I. WPROWADZENIE DO PROJEKTU[1]

Algorytmy *DRL* (głębokiego uczenia ze wzmocnieniem) to systemy sztucznej inteligencji, które uczą się rozwiązywać zadane im problemy poprzez bezpośrednią interakcję ze środowiskiem. Dokonują tego metodą prób i błędów, otrzymując od środowiska nagrody, które służą kierowaniu procesu uczenia[2].

Problemy tego rodzaju są nazywane *MDP* (*Markov decision problem* - problemy decyzyjne Markowa). Prostsze z nich mogą być określane poprzez tablice lub grafy, które określają: występujące stany, przejścia pomiędzy nimi i nagrody. Niestety, w przypadku dużej ilości problemów istnieje problem ich reprezentacji – są one po prostu zbyt skomplikowane. W związku z tym zaczęto wykorzystywać techniki oparte na sieciach neuronowych - *deep learning* (uczenie głębokie).

W ramach projektu utworzony został agent sztucznej inteligencji uczący się grać w grę *„Connect 4”*. Algorytmy *DRL* (*deep reinforcement learning* – głębokie uczenie ze wzmocnieniem) już od początku swojej historii są związane z grami planszowymi. Jednym z pierwszych i lepiej znanych przypadków zastosowania takich algorytmów jest program *„TD-Gammon”*, który poprzez grę z samym sobą osiągnął w 1992 roku poziom najlepszych ludzkich graczy[3]. W ostatnich czasach rozwój tych algorytmów znacznie przyśpieszył – dzisiaj są one w stanie rozwiązywać znacznie bardziej skomplikowane gry. Przykładem jest np. *„Go”*[4] czy *„Dota 2”*[5].

Rozwiązywana w ramach projektu gra jest znacznie prostsza niż podane przykłady[6]. Posiada ona planszę złożoną z 6 wierszy i 7 kolumn, a jej celem jest ułożenie 4 krążków w linii. Jest ona również rozwiązana w sposób silny – istnieje optymalne jej rozwiązanie, w którym pierwszy gracz jest w stanie wymusić wygraną na przeciwniku[7]. Rzeczywiście, projekt był rozpatrywany w ramach internetowego konkursu zorganizowanego przez firmę *Kaggle* i pierwsze miejsca w tym konkursie

[1] Dostęp do wszystkich źródeł elektronicznych został sprawdzony na dzień 22.01.2022

[2] https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf, strona 1

[3] Gerald Tesauro, 1995, *“Temporal difference learning and TD-Gammon”*

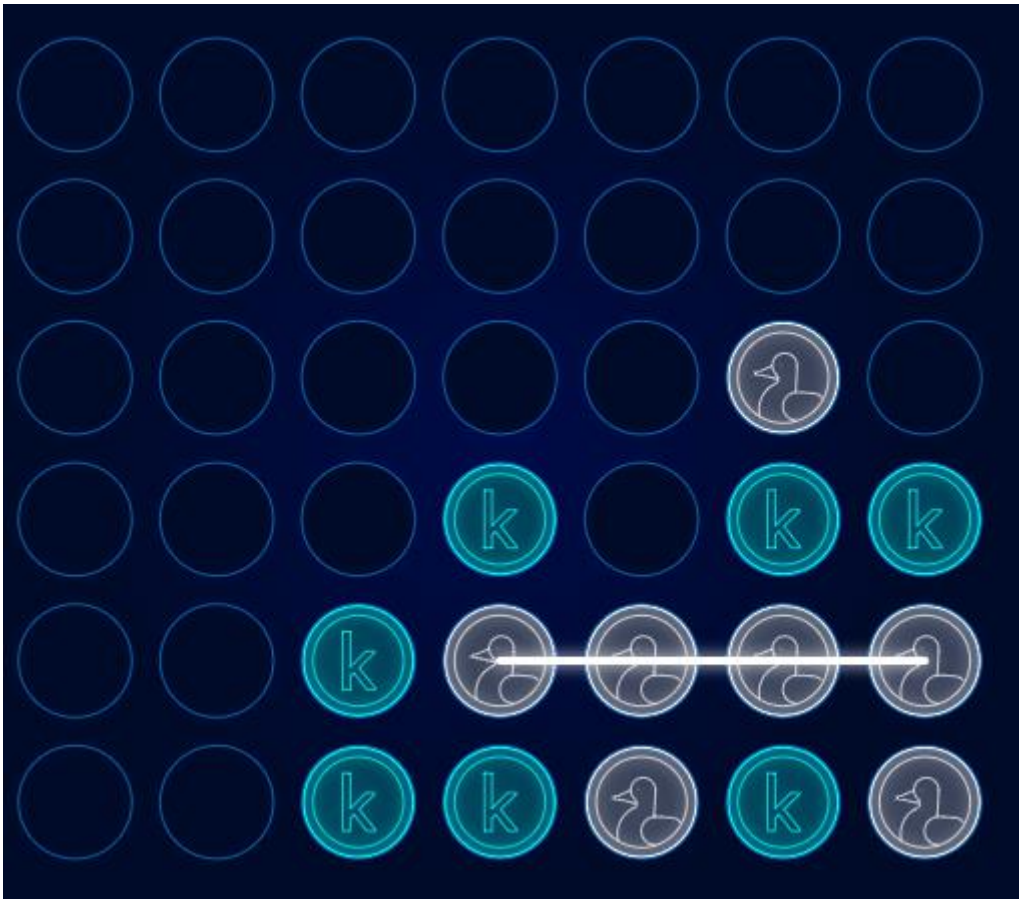
[4] AlphaGo - https://deepmind.com/research/case-studies/alphago-the-story-so-far

[5] OpenAI Five - https://openai.com/five/

[6] https://en.wikipedia.org/wiki/Game_complexity

[7] https://en.wikipedia.org/wiki/Solved_game, https://en.wikipedia.org/wiki/Connect_Four

zajmują rozwiązania heurystyczne, które przeszukują drzewo stanów gry, do jej końca[8]. Takie rozwiązanie jest w tym przypadku prostsze i bardziej skuteczne, ponieważ agent *DRL* musiałby się nauczyć perfekcyjnie oceniać 4 531 985 219 092 możliwych pozycji na planszy.



Źródło: https://www.kaggle.com/c/connectx/overview

Rys. 1 Przykład planszy skończonej gry „Connect 4". Wygrana przez drugiego gracza (oznaczonego krążkami z kaczką).

[8] https://www.kaggle.com/c/connectx/discussion/188305

II. IMPLEMENTACJA PROJEKTU

Projekt został utworzony w języku *Python*, w ramach interaktywnego środowiska *Jupyter Notebook*, które jest również dostępne na platformie *Kaggle*. Przyjęte rozwiązanie umożliwia w prosty sposób bogatą wizualizację uzyskanych wyników.

W projekcie zostały użyte następujące biblioteki:

- *OpenAI Gym*[9] – utworzona przez firmę *OpenAI* biblioteka, która umożliwia łatwiejszą standaryzację środowisk reprezentujących problemy *MDP*. Umożliwia to łatwiejszą implementację algorytmów, które je rozwiązują.
- *NumPy*[10] - biblioteka implementująca typy zmiennych dla języka *Python,* które ułatwiają i przyśpieszają wykonywanie obliczeń.
- *PyTorch*[11] - biblioteka udostępniająca moduły do budowy szybkich i wydajnych sieci neuronowych.
- *Stable-Baselines3*[12] - biblioteka algorytmów *DRL* utworzonych w oparciu o środowiska *OpenAI Gym*. Stanowi ona rozwinięcie projektu *OpenAI Baselines*[13], którego celem jest ułatwienie replikacji wyników osiągniętych w ramach badań naukowych nad algorytmami *DRL*, co stanowi spory problem[14].
- *Kaggle Environments*[15] - biblioteka z funkcjami pomocniczymi, które ułatwiają tworzenie rozwiązań w ramach konkursów *Kaggle*.

W projekcie zaimplementowano środowisko do gry w grę *„Connect 4”*, które ustandaryzowano przy pomocy bibliotek *OpenAI Gym* i *Kaggle Environments*. Środowisko to posiada zdefiniowane w ramach *MDP* nagrody, możliwe akcje agenta oraz stan planszy. Najważniejsze funkcje tego środowiska to *reset* i *step*, które

[9] https://gym.openai.com/

[10] https://numpy.org/

[11] https://pytorch.org/

[12] https://stable-baselines3.readthedocs.io/en/master/index.html

[13] https://github.com/openai/baselines

[14] Deep Reinforcement Learning that Matters - https://arxiv.org/pdf/1709.06560.pdf

[15] https://github.com/Kaggle/kaggle-environments

odpowiednio decydują o stanie startowym gry oraz mechanice przejścia pomiędzy stanami. Środowisko nie umożliwia grania agenta ze samym sobą[16].

W celu lepszej wizualizacji procesu uczenia dodane zostały funkcje *callback*, które są wyzwalane w środowisku przy określonych etapach uczenia. Pierwsza z nich *ProgressBarCallback* dodaje pasek ładowania, który wskazuje ile jeszcze potrwa uczenie. Druga funkcja – *EvalCallback* sprawdza uczonego agenta co określoną ilość kroków i zapisuje najlepszy model, który został uzyskany w procesie uczenia. Procesy uczenia *DRL* są często niestabilne, w związku z czym warto zapisywać modele, które uzyskały dobre rezultaty[17]. Ponadto, po zakończeniu procesu uczenia wyświetlony jest wykres, który przedstawia jak zmieniały się otrzymywane przez agenta nagrody[18].

Prawdopodobnie najważniejszą częścią programu jest algorytm *RL* i jego głęboka sieć neuronowa. W ostatecznej wersji programu użyty został algorytm *Proximal Policy Optimization* (*PPO*). Jest to algorytm z 2017 roku, który został utworzony przez firmę *OpenAI*. Opiera się on na zmienianiu strategii[19] w odniesieniu do estymowanej nagrody poprzez zastosowanie gradientu (znane szerzej, jako *policy gradients*)[20]. Standardowe metody *policy gradient* są bardzo wrażliwe na zmiany parametru ich działania takich jak np. szybkość uczenia czy długość kroku gradientu. Jest to o tyle problematyczne, że to w jaki sposób zmieniany strategie wybierania kroków agenta wpływa również na uzyskiwane obserwacje. Obserwacje sprawiają, że używamy danej strategii, a dana strategia sprawia, że otrzymujemy takie, a nie inne obserwacje. Prowadzi to do dodatkowej niestabilności.

Algorytm *PPO* naprawia to, poprzez wprowadzenie dodatkowych zabezpieczeń, które sprawiają, że zmiana gradientu nie będzie zbyt duża. Algorytm *PPO* używa głównie następującej funkcji do obliczenia spadku gradientu:

[16] Wymagałoby to znacznej modyfikacji środowiska i algorytmów
[17] Deep Reinforcement Learning that Matters - https://arxiv.org/pdf/1709.06560.pdf
[18] Średnie nagrody w danym okresie
[19] Sposobu reakcji w stosunku do zaobserwowanego stanu środowiska
[20] Proximal Policy Optimization Algorithms - https://arxiv.org/pdf/1707.06347.pdf

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)]$$

θ – sparametryzowana strategia

t – *timestep* (czas)

$\hat{E}_t$ – spodziewany rezultat danej strategii

r_t – stosunek prawdopodobieństwa nowej i starej strategii[21]

$\hat{A}_t$ – estymowany zysk z danej strategii[22]

ε – hiperparametr zadany algorytmowi, który wpływa na szybkość zmiany strategii

Pozwala to na zmianę strategii w taki sposób, że jeżeli dana strategia okaże się dużo lepsza od obecnej[23], to zmiany w sieci neuronowej będą ograniczone. W podobnej sytuacji, jeżeli nowa strategia okaże się dużo gorsza, to nie zostanie ona całkowicie zignorowana w przyszłych etapach uczenia. Ostateczny algorytm używa jeszcze dwie dodatkowe funkcje i wygląda następująco:

$$L^{CLIP+VF+S}(\theta) = \hat{E}_t[L^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

$L_t^{VF}(\theta)$ – funkcja błędu wynikająca z używania tych samych sieci neuronowych do wybierania strategii i obliczania wartości stanu s_t

$S[\pi_\theta](s_t)$ – funkcja dodająca entropii do procesu eksploracji stanów problemu *MDP*[24]

c_1, c_2 – hiperparametr zadany algorytmowi

Jednakże największe znaczenie dla algorytmu ma zdecydowanie pierwsza część tego równania. Pozostałe elementy będą miały coraz mniejsze znaczenie w miarę postępu procesu uczenia.

[21] Czy nowa strategia daje lepsze rezultaty od poprzedniej

[22] Różnica pomiędzy rzeczywistą wartością obserwacji, a jej estymacją przez algorytm

[23] Na podstawie $\hat{A}_t$

[24] Jeżeli dany stan nie był sprawdzony, to funkcja ta zwiększa estymowaną wartość strategii, która go sprawdzi

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Źródło: *Proximal Policy Optimization Algorithms* - https://arxiv.org/pdf/1707.06347.pdf

Rys. 2 Pseudokod przedstawiający uproszczone działanie algorytmu *PPO*

Kolejnym ważnym elementem programu jest głęboka sieć neuronowa. Użyta w *Stable-baselines3* implementacja algorytmu *PPO* polega na szerzej znanej klasie algorytmów – *ActorCritic*. Posiadają one wynikające z nazwy elementy, to znaczy sieci neuronowe odpowiedzialne za wybór akcji (aktora) i estymację wartości stanu (krytyka). W ramach programu dodana została do niej dodatkowo sieć, której zadaniem jest znalezienie bardziej wartościowych elementów obserwacji[25].

Sieć ta na wejściu wykonuje operacje splotu (*convolution*) w ramach jądra o wielkości *4 x 4*, co oznacza, że przechodzi ona po całej obserwacji stosując filtry z różnymi macierzami o wielkości *4 x 4*[26]. Filtry takiej wielkości były zastosowane ze względu na to, że celem gry jest ustawienie 4 krążków w linii. Dodatkowo zastosowana została normalizacja uzyskanych wyników, która powinna przyśpieszyć szybkość i stabilność procesu uczenia[27].

Następnie rezultat operacji splotu jest transformowany do 1 wymiaru i przekazywany do sieci w pełni połączonej typu *MLP* (*multilayer perceptron* – perceptron wielowarstwowy). Dzieli się ona tutaj na wspomnianego wcześniej aktora i krytyka.

Wszystkie warstwy i części sieci neuronowej są aktywowane za pomocą tej samej funkcji *ReLu* (*Rectified Linear Unit* – prostownik funkcji liniowej). Jest

[25] Macierzy przedstawiającej symboliczną reprezentację planszy gry
[26] https://pytorch.org/docs/1.9.1/generated/torch.nn.Conv2d.html
[27] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift - https://arxiv.org/pdf/1502.03167.pdf

to funkcja, która dla wartości ujemnych zwraca wartość 0 a dla reszty wartość liniową. Użycie tej funkcji do aktywacji pozwala na wygaszanie neuronów, przy jednoczesnym zachowaniu liniowości, co teoretycznie powinno ułatwić uczenie. Dodatkowo jest ona prostsza w implementacji, ponieważ sprowadza się ona do wybrania wartości maksymalnej z 0 i funkcji liniowej[28].

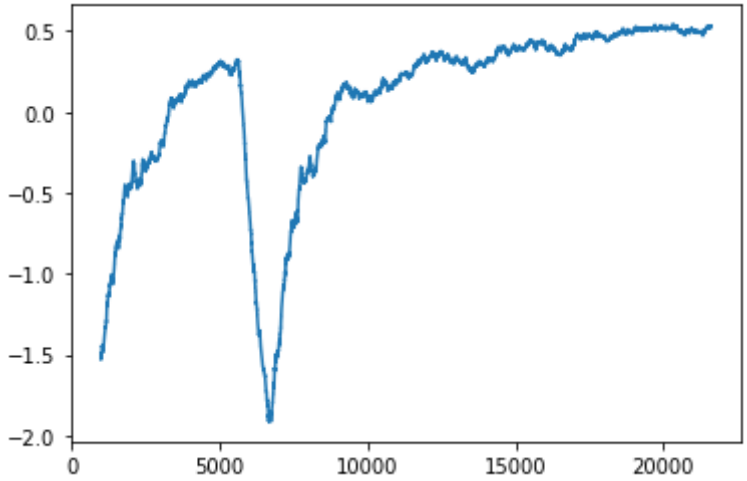
W programie został również zaimplementowany agent heurystyczny, oparty na metodzie *minimax*[29]. Agent ten wybiera ruch poprzez przeszukiwanie drzewa wszystkich możliwych stanów gry, stosując zasadę, że należy wybierać ruchy, które są najlepsze z najgorszych. Wynika to z tego, że zakładając, że przeciwnik gra optymalnie, to wybierze on najlepszy z możliwych ruchów – to znaczy najgorszy dla nas. Heurystyka oceniająca wartość konkretnych ruchów została dobrana arbitralnie. Głównym celem dodania tego agenta było sprawdzenie jak poradzi on sobie z agentem uczonym algorytmem *DRL*.

[28] Deep Sparse Rectifier Neural Networks - http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf
[29] https://en.wikipedia.org/wiki/Minimax

III. UZYSKANE REZULTATY

W ramach wykonania programu agent *DRL* uczył się przez 100 000 kroków, używając wyłącznie procesora dostępnego w środowisku *Kaggle*. To znaczy, że wykonał on 100 000 ruchów i zaobserwował tyle samo stanów planszy gry. Hiperparametry nie były zmieniane w odniesieniu do zdefiniowanych jako podstawowe przez bibliotekę *Stable-baselines3*[30]. W czasie treningu jego przeciwnik wykonywał losowe, prawidłowe ruchy. Przejście przez cały trening zajmuje około 15 minut.

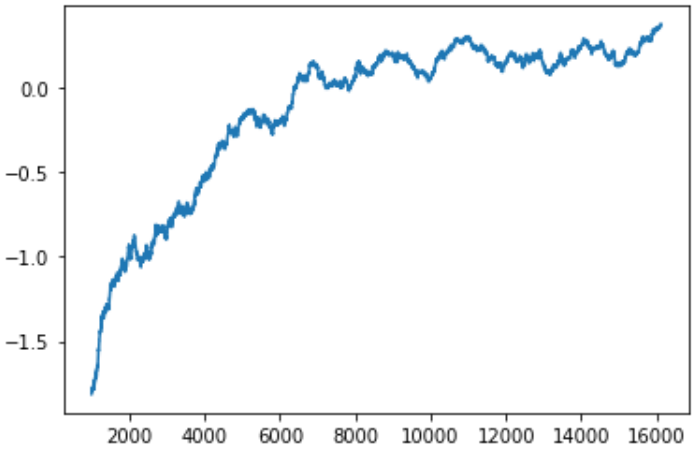
Na początku warto sprawdzić jak idzie trening podstawowej sieci neuronowej *MLP* zdefiniowanej przez bibliotekę. W czasie treningu jego przeciwnik wykonywał losowe, prawidłowe ruchy.



***Rys. 3* Wykres przedstawiający średnie wartości nagród uzyskiwanych przez agenta *PPO* z standardową siecią neuronową[31] podczas uczenia w okienkach po 1000 rozegranych gier (Y) w odniesieniu do ilości rozegranych gier (X)**

Jak można zaobserwować na rys. 3 agent nauczył się grać lepiej, ale trening zdecydowanie był niestabilny. W pewnym momencie, agent zaczął grać gorzej niż przed rozpoczęciem uczenia. Dlatego do projektu została dodana opisana wcześniej sieć neuronowa.

[30] https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html#stable_baselines3.ppo.PPO
[31] Zdefiniowaną w bibliotece



Źródło: Opracowanie własne

Rys. 4 Wykres przedstawiający średnie wartości nagród uzyskiwanych przez agenta *PPO* z siecią neuronową zdefiniowaną przez projekt podczas uczenia w okienkach po 1000 rozegranych gier (Y) w odniesieniu do ilości rozegranych gier (X)

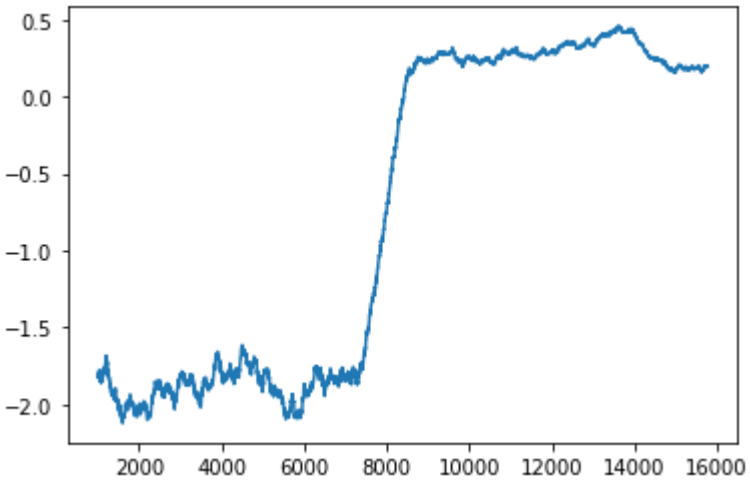
Jak można zauważyć na rys. 4, stabilność treningu znacznie wzrosła, w związku z czym w dalszej części projektu używana była własna sieć neuronowa.

W czasie treningu agent otrzymywał następujące nagrody: za nieprawidłowy ruch -10 i uznanie rozgrywki za skończoną, za przegraną -1, za remis 0 i za wygraną 1. Widać w związku z tym, że agent na początku wykonywał ruchy niedozwolone, ale z czasem nauczył się ich unikać i wygrywać. Ostatecznie najlepszy model podczas tego treningu uzyskał średnią nagrodę 0.92 +/- 0.39[32].

W ramach oceny agenta, po zakończeniu uczenia rozegrał on 100 gier z agentem całkowicie losowym i zaimplementowanym algorytmem heurystycznym. W ramach tych rozgrywek nauczony agent wygrał 75% gier przeciwko agentowi losowemu oraz 23% przeciw agentowi heurystycznemu. Wykonał on również w sumie 10 ruchów nieprawidłowych. Przed uczeniem było to odpowiednio 38%, 20% i w sumie 17 ruchów nieprawidłowych.

W ramach porównania sprawdzony został również dobrze znany algorytm *DQN* (*Deep Q-Network)*. Sieć neuronowa pozostała ta sama, a hiperparametry również nie były zmieniane[33]. Przy uczeniu przez tą samą ilość kroków, algorytm ten osiągnął porównywalne rezultaty.

[32] Wyliczone na podstawie 100 gier z agentem losowym, które nie były objęte procesem uczenia
[33] https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html#stable_baselines3.dqn.DQN



Źródło: Opracowanie własne

Rys. 5 Wykres przedstawiający średnie wartości nagród uzyskiwanych przez agenta *DQN* podczas uczenia w okienkach po 1000 rozegranych gier (Y) w odniesieniu do ilości rozegranych gier (X)

DQN opiera swoje uczenie na podstawie zebranych obserwacji podczas wcześniejszych gier. Stanowi to inne podejście niż w przypadku *PPO*, który uczy się w czasie gry[34]. Algorytm zaczął się uczyć po 50 000 ruchach i najlepszy model osiągną średnią nagrodę 0.68 +/- 0.73. Porównywalne były również wyniki 100 rozgrywek, w których wygrał on 73% gier przeciwko agentowi losowemu. *DQN* wygrał jednak znacznie więcej gier przeciwko algorytmowi heurystycznemu - 49%. Wykonał on również znacznie mniej nieprawidłowych ruchów. Należy pamiętać, że zarówno rozgrywki jak i samo uczenie podlega znacznej losowości, co wpływa na wyniki.

[34] Możliwa jest jednak również implementacja ucząca się z buforu, podobnie jak algorytm *DQN*

IV. ANALIZA WYNIKÓW

W ramach projektu przedstawiono agenta uczonego metodami *DRL*. Pokazane rezultaty wskazują, że trenowany agent *PPO* (oraz *DQN*) nauczył się grać lepiej. Niestety jednak daleko mu do ideału.

W kontekście samej gry *„Connect 4”* prostszym i skuteczniejszym rozwiązaniem jest agent heurystyczny. Ten, który został zaimplementowany do testów nie był w zoptymalizowany i analizował drzewo stanów gry jedynie 3 kroki w przód, ale i to wystarczyło częściej wygrywać niż agent *DRL*.

Z implementacji agenta heurystycznego można jednak wyciągnąć wnioski, które pozwalałyby na stworzenie znacznie lepszego agenta *DRL*. Istnieją algorytmy *DRL*, które również wykorzystują przeszukiwanie stanów gry. Jednym z nich jest np. *MuZero*, który opanował grę w *Go* na poziomie pozwalającym pokonać najlepszych ludzkich graczy. Ponadto, dokonał to ucząc się samemu zasad gry. Wykorzystał on do tego między innymi algorytm *Monte-Carlo Tree Search*[35].

Jednym z niewątpliwych problemów w zastosowaniu np. algorytmu *MuZero* jest wymagana moc obliczeniowa. Do osiągnięcia wspomnianego wcześniej rezultatu *MuZero* uczył się przez 1 000 000 kroków, używając 1000 jednostek *TPU*[36] (*Tensor Processing Unit),* które są znacznie wydajniejsze od użytego w projekcie procesora. Posiadał on również znacznie bardziej złożoną sieć neuronową. Podczas tworzenia projektu zaobserwowano również, że przyspieszenie procesu uczenia poprzez jego zrównoleglenie nie jest proste, ponieważ największym problemem jest uzyskiwanie obserwacji z symulacji i związane z tym przesyłanie danych[37]. Istnieją prace nad polepszeniem wydajności algorytmów *DRL* i ich wyniki są obiecujące[38].

Powinny być jednak prostsze rozwiązania, które mogą znaleźć zastosowanie konkretnie w grze *„Connect 4”*. Wspomniany wcześniej algorytm *DQN* uczył się i oceniał wyłącznie obecną obserwację – bez przeszukiwania przyszłych stanów gry. Mógłby on jednak zostać połączony z metodą użytą w agencie heurystycznym

[35] Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model - https://arxiv.org/pdf/1911.08265.pdf
[36] https://cloud.google.com/tpu
[37] Np. pomiędzy GPU a CPU
[38] Mastering Atari Games with Limited Data - https://arxiv.org/pdf/2111.00210.pdf

i oceniać obserwację stosując technikę *minimax*[39]. Jest to również rozwiązanie problemu związanego z uczeniem się tylko na podstawie agenta losowego. Uczony w ten sposób agent może być co najwyżej lepszy od losowego i nie osiągnie on ogólnie wysokiego poziomu. Uczenie agentów poprzez trenowanie z samym sobą sprawia, że proces ten staje się bardziej niestabilny[40], ale to również jest rozwiązywane przez metodę *minimax*. Ponadto, można wykorzystać symetryczność planszy[41], do podwojenia ilości obserwacji.

Rezultat projektu jest zadowalający, ale na podstawie przedstawionej analizy otrzymanych rezultatów oraz innych prac naukowych można jednoznacznie ocenić, że istnieją znaczne możliwości poprawy.

[39] A Theoretical Analysis of Deep Q-Learning - https://arxiv.org/pdf/1901.00137.pdf
[40] Emergent Complexity via Multi-Agent Competition - https://arxiv.org/pdf/1710.03748.pdf
[41] Lustrzane odbicie planszy sprowadza się do tego samego problemu w kontekście następnego ruchu