Plan realizacji przedmiotu obieralnego w ramach koła naukoweg

Zima 2023/2024

26 stycznia 2024

Streszczenie

Przystosowanie sztucznej inteligencji do gry w prosty symulator przetrwania oraz analiza zachowania agentów z coraz większym współczynnikiem przeżywalności

1 Projekt symulacji

Określona ilość osobników osobników zostanie umieszczona na mapie o określonej wielkości, z elementami zarówno stałymi, jak i różniącymi się z rozgrywki na rozgrywkę:

1. Pola z jedzeniem (grzyby - niektóre mogą być trujące, pola z jedzeniem odradzają się po odpowiednim czasie)
2. Rzeki/jeziora (osobnik nie umie pływać, przeszkoda)
3. Lasy z drapieżnikami
4. bagna spowalniające gracza

Osobnik może zginąć:

- Z głodu (każdy ruch kosztuje energię)
- Zatruty trującym jedzeniem (musi nauczyć się sprawdzać, co jednak kosztuje punkty energii)
- Utopiony w rzecze (musi nauczyć się omijać wodę)
- Zabity w lesie z pewnym prawdopodobieństwem (ale w lesie jest więcej jedzenia)

Po określonym czasie osobniki, które przeżyją zrodzą potomstwo.

2 Implementacja

2.0.1 Szata graficzna

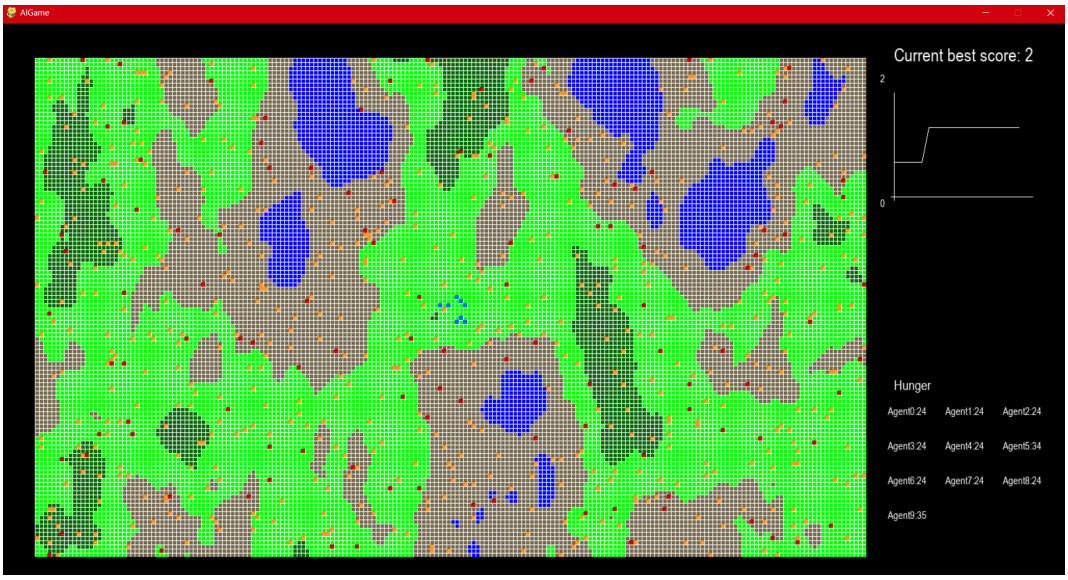
Szata graficzna gry pozostała utrzymana w prostocie - biblioteka pygame nie zachwyca możliwościami graficznymi. Program oferuje jednak dostateczne funkcje

2.0.2 Mapa

Świat użyty w grze jest generowany proceduralnie przez specjalnie wydzielony moduł za pomocą szumu Perlina - jednego z najpowszechniej używanych sposobów generowania światów w grach komputerowych. Mapa generowana jest za pomocą "ziarna"(*ang. seed*), co pozwala na inicjalizowanie tej samej mapy w różnych rozgrywkach oraz znacząco zmniejsza wielkość plików zapisu stanu gry, chociaż za cenę dłuższej inicjalizacji. Na mapie za pomocą *colormap* wyznaczane są następujące tereny (rys. 1)

- woda - na którą gracz nie może wejść (pewna śmierć)

- las, w którym gracz jest narażony na drapieżniki (prawdopodobieństwo śmierci, na obecnym etapie, 2%), ale również mają więcej szans na *spawn* jedzenia
- bagna, które spowalniają gracza
- trawa



Rysunek 1: FInalny wygląd gry [**?**]

2.0.3 Szum Perlina

Szum Perlina jest techniką generowania pseudolosowych, przyjemnych dla oka, nieregularnych wzorców. Znajduje szerokie zastosowanie w tworzeniu realistycznych krajobrazów, animacji, efektów specjalnych oraz tekstur o naturalnym wyglądzie ([**?**]).

Podstawowe równania szumu Perlin są oparte na interpolacji gradientów w przestrzeni. Dla jednego wymiaru, można opisać szum Perlin następującym równaniem:

$$\text{szum}(x) = \sum_{i=0}^{n} \text{gradient}(i) \cdot \text{skalar\_produkty}(x - i) \cdot \text{fade}(x - i)$$

gdzie:

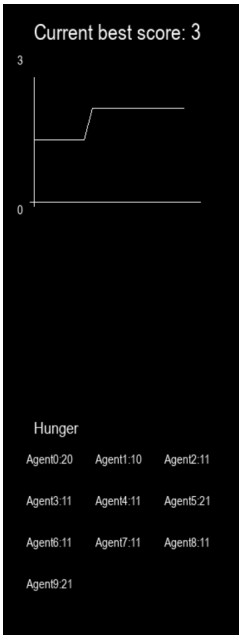
- $\text{gradient}(i)$ to gradient losowy w punkcie i,
- $\text{skalar\_produkt}(x - i)$ to iloczyn skalarny wektora $(x - i)$ i gradientu w punkcie i,
- $\text{fade}(t)$ to funkcja fade, która pomaga w płynnym przejściu pomiędzy całkowitymi wartościami.

W przypadku dwóch wymiarów równanie to można rozszerzyć do dwóch wymiarów, a dla trzech wymiarów - do trzech. Ogólnie szum Perlin jest wykorzystywany jako funkcja wielowymiarowa, co pozwala na generowanie bardziej złożonych, naturalnych struktur.

W projekcie szum perlina został zaimplementowany do generacji świata przy pomocy biblioteki perlin-noise [**?**] oraz specjalnie stworzonego modułu `world_generator`, który, używając specjalnie dobranej mapy kolorów, tworzy na podstawie szumu Perlina odpowiednie kolory na mapie.

2.0.4 Statystyki

Program wyświetla statystyki obecnej "tury"gry: wyświetla wykres przedstawiający rosnący rekord punktów (ilości zjedzonych elementów). Ponadto widoczny jest poziom "głodu"każdego z uczestników (rys. 2)



Rysunek 2: Statystyki agentów

2.1 Wersja gry dla ludzkiego użytkownika

Projekt zawiera wersję gry dla ludzkiego użytkownika. Służy ona porównaniu sprawności człowieka i komputera oraz była przydatna w procesie debugowania.

2.1.1 Kontrola

Gracz kontroluje agentem za pomocą przycisków "strzałek".

2.1.2 Zasady gry

Gracza obejmują te same zasady gry, co sztuczną inteligencję, z wyjątkiem głodu - ludzki gracz ma podniesiony poziom trudności, ze względu na przewagę od początku rozgrywki.

2.2 Sztuczna inteligencja

Głównym przedmiotem tego projektu były badania zachowania sztucznej inteligencji w stworzonej grze.

2.2.1 Użyte biblioteki

`Modele`

2.2.2 Klasa Linear_QNet

Klasa będąca modelem każdego z agentów. Składa się z następujących komponentów:

- **Konstruktor (`__init__`)**

```
1             def __init__(self, input_size, hidden_size,
                  ↪ output_size):
```

  Konstruktor inicjalizuje sieć neuronową typu feedforward z trzema warstwami ukrytymi. Rozmiary warstw wejściowej, ukrytej i wyjściowej są określone przez parametry `input_size`, `hidden_size` i `output_size`.

- **Metoda Forward (forward)**

```
def forward(self, x):
```

Metoda `forward` definiuje przebieg przód sieci neuronowej. Stosuje funkcję aktywacji ReLU do wyjścia każdej warstwy liniowej, z wyjątkiem ostatniej.

- **Metoda Zapisz (save)**

```
def save(self, file_name='model.pth'):
```

Metoda `save` zapisuje słownik stanu modelu do pliku o nazwie określonej przez `file_name`. Tworzy folder o nazwie `model`, jeśli nie istnieje.

- **Metoda Pobierz Wagi jako Tablica (get_weights_as_array)**

```
def get_weights_as_array(self):
```

Ta metoda zwraca wagi modelu jako spłaszczoną tablicę NumPy.

- **Metoda Wczytaj Wagi z Tablicy (load_weights_from_array)**

```
def load_weights_from_array(self, weights_array):
```

Metoda `load_weights_from_array` wczytuje wagi z przekazanej tablicy do modelu.

2.2.3 Klasa QTrainer

Klasa odpowiedzialna za uczenie modelu agenta. Składa się z następujących komponentów:

- **Konstruktor (__init__)**

```
def __init__(self, model, lr, gamma):
```

Konstruktor inicjalizuje obiekt QTrainer z modelem sieci neuronowej, współczynnikiem uczenia (`lr`) i współczynnikiem dyskontowym (`gamma`). Ustawia również optymalizator Adam i funkcję straty Mean Squared Error (MSE).

- **Metoda Krok Treningowy (train_step)**

```
def train_step(self, state, action, reward, next_state
    ↪ , done):
```

Metoda `train_step` wykonuje pojedynczy krok treningowy algorytmu Q-learning. Oblicza przewidziane wartości Q, oblicza docelowe wartości Q i aktualizuje wagi modelu przy użyciu optymalizatora Adam.

2.2.4 Ewolucja

Reinforecement Learning Uczenie ze wzmocnieniem (ang. Reinforcement Learning, RL) to obszar sztucznej inteligencji, który koncentruje się na rozwijaniu algorytmów i systemów, zdolnych do samodzielnego uczenia się poprzez interakcję z otoczeniem. W kontekście uczenia ze wzmocnieniem, agent, czyli program komputerowy lub system, podejmuje decyzje w środowisku w celu osiągnięcia określonych celów. Kluczowym elementem tego podejścia jest ideologia nagrody i kary, gdzie agent stara się maksymalizować zbierane nagrody i minimalizować kary poprzez eksplorację różnych działań.

W trakcie procesu uczenia ze wzmocnieniem, agent podejmuje akcje, a środowisko reaguje na te akcje, dostarczając z powrotem informacje zwrotną w postaci nagród lub kar. Celem agenta jest odkrywanie strategii, które umożliwiają mu maksymalizację sumy nagród w dłuższej perspektywie czasowej. Algorytmy uczenia ze wzmocnieniem często wykorzystują tzw. funkcje wartości, które pomagają agentowi oceniać, jak dobre są różne akcje w danym kontekście.

W projekcie zostało zastosowane uczenie ze wzmocnieniem. W każdej turze agent musi podjąć decyzję dotyczącą następnego kroku. Za akcje dostaje informacje zwrotne w postaci kar bądź nagród:

- za śmierć otrzymuje karę -10
- za zjedzenie jednego "grzybka"dostaje 10 punktów oraz bonus za to, jak szybko od początku generacji to zrobił. Jedzenie na bagnach liczy się podwójnie.

Każdy agent pamięta swój najlepszy wynik.

2.2.5 Algorytmy genetyczne

Metoda `evolve` odpowiada za ewolucję populacji agentów w systemie. W zależności od liczby najlepszych agentów w populacji, metoda przeprowadza ewolucję, uwzględniając operacje selekcji turniejowej, krzyżowania, mutacji i tworzenia nowych agentów.

```
    def evolve(self):
```

Brak Najlepszych Agentów Jeśli liczba najlepszych agentów wynosi 0, metoda kończy się bez zmian w populacji.

```
    if len(self.best_agents)  == 0:
        return
```

Jeden Najlepszy Agent Jeśli istnieje tylko jeden najlepszy agent, metoda zastępuje populację nowymi agentami, przy czym większość z nich dziedziczy wagi od tego jednego najlepszego agenta.

```
    if len(self.best_agents) == 1:
        self.agents = []
        for i in range(self.num_agents):
            agent = Agent(i)
            x = random.randint(0, 10)
            if x > 3:
                agent.model.load_weights_from_array(self.best_agents
                    ↪ [0].model.get_weights_as_array())
            self.agents.append(agent)
```

Więcej Niż Jeden Najlepszy Agent W przypadku, gdy istnieje więcej niż jeden najlepszy agent, metoda przeprowadza selekcję turniejową, krzyżowanie, mutację i tworzenie nowych agentów.

```
    else:
        selected_agents = self.tournament_selection(self.agents, 2)
        offspring = []
        for i in range(0, len(selected_agents)-2):
            child = Agent(i)
            child_weights = self.crossover(selected_agents[i],
                ↪ selected_agents[i + 1])
            child.model.load_weights_from_array(child_weights)
            offspring.append(child)

        # Mutation
        mutated_offspring = [self.mutate(child) for child in offspring
            ↪ ]
        if len(mutated_offspring) < self.num_agents:
            for i in range(len(mutated_offspring), self.num_agents-1):
                mutated_offspring.append(Agent(i))
        self.agents = mutated_offspring
```

Poniżej znajdują się krótkie opisy innych metod używanych w procesie ewolucji.

Metoda `tournament_selection` Metoda przeprowadza selekcję turniejową, gdzie z określonej liczby agentów wybierany jest zwycięzca na podstawie rekordu.

```
    def tournament_selection(self, agents, tournament_size):
```

Metoda `negative_tournament_selection` Metoda przeprowadza selekcję turniejową, ale wybierany jest przegrany, czyli agent z najniższym rekordem.

```
def negative_tournament_selection(self, agents, tournament_size):
```

Metoda `crossover` Metoda przeprowadza krzyżowanie pomiędzy dwoma rodzicami, tworząc potomstwo.

```
def crossover(self, parent1, parent2):
```

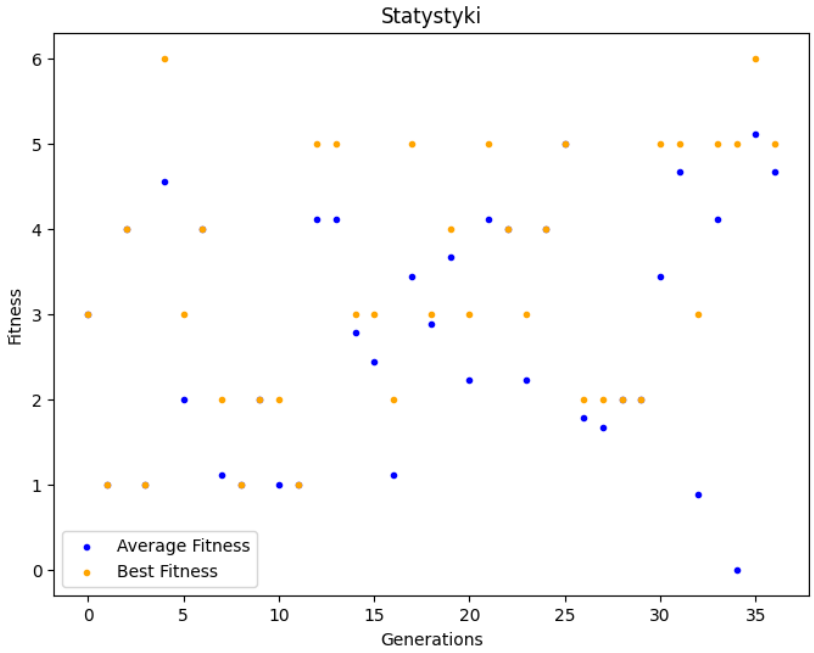
Metoda `mutate` Metoda wprowadza mutacje do wag agenta, zmieniając je o niewielką losową wartość.

```
def mutate(self, agent):
```

3 Obserwacje

3.1 Regularność modelu

Szczególnie w pierwszysch fazach nauki, model nie wykazuje szczególnych regularności, jeżeli chodzi o wyniki. Chociaż zazwyczaj wykazuje tendencję wzrostową (przykładowo rys.3), ze względu na losowość mapy oraz algorytmów genetycznych (mutacje, możliwosc crossoveru ze słabszym osobnikiem) jego wyniki bywają chaotyczne (rys. 4). Oczywiście, wprowadzenie entropii do gry wiązało się z takim



Rysunek 3: Średnie i najlepsze wyniki agentów podczas pierwszej symulacji

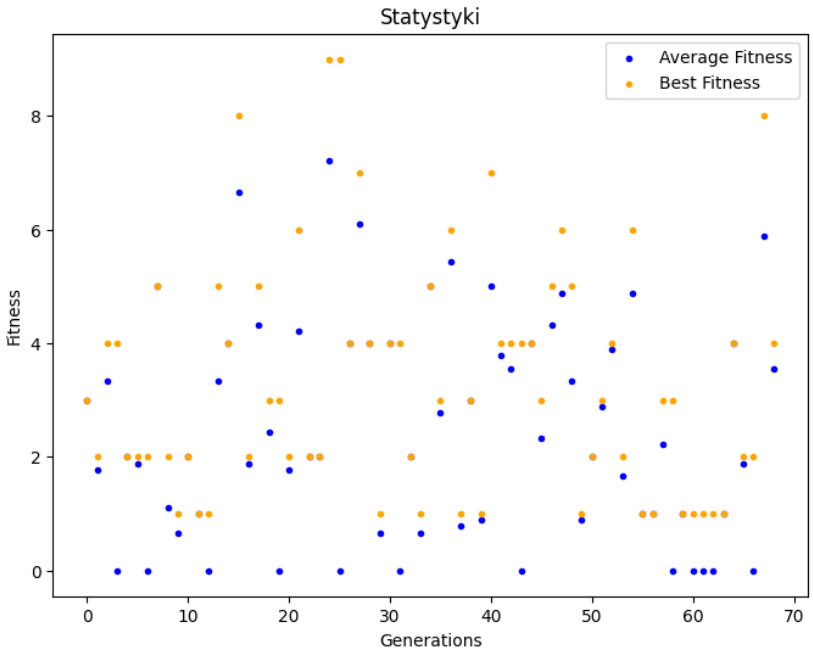
scenariuszem. Dobrze dostosowane osobniki mogą zostać wyeliminowane przez brak szczęścia:

- losowa śmierć na jednym z trudniejszych terenów
- brak jedzenia w pobliżu
- zła decyzja zjedzenia trucizny

3.2 Użycie różnych modeli sieci neuronowych

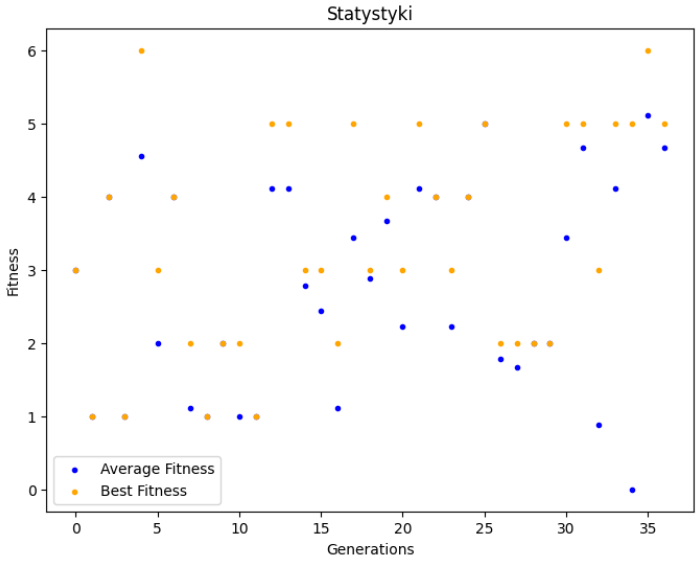
Następujące rysunki przedstawiają wyniki półgodzinnego uczenia niewielkiej liczby agentów (10 agentów) w czasie pół godziny przy użyciu modeli o odpowiednio:

- 4 warstwach ukrytych (rys. 5)



Rysunek 4: Średnie i najlepsze wyniki agentów podczas drugiej symulacji

- 6 warstwach ukrytych (rys. 6)
- 8 warstwach ukrytych (rys. 7)

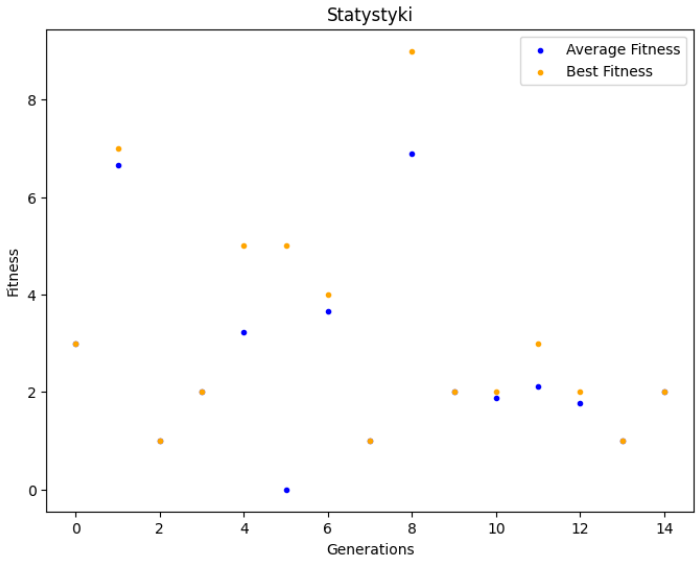


Rysunek 5: Średnie i najlepsze wyniki agentów podczas pierwszej symulacji

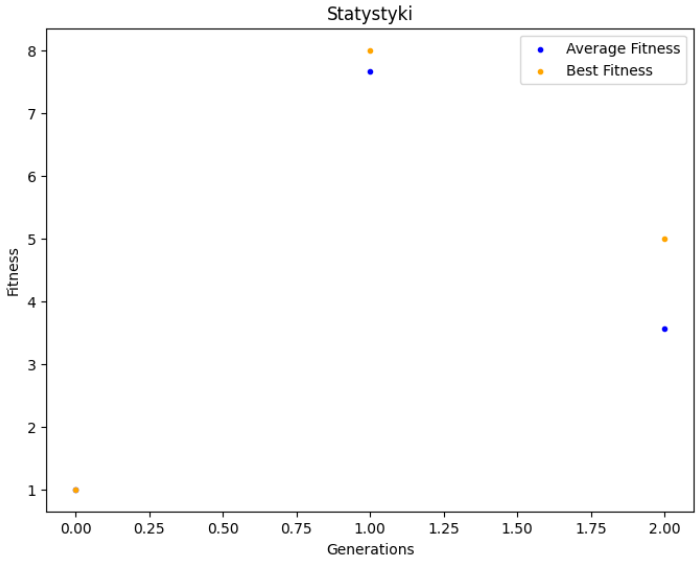
Na podstawie tych statystyk można wysnuć jeden prosty wniosek: użyty sprzęt jest niewystarczający. Zwiększanie ilości warstw ukrytych przyniosło spodziewane zwiększenie czasu potrzebnego na obliczenia - skok jest jednak tak drastyczny, że znacząco utrudnia pracę.

3.3 Zachowania agentów

Agenty podczas przeprowadzania symulacji przedstawiały często następujące zachowania:



Rysunek 6: Średnie i najlepsze wyniki agentów podczas pierwszej symulacji



Rysunek 7: Średnie i najlepsze wyniki agentów podczas pierwszej symulacji

- tendencja do trzymania się w kupie wpocząkowych fazach uczenia - szczególnie podczas pierwszych faz uczenia agenty przejawiają tendencję to pozostawania w jednym miejscu - wynika to z jednolitości modelu. Agenty są wtedy "jednomyślne"i faworyzują poruszanie się na niewielkim terenie. W miarę nauki agenty stają się coraz bardziej "ciekawskie"i osiągają większe odległości od miejsca startowego
- rozpoznawanie trucizny - agenty stosunkowo szybko uczą się rozpoznawać truciznę

3.4 Człowiek kontra komputer

Oczywiście, nie będzie zaskoczeniem stwierdzenie, że człowiek jest nieporównywalnie lepszy od prostego, uczącego się modelu. Przeciętny gracz potrzebuje zaledwie chwili do zrozumienia zasad i osiągnięcia płynności w grze - nie jest ona skomplikowana dla ludzkiego mózgu. Pomijając niesprzyjające ułożenia losowe, ludzki gracz jest w stanie prowadzić nieprzerwaną rozgrywkę parę-parenaście minut.

Kontrastowo, do wyuczenia tak dobrego modelu potrzebne są długie godziny. Jednakże odpowiednie dobranie hiperparametrów i ilości agentów, na odpowiednio mocnym sprzęcie, pozwoliłoby teoretycznie wytworzenie modelu, który byłby lepszy od człowieka.

4 Napotkane trudności

Największym wyzwaniem w projekcie okazały się wymagania sprzętowe. Niestety, posiadany sprzęt okazał się zbyt słaby, aby osiągnąć w szybkim czasie imponujące wyniki. Sprawiał też bardzo duże ograniczenia dotyczące ilości agentów (niewystarczająca ilość pamięci RAM)

5 Podsumowanie

W ramach projektu udało się stworzyć program umożliwiający uczenie sztucznej inteligencji naukę w grę typu survival, z wersją dla gracza ludzkiego dla porównania.

Ze względu na ograniczenia czasowe jak i przede wszystki sprzętowe udało się wyszkolić jedynie podstawowe modele - niebędące w stanie dogonić ludzkiego człowieka w wynikach. Agenty jednak powoli, ale z pewną skutecznością uczyły się zasad gry.

Projekt jest gotowym produktem pozwalającym na prowadzenie badań nad sztuczną inteligencją.