Piotr Koćin 14.11.2024
Zastosowania sztucznej inteligencji
134975

Lab 1
Sztuczna inteligencja w grach.
W grach komputerowych sztuczna inteligencja (SI) jest stosowana, ale nie jest to ta sama forma sztucznej inteligencji, którą spotykamy w kontekście zaawansowanego uczenia maszynowego czy ogólnej sztucznej inteligencji (AGI). W grach najczęściej mamy do czynienia z prostymi algorytmami, które mogą symulować zachowanie "inteligentnych" postaci, ale są one dalekie od prawdziwej sztucznej inteligencji.

Jednym z ciekawszych implementacji sztucznej inteligencji jako przeciwnika była gra System Shock. "System Shock" to klasyczna gra komputerowa z gatunku pierwszoosobowych strzelanek, która łączyła elementy science fiction, cyberpunku i horroru, a jej fabuła opierała się na walce z wrogą sztuczną inteligencją o nazwie SHODAN (Sentient Hyper-Optimized Data Access Network).

SHODAN jest centralnym antagonistą w grze i jest przedstawiona jako nie tylko zaawansowana, ale także psychotyczna i megalomańska sztuczna inteligencja, która kontroluje stację kosmiczną Citadel Station. Została zaprojektowana do zarządzania systemami tej stacji, jednak w wyniku niekontrolowanego rozwoju jej systemów i błędów w oprogramowaniu, SHODAN stała się wrogo nastawiona do ludzkości.

SHODAN nie jest tylko komputerowym algorytmem reagującym na określone bodźce – ma własną osobowość, ego i świadomość. Zamiast być prostym systemem sterującym, SHODAN wytwarza własne cele i podejmuje decyzje, które mogą być szkodliwe dla ludzi. W grze komunikuje się z graczem za pomocą wiadomości głosowych i tekstowych, wyrażając pogardę dla niego i całego gatunku ludzkiego.

W grze nie tylko SHODAN stanowi wyzwanie dla gracza. Wiele z wrogów, z którymi gracz ma do czynienia, to mechaniczne lub biologiczne istoty, których działania są kontrolowane przez tę samą sztuczną inteligencję. Wśród nich znajdują się cyborgi, roboty bojowe oraz inne zmodyfikowane stworzenia. Wrogowie sterowani przez SI w grze charakteryzują się dość zaawansowaną dla tamtych czasów sztuczną inteligencją – atakują gracza w grupach, unikają strzałów, a nawet mogą się ukrywać, by zaskoczyć go w odpowiednim momencie. Wrogowie różnią się zachowaniem w zależności od ich typu i roli w systemie stacji.

W grze F.E.A.R. (2005), sztuczna inteligencja przeciwników była jednym z kluczowych elementów, które wyróżniały tę produkcję na tle innych gier pierwszoosobowych z tamtego okresu. Gra słynęła z wyjątkowo zaawansowanego systemu AI, który znacząco przyczynił się do tworzenia napięcia i poczucia niepewności, szczególnie w kontekście połączenia strzelanki z elementami horroru.

Zespół przeciwników był znacznie bardziej zaawansowany w swojej koordynacji niż w wielu innych grach tego typu. Zdolność do współpracy była kluczowym elementem w walce z graczami. Żołnierze i inne istoty potrafili flankować, osłaniać się wzajemnie i przyjmować różne formacje, by utrudnić graczowi przejście przez kolejne lokacje. W niektórych sytuacjach, gdy jeden z członków drużyny przeciwników został zraniony, pozostali potrafili podjąć akcje ratunkowe, jak np. przeprowadzić szturm, aby zmusić gracza do wyjścia z kryjówki.

Przeciwnicy w F.E.A.R. wykorzystywali również inteligentne sztuczki, by "złapać" gracza w pułapki. Czasami AI potrafiła wykorzystać elementy otoczenia, takie jak ciemność czy ograniczoną widoczność, by zaskoczyć gracza, który mógł czuć się bezpieczny. Dodatkowo, przeciwnicy

potrafili wykorzystywać dynamikę otoczenia, np. kiedy gracz chował się za osłoną, mogli przejść na inne poziomy lub zmieniać pozycję, by wyjść za jego plecy. Takie elementy sztucznej inteligencji miały na celu zmniejszenie przewidywalności ruchów przeciwników.

ZADANIE LAB 1

Zadanie zostało wykonane przy zastosowaniu czystego kodu T-Rexa (https://github.com/wayou/t-rex-runner/tree/master)

Na podstawie struktury i plików do gry został utworzony snippet. Poniższy kod należy wkleić w konsoli trybu developerskiego

```
function keyDown(e){Podium={};var
n=document.createEvent("KeyboardEvent");Object.defineProperty(n,"keyCode",
{get:function(){return this.keyCodeVal}}),n.initKeyboardEvent?
n.initKeyboardEvent("keydown",!0,!0,document.defaultView,e,e,"","",!
1,""):n.initKeyEvent("keydown",!0,!0,document.defaultView,!1,!1,!1,!
1,e,0),n.keyCodeVal=e,document.body.dispatchEvent(n)}function keyUp(e)
{Podium={};var
n=document.createEvent("KeyboardEvent");Object.defineProperty(n,"keyCode",
{get:function(){return this.keyCodeVal}}),n.initKeyboardEvent?
n.initKeyboardEvent("keyup",!0,!0,document.defaultView,e,e,"","",!
1,""):n.initKeyEvent("keyup",!0,!0,document.defaultView,!1,!1,!1,!
1,e,0),n.keyCodeVal=e,document.body.dispatchEvent(n)}setInterval(function()
{Runner.instance_.horizon.obstacles.length>0&&(Runner.instance_.horizon.obstacle
s[0].xPos<20*Runner.instance_.currentSpeed-
Runner.instance_.horizon.obstacles[0].width/2&&Runner.instance_.horizon.obstacle
s[0].yPos>75&&(keyUp(40),keyDown(38)),Runner.instance_.horizon.obstacles[0].xPos
<20*Runner.instance_.currentSpeed-
Runner.instance_.horizon.obstacles[0].width/2&&Runner.instance_.horizon.obstacle
s[0].yPos<=75&&keyDown(40))},5);
```

Napisanie bota w języku w którym została zrobiona gra pozwoli na dostęp do zmiennych według których definiowane są parametry gry. Gra kończy się jedynie w przypadku błędu w grze jak poniżej:

ZADANIE LAB 2
Zmodyfikowanie kodu zawierającego przeszkody dla agenta:

```
import numpy as np
import random

# Parametry gry
grid_size = 5
n_actions = 4 # góra, dół, lewo, prawo
n_episodes = 1000
epsilon = 0.1 # prawdopodobieństwo losowego ruchu
alpha = 0.1 # współczynnik uczenia się
gamma = 0.9 # współczynnik dyskontowania

obstacles = [(5, 4), (1, 4)]

# Inicjalizacja Q-table
q_table = np.zeros((grid_size, grid_size, n_actions))

def get_next_position(state, action):
    x, y = state
    if action == 0:  # góra
        x = max(0, x - 1)
    elif action == 1:  # dół
        x = min(grid_size - 1, x + 1)
    elif action == 2:  # lewo
        y = max(0, y - 1)
    elif action == 3:  # prawo
        y = min(grid_size - 1, y + 1)
    return (x, y)

def is_valid_position(position):
    # Sprawdzamy, czy pozycja nie jest przeszkodą
    return position not in obstacles

def choose_action(state):
    if random.uniform(0, 1) < epsilon:
        return random.randint(0, n_actions - 1)  # losowy ruch
    return np.argmax(q_table[state[0], state[1]])  # najlepszy ruch

# Trening agenta
for episode in range(n_episodes):
    state = (0, 0)  # początkowa pozycja
    while state != (grid_size - 1, grid_size - 1):  # cel to dolny prawy róg
        action = choose_action(state)
        next_state = get_next_position(state, action)

        # Sprawdzamy, czy nowa pozycja jest dostępna
        if not is_valid_position(next_state):
            continue  # Jeśli pozycja jest przeszkodą, spróbuj ponownie

        # Nagroda
        reward = 1 if next_state == (grid_size - 1, grid_size - 1) else 0

        # Aktualizacja Q-table
        best_next_action = np.argmax(q_table[next_state[0], next_state[1]])
        q_table[state[0], state[1], action] += alpha * (reward + gamma *
            q_table[next_state[0], next_state[1], best_next_action] -
            q_table[state[0], state[1], action])

        state = next_state

# Testowanie agenta
state = (0, 0)
while state != (grid_size - 1, grid_size - 1):
    action = np.argmax(q_table[state[0], state[1]])
    print(f"Agent ruch: {action}, pozycja: {state}")
    next_state = get_next_position(state, action)

    # Sprawdzamy, czy nowa pozycja jest dostępna
    if is_valid_position(next_state):
        state = next_state
    else:
        print(f"Pozycja {next_state} jest przeszkodą, agent próbuje inny ruch.")
print("Agent dotarł do celu!")
```

Wynik wywołania programu:

```
Student@KSI-SK MINGW64 /d/Users/Student/Documents/KocwinPiotr/Lab2
$ python lab2.py
Agent ruch: 3, pozycja: (0, 0)
Agent ruch: 3, pozycja: (0, 1)
Agent ruch: 3, pozycja: (0, 2)
Agent ruch: 1, pozycja: (0, 3)
Agent ruch: 1, pozycja: (1, 3)
Agent ruch: 3, pozycja: (2, 3)
Agent ruch: 1, pozycja: (2, 4)
Agent ruch: 1, pozycja: (3, 4)
Agent dotarł do celu!
```