle is between 0 and 2*pi
    angle = angle % (2 * np.pi)
    # Calculate the distance from the closest angle (0 or pi)
    angle_distance = min(abs(angle - 0), abs(angle - np.pi))

    # Map the distance to a value between 0 and 100 (closer to 0 or pi results in higher values)
    scaled_value = max_reward - (((2 * angle_distance) / np.pi) * max_reward)

    # Ensure the result is between 0 and 100
    return max(0, min(100, scaled_value))

przeros
def final_reward(global_variables, state):
    if is_in_parking_place(global_variables, state):
        return 100.0 + final_angle_reward(state.car_angle)
    else:
        return 0.0

new *
def is_in_parking_place(global_variables, state: State):
    return (-global_variables.place_width / 2.0 < state.x < global_variables.place_width / 2.0
     and -global_variables.park_depth / 2.0 < state.y < global_variables.park_depth / 2)
```

Klasa zapobiegająca stagnacji samochodu (poruszania w miejscu):



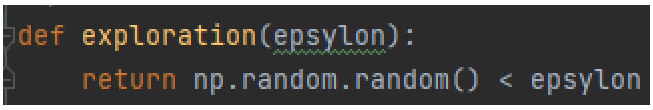
```
class StateStagnationHandler(object):
    closest_distance = None
    def __init__(self, global_variables):
        self.closest_distance = None
        self.smallest_angle = None
        self.max_distance_squared = (global_variables.street_width * global_variables.street_width) \
                                    + (global_variables.street_length * global_variables.street_length)

    def get_reward_relative_to_closest_distance_achieved(self, state: State):
        if self.closest_distance is None:
            self.closest_distance = get_distance_from_parking(state)
            return 0
        else:
            reward = self.closest_distance - get_distance_from_parking(state)
            self.closest_distance = min(get_distance_from_parking(state), self.closest_distance)
            return reward

    def get_reward_relative_to_smallest_angle_achieved(self, state: State):
        if self.smallest_angle is None:
            self.smallest_angle = state.car_angle
            return 0
        else:
            reward = self.smallest_angle - min(abs(state.car_angle), abs(np.pi - state.car_angle))
            self.smallest_angle = min(min(abs(state.car_angle), abs(np.pi - state.car_angle)), self.smallest_angle)
            return reward
```

Wybór najlepszej akcji:

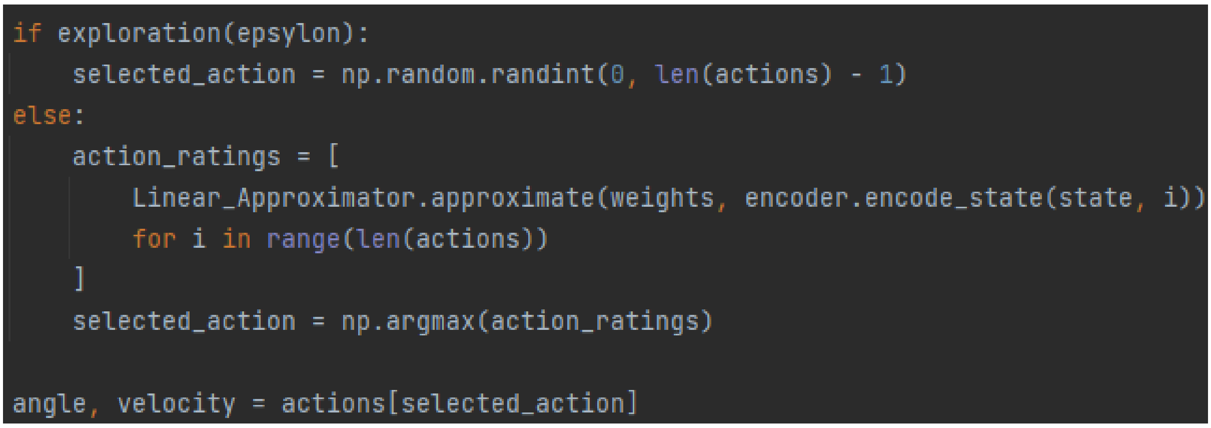


```
def choose_action(state, approximator):
    actions = approximator.encoder.get_actions()
    actions_ratings = []
    for i in range(len(actions)):
        actions_ratings.append(Linear_Approximator.approximate(approximator.weights, approximator.encoder.encode_state(state, i)))
    best_action = actions[np.argmax(actions_ratings)]
    angle, velocity = best_action
    return angle, velocity
```

Eksploracja vs Eksploatacja:

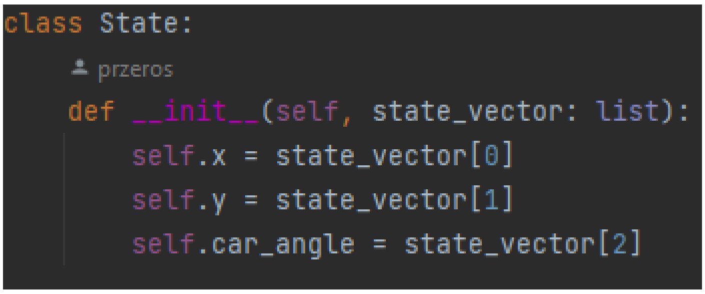


```
def exploration(epsylon):
    return np.random.random() < epsylon
```



```
if exploration(epsylon):
    selected_action = np.random.randint(0, len(actions) - 1)
else:
    action_ratings = [
        Linear_Approximator.approximate(weights, encoder.encode_state(state, i))
        for i in range(len(actions))
    ]
    selected_action = np.argmax(action_ratings)

angle, velocity = actions[selected_action]
```

Klasa stanu:



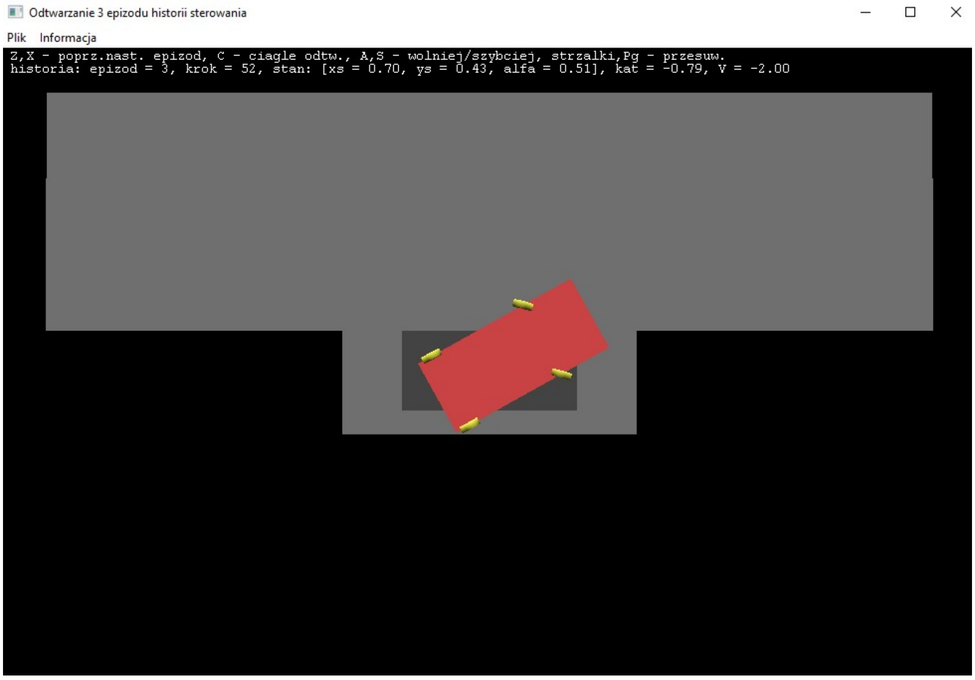
```
class State:
    def __init__(self, state_vector: list):
        self.x = state_vector[0]
        self.y = state_vector[1]
        self.car_angle = state_vector[2]
```

Hiperparametry:

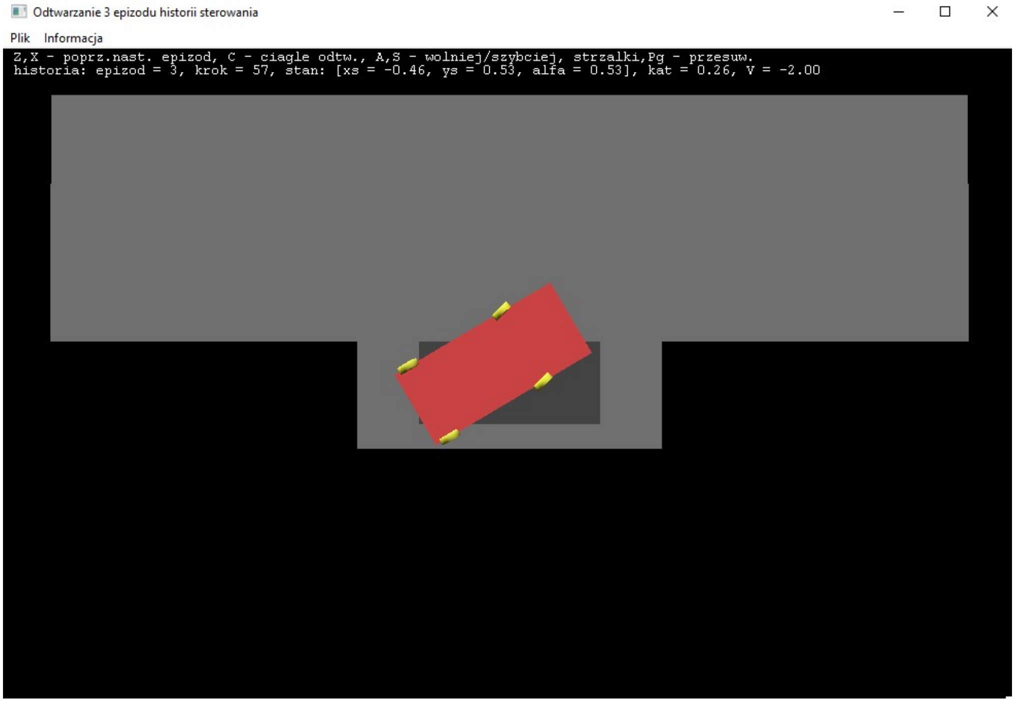
- liczba wartości kąta samochodu: 10
- liczba wartości kąta kół: 10
- wartości prędkości: [-2, 2]
- ilość obszarów w szerokości: 10
- ilość obszarów w wysokości: 6
- ilość projekcji mapy: 3
- przesunięcia projekcji: (-0.5, -0.5, -0.5), (0, 0, 0), (0.5, 0.5, 0.5)
- liczba epok: 2000
- alpha: 0.1
- początkowa wartość epsylon: 1

- zanik epsylon na epokę: 0.0005
- gamma: 0.95

Efekty uczenia:



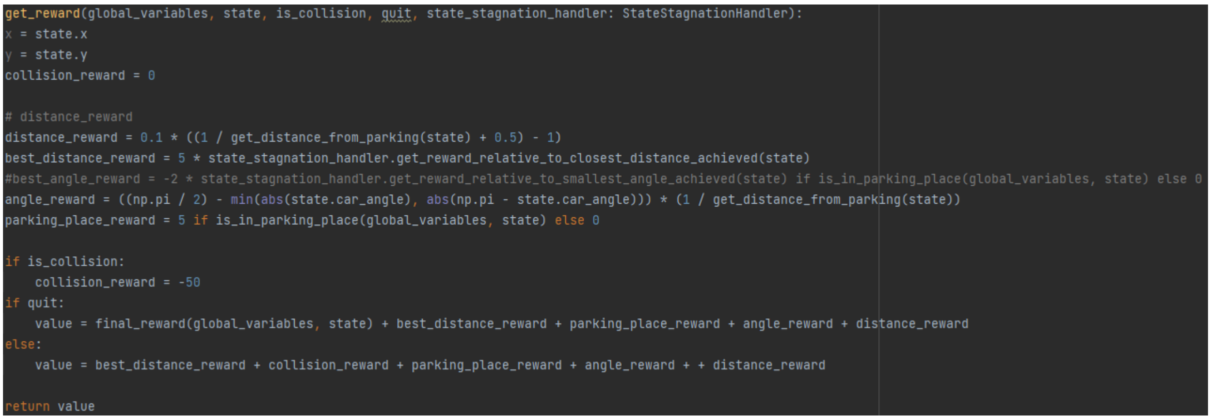
```
Z,X - poprz.nast. epizod, C - ciagle odtw., A,S - wolniej/szybciej, strzalki,Pg - przesuw.
historia: epizod = 3, krok = 52, stan: [xs = 0.70, ys = 0.43, alfa = 0.51], kat = -0.79, V = -2.00
```



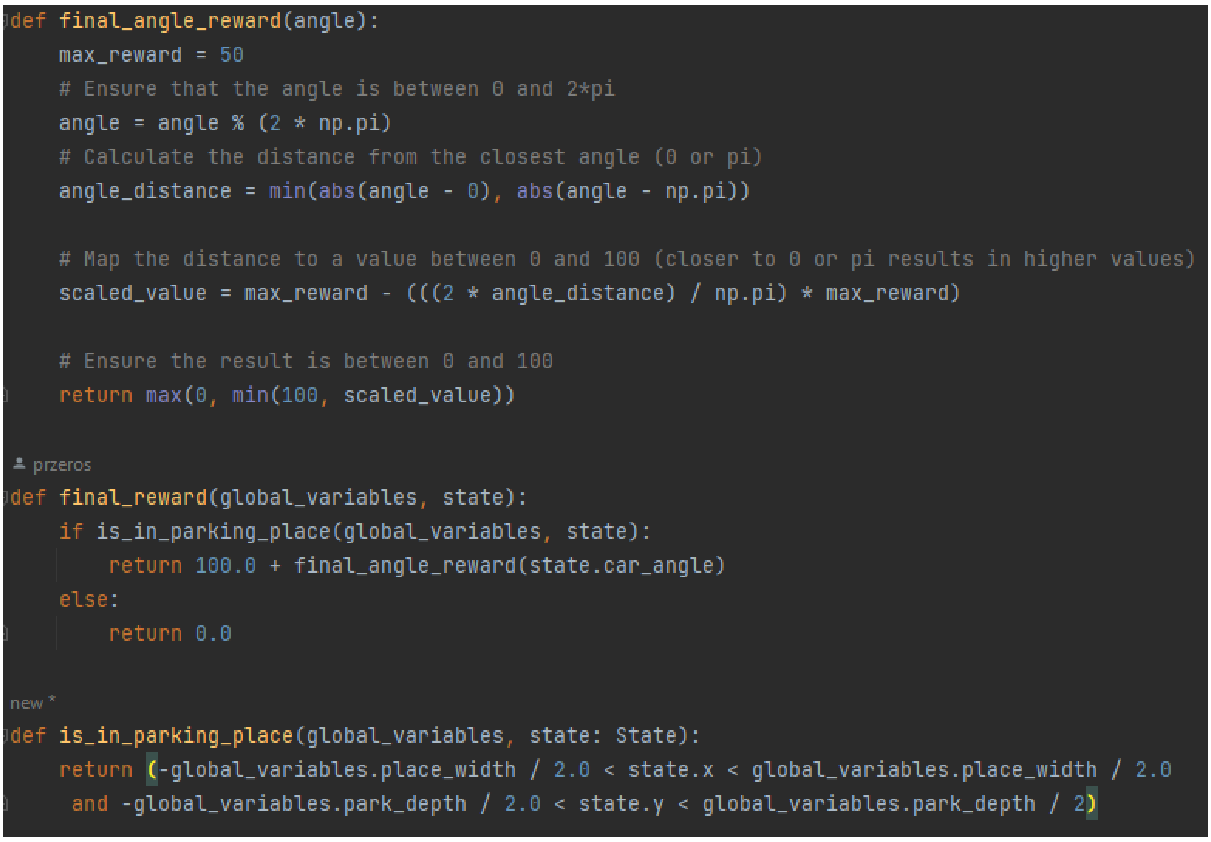
```
Z,X - poprz.nast. epizod, C - ciagle odtw., A,S - wolniej/szybciej, strzalki,Pg - przesuw.
historia: epizod = 3, krok = 57, stan: [xs = -0.46, ys = 0.53, alfa = 0.53], kat = 0.26, V = -2.00
```

Kodowanie metodą prototypową:

Funkcja nagrody:

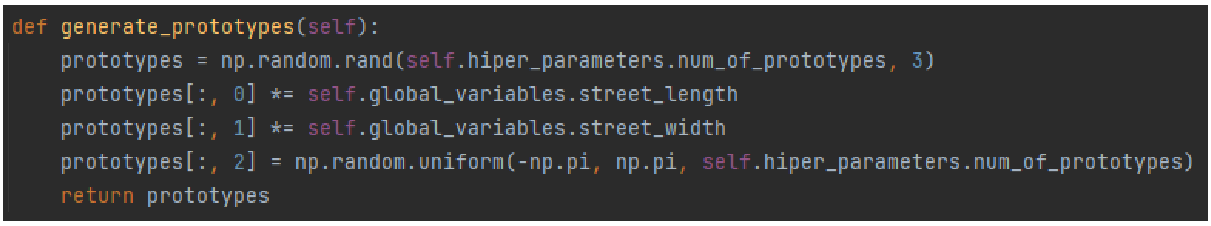


```
get_reward(global_variables, state, is_collision, quit, state_stagnation_handler: StateStagnationHandler):
x = state.x
y = state.y
collision_reward = 0

# distance_reward
distance_reward = 0.1 * ((1 / get_distance_from_parking(state) + 0.5) - 1)
best_distance_reward = 5 * state_stagnation_handler.get_reward_relative_to_closest_distance_achieved(state)
#best_angle_reward = -2 * state_stagnation_handler.get_reward_relative_to_smallest_angle_achieved(state) if is_in_parking_place(global_variables, state) else 0
angle_reward = ((np.pi / 2) - min(abs(state.car_angle), abs(np.pi - state.car_angle))) * (1 / get_distance_from_parking(state))
parking_place_reward = 5 if is_in_parking_place(global_variables, state) else 0

if is_collision:
    collision_reward = -50
if quit:
    value = final_reward(global_variables, state) + best_distance_reward + parking_place_reward + angle_reward + distance_reward
else:
    value = best_distance_reward + collision_reward + parking_place_reward + angle_reward + + distance_reward

return value
```



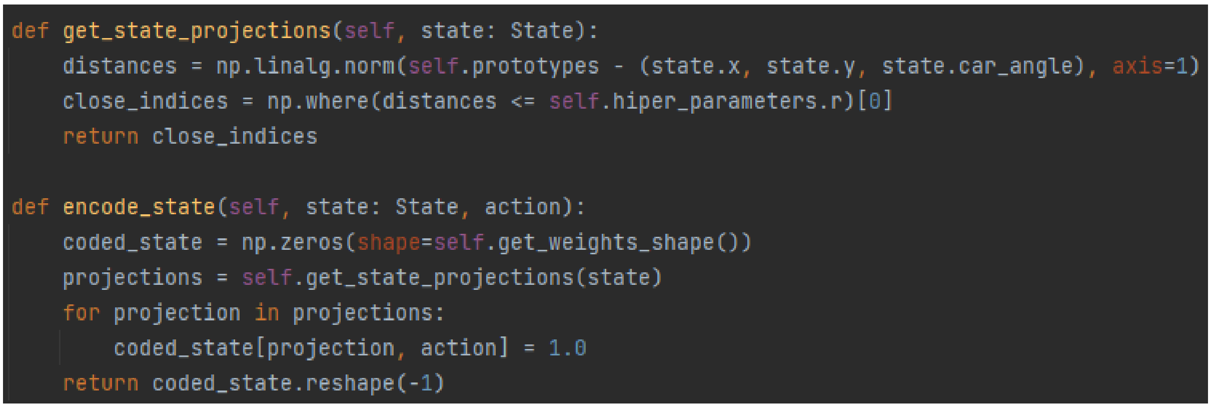
```
def final_angle_reward(angle):
    max_reward = 50
    # Ensure that the angle is between 0 and 2*pi
    angle = angle % (2 * np.pi)
    # Calculate the distance from the closest angle (0 or pi)
    angle_distance = min(abs(angle - 0), abs(angle - np.pi))

    # Map the distance to a value between 0 and 100 (closer to 0 or pi results in higher values)
    scaled_value = max_reward - (((2 * angle_distance) / np.pi) * max_reward)

    # Ensure the result is between 0 and 100
    return max(0, min(100, scaled_value))

def final_reward(global_variables, state):
    if is_in_parking_place(global_variables, state):
        return 100.0 + final_angle_reward(state.car_angle)
    else:
        return 0.0

def is_in_parking_place(global_variables, state: State):
    return (-global_variables.place_width / 2.0 < state.x < global_variables.place_width / 2.0
     and -global_variables.park_depth / 2.0 < state.y < global_variables.park_depth / 2)
```

Generowanie prototypów:



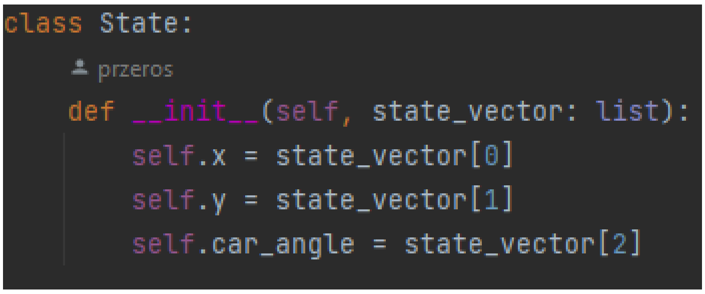
```
def generate_prototypes(self):
    prototypes = np.random.rand(self.hiper_parameters.num_of_prototypes, 3)
    prototypes[:, 0] *= self.global_variables.street_length
    prototypes[:, 1] *= self.global_variables.street_width
    prototypes[:, 2] = np.random.uniform(-np.pi, np.pi, self.hiper_parameters.num_of_prototypes)
    return prototypes
```

Kodowanie prototypów:



```
def get_state_projections(self, state: State):
    distances = np.linalg.norm(self.prototypes - (state.x, state.y, state.car_angle), axis=1)
    close_indices = np.where(distances <= self.hiper_parameters.r)[0]
    return close_indices

def encode_state(self, state: State, action):
    coded_state = np.zeros(shape=self.get_weights_shape())
    projections = self.get_state_projections(state)
    for projection in projections:
        coded_state[projection, action] = 1.0
    return coded_state.reshape(-1)
```

Klasa stanu:

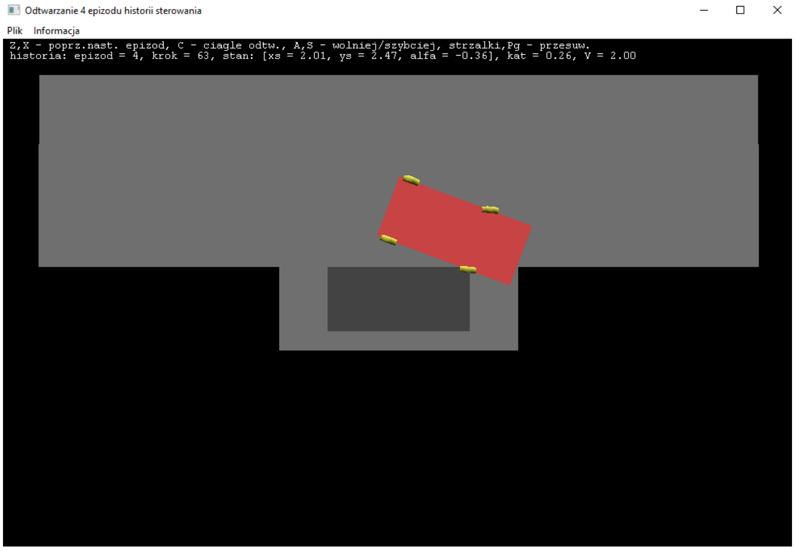


```
class State:
    def __init__(self, state_vector: list):
        self.x = state_vector[0]
        self.y = state_vector[1]
        self.car_angle = state_vector[2]
```

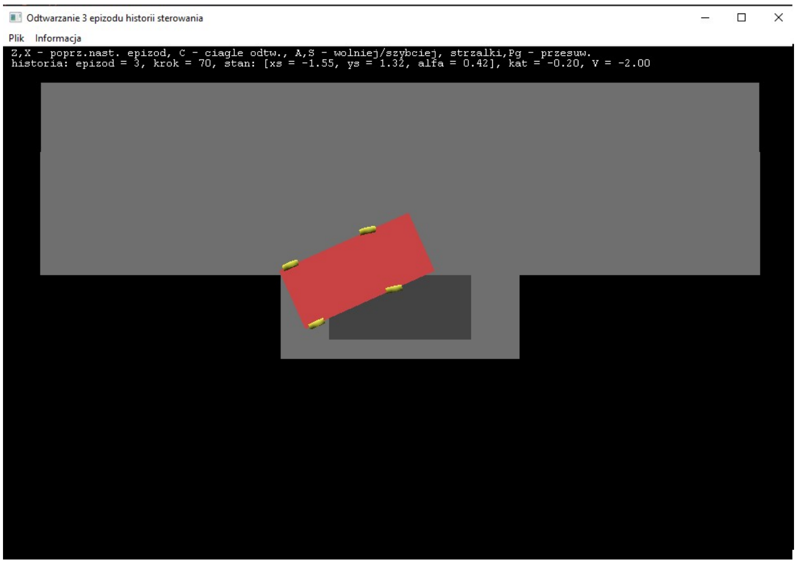
Hiperparametry:

- liczba wartości kąta samochodu: 10
- liczba wartości kąta kół: 10
- wartości prędkości: [-2, 2]
- liczba prototypów: 2000
- promień: 1
- liczba epok: 2000
- alpha: 0.1
- początkowa wartość epsylon: 1
- zanik epsylon na epokę: 0.0005
- gamma: 0.95

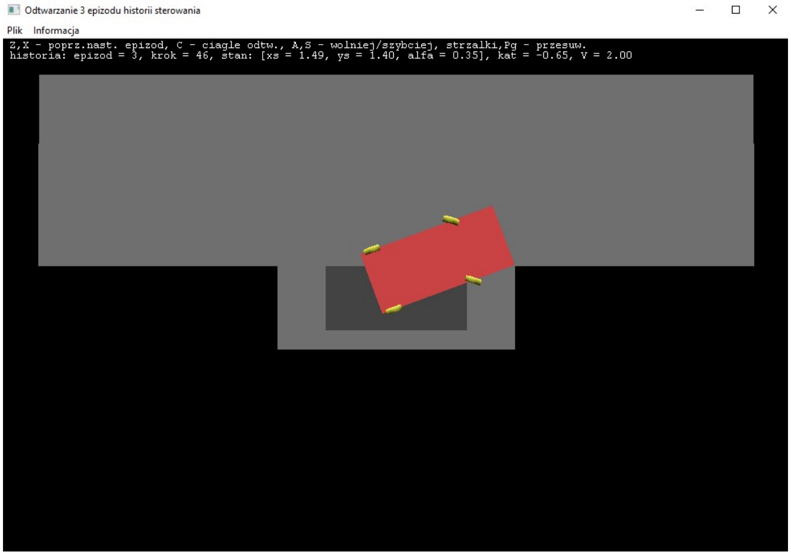
Efekty uczenia:



```
Z,X - poprz.nast. epizod, C - ciagle odtw., A,S - wolniej/szybciej, strzalki,Pg - przesuw.
historia: epizod = 4, krok = 63, stan: [xs = 2.01, ys = 2.47, alfa = -0.36], kat = 0.26, V = 2.00
```



```
Z,X - poprz.nast. epizod, C - ciagle odtw., A,S - wolniej/szybciej, strzalki,Pg - przesuw.
historia: epizod = 3, krok = 70, stan: [xs = -1.55, ys = 1.32, alfa = 0.42], kat = -0.20, V = -2.00
```



```
Z,X - poprz.nast. epizod, C - ciagle odtw., A,S - wolniej/szybciej, strzalki,Pg - przesuw.
historia: epizod = 3, krok = 46, stan: [xs = 1.49, ys = 1.40, alfa = 0.35], kat = -0.65, V = 2.00
```