Sieci samouczące się

Laboratorium 7, 8

Przemysław Rośleń 180150

Problem parkowania samochodu rozwiązałem stosując aproksymator neuronowy z kodowaniem metodą prototypową i aktualizacją wag zgodnie z algorytmem Q-learning.

Poniżej przedstawiłem:

- opis architektury użytej sztucznej sieci neuronowej
- opis metody uczenia z użyciem algorytmu Q-learning
- przedstawienie metod zapewniania stabilności procesu uczenia:
    - minibatch
    - curriculum learning
- istotne fragmenty kodu dla każdej z metod
- wyniki w postaci wizualizacji zachowania samochodu

Architektura sieci:

```
# Initialize the neural network approximator
input_shape = (3,)
output_shape = len(actions)
deep_nn_approximator = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=input_shape),
    # czy tu powinien byc relu?
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(output_shape, activation='softmax')
])
deep_nn_approximator.compile(optimizer='adam', loss='mse')
deep_nn_approximator.summary()
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                256

 dense_1 (Dense)             (None, 64)                4160

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 18)                594

=================================================================
Total params: 7,090
Trainable params: 7,090
Non-trainable params: 0
_________________________________________________________________
```

Sieć składa się z następujących warstw:

- warstwa wejściowa przyjmująca 3-wymiarowy wektor (x, y, alfa)
- 3 warstwy gęste o liczbach neronów:
    - 64
    - 64
    - 32
- warstwa wyjściowa przyjmująca wektor o wymiarze zależnym od ilości możliwych akcji

W celu minimalizacji funkcji kosztu w postaci MSE zastosowano optymalizator Adam.

Łączna ilość uczących się parametrów w sieci wynosi 7090.

Metoda uczenia z użyciem Q-learning:

Funkcja kosztu została przeze mnie zaprojektowana w oparciu o wzór:

$$R + \gamma \max_a \widehat{Q}(S', a, \tilde{\mathbf{w}}) - \widehat{Q}(S, A, \mathbf{w})$$

Aktualizacja wag w sieci odbywa się przy pomocy optymalizatora Adam z parametrem learning rate = 0.001. Proces aktualizacji wag odzwierciedla wzór:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \max_a \widehat{Q}(S', a, \tilde{\mathbf{w}}) - \widehat{Q}(S, A, \mathbf{w})]\nabla_{\mathbf{w}} \widehat{Q}(S, A, \mathbf{w})$$

Funkcja realizująca proces uczenia sieci:

```python
def update_model_Q_learning(state, new_state, selected_action, reward, deep_nn_approximator: Sequential, encoder, gamma):
    # Encode the current and next state
    current_state_input = np.array([state.x, state.y, state.car_angle])
    target_q_values = reward + gamma * get_best_rating_of_state_actions(approximator=deep_nn_approximator, new_state=new_state)
    with tf.GradientTape() as tape:
        current_q_values = deep_nn_approximator(np.array([current_state_input]))
        loss = MSE([target_q_values], [current_q_values[0, selected_action]])
    grads = tape.gradient(loss, deep_nn_approximator.trainable_variables)
    deep_nn_approximator.optimizer.apply_gradients(zip(grads, deep_nn_approximator.trainable_variables))

    return deep_nn_approximator
```

Metoda poprawy stabilności minibatch:

```python
class ReplayBuffer:
    przeros
    def __init__(self, max_size):
        self.buffer = []
        self.max_size = max_size
        self.next_idx = 0

    przeros
    def add(self, experience):
        if len(self.buffer) < self.max_size:
            self.buffer.append(experience)
        else:
            self.buffer[self.next_idx] = experience
            self.next_idx = (self.next_idx + 1) % self.max_size

    przeros
    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)
```

```python
# wyznaczenie nowego stanu:
new_state, useless, is_collision = pm.model_of_car(state, angle, velocity, global_variables)
new_state = State(new_state)

if is_collision|(step >= global_variables.max_number_of_steps):
    quit = True

reward = get_reward(global_variables, new_state, is_collision, quit, state_stagnation_handler)

buffer.add((state, selected_action, reward, new_state, quit))

# Aktualizujemy wartosci Q dla aktualnego stanu i wybranej akcji:
# .....................................................
# .....................................................
# w = w + ...
if len(buffer.buffer) > batch_size:
    deep_nn_approximator = update_model_Q_learning_minibatch(buffer, batch_size, deep_nn_approximator, encoder, gamma)

state = new_state
```

```python
def update_model_Q_learning_minibatch(buffer, batch_size, deep_nn_approximator, encoder, gamma):
    minibatch = buffer.sample(batch_size)
    for experience in minibatch:
        state, action, reward, next_state, done = experience

        # Przetwarzanie stanów i akcji
        current_state_input = np.array([state.x, state.y, state.car_angle])
        next_state_input = np.array([next_state.x, next_state.y, next_state.car_angle])

        # Obliczanie docelowych wartości Q
        target_q = reward
        if not done:
            target_q += gamma * get_best_rating_of_state_actions(deep_nn_approximator, next_state)

        # Aktualizacja sieci neuronowej
        with tf.GradientTape() as tape:
            current_q_values = deep_nn_approximator(np.array([current_state_input]))
            loss = MSE([target_q], [current_q_values[0, action]])
        grads = tape.gradient(loss, deep_nn_approximator.trainable_variables)
        deep_nn_approximator.optimizer.apply_gradients(zip(grads, deep_nn_approximator.trainable_variables))

    return deep_nn_approximator
```

Metoda poprawy stabilności curriculum learning:

Zastosowałem mechanizm dostosowywania nagrody na podstawie aktualnej epoki uczenia. Nagroda za ustawienie się pod dobrym kątem jest stopniowo zwiększana wraz z trwaniem procesu uczenia.

```python
def update_model_Q_learning_curriculum(state, new_state, selected_action, reward, deep_nn_approximator: Sequential, encoder, gamma, epoch, num_of_epochs):
    # Dostosowanie nagrody na podstawie epoki
    curriculum_factor = epoch / num_of_epochs  # Procentowy postęp treningu
    adjusted_reward = reward * (1 + curriculum_factor)  # Zwiększenie nagrody w miarę postępu treningu

    # Przetwarzanie stanów i akcji
    current_state_input = np.array([state.x, state.y, state.car_angle])
    target_q_values = adjusted_reward + gamma * get_best_rating_of_state_actions(approximator=deep_nn_approximator, new_state=new_state)

    with tf.GradientTape() as tape:
        current_q_values = deep_nn_approximator(np.array([current_state_input]))
        loss = MSE([target_q_values], [current_q_values[0, selected_action]])
    grads = tape.gradient(loss, deep_nn_approximator.trainable_variables)
    deep_nn_approximator.optimizer.apply_gradients(zip(grads, deep_nn_approximator.trainable_variables))

    return deep_nn_approximator
```

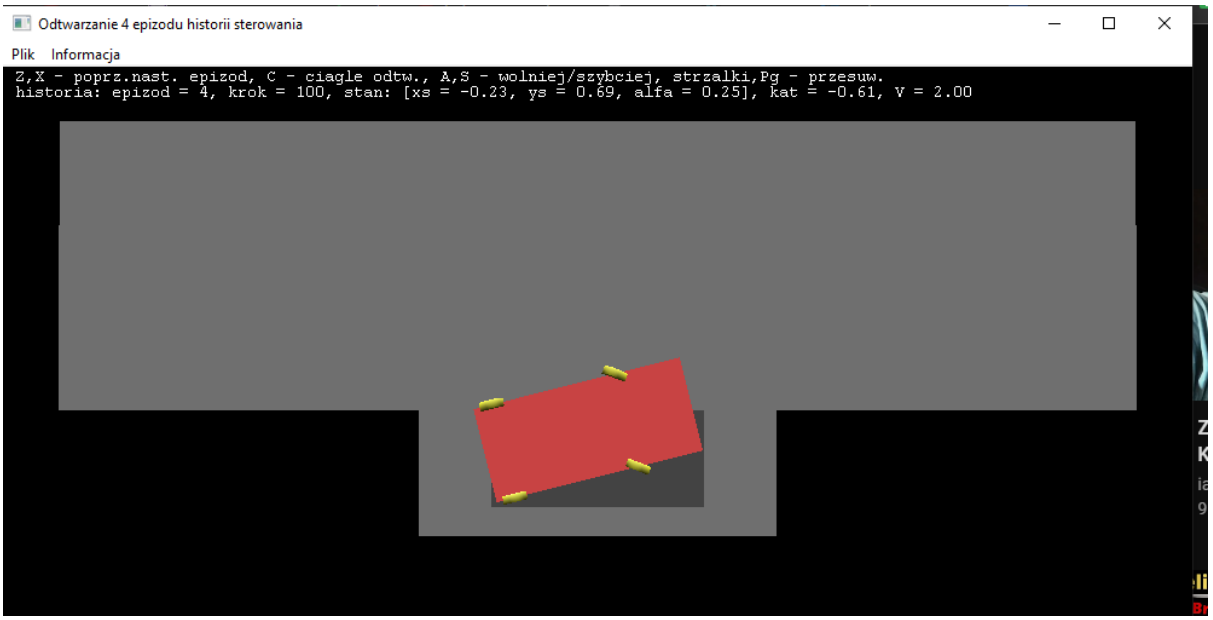
```python
def get_reward(physical_parameters, state, is_collision, is_stopped, recorder, epoch, num_of_epochs):
    # Dostosowanie nagrody na podstawie epoki
    curriculum_factor = epoch / num_of_epochs  # Procentowy postęp treningu
```

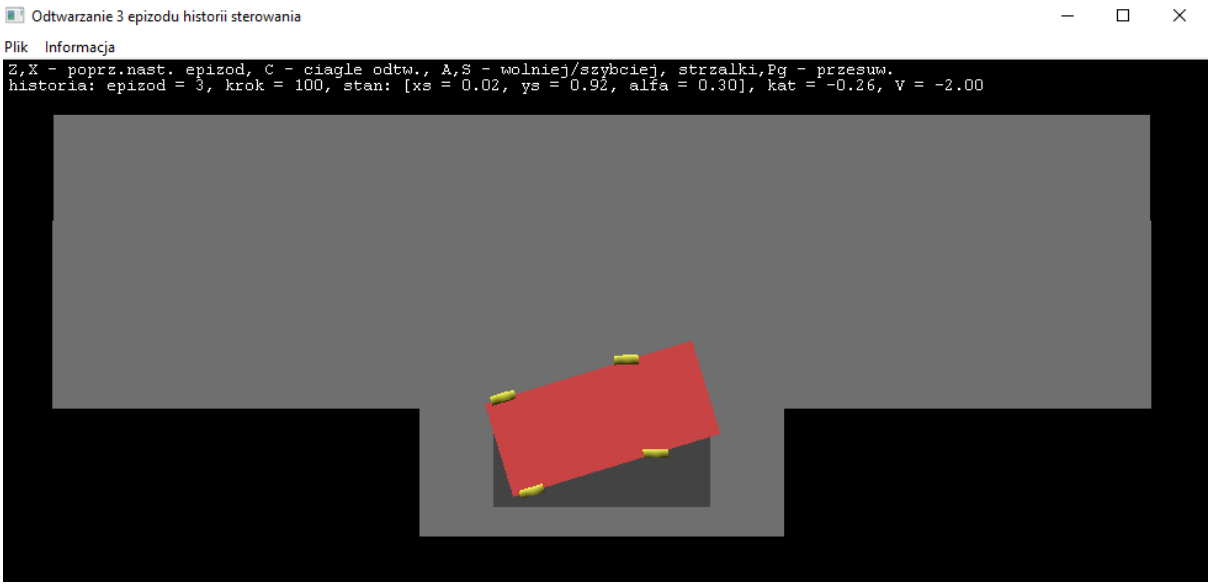
```python
if is_collision:
    value = -10.0
elif is_stopped:
    # Zwiększenie nagrody w miarę postępu treningu
    value = distance_reward + (4 * (1 + curriculum_factor) * alpha_reward) + get_final_score(physical_parameters, state)
else:
    value = distance_reward

return value
```

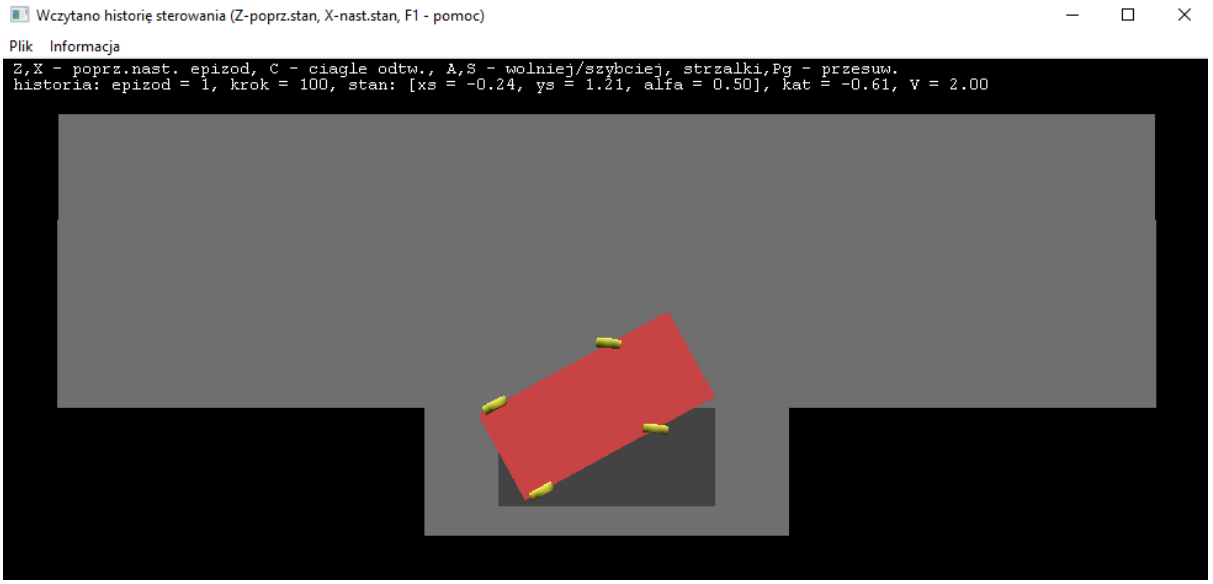
Efekty uczenia:

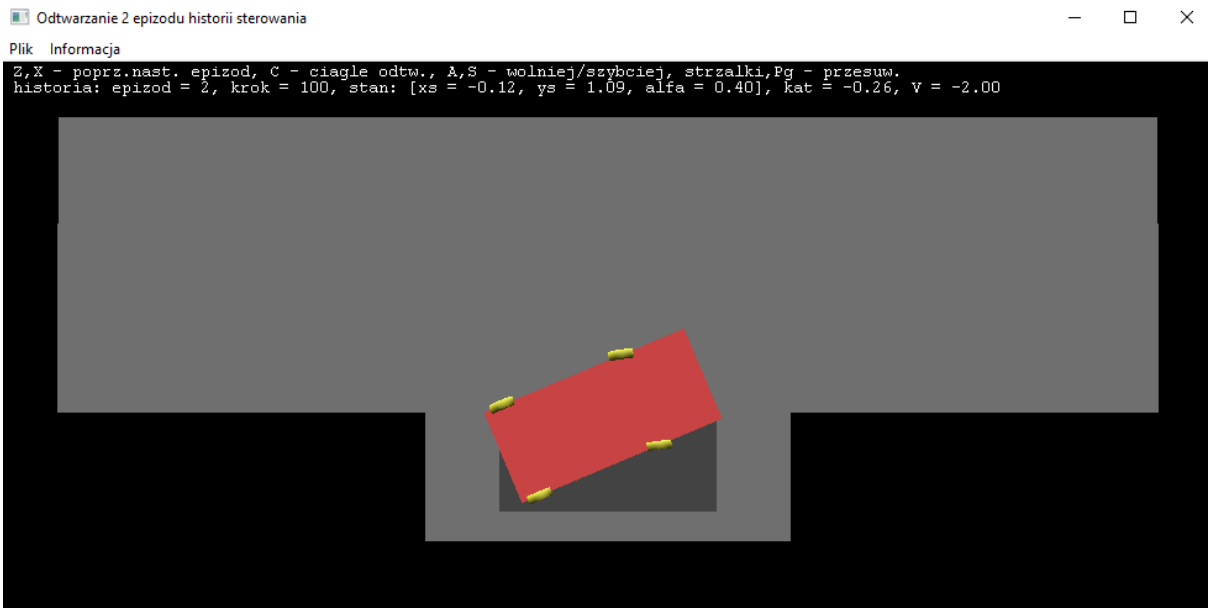
Standardowy Q-learning:





Metoda Minibatch:





Metoda Curriculum learning: