Implementacja algorytmu uczenia ze wzmocnieniem dla wybranej gry strategicznej

Thesis · February 2024

DOI: 10.13140/RG.2.2.25938.32969

1 author:

Jakub Kubiak
Military University of Technology
2 PUBLICATIONS **0** CITATIONS

SEE PROFILE

WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

PRACA DYPLOMOWA

STUDIA I^{o}

Temat pracy: **IMPLEMENTACJA ALGORYTMU UCZENIA ZE WZMOCNIENIEM DLA WYBRANEJ GRY STRATEGICZNEJ**

INFORMATYKA

(kierunek studiów)

INŻYNIERIA SYSTEMÓW

(specjalność)

Dyplomant:

Jakub Kubiak

Promotor:

dr inż. Marcin Mazurek

Warszawa 2024

OŚWIADCZENIE

Wyrażam zgodę / ~~nie wyrażam zgody~~ *
na udostępnianie mojej pracy przez Archiwum WAT

Dnia ....

(podpis)

* *Niepotrzebne skreślić*

Spis treści

Wstęp

Praca inżynierska zatytułowana "Implementacja algorytmu uczenia ze wzmocnieniem dla wybranej gry strategicznej"zgłębia ideę zastosowania sztucznej inteligencji w strategicznych grach, w szczególności przez pryzmat gry planszowej Blokus. Badania te wynikają z fascynacji strategiczną głębią i złożonością gier planszowych oraz intrygującym potencjałem sztucznej inteligencji do opanowania takich domen.

Jako entuzjasta szachów, moją inspiracją do tego przedsięwzięcia były przełomowe osiągnięcia AlphaZero w szachach. Zdolność AlphaZero do samodoskonalenia się i osiągania nadludzkich wyników, zaczynając od zera, pokazała ogromne możliwości sztucznej inteligencji w strategicznym myśleniu i podejmowaniu decyzji. To nie tylko zrewolucjonizowało podejście do sztucznej inteligencji w grach, ale także otworzyło pole do zastosowania tych technik w innych złożonych, strategicznych środowiskach. Blokus, ze swoją prostotą zasad, a zarazem wieloaspektową głębią strategiczną, stanowi idealne wyzwanie do zbadania tych możliwości.

Praca zaczyna się od przybliżenia Blokusa, opisując mechanikę i strategiczne wymiary gry. Ta sekcja służy do osadzenia w kontekście kolejnych eksperymentów, przedstawiając Blokusa nie tylko jako grę, ale także jako złożone środowisko decyzyjne.

Wstęp teoretyczny, kładzie podwaliny pod lepsze zrozumienie podstawowych metodologii zastosowanych w tej pracy. Wyjaśnia on zasady Proksymalnej Optymalizacji Polityki (PPO) i AlphaZero, dwóch kluczowych algorytmów w nowoczesnym uczeniu ze wzmocnieniem. Sekcja ta ma za zadanie przedstawić założenia stojące za tymi algorytmami, prezentując ich zasadnicze mechanizmy działania, potencjalne zastosowania oraz dokonuje analizy ich atutów i ograniczeń w kontekście ich praktycznego użycia.

Rdzeniem pracy jest trzeci rozdział, "Implementacja", w której przedstawiono kompleksowy opis przeprowadzonych eksperymentów. Sekcja ta szczegółowo opisuje zastosowanie PPO i AlphaZero w różnych konfiguracjach Blokusa, od uproszczonej planszy $7x7$ do pełnowymiarowego układu $20x20$. Eksperymenty są nie tylko technicznym sprawozdaniem z przeprowadzonych prac, ale także potwierdzeniem zdolności adaptacyjnych i skalowalności sztucznej inteligencji w strategicznych ustawieniach.

Podsumowanie syntetyzuje spostrzeżenia zebrane z tych eksperymentów, zastanawiając się nad osiągnięciami i wyciągając szersze wnioski na temat potencjału sztucznej inteligencji w strategicznej rozgrywce. Co więcej, otwiera ono drogę do przyszłych badań, sugerując, w jaki sposób wyniki tej pracy można ekstrapolować na inne dziedziny, co w konsekwencji podkreśla wszechstronność i potencjał transformacyjny sztucznej inteligencji w różnych złożonych systemach.

Użyteczność tych badań wykracza poza akademickie ćwiczenie polegające na zastosowaniu uczenia ze wzmocnieniem do Blokusa. Opracowane metodologie i uzyskane spostrzeżenia mają praktyczne implikacje w obszarach, w których strategiczne podej-

mowanie decyzji, rozpoznawanie wzorców i adaptacyjne uczenie się mają kluczowe znaczenie. Na przykład algorytmy i techniki udoskonalone w ramach tej pracy mogą być stosowane do optymalizacji procesów decyzyjnych w logistyce, gdzie ważne jest dynamiczne formułowanie strategii. Ponadto, zbadane zasady mają potencjalne zastosowanie w opracowywaniu metod opartych na sztucznej inteligencji do rozwiązywania złożonych problemów w sektorach takich jak finanse, opieka zdrowotna i planowanie urbanistyczne. Niniejsza praca stanowi zatem wkład nie tylko w dziedzinę zastosowania sztucznej inteligencji w grach, ale także w szerszy zakres dyscyplin, w których zastosowanie mają zasady analizy strategicznej i podejmowania decyzji w warunkach niepewności. Zdolność adaptacji zastosowanych podejść do różnych przestrzeni problemowych podkreśla wszechstronność sztucznej inteligencji i otwiera drogę do innowacyjnych zastosowań w różnych scenariuszach świata rzeczywistego.

Rozdział I. Opis problemu

Celem tej pracy było zaimplementowanie i wytrenowanie agenta do gry planszowej blokus[1] stosując uczenie ze wzmocnieniem.

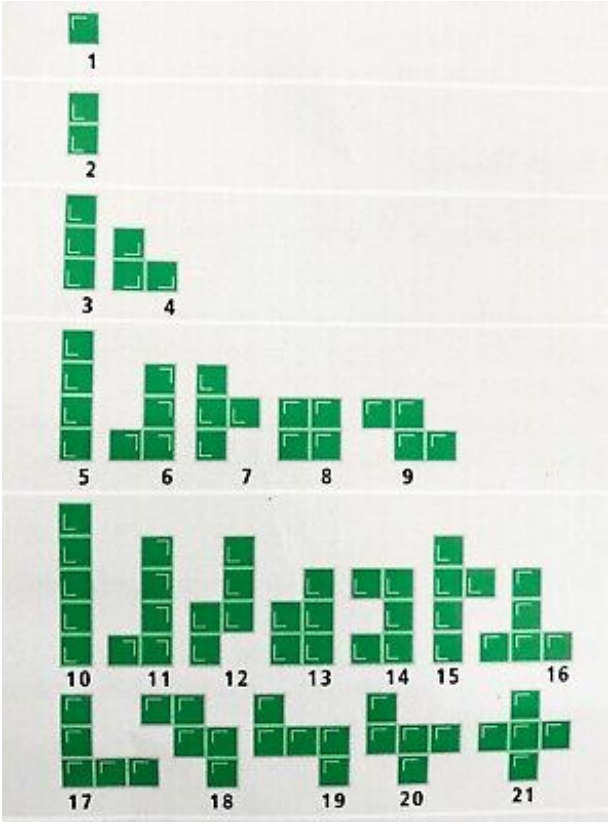
Rys. 1. Plansza Blokus

Źródło: https://www.krainazabawek24.pl/media/products/19a5b0a2c6b43ca74f47883a26853227/images/thumbnail/large_krainazabawek24-pl-mattel-gra-STRATEGICZNA-blokus-bjv44-ALT7.jpg?lm=1644423921

I.1. Opis gry

Blokus podobnie jak szachy czy Go jest strategiczną grą planszową z stosunkowo prostymi zasadami, które jednak mogą prowadzić do bardzo skomplikowanych strategii i dają niezliczoną ilość możliwości rozgrywki. Blokus kładzie nacisk na myślenie przestrzenne i strategiczne planowanie swoich akcji. Gra w klasycznej wersji przeznaczona jest dla 4 graczy, a rozgrywka toczy się na planszy o rozmiarze $20x20$. Każdy z graczy ma do dyspozycji 21 elementów, gdzie każdy zestaw różni się od siebie kolorem i zawiera różne kształty elementów składające się z od 1 do 5 kwadratów. Celem rozgrywki jest wyłożenie jak największej liczby klocków na planszę.

[1] https://scheherazade.znadplanszy.pl/2018/03/31/blokus/)



Rys. 2. Dostępne elementy w grze Blokus

Źródło: `https://i.ebayimg.com/images/g/cj8AAOSwdXhekQcT/s-1400.jpg`

I.2. Rozgrywka

Na początku gracze wybierają swój kolor i otrzymują zestaw 21 elementów tego koloru. Rozgrywka toczy się w turach, w których gracz może dołożyć jeden klocek do planszy. Pierwszy pionek zagrany przez każdego gracza musi dotykać rogu planszy przy którym ten gracz siedzi. Następne elementy wykładane są przez graczy w taki sposób aby stykały się rogami z co najmniej jednym pionkiem tego samego koloru. Kontakt krawędziami jest niedozwolony, nie ma jednak żadnego ograniczenia w kontakcie z pionkami przeciwnika.

Gra kończy się kiedy żaden z graczy nie może umieścić więcej elementów na planszy. Następnie gracze zliczają liczbę pól (małych kwadratów), które pozostały "na ręku", gracz z najmniejszą ilością punktów wygrywa.

I.3. Blokus a sztuczna inteligencja

Blokus jest szczególnie ceniony za swoją prostotę, a zarazem głęboką strategię. Jego abstrakcyjna natura i poleganie na rozumowaniu przestrzennym sprawiają, że jest to wymagający sprawdzian dla sztucznej inteligencji.

Rozdział II. Przegląd algorytmów

W tym rozdziale skupię się na zagadnieniach teoretycznych związanych z uczeniem ze wzmocnieniem. W szczególności omówię algorytmy Proksymalnej Optymalizacji Polityki (PPO) i AlphaZero, które wykorzystałem w swoich eksperymentach.

II.1. Proksymalna Optymalizacja Polityki

Algorytm Proksymalnej Optymalizacji Polityki[22], opracowany przez zespół badaczy z OpenAI[2] stanowi znaczący postęp w sferze uczenia ze wzmocnieniem (RL), w szczególności w dziedzinie metod gradientu polityki[27]. PPO charakteryzuje się prostym, a zarazem bardzo skutecznym podejściem do aktualizacji polityki, które stawia nacisk na stabilność procesu uczenia przez metodę "małych kroków".

W PPO można wyróżnić trzy kluczowe aspekty: stabilność treningu, efektywne wykorzystanie zebranych danych oraz łatwość implementacji. Uzyskuje się to poprzez zastosowanie odpowiedniej funkcji celu. Funkcja ta została nazwana "odciętą funkcją celu", która tak jak wskazuje nazwa odcina krok aktualizacji do pewnego poziomu nie pozwalając na zbyt dużą zmianę wag polityki w stosunku do poprzedniego stanu. Prostota tej metody na optymalizację polityki sprawia, że PPO jest szczególnie atrakcyjny dla praktycznych zastosowań, ponieważ zapewnia bardziej stabilne i niezawodne uczenie się agentów RL[22].

II.1.1. Opis działania

PPO aktualizuje swoją politykę, stosując się do odciętej funkcji celu. W pętli treningowej PPO można wyróżnić 5 istotnych etapów:

- **Inicjalizacja Parametrów**: Proces zaczyna się od inicjalizacji parametrów polityki θ oraz parametrów funkcji wartości ϕ.
- **Zbieranie danych**: W każdej iteracji agent zbiera informacje przez interakcje ze środowiskiem. Agent na podstawie stanu środowiska wykonuje akcje i w postaci informacji zwrotnej otrzymuje kolejny stan oraz nagrodę[3]. Ilość tych epizodów jest określona przez liczbę kroków czasowych T, a następnie obliczane są szacowane korzyści $A_1, A_2, \ldots, A_T$
- **Optymalizacja Funkcji Zastępczej**: Po zebraniu odpowiedniej ilości danych ze środowiska następuje optymalizacja funkcji zastępczej L względem parametrów polityki θ. Proces ten jest realizowany przez określoną liczbę epok K przy użyciu partii danych ze środowiska określonych rozmiarem M.
- **Aktualizacja Parametrów**: W tym kroku następuje optymalizacja właściwych parametrów polityki θ. Wykonywane jest to przez maksymalizację funkcji zastępczej L. W tym kroku wykorzystywana jest odcięta funkcja celu wraz z parametrem ε, który zapobiega zbyt drastycznym aktualizacją wag polityki.

[2] https://openai.com/

- **Dopasowanie Funkcji Wartości**: Ostatnim krokiem jest poprawienie funkcji wartości poprzez regresję na błędzie średnio kwadratowym:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

Poniżej pseudo kod z pętlą treningową w PPO:

Alg. 1. Pętla treningowa algorytmu Proksymalnej Optymalizacji Polityki

```
    Inicjalizacja parametrów polityki theta
    Inicjalizacja parametrów funkcji wartości phi
    dla każdej iteracji wykonaj
        dla każdego aktora wykonaj
            Uruchom politykę pi_theta przez T kroków
            Oblicz szacowane korzyści A_1, A_2, ..., A_T
        koniec dla
        Optymalizuj funkcję zastępczą L względem theta
        theta_stara <- theta
        dla każdej epoki wykonaj
            dla każdej partii w 1, 2, ..., NT/M wykonaj
                Zaktualizuj theta maksymalizując L
            koniec dla
        koniec dla
        Dopasuj funkcję wartości
    koniec dla
```

Źródło: Na podstawie: [22]

II.1.2. Odcięta funkcja straty

Ogólna funkcja celu optymalizacji polityki w uczeniu ze wzmocnieniem ma następującą postać:

$$L^{PG}(\theta) = \mathbb{E}_t\left[\log \pi_\theta(a_t|s_t) * A_t\right] \tag{2}$$

Gdzie:

- $\log \pi_\theta(a_t|s_t)$ - logarytm prawdopodobieństwo podjęcia akcji a_t w stanie s_t
- A_t - Korzyść jeśli $A_t > 0$, ta akcja jest lepsza niż inne możliwe akcje w tym stanie

PPO modyfikuje tą funkcję wprowadzając mechanizm odcięcia. Niech stosunek prawdopodobieństwa będzie $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$, gdzie π_θ to aktualna polityka, $\pi_{\theta_{old}}$ to stara polityka, a to akcja, a s to stan. Funkcja celu z ograniczeniem, $L^{CLIP}(\theta)$, jest dana przez:

$$\mathrm{L}^{\mathrm{CLIP}}(\theta) = \mathbb{E}_t\left[\min(r(\theta)A_t, \mathrm{clip}(r(\theta), 1-\varepsilon, 1+\varepsilon)A_t)\right] \tag{3}$$

Gdzie:

- A_t to funkcja korzyści w czasie t.
- $\text{clip}(r(\theta), 1-\varepsilon, 1+\varepsilon)$ ogranicza stosunek $r(\theta)$ do przedziału $[1-\varepsilon, 1+\varepsilon]$.
- ε to hiper-parametr, zazwyczaj mała wartość jak 0.1 lub 0.2.
- Oczekiwanie $\mathbb{E}_t$ jest obliczane na podstawie skończonej liczby próbek.

Celem $L^{CLIP}(\theta)$ jest minimalizowanie różnicy między nową a starą polityką, jednocześnie karząc za zmiany, które spowodowałyby, że stosunek prawdopodobieństwa wypadłby poza przedział określony przez $1 \pm \varepsilon$.

II.1.3. Zastosowanie

PPO przez swoją prostotę, elastyczność i zdolność do rozwiązywania złożonych i wielowymiarowych środowisk znalazło zastosowanie w wielu dziedzinach. Poniżej przedstawiam kilka zastosowań:

- **Gry typu Atari**: PPO okazał się bardzo skuteczny w teście wydajności algorytmów uczenia ze wzmocnieniem jakim jest zbiór gier Atari 2600. Osiągnął on lepsze wyniki od innych algorytmów opartych na gradiencie polityki takich jak: "Advantage Actor Critic"(A2C)[14] czy "Actor-Critic with Experience Replay"[31] wygrywając w 30 z 49 gier[22].

- **Dota 2**: Zastosowanie PPO obejmuje również złożone wieloosobowe gry bitewne online, w szczególności Dota 2[3]. W 2019 zespół naukowców z OpenAI stworzył OpenAI Five, zespół agentów AI, którzy byli w stanie rywalizować z profesjonalnymi ludzkimi graczami i nawet ich pokonać[15]. Dota 2 charakteryzuje się wielowymiarową przestrzenią stanów i akcji, które w połączaniu z koniecznością zaplanowania i zastosowania długoterminowej zaawansowanej strategii były wielkim wyznawaniem dla algorytmów uczenia ze wzmocnieniem, któremu PPO sprostało.

- **Dostrajanie dużych modeli językowych**: Kolejną dziedziną, w której PPO zostało użyte jest przetwarzanie języka naturalnego. Algorytm ten jest stosowany w procesie "uczenia ze wzmocnieniem z ludzką informacją zwrotna"(RLHF)[16], który polega na dostrojeniu dużego modelu językowego tak aby teksty generowane przez ten model były bardziej ludzko wyglądające oraz w celu zmniejszenia ryzyka na generowanie obraźliwych, szkodliwych czy toksycznych odpowiedzi przez ten model.

[3] `https://www.dota2.com/home`

II.1.4. Zalety i wady

Zalety PPO:

- **Stabilność treningu**: Dzięki zastosowaniu odciętej funkcji straty II.1.2 trening PPO unika drastycznych aktualizacji polityki, które mogły by doprowadzić do załamania wydajności.

- **Prostota implementacji**: W porównaniu do innych gradientowych polityk takich jak Optymalizacja Polityki Regionu Zaufania[23] czy Tęcza[9] PPO wyróżnia się względną prostotą w swoich założeniach i implementacji.

- **Wszechstronność i skuteczność**: Tak jak zostało przedstawione w zastosowaniach II.1.3 PPO może być z sukcesem zaaplikowane w wielu dziedzinach.

Wady PPO:

- **Wrażliwość na hiper-parametry** Wydajność algorytmu mocno uzależniona jest od precyzyjnej kalibracji hiper-parametrów, w szczególności progu odcięcia. Zły dobór hiper-parametrów może znacznie zmniejszyć osiągi agenta lub w skrajnych przypadkach algorytm nie będzie w stanie zbiec do minimum.

- **Wymagania obliczeniowe** Szczególnie w wielowymiarowych i skomplikowanych środowiskach PPO wymaga dużych mocy obliczeniowych składających się z wielu graficznych jednostek obliczeniowych.

- **Zbieżność do optimum**: PPO może nie być konsekwentne w dążeniu do globalnego minimum. Również przez odcinanie zbyt dużej aktualizacji wag polityki algorytm jest podatny na pozostanie w lokalnym minimum.

II.2. AlphaZero

AlphaZero jest algorytmem opracowanym przez zespół DeepMind[4] w 2017 roku. Jest on przełomem jeśli chodzi o zastosowanie uczenia ze wzmocnieniem w grach planszowych. Osiągnął nadludzki poziom w złożonych grach strategicznych takich jak szachy, shogi czy Go. AlhpaZero zaczyna naukę od "czystej tablicy"[5] i poprzez metodologię samouka, bez żadnej specyficznej wiedzy w danej dziedzinie poza zasadami iteracyjne doskonali swoje umiejętności[26]. AlphaZero wykorzystuje uogólnioną adaptację algorytmu Przeszukiwanie Drzewa Monte Carlo (MCTS)[6] zintegrowaną z głębokimi sieciami neuronowymi[20]. Wyróżniającą cechą AlphaZero na tle swoich poprzedników jest jego uniwersalność i możliwość adaptacji do dowolnej gry dwuosobowej z doskonałą informacją[6][25].

[4] https://deepmind.google/
[5] https://pl.wikipedia.org/wiki/Tabula_rasa
[6] https://en.wikipedia.org/wiki/Perfect_information

II.2.1. Opis działania

W procesie treningu AlphaZero można wyróżnić kilka kluczowych etapów:

- **Inicjalizacja sieci neuronowych**: Inicjalizacja wag dla sieci polityki π, która przewiduje kolejną akcję na podstawie obecnego stanu, oraz inicjalizacja funkcji wartości ϕ odpowiedzialnej za przewidywanie nagrody dla danej akcji.
- **Gra sam ze sobą**: W tym etapie AlphaZero rozgrywa ze sobą N epizodów samodzielnej gry używając algorytmu MCTS do eksploracji przestrzeni stanów.
- **Ocena i wybór ruchów**: W każdym ruchu algorytm ocenia możliwe ruchy na podstawie przeprowadzonych M symulacji MCTS oraz wykorzystuje sieć polityki do wyboru najbardziej obiecującego ruchu.
- **Aktualizacja sieci neuronowych**: Po serii gier następuje aktualizacja polityki i funkcji wartości, aby poprawić zarówno jakość ruchów, jak i ocenę stanów, na podstawie wyników gier.
- **Arena porównawcza** Nowa wersja modelu jest porównywana z obecnie najlepszą wersją agenta. Rozgrywane jest G gier, jeśli nowy model przekroczy próg 55% wygranych to zostaje zaakceptowany jako nowy najlepszy model.

Poniżej przedstawiono pseudokod dla pętli treningowej w AlphaZero:

Alg. 2. Pętla treningowa algorytmu AlphaZero

```
1     Inicjalizacja sieci neuronowej polityki i oceny stanów
2     dla każdego epizodu N wykonaj
3         dla każdego M wykonaj
4             Uruchomienie symulacji MCTS
5         koniec dla
6         Użycie danych z MCTS do aktualizacji sieci
7         Arena porównawcza
8         Akceptacja lub odrzucenie nowego modelu
9     koniec dla
```

Źródło: Na podstawie: [26], [25]

II.2.2. Przeszukiwanie Drzewa Monte Carlo

Monte Carlo Tree Search jest heurystycznym algorytmem stosowanym do przeszukiwania przestrzeni akcji i podejmowania decyzji. Połączanie tej metody z głębokimi sieciami neuronowymi w AlphaZero znacznie usprawniło proces wyboru akcji przez agenta w złożonych grach strategicznych. MCTS składa się z następujących kroków:

1. **Selekcja**: Proces selekcji rozpoczyna się od korzenia, algorytm przechodzi po drzewie gry[30] poprzez podrzędne węzły aż do węzła liścia. W celu zrównoważenia eksploracji i eksploatacji algorytm kieruję się górną granicą ufności (UCB)[2] stosowaną do drzew:

$$UCB = \bar{X}_j + C_p \sqrt{\frac{\ln t}{n_j}} \tag{4}$$

gdzie:

- $\bar{X}_j$ to średnia nagroda węzła
- n_j to liczba odwiedzin węzła j
- t to całkowita liczba symulacji
- C_p to parametr eksploracji

Dla AlphaZero jako że operuje na sieciach neuronowych polityki i funkcji wartości, a nie tylko na heurystycznym wyborze akcji, równanie UCB nr 4 zostało przekształcone do następującej formy:

$$a_t = \arg\max_a (Q(s,a) + u(s,a)) \tag{5}$$

gdzie:

- a_t to akcja wybrana w kroku czasowym t.
- $Q(s,a)$ to szacowana wartość podjęcia akcji a w stanie s.
- $u(s,a)$ to termin eksploracyjny dla akcji a w stanie s.

$$u(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{6}$$

gdzie:

- c_{puct} to stała określająca poziom eksploracji.
- $P(s,a)$ to wcześniejsze prawdopodobieństwo akcji a w stanie s dane przez sieć polityki.
- $N(s,b)$ to liczba wizyt dla wszystkich akcji b ze stanu s.
- $N(s,a)$ to liczba wizyt dla akcji a ze stanu s.

Wyjście sieci polityki i wartości w AlphaZero:

$$(P(s,\cdot), v) = f_\theta(s) \tag{7}$$

gdzie:

- $P(s,\cdot)$ to wektor prawdopodobieństw ruchów dla stanu s. Wektor ten jest wyjściem sieci polityki.
- v to szacowana wartość dla stanu s.

- $f_\theta(s)$ to funkcja reprezentowana przez sieć neuronową z parametrami θ, która na wyjściu zwraca zarówno prawdopodobieństwa jak i szacowaną wartość dla stanu s.

2. **Ekspansja**: W fazie ekspansji algorytm wybiera kolejne węzły z drzewa gry do dalszego rozwoju. W AlphaZero za węzeł rozumiana jest pozycja w grze, która została osiągnięta podczas fazy selekcji ale nie została jeszcze zbadana. Kolejny węzeł w tej fazie jest wybierany na podstawie poprzednich symulacji i sieci polityki, która na wyjściu przewiduje rozkład prawdopodobieństwa dostępnych akcji. Po wyborze węzła, do drzewa dołączany jest jeden lub więcej węzłów podrzędnych, z których każdy reprezentuje ruch z danej pozycji. Proces ten przedstawia równanie 7.

3. **Symulacja**: Faza symulacji w MCTS zaczyna się od nowo rozwiniętych węzłów w fazie ekspansji. Przeprowadzana jest seria symulowanych pełnych gier, aż do stanu końcowego. Ruchy są wybierane na podstawie polityki agenta. Etap ten jest kluczowy dla oceny wartości węzła, ponieważ pozwala dostarczać konkretnych informacji o wynikach rozgrywek z danego punktu przestrzeni. Poniżej pseudokod procedury symulacji MCTS:

Alg. 3. Procedura symulacji w MCTS

```
1          funkcja Symulacja(węzeł)
2              dopóki nie KońcowyStan(węzeł.stan) do
3                  akcje = DostępneAkcje(węzeł.stan)
4                  akcja = WybierzAkcje(akcje, węzeł.polityka)
5                  węzeł = ZastosujAkcje(węzeł, akcja)
6              koniec dopóki
7              zwrot Wynik(węzeł.stan)
8          koniec funkcja
9
```

Źródło: Na podstawie: [26]

gdzie:

- `KońcowyStan` sprawdza, czy stan gry w węźle jest stanem końcowym.
- `DostępneAkcje` zwraca dostępne akcje dla danego stanu.
- `WybierzAkcje` wybiera akcję do wykonania, zgodnie z zasadami polityki symulacji.
- `ZastosujAkcje` aktualizuje stan węzła na podstawie wybranej akcji.
- `Wynik` ocenia wynik gry po osiągnięciu stanu końcowego.

4. **Propagacja wsteczna**: Po fazie symulacji następuje etap propagacji wyników z powrotem w górę drzewa gry. Wartości w kolejnych węzłach są aktualizowane nowymi oszacowaniami. Konkretnie, estymowana wartość Q w danym węźle jest aktualizowana na podstawie propagowanych wyników symulacji, a liczba wizyt węzła N zwiększa się. Matematycznie proces ten może być opisany następującymi regułami:

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)}(V(s') - Q(s,a)) \tag{8}$$

$$N(s,a) \leftarrow N(s,a) + 1 \tag{9}$$

gdzie:

- $Q(s,a)$ to aktualne oszacowanie wartości podjęcia akcji a w stanie s
- $N(s,a)$ ile razy akcja a została podjęta ze stanu s
- $V(s')$ to wartość uzyskana z symulacji rozpoczynającej się w stanie s i podejmującej akcję a

II.2.3. Sieć neuronowa

Po zebraniu odpowiedniej ilości danych ze środowiska przez rozgrywanie epizodów samodzielnej gry następuję czas na aktualizacje wag sieci polityki i funkcji wartości. W tym celu wykorzystuje się metodę optymalizacji gradientów w procesie wstecznej propagacji[19]. Architektura sieci może znacznie różnić się między grami jednak w AlhpaZero zazwyczaj stosowane są sieci splotowe[13]. Sieci tego typu mają szerokie zastosowanie w wizji komputerowej i są zdolne do wyciągania cech i złożonych zależności ze struktur wielowymiarowych takich jak obrazy. Wejściem do modelu w AlhpaZero jest zazwyczaj macierz o wymiarach planszy z oznaczonym na niej rozmieszczeniem elementów. Wejście przechodzi przez stos warstw splotowych i na końcu znajdują się dwie głowy modelujące, odpowiednio dla polityki i funkcji wartości. Dla polityki jest to funkcja liniowa o rozmiarze wyjściowym przestrzeni akcji. Na podstawie logitów sieci polityki samplowana jest kolejna akcja z rozkładu prawdopodobieństwa dostępnych akcji. Głowa modelująca dla funkcji wartości jest również warstwą liniową, która przewiduje oczekiwaną wartość zwrotu z obecnego stanu. Na podstawie wyjść z modelu wyliczana jest funkcja straty:

- **Funkcja straty polityki** jest zazwyczaj definiowana jako negatywny logarytm prawdopodobieństwa:

$$\mathscr{L}_{\text{Polityka}} = -\sum_i \log(p(a_i|s_i)) \cdot \pi(a_i|s_i) \tag{10}$$

  gdzie:

  – $p(a_i|s_i)$ to przewidywane przez sieć prawdopodobieństwo podjęcia działania a_i w stanie s_i.

  - $\pi(a_i|s_i)$ jest prawdopodobieństwem akcji a_i z MCTS używanego jako polityka docelowa.

- **Funkcja straty wartości** jest to zazwyczaj średni błąd kwadratowy między przewidywaną wartością, a rzeczywistym wynikiem

$$\mathscr{L}_{\text{Wartość}} = \frac{1}{N}\sum_i (v(s_i) - z_i)^2 \tag{11}$$

  gdzie:

  - $v(s_i)$ jest oszacowaniem wartości dla stanu s_i przewidywanym przez sieć.
  - z_i to rzeczywisty wynik gry ze stanu s_i (np. $+1$ dla wygranej, 0 dla remisu, -1 dla przegranej).
  - N to liczba punktów danych.

- **Całkowita funkcja straty** jest ważoną sumą strat polityki i wartości:

$$\mathscr{L} = \lambda \mathscr{L}_{\text{Polityka}} + (1-\lambda)\mathscr{L}_{\text{Wartość}} \tag{12}$$

  gdzie λ jest hiper-parametrem, który równoważy dwa składniki.

Na podstawie otrzymanej całkowitej wartości straty wyliczany jest gradient dla każdej kolejnej warstwy sieci i następnie przy pomocy jednej z metod optymalizacji np. Adam[12] wagi modelu są aktualizowane. Poniżej ogólne równanie aktualizacji wag modelu na podstawie metody zejścia gradientu:

$$\theta_{\text{nowy}} = \theta_{\text{dawny}} - \alpha \nabla_\theta \mathscr{L}(\theta) \tag{13}$$

gdzie:

- θ_{nowy} i θ_{dawny} odpowiednio nowe i stare wagi modelu
- α wskaźnik uczenia się, hiper-parametr, który kontroluje rozmiar kroku podczas procesu optymalizacji.
- $\nabla_\theta \mathscr{L}(\theta)$ to gradient całkowitej funkcji straty w odniesieniu do wag θ.

II.2.4. Zastosowanie

AlphaZero poza swoją uniwersalnością w tradycyjnych grach z doskonałą informacją znalazł zastosowanie w innych dziedzinach nauki takich jak dynamika kwantowa. Artykuł opublikowany w "npj Quantum Information"[7] bada użyteczności AlphaZero w zadaniach optymalizacji parametrów w dynamice kwantowej porównując go z innym algorytmem optymalizacji opartym na gradiencie (GRAPE). Wyniki pokazały, że AlhpaZero systematycznie przewyższał GRAPE w symulacjach wymagających wysokiej precyzji.

II.2.5. Zalety i wady

Zalety:

- **Uniwersalność**: Jest w stanie nauczyć się dowolnej gry dwuosobowej z doskonałą informacją[25]
- **Samodoskonalenie**: Nie wymaga żadnej wiedzy eksperckiej, opracowuje strategie jedynie na podstawie grania sam ze sobą
- **Osiągi**: W stosunkowo krótkim czasie osiąga nadludzki poziom w złożonych grach strategicznych

Wady:

- **Zasoby obliczeniowe**: Wymaga przeprowadzenia olbrzymiej ilości symulacji co wymaga wielu wyspecjalizowanych jednostek obliczeniowych, których ilość na jeden trening może iść w setki lub tysiące
- **Ograniczenia**: AlphaZero został zaprojektowanych głównie z myślą o grach turowych z dyskretną przestrzenią akcji co ogranicza jego bezpośrednie zastosowanie w innych typach problemów np. z ciągłą przestrzenią akcji

Rozdział III. Implementacja

III.1. Eksperymenty z PPO

Po zapoznaniu się z teorią uczenia ze wzmocnieniem i dostępnymi algorytmami postanowiłem zaimplementować jeden z nich. W pierwszej kolejności wybór padł na algorytm Proksymalnej Optymalizacji Polityki II.1. Algorytm ten dawał bardzo dobre wyniki w grach typu Atari, więc postanowiłem sprawdzić jak zachowa się w grze planszowej jaką jest Blokus. Dodatkowo implementacja tego algorytmu jest stosunkowo prosta w porównaniu do innych algorytmów opartych na polityce na przykład takich jak Tęcza[9]. Swoją implementację wzorowałem na otwarto źródłowej wersji z projektu "cleanrl"[10]. W repozytorium "cleanrl"[7] znajduję się implementacja wielu algorytmów uczenia ze wzmocnieniem bazujących zarówno na funkcji wartości jak i na polityce. Wyróżniającą cechą tego projektu są jedno plikowe implementację algorytmów, które ułatwiają ponowne ich użycie w innym miejscu oraz w pracach badawczych.

III.1.1. Gymnasium

Gymnasium[8] jest to API do języka Python z gotowymi środowiskami dla uczenia ze wzmocnieniem. Pakiet ten zapewnia jednolity Python'owy interfejs dla ogólnych problemów uczenia ze wzmocnieniem oraz zawiera szeroką kolekcję środowisk, takich jak CartPole, Lunar Lander czy gry Atari. Biblioteka ta posiada kilka kluczowych funkcji API takie jak:

- **make**: Inicjalizuje obiekt środowiska
- **reset**: Resetuje stan środowiska do pierwotnej formy
- **step**: Wykonuje akcję i zwraca artefakty o stanie środowiska
- **render**: Wizualizuje obecny stan środowiska

III.1.2. Architektura sieci neuronowej

Sieć neuronowa w ramach PPO służy do dwóch celów: działa jako sieć polityki, która decyduje o następnym ruchu, oraz jako sieć wartości, która ocenia jakość ruchu. W tym eksperymencie skorzystałem z dwóch architektur sieci: wielowarstwowego perceptronu (MLP)[18] i splotowej sieci neuronowej (CNN). Poniżej prezentuje szczegóły architektury modeli:

[7] https://github.com/vwxyzjn/cleanrl
[8] https://gymnasium.farama.org/

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
├─MLP: 1-1                               [-1, 919]                 --
|    └─Sequential: 2-1                   [-1, 919]                 --
|    |    └─Linear: 3-1                  [-1, 64]                  3,200
|    |    └─Dropout: 3-2                 [-1, 64]                  --
|    |    └─ReLU: 3-3                    [-1, 64]                  --
|    |    └─Linear: 3-4                  [-1, 128]                 8,320
|    |    └─Dropout: 3-5                 [-1, 128]                 --
|    |    └─ReLU: 3-6                    [-1, 128]                 --
|    |    └─Linear: 3-7                  [-1, 64]                  8,256
|    |    └─Dropout: 3-8                 [-1, 64]                  --
|    |    └─ReLU: 3-9                    [-1, 64]                  --
|    |    └─Linear: 3-10                 [-1, 919]                 59,735
==========================================================================
Total params: 79,511
Trainable params: 79,511
Non-trainable params: 0
Total mult-adds (M): 0.24
==========================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.30
Estimated Total Size (MB): 0.31
==========================================================================
```

Rys. 3. Architektura sieci wielowarstwowego perceptronu w PPO

Źródło: Badania własne

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─ConvBlock: 1-1                         [-1, 64, 7, 7]            --
|    └─Sequential: 2-1                   [-1, 64, 7, 7]            --
|    |    └─Conv2d: 3-1                  [-1, 64, 7, 7]            640
|    |    └─Dropout: 3-2                 [-1, 64, 7, 7]            --
|    |    └─ReLU: 3-3                    [-1, 64, 7, 7]            --
|    |    └─Conv2d: 3-4                  [-1, 64, 7, 7]            36,928
|    |    └─Dropout: 3-5                 [-1, 64, 7, 7]            --
|    |    └─ReLU: 3-6                    [-1, 64, 7, 7]            --
|    |    └─Conv2d: 3-7                  [-1, 64, 7, 7]            36,928
|    |    └─Dropout: 3-8                 [-1, 64, 7, 7]            --
|    |    └─ReLU: 3-9                    [-1, 64, 7, 7]            --
|    |    └─Conv2d: 3-10                 [-1, 64, 7, 7]            36,928
|    |    └─Dropout: 3-11                [-1, 64, 7, 7]            --
|    |    └─ReLU: 3-12                   [-1, 64, 7, 7]            --
├─MLP: 1-2                               [-1, 919]                 --
|    └─Sequential: 2-2                   [-1, 919]                 --
|    |    └─Linear: 3-13                 [-1, 64]                  200,768
|    |    └─Dropout: 3-14                [-1, 64]                  --
|    |    └─ReLU: 3-15                   [-1, 64]                  --
|    |    └─Linear: 3-16                 [-1, 128]                 8,320
|    |    └─Dropout: 3-17                [-1, 128]                 --
|    |    └─ReLU: 3-18                   [-1, 128]                 --
|    |    └─Linear: 3-19                 [-1, 64]                  8,256
|    |    └─Dropout: 3-20                [-1, 64]                  --
|    |    └─ReLU: 3-21                   [-1, 64]                  --
|    |    └─Linear: 3-22                 [-1, 919]                 59,735
==========================================================================================
Total params: 388,503
Trainable params: 388,503
Non-trainable params: 0
Total mult-adds (M): 6.50
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.10
Params size (MB): 1.48
Estimated Total Size (MB): 1.59
==========================================================================================
```

Rys. 4. Architektura sieci splotowej w PPO

Źródło: Badania własne

Model oparty na MLP (patrz rysunek 3) jest siecią sekwencyjną, która zaczyna się od w pełni połączonej warstwy liniowej przekształcającej dane wejściowe w 64 wymiarową przestrzeń. Następnie występuje warstwa porzucenia i po niej Poprawiona Funkcja Liniowa (ReLU) jako funkcja aktywacji. Proces ten jest powtarzany, zwiększając wymiarowość do 128, a następnie zmniejszając ją znowu do 64 wymiarów. Ostatnia transformacja liniowa przekształca 64 wymiarowy wektor do przestrzeni wyjściowej, która jest przestrzenią stanów o wielkości 919. Model ten ma 79511 parametrów.

Drugi model oparty na CNN (patrz rysunek 4) zaczyna się od bloku konwolucyjnego, który składa się z kilku warstw konwolucyjnych przeplatanych funkcjami aktywa-

cji porzucenia i ReLU. Warstwy konwolucyjne służą do wyciągnięcia cech przestrzennych z obserwacji, którą jest dwu-wymiarowa plansza z oznaczonymi na niej pozycjami pionków graczy. Następnie dane wyjściowe tego bloku są spłaszczane i przetwarzane przez wielowarstwowy perceptron, którym jest pierwszy model MLP. Model ten składa się z 388503 parametrów.

III.1.3. CartPole

Pierwszym krokiem było sprawdzenie czy algorytm działa poprawnie. Do tego celu wykorzystałem środowisko CartPole-v1[9]. Środowisko to polega na utrzymaniu równowagi drążka na wózku. Wózek może poruszać się w lewo lub prawo. Za każdym razem gdy drążek przekroczy kąt 15 stopni lub wózek wyjedzie poza obszar środowiska gra się kończy. Celem jest utrzymanie drążka w pionie jak najdłużej, maksymalną nagrodą w tym środowisku jest 500 co oznacza, że agent osiągnął najwyższy poziom dla tej gry. Poniżej specyfikacja parametrów tego środowiska:

Tab. 1. Specyfikacja środowiska CartPole-v1

Przestrzeń akcji	Dyskretna(2)
Kształt obserwacji	(4,)
Maksymalna wartość obserwacji	$[4.8, \infty, 0.42, \infty]$
Minimalna wartość obserwacji	$[-4.8, -\infty, -0.42, -\infty]$

Źródło: `https://www.gymlibrary.dev/environments/classic_control/cart_pole`

Tab. 2. Przestrzeń akcji CartPole-v1

Wartość	**Akcja**
0	Popchnij wózek w lewo
1	Popchnij wózek w prawo

Źródło: `https://www.gymlibrary.dev/environments/classic_control/cart_pole`

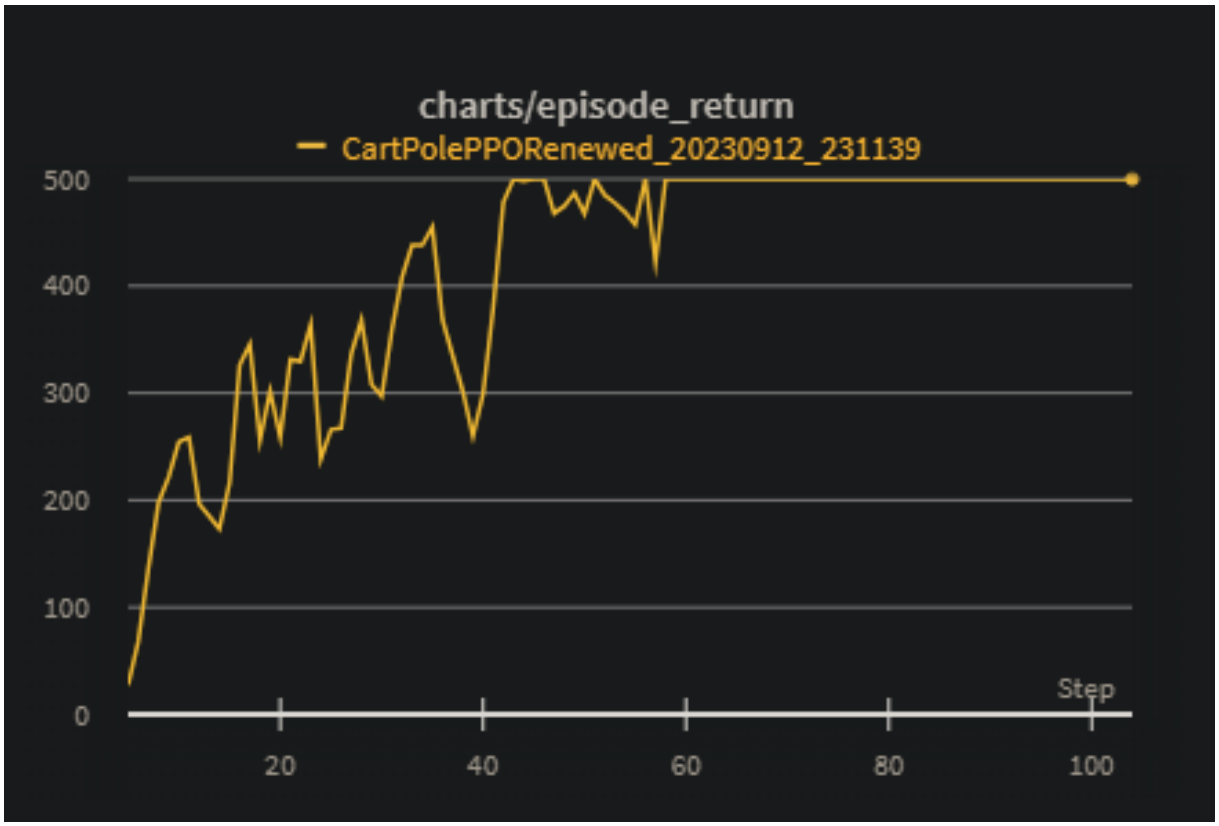
[9] `https://gymnasium.farama.org/environments/classic_control/cart_pole/`

Tab. 3. Przestrzeń obserwacji CartPole-v1

Wartość	Obserwacja	Min	Max
0	Pozycja wózka	-4.8	4.8
1	Prędkość wózka	$-\infty$	∞
2	Kąt biegunowy	$-0.418\,\text{rad}\,(-24°)$	$0.418\,\text{rad}\,(24°)$
3	Prędkość kątowa bieguna	$-\infty$	∞

Źródło: `https://www.gymlibrary.dev/environments/classic_control/cart_pole`

Środowisko to jest bardzo proste, więc algorytm powinien nauczyć się w nie grać w stosunkowo krótkim czasie. Otrzymane wyniki przedstawiam poniżej:

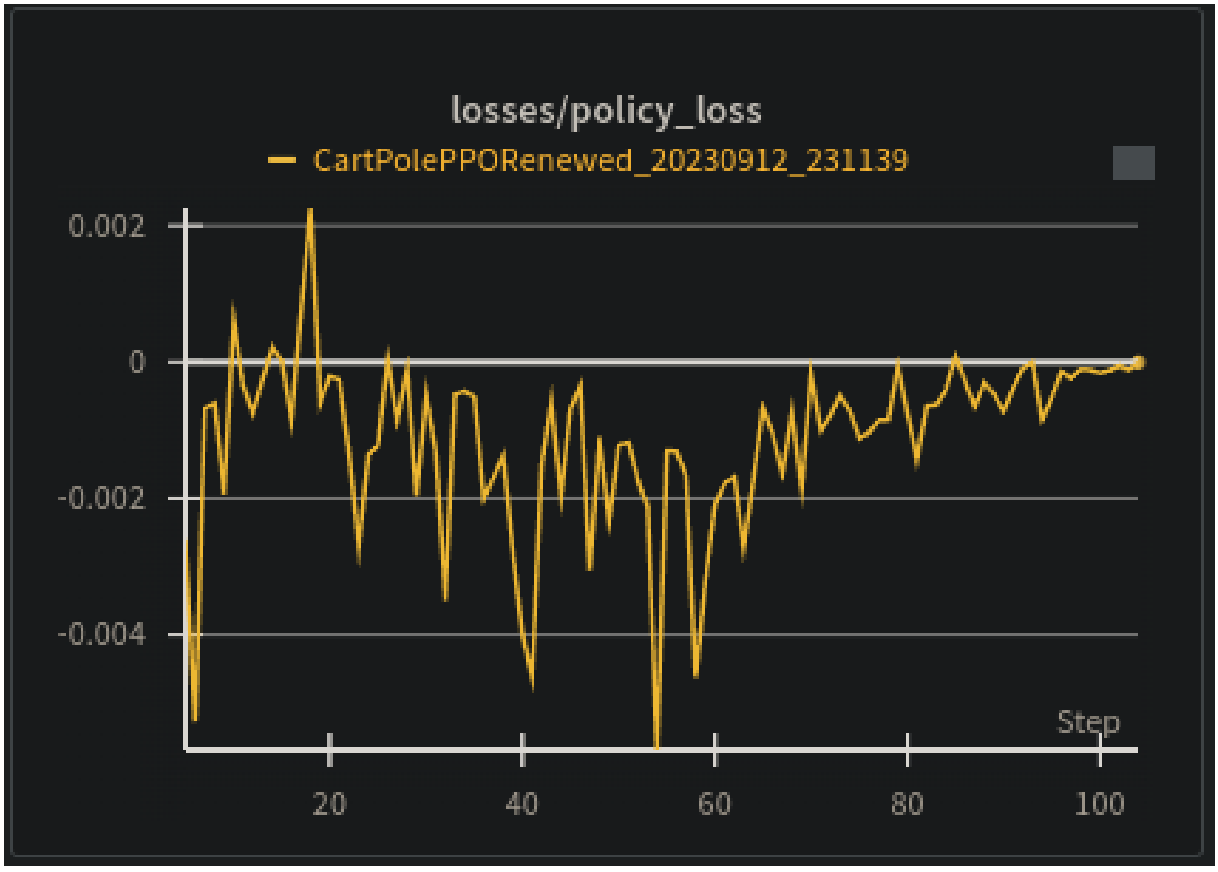


Rys. 5. Wykres nagród dla kolejnych epizodów w CartPole dla PPO

Źródło: Badania własne

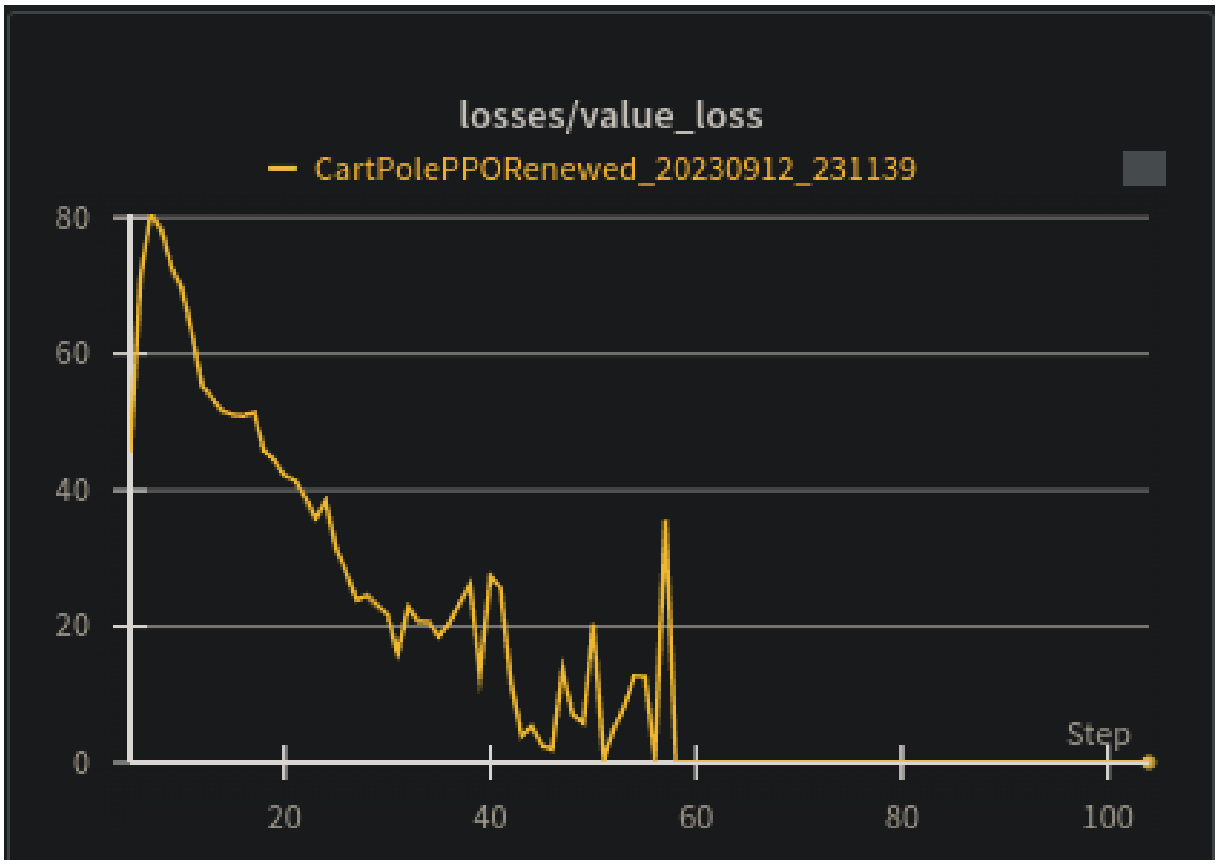
Wykres nr 5 przedstawia otrzymane nagrody w kolejnych epizodach treningowych. W tym środowisku nagroda jest równa długości epizodu, przez który drążek był utrzymywany w poziomie. Już w okolicach 60 epizodu algorytm zaczął osiągać optimum,

zdobywając w każdej kolejnej rozgrywce maksymalną ilość punktów, która wynosi 500 punktów.



Rys. 6. Wykres straty polityki kolejnych epizodów treningowych w CartPole dla PPO

Źródło: Badania własne



Rys. 7. Wykres straty funkcji wartości kolejnych epizodów treningowych w CartPole dla PPO

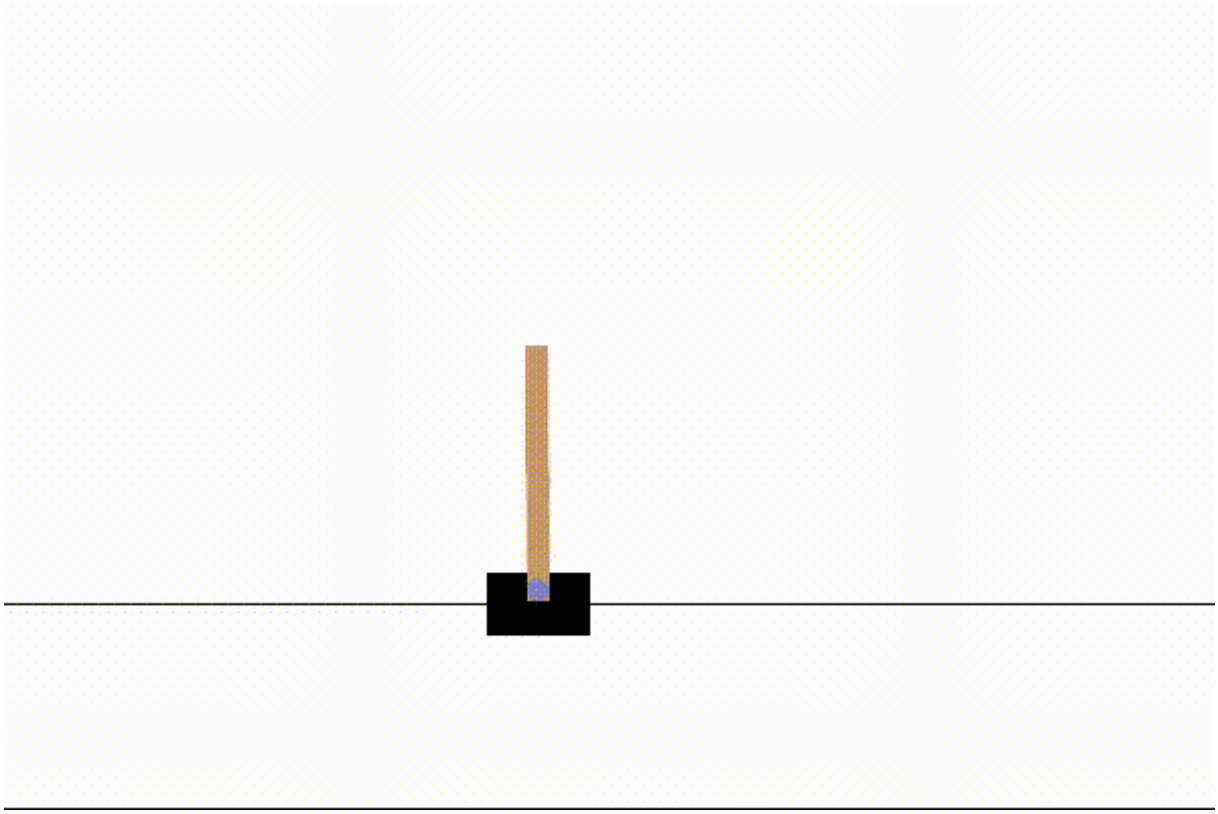
Źródło: Badania własne

Wykres straty polityki dla PPO pokazuje wahania wokół wąskiego zakresu wartości bliskich zeru. Może to wskazać, że model bada różne strategie i w tym samym czasie wprowadza ulepszenia do obecnej polityki. Pomimo braku stabilnego spadku funkcji starty na wykresie nr 6, konwergencja (patrz rysunek 5) wykazuje, że agent nauczył się stabilnej polityki, która działa dobrze w środowisku CartPole.

Wykres funkcji straty (patrz rysunek nr 7) wykazuje bardziej przewidywalne zachowanie. Przez większość czasu miał spadkową tendencje w efekcie, już w okolicach 60 kroku przewidywał niemal perfekcyjnie oczekiwaną wartość zwrotu z wykonanej akcji.

Jak przedstawiają wykresy (patrz rysunek nr 5, 6 5), algorytm stosunkowo szybko dostosował strategie do tej gry. Warto zauważyć, że w przypadku tego środowiska nie ma potrzeby wykorzystywania sieci splotowych, więc sieć neuronowa składa się tylko z warstw liniowych. W tym eksperymencie użyłem prostej architektury sieci MLP (patrz rysunek nr 3) jednak w przypadku środowiska Blokus konieczne będzie wykorzystanie większych i bardziej złożonych sieci splotowych.

Dodatkowo zamieszczam zrzut z rozgrywki w celu wizualizacji graficznego interfejsu gry:



Rys. 8. Środowisko CartPole

Źródło: Badania własne

III.1.4. Blokus $7x7$

Po upewnieniu się że algorytm działa poprawnie w środowisku CartPole przyszedł czas na sprawdzenie jak zachowa się w środowisku Blokus. Środowisko to jest dużo bardziej skomplikowane niż CartPole, więc algorytm będzie potrzebował więcej czasu na znalezienie odpowiedniej strategi. Dodatkową trudnością jest to ze Blokus jako gra planszowa ma zmienną liczbę akcji w zależności od stanu gry. W przypadku CartPole liczba akcji jest stała i wynosi 2 (prawo/lewo). Rozwiązałem ten problem przez maskowanie niedostępnych akcji, więc agent nie może wybrać akcji która jest zakryta. W praktyce wyjście z sieci polityki ma stały rozmiar równy przestrzeni akcji i na końcu akcje, które są niemożliwe do wykonaniu w danym stanie są maskowane wartością 0. Następnie próbkujemy akcje z logitów modelu. Również system nagród jest inny, w CartPole nagroda jest przyznawana za każdy krok, natomiast w Blokus informacja o nagrodzie jest tylko na koniec gry. W przypadku uproszczonej wersji gry Blokus dla dwóch osób, za wygraną dostaje się 1 punkt za remis 0 i za przegraną -1.

W celach testowych jak PPO poradzi sobie z grą Blokus zacząłem od uproszczonej wersji gry to znaczy od dwóch graczy i planszy $7x7$. Przestrzeń akcji jest dziedziną wszystkich ruchów z jakimi agent może się spotkać podczas interakcji ze środowiskiem. Przestrzeń akcji w Blokus'ie można zdefiniować przez wyznaczenie dla każdego klocka wszystkich możliwych ruchów, na które ten element może legalnie zostać wyłożony na

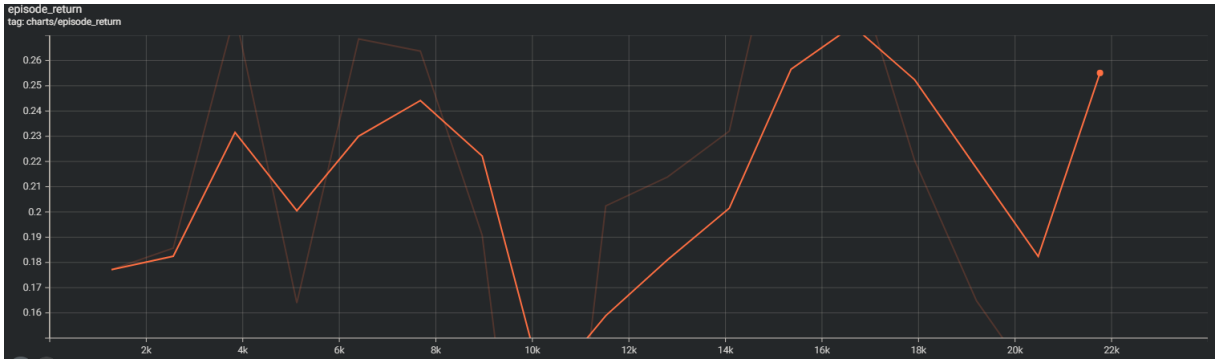
planszy uwzględniając jego rotacje. W tych ustawieniach przestrzeń akcji wynosi 919. Skorzystałem z implementacji środowiska dostępnej przez Pythonowe API gymnasium. Obserwacją jest tutaj dwuwymiarowa macierz o rozmiarze planszy 7 na 7, wolne miejsca są reprezentowane przez 0, kwadraty zajęte przez agenta to 1, a przeciwnika -1. W tej wersji środowiska nie implementowałem jeszcze mechanizmu samodzielnej gry i agent grał przeciwko losowemu graczowi.

Tab. 4. Specyfikacja środowiska Blokus $7x7$

Przestrzeń akcji	Dyskretna(919)
Kształt obserwacji	(7,7)
Maksymalna wartość obserwacji	$[1, \ldots 1]$
Minimalna wartość obserwacji	$[-1, \ldots, -1]$

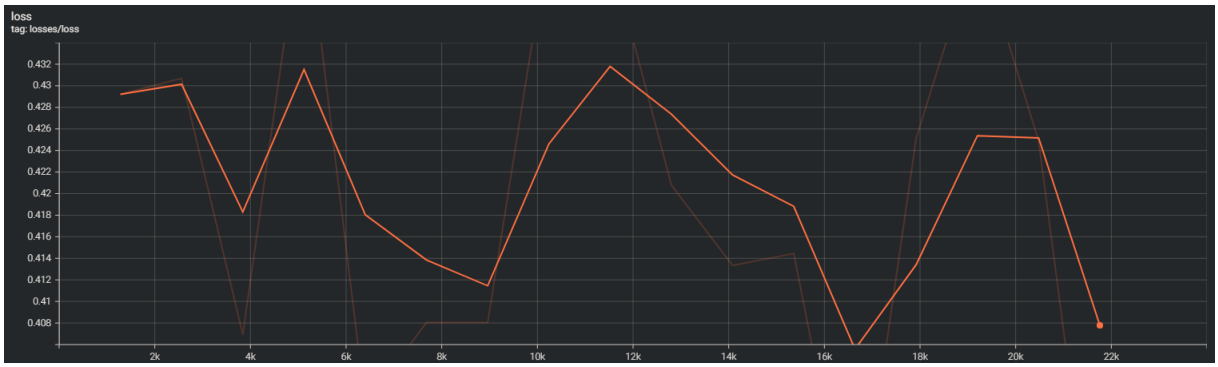
Źródło: Badania własne

Niestety ten eksperyment zakończył się niepowodzeniem. Agent nie radził sobie nawet grając z przeciwnikiem, którego ruchy były losowane uzyskując średni wynik nagród niewiele przekraczający 0. Dodatkowo cały proces treningu był bardzo niestabilny i już po około $20k$ iteracji doszło do zjawiska zanikającego gradientu[4] i wartość funkcji straty stała się NaN'em co zakończyło trening. Poniżej przedstawiam wykresy z treningu:



Rys. 9. Wykres nagród kolejnych epizodów w środowisku Blokus dla agenta PPO

Źródło: Badania własne



Rys. 10. Wykres straty kolejnych epizodów w środowisku Blokus dla agenta PPO

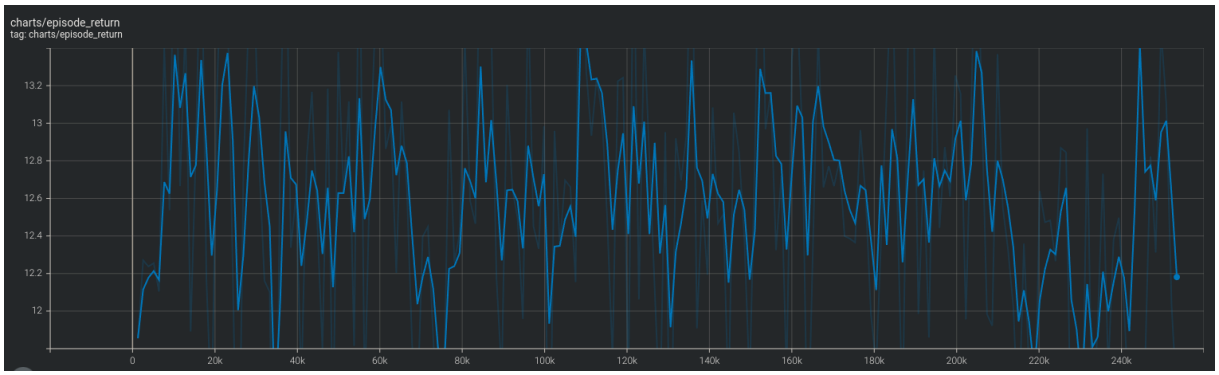
Źródło: Badania własne

Wartości zarówno na wykresie zwrotu (patrz rysunek nr 9) ani funkcji straty (patrz rysunek nr 10) nie wykazują aby model zbiegał do minimum. Biorąc pod uwagę, że przeciwnik wybierał ruchy na podstawie losowania to można stwierdzić że agent nie był w stanie nauczyć się polityki lepszej od losowej.

Po analizie problemu doszedłem do wniosku, że agent PPO zmaga się zbyt dużą wariacją środowiska aby był w stanie poprawnie oceniać sytuacje i dobierać odpowiednią strategie do niej. W przypadku gier Atari czy innych, w których PPO wykazywał dobre rezultaty nagroda jest przyznawana od razu za każdą wykonaną akcję. W grze Blokus nagroda jest epizodyczna, przyznawana tylko na końcu partii, utrudnia to znacznie agentowi ocenę pojedynczych posunięć i ich wpływ na ostateczny rezultat. Dlatego postanowiłem przeprowadzić jeszcze jeden eksperyment ze zmienionym systemem nagród.

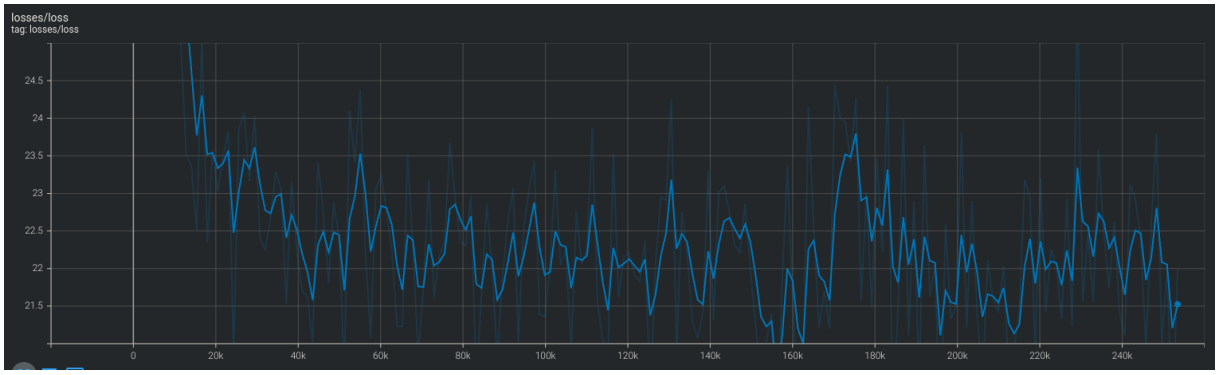
III.1.5. Blokus $7x7$ z natychmiastową nagrodą

PPO w środowiskach z epizodyczną nagrodą cierpi na zbyt małą ilość informacji zwrotnej co w poprzednim eksperymencie objawiło się dużą niestabilnością treningu i ostateczną eksplozją gradientu. Aby ułatwić algorytmowi dostosowanie strategii w Blokus'ie postanowiłem wprowadzić zmiany w systemie nagrody. W nowej wersji agent za każdy wyłożony klocek otrzymuje w nagrodę tyle punktów jakiej wielkości był dany element. Powoduje to że nagroda może wahać się od 1 punktu za najmniejszy kwadrat, aż do 5. Dodatkowo aby zaakcentować znaczenie wyniku końcowego epizodu, za wygraną grę agent dostawał 10 punktów za remis 0 i odpowiednio za przegraną -10. Wartości te wybrałem po przeprowadzeniu kilku wstępnych prób z liczbami z zakresu 10 do 50 za wygraną i odpowiednio -10 do -50 za przegraną. Pozostałe parametry środowiska, sieci neuronowej i hiper-parametry treningowe zostały takie same jak w poprzednim eksperymencie III.1.4. Poniżej wykresy z treningu:



Rys. 11. Wykres nagród kolejnych epizodów w środowisku Blokus z natychmiastową nagrodą dla agenta PPO

Źródło: Badania własne



Rys. 12. Wykres straty kolejnych epizodów w środowisku Blokus z natychmiastową nagrodą dla agenta PPO

Źródło: Badania własne

Zmiana systemu nagród pomogła ustabilizować trening i potwierdziła hipotezę, że zbyt mała ilość informacji ze środowiska powodowała zanik gradientu w poprzednim eksperymencie. Jednak jak wykazuje wykres nagród (patrz rysunek nr 11) oraz wykres funkcji straty (patrz rysunek nr 12) algorytm nie był w stanie dostosować strategii dla tego środowiska i pomimo wykonania ponad 250 tysięcy kroków wariacja jest bardzo dużo i nie widać aby model zbiegał do minimum.

III.1.6. Podsumowanie

Zarówno w ustawieniach z nagrodą epizodyczną jak i natychmiastową PPO wykazał duże trudności w dostosowaniu strategii dla gry planszowej jaką jest Blokus. Pomimo

małej planszy i stosunkowo niedużego skomplikowania środowiska algorytm nie był w stanie przeważyć nad przeciwnikiem, którego ruchy były próbkowane z równomiernego rozkładu. Duża wariacja środowiska może być spowodowana przez stale zmieniający się stan dostępnych akcji. Jest to odmienne od gier Atari, gdzie przestrzeń akcji była stała przez cały czas rozgrywki na przykład dwie akcje lewo/prawo w CartPole. Istotnym czynnikiem niepowodzenia PPO w tej grze planszowej jest brak mechanizmu symulacji w przyszłość. Przez swoją złożoność i strategiczny charakter gry takie jak szachy, Go czy Blokus wymagają dobrania długofalowej strategii i przewidywania tego co może się wydarzyć nie tylko w następnym ruchu ale nawet kilka posunięć w przód. Jest to analogiczny sposób myślenia stosowany przez człowieka, w którym decyzje podejmowane są na podstawie analizy określonej liczby ruchów kandydatów i analizy pozycji, która może nastąpić po wykonaniu tego ruchu. Głębia analizy i odpowiednia ocena końcowej pozycji jest kluczowym czynnikiem pozwalającym na wybranie najbardziej optymalnego ruchu w danej pozycji[5]. Algorytm PPO jest pozbawiony tego mechanizmu i podejmuje decyzje tylko na podstawie obecnej obserwacji co w połączeniu z brakiem natychmiastowej nagrody może być zbyt słabą informacją i dawać zbyt dużą wariacje, aby sieć neuronowa była w stanie zrozumieć tą grę.

III.2. Eksperymenty AlphaZero dla dwóch graczy

Po niepowodzeniu z PPO postanowiłem zaimplementować algorytm AlphaZero II.2. AlphaZero w oryginalnej formie[26] jest zaprojektowany dla gier dwuosobowych i od takiej konfiguracji środowiska postanowiłem zacząć ten eksperyment. Tak, jak w eksperymencie z PPO III.1.4 przed przejściem do cięższego zadania chciałem sprawdzić czy implementacja działa poprawnie w łatwiejszym środowisku. Środowisko Blokus ma takie same parametry jak w poprzednim eksperymencie III.1.4. Główna, zmianą w implementacji środowiska było zrezygnowanie z biblioteki Gymnasium i zamiast tego dodanie adaptacyjnego interfejsu, który umożliwia bezpośrednią interakcje ze środowiskiem bez warstwy pośredniej. W tej wersji agent grał sam przeciwko sobie i uczył się na podstawie swoich ruchów. Po określonej ilości epizodów samodzielnej gry sieć agenta była uczona na dostępnych próbkach po czym nowo wytrenowany model grał areny kontrolne przeciwko swojej poprzedniej wersji. Jeśli nowy agent wygrał określony procent gier był akceptowany jako nowy najlepszy model. W przeciwnym wypadku był odrzucany i proces uczenia rozpoczynał się od nowa. Swoją implementacje bazowałem na implementacji AlphaZero dla gry Othello[28].

Użyłem tej samej architektury sieci splotowej (patrz rysunek nr 4) co w poprzednim eksperymencie z PPO. Pozostałe ustawienia środowiska takie jak wielkość planszy, rozmiar obserwacji, przestrzeń akcji czy funkcja nagrody pozostały takie same jak w uprzednim doświadczeniu III.1.4.

III.2.1. Parametry eksperymentu

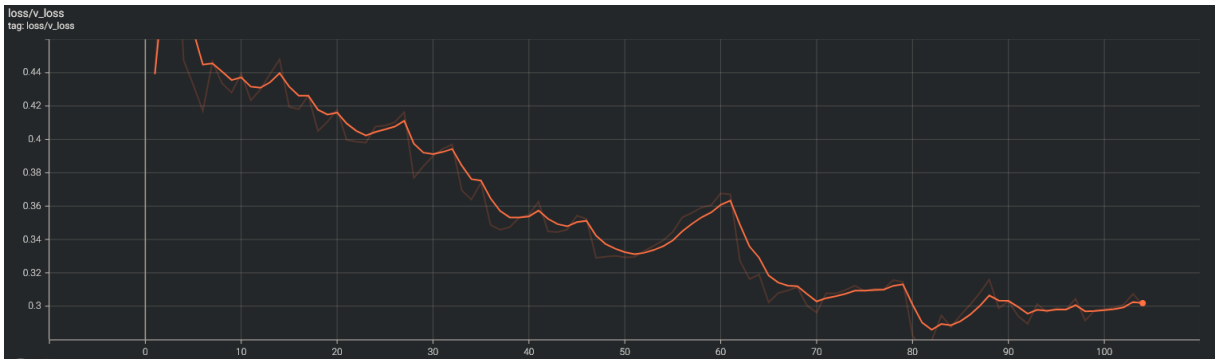
AlphaZero poza dobraniem odpowiednich parametrów typowych dla treningu głębokich sieci neuronowych takich jak wielkość partii czy ilość epok wymaga także dostrojenia parametrów związanym z interakcją agenta ze środowiskiem. Najbardziej ogólnym atry-

butem jest ilość gier jaką agent ma rozegrać w epizodzie samodzielnej gry. W każdej rozgrywce ruchy agenta podejmowane są na podstawie sieci polityki, funkcji wartości oraz symulacji MCTS. Parametr ilości symulacji MCTS określa ile gier symulacyjnych ma być rozegranych przed tym jak zostanie wybrany jeden właściwy ruch na podstawie otrzymanej struktury drzewa MCTS. Po procesie zbierania informacji ze środowiska następuje etap uczenia modelu agenta, proces ten został opisany w sekcji o sieciach neuronowych II.2.3. W ostatnim kroku nowy model porównywany jest z obecnie najlepszym agentem przez rozgrywanie gier kontrolnych jeśli nowy model przekroczy określony próg zdobytych punktów to zostaje nowym najlepszym agentem. Poniżej przedstawiam listę parametrów jakie dobrałem dla tego eksperymentu:

- Liczba gier w epizodzie samodzielnej gry: 100
- Ilość symulacji MCTS podczas samodzielnej gry: 25
- Ilość epok uczenia: 10
- Rozmiar partii: 64
- Wskaźnik uczenia się: 0.001
- Ilość rozegranych aren kontrolnych: 40
- Próg akceptacji nowego modelu: 60%

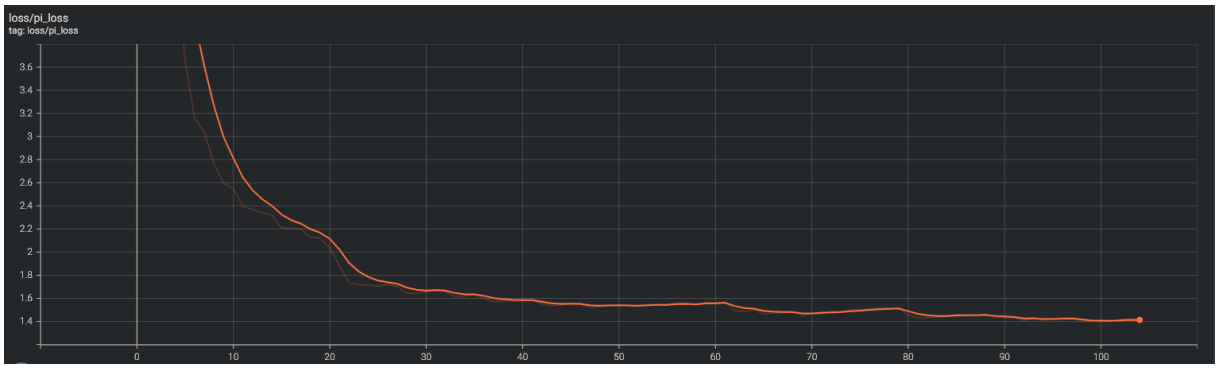
III.2.2. Trening agenta

Po wykonaniu 103 pętli treningowych, które zajęły około 160h na karcie graficznej NVIDIA GeForce RTX 3070Ti o pojemności pamięci 8GB, zdecydowałem się zatrzymać trening. Poniżej przedstawiam wykresy:



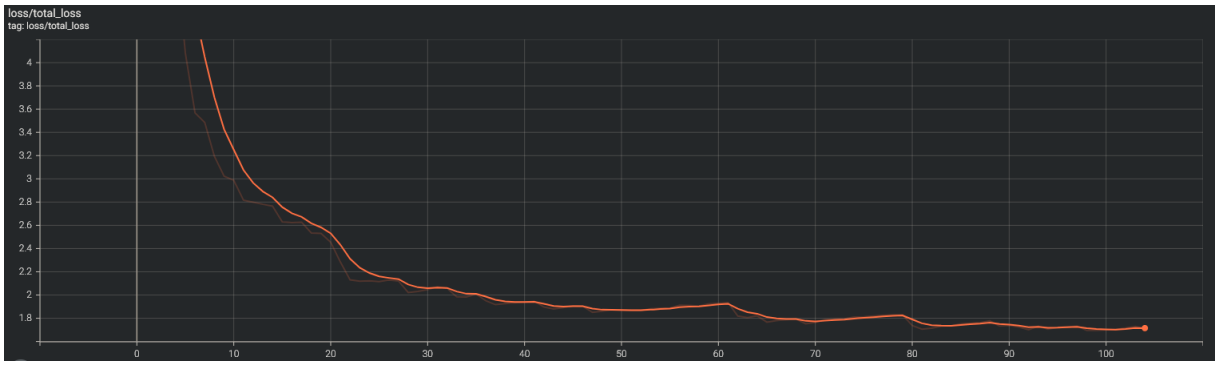
Rys. 13. Wykres straty funkcji wartości w środowisku Blokus $7x7$ dla AlhpaZero

Źródło: Badania własne



Rys. 14. Wykres straty polityki w środowisku Blokus $7x7$ **dla AlhpaZero**

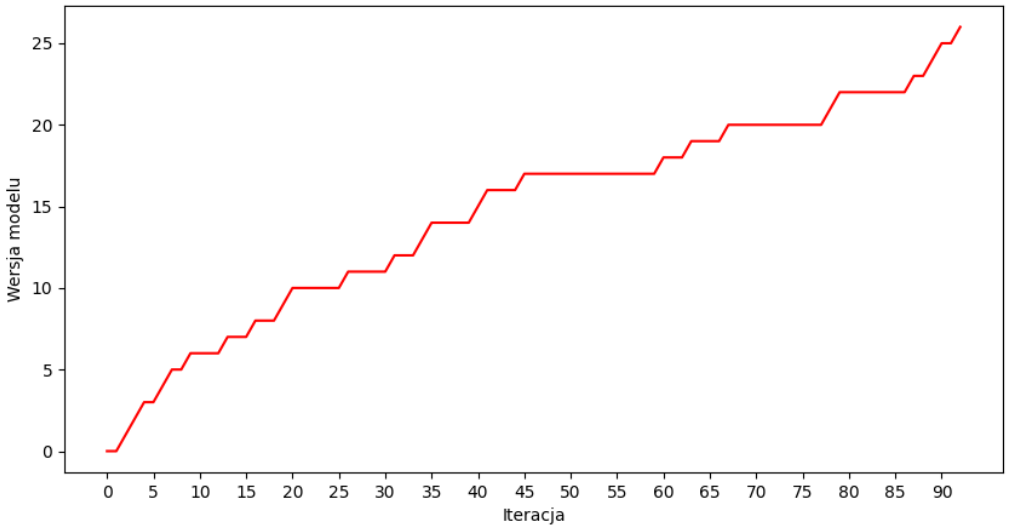
Źródło: Badania własne



Rys. 15. Wykres całkowitej funkcji straty w środowisku Blokus $7x7$ **dla AlhpaZero**

Źródło: Badania własne

W tym przypadku nie było problemów z stabilnością treningu tak jak miało to miejsce w III.1.4. Wszystkie trzy wykresy funkcji straty (patrz rysunki nr 13,14,15) nie wykazują dużej wariancji i wyglądają że zbliżyły się do swojego minimum, jednak pozostało jeszcze miejsce na potencjalną poprawę jeśli trening potrwałby dłużej. Strata funkcji wartości (patrz rysunek nr 13) jest obliczana jako średni błąd kwadratowy między przewidywaną wartością, a rzeczywistym wynikiem. Natomiast strata polityki (patrz rysunek nr 14) jest obliczana przy użyciu entropii krzyżowej między przewidywanym prawdopodobieństwem ruchu, a faktycznym wyborem ruchu w samodzielnej grze. Całkowita funkcja straty (patrz rysunek nr 15 jest sumą straty wartości i polityki. Bardziej szczegółowy opis na powyższe funkcje straty został przedstawiony w paragrafie o sieciach neuronowych II.2.3.



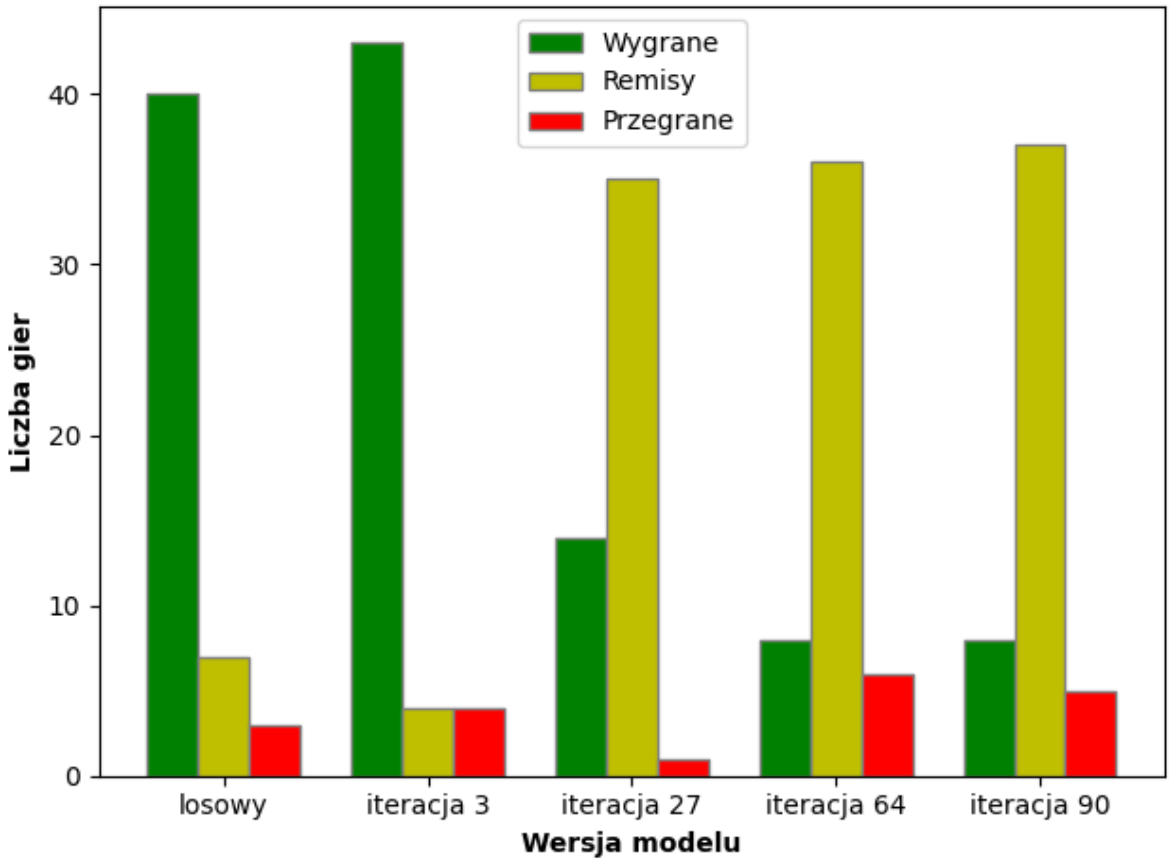
Rys. 16. Wykres akceptacji nowego modelu AlphaZero dla środowiska Blokus $7x7$

Źródło: Badania własne

Wykres nr 16 przedstawia aktualizacje wersji modelu w kolejnych iteracjach treningowych. Oznacza to że model, który w arenach kontrolnych przekroczył próg 60% procent zdobytych punktów stawał się nowym najlepszym modelem i wersja została podnoszona. Na 103 cykle uczenia powstało 26 wersji modelu, które były lepsze od swoich poprzedników.

III.2.3. Ewaluacja agenta

W celu ewaluacji skuteczności tego eksperymentu przeprowadziłem serie 5 meczów między najlepszym modelem z iteracji 103 przeciwko jego poprzednikom. Wziąłem pod uwagę 5 wersji, które są dość dobrą reprezentatywną próbką postępu i ulepszeń agenta w czasie. Konkretnie wybrałem losowego gracza będącego poziomem wyjścia i agentów z iteracji: 3, 27, 64 i 90. Każdy mecz składał się z 50 gier, poniżej przedstawiam wyniki tych pojedynków:



Rys. 17. Wyniki najlepszego modelu przeciwko poprzednim wersjom

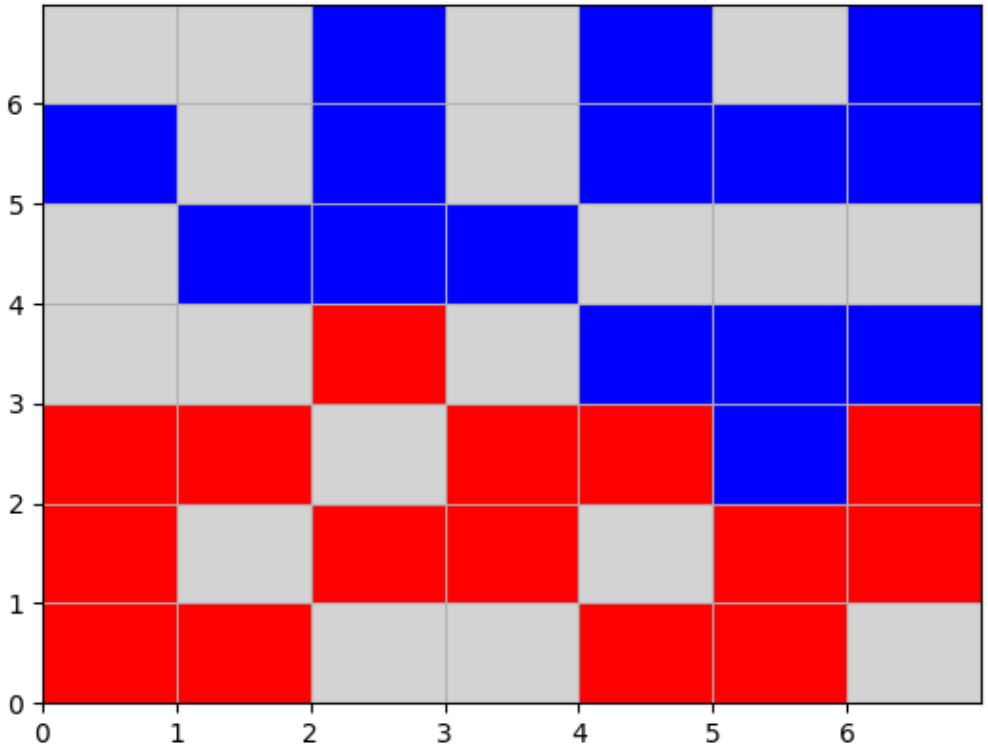
Źródło: Badania własne

Poniżej analiza wyników17

- Przeciwko losowemu graczowi, ostateczny model osiągnął 40 zwycięstw, 7 remisów i zaledwie 3 porażki, wykazując znaczną przewagę.
- Model z iteracji 3 został pokonany z 43 zwycięstwami, co wskazuje na bardziej wyrafinowane podejście strategiczne modelu końcowego, wraz z 4 remisami i 4 porażkami.
- Znaczący wzrost liczby remisów zaobserwowano w konfrontacji z modelem z iteracji 27, co zaowocowało 14 zwycięstwami modelu końcowego, 35 remisami i tylko jedną porażką, co sugeruje bliższe dopasowanie zdolności strategicznych.
- Spotkania z modelami z iteracji 64 i 90 przedstawiały większość remisów - odpowiednio 36 i 37 - z niewielką przewagą modelu końcowego pod względem zwycięstw i marginalnym wzrostem strat.

Stała wysoka liczba remisów, zwłaszcza na późniejszych etapach, podkreśla iteracyjne udoskonalanie procesu decyzyjnego modelu. Dane odzwierciedlają konwergencję w kierunku solidnej strategii, ponieważ ostatnia iteracja modelu nie znacząco przewyższyła

poprzednie wersje, wskazując na plateau krzywej uczenia się i sugeruje zwiększanie zdolności modelu do gry w środowisku Blokus. Poniżej przedstawiam przykładowy obraz z przebiegu gry:



Rys. 18. Przykład rozgrywki w środowisku Blokus $7x7$

Źródło: Badania własne

III.2.4. Podsumowanie

Eksperyment z AlhpaZero wykazały, że algorytm jest w stanie dostosować strategie dla środowiska Blokus i wykazał znaczny wzrost poziomu względem bazowemu losowemu graczowi. Największym problemem jaki pojawił się w trakcie eksperymentu jest ilość mocy obliczeniowej jaka jest potrzebna do osiągnięcia konwergencji algorytmu AlhpaZero. Pomimo że środowisko było stosunkowo małe, przestrzeń akcji wynosiła tylko 919 to algorytm wymagał ponad 160 godzin aby wykonać 100 iteracji treningowych. Również wraz z wzrostem skomplikowania środowiska będzie potrzebne wykorzystanie głębszej i bardziej złożonej architektury sieci neuronowej.

III.3. Eksperymenty AlphaZero dla czterech graczy

Po udanych próbach z uproszczoną wersją gry postanowiłem przeprowadzić eksperyment z pełną wersją gry. W tym przypadku grało 4 graczy, a plansza miała rozmiar $20x20$ co dawało przestrzeń akcji 30433, jest to ponad 33 razy więcej niż w Blokus $7x7$.

Dodatkową trudnością jaką musiałem rozwiązać było to że obecna wersja kodu była zaprojektowana z myślą o grze dwuosobowej, więc musiałem dokonać kilku znaczących zmian w kodzie. Zainspirowałem się artykułem o implementacji AlhpaZero dla wielu graczy[17], który wprowadza zmiany w mechanizmie samodzielnej gry aby uwzględniał większą liczbę graczy. Dodatkowo dokonałem wektoryzacji algorytmu MCTS w celu przyśpieszenia obliczeń.

III.3.1. Przystosowanie środowiska Blokus

Innym aspektem, który wymagał aktualizacji, było same środowisko Blokus. Obecna implementacja sprawdzała się dobrze dla dwóch graczy, jednak w pełnowymiarowej wersji gry działał zbyt wolno. Z tego powodu skorzystałem ze środowiska dostępnego w bibliotece colosseumrl[24]. Pakiet ten zawiera implementacje środowisk do uczenia ze wzmocnieniem w grach wieloosobowych. Posiada on API do języka Python co pozwala na elastyczne zaadaptowanie go do problemów uczenia ze wzmocnieniem, a dodatkowo w celach optymalizacji kluczowe funkcjonalności są napisane w języku *C*. Aby dostosować środowisko Blokus z tej biblioteki stworzyłem klasę osłonową, która implementuje poniższy interfejs AlphaZero:

inicjalizacja_planszy(self) Zwraca początkowy stan gry, w tym początkową konfigurację planszy i pierwszego gracza.

następny_stan(self, obecny_stan, obecny_gracz, id_akcji) Stosuje akcję do bieżącego stanu i zwraca następny stan oraz gracza, który zagra jako następny.

reprezentacja_łańcuchowa(self, stan) Konwertuje stan gry na reprezentację łańcuchową do celów haszowania w algorytmie MCTS.

liczność_akcji(self) Zwraca całkowitą liczbę możliwych akcji w grze.

dostępne_ruchy(self, obecny_stan, obecny_gracz) Generuje wektor masek wskazujący prawidłowe ruchy dla bieżącego stanu i gracza.

obserwacja(self, stan, gracz) Konwertuje stan gry na obserwację odpowiednią dla danego gracza.

render(self, stan) Wizualizuje aktualny stan planszy.

czy_koniec_gry(self, stan) Określa, czy gra się zakończyła i zwraca zwycięzcę, jeśli taki istnieje.

punktacja(self, wygrani) Oblicza i zwraca wyniki na podstawie zwycięzców gry.

W przypadku czterech graczy system nagród zero jedynkowy nie sprawdza się dobrze. Z tego powodu zdecydowałem się na przyznawanie trzech punktów dla gracza jeśli był jedynym zwycięzcą, w przypadku remisu dwóch graczy obaj dostają 1, a przegrani zawodnicy zawsze otrzymują -1 punkt.

III.3.2. Architektura sieci neuronowej

Wraz ze wzrostem złożoności środowiska potrzebne było zwiększenie rozmiaru sieci neuronowej odpowiedzialnej za podejmowanie decyzji przez agenta. Użyłem architektury podobnej do tej z sieci ResNet[8], która jest szeroko stosowana w zadaniach związanych z wizją komputerową. Model składa się z zestawionych bloków warstw konwolucyjnych przeplatanych normalizacją baczową[11] i Poprawionej Funkcji Liniowej jako funkcji aktywacji. Poniżej szczegóły architektury modelu:

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
├─Conv2d: 1-1                            [-1, 64, 20, 20]          4,672
├─BatchNorm2d: 1-2                       [-1, 64, 20, 20]          128
├─Sequential: 1-3                        [-1, 64, 20, 20]          --
|    └─Sequential: 2-1                   [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-1                  [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-2             [-1, 64, 20, 20]          128
|    |    └─ReLU: 3-3                    [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-4                  [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-5             [-1, 64, 20, 20]          128
|    └─Sequential: 2-2                   [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-6                  [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-7             [-1, 64, 20, 20]          128
|    |    └─ReLU: 3-8                    [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-9                  [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-10            [-1, 64, 20, 20]          128
|    └─Sequential: 2-3                   [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-11                 [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-12            [-1, 64, 20, 20]          128
|    |    └─ReLU: 3-13                   [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-14                 [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-15            [-1, 64, 20, 20]          128
|    └─Sequential: 2-4                   [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-16                 [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-17            [-1, 64, 20, 20]          128
|    |    └─ReLU: 3-18                   [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-19                 [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-20            [-1, 64, 20, 20]          128
|    └─Sequential: 2-5                   [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-21                 [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-22            [-1, 64, 20, 20]          128
|    |    └─ReLU: 3-23                   [-1, 64, 20, 20]          --
|    |    └─Conv2d: 3-24                 [-1, 64, 20, 20]          36,928
|    |    └─BatchNorm2d: 3-25            [-1, 64, 20, 20]          128
├─Conv2d: 1-4                            [-1, 2, 20, 20]           130
├─BatchNorm2d: 1-5                       [-1, 2, 20, 20]           4
├─Linear: 1-6                            [-1, 30433]               24,376,833
├─Conv2d: 1-7                            [-1, 1, 20, 20]           65
├─BatchNorm2d: 1-8                       [-1, 1, 20, 20]           2
├─Linear: 1-9                            [-1, 64]                  25,664
├─Linear: 1-10                           [-1, 4]                   260
==========================================================================
Total params: 24,778,318
Trainable params: 24,778,318
Non-trainable params: 0
Total mult-adds (M): 174.49
==========================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 4.55
Params size (MB): 94.52
Estimated Total Size (MB): 99.08
==========================================================================
```

Rys. 19. Architektura sieci typu ResNet w AlphaZero

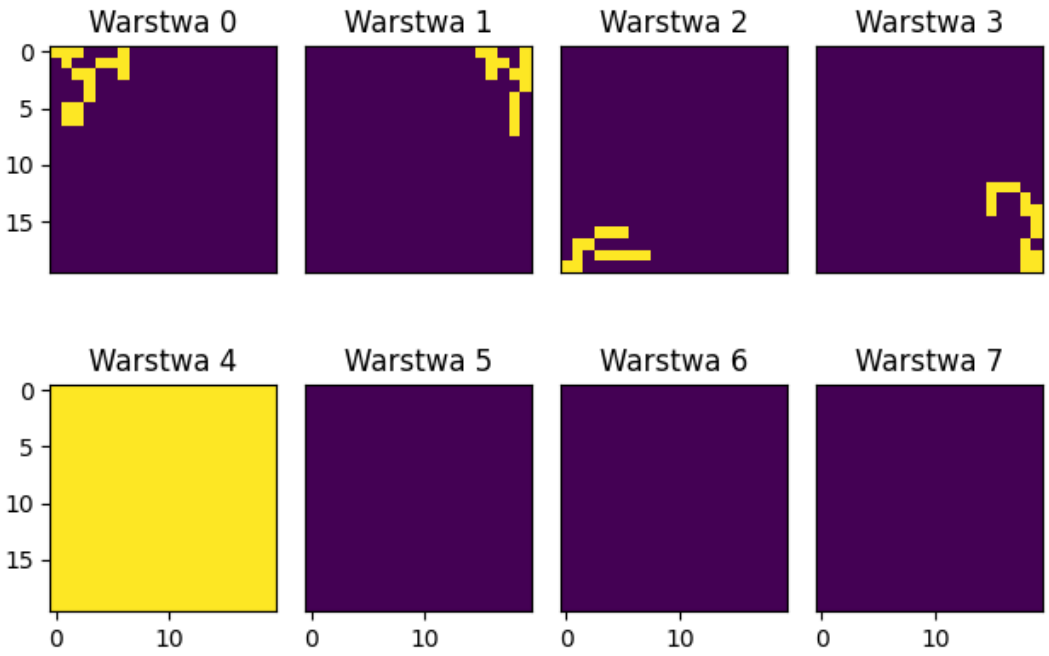
Źródło: Badania własne

Sieć tak jak w poprzedniej wersji (patrz rysunek nr 4) posiada na wyjściu dwie głowy modelujące: jedną dla polityki i drugą dla wartości. Warstwa liniowa polityki na wyjściu ma rozmiar 30433 i podaje rozkład prawdopodobieństwa następnego ruchu, a warstwa liniowa dla wartości składa się z 4 neuronów, które przewidują wynik gry dla każdego gracza z danego stanu. Łączna ilość parametrów modelu wynosi 24.28M.

III.3.3. Reprezentacja stanu gry

Podczas adaptacji formatu wejściowego stanu gry w celu uwzględnienia zwiększonej liczby graczy, konwencjonalna reprezentacja w której pionki aktywnego gracza są oznaczone 1, a przeciwnika -1 nie sprawdziłaby się. Takie binarne rozróżnienie było niewystarczające, szczególnie w scenariuszach, w których identyfikacja graczy za pomocą etykiet numerycznych mogłaby nieumyślnie przypisać sztuczne znaczenie kolejności graczy. W szczególności, numeryczne identyfikatory od 1 do 4 mogłyby mylnie sugerować hierarchiczne znaczenie, co jest sprzeczne z zasadą gry, że wszystkie ruchy graczy mają równoważną wartość strategiczną.

Aby obejść tę kwestię, zaadaptowałem podejście zaproponowane w "Multiplayer AlphaZero"[17], wykorzystując metodę wielokanałową do wyraźnego reprezentowania pozycji każdego gracza i bieżącej tury gracza. W rezultacie dane wejściowe obserwacji do modelu to macierz o wymiarach $8x20x20$. Pierwsze cztery kanały reprezentują pozycje na planszy zajmowane przez pionki każdego gracza, podczas gdy kolejne cztery kanały służą jako wskaźnik tury. Kanał wyróżnika ruchu wypełniony wartością 1 oznacza turę obecnego gracza, zapewniając jasną i bezstronną reprezentację stanu gry bez nadawania nieuzasadnionej wagi żadnemu konkretnemu graczowi. Ta wielokanałowa reprezentacja skutecznie zachowuje bezstronność znaczenia graczy, zapewniając jednocześnie modelowi sieci neuronowej kompleksowe i dyskretne przedstawienie dynamiki gry. Poniżej przedstawiam przykład takiej obserwacji:



Rys. 20. Przykład obserwacji stanu gry w Blokus $20x20$

Źródło: Badania własne

Przykładowa obserwacja (patrz rysunek nr 20) pokazuje stan planszy po wykonaniu ruchu pierwszego gracza w czwartej turze.

III.3.4. Optymalizacja hiper-parametrów

Wraz ze wzrostem złożoności i ilości parametrów sieci neuronowej większego znaczenia nabiera odpowiednie dostrojenie hiper-parametrów treningowych. Pozwoli to na szybszą konwergencje modelu co powinno się przełożyć na lepsze osiągi agenta w grze. W tym celu skorzystałem z pakietu Optuna[1], jest to otwarto źródłowy framework do zautomatyzowania procesu optymalizacji hiper-parametrów. Pakiet ten pozwala na korzystanie z różnych metod maksymalizacji efektywności takich jak optymalizacja bayesowska, zejście gradientu i innych zaawansowanych algorytmów, poprzez przyjazny dla użytkownika Python'nowe API. Do przygotowania przestrzeni przeszukiwania hiper-parametru określa się jego dziedzinę, która może być ciągła lub dyskretna, definiuje się funkcję kryterium wraz z kierunkiem optymalizacji czy ma być zmaksymalizowana czy zminimalizowana.

Przed przystąpieniem do optymalizacji hiper-parametrów potrzebowałem zebrać odpowiednią ilość danych ze środowiska. W tym celu przeprowadziłem 10 epizodów samodzielnej gry z następującymi parametrami:

- Liczba gier w epizodzie samodzielnej gry: 100
- Ilość symulacji MCTS podczas samodzielnej gry: 25

Po skompletowaniu odpowiedniej ilości danych przeszedłem do zdefiniowania przestrzeni przeszukiwań hiper-parametrów, które chciałem dostroić: Po 100 iteracjach opty-

Tab. 5. Przestrzeń hiper-parametrów dla próby optymalizacji Optuna

Hiper-parametr	Typ	Zakres
Współczynnik uczenia	Zmienna rzeczywista (skala logarytmiczna)	$[10^{-5}, 10^{-1}]$
Rozmiar partii	Kategoryczny	$\{16, 32, 64, 128\}$
Liczba bloków modelu	Całkowity	$[1, 5]$
Epoki	Całkowity	$[10, 100]$

malizacji otrzymałem następujące wartości:

Tab. 6. Dostrojone hiper-parametry przez optymalizację Optuna

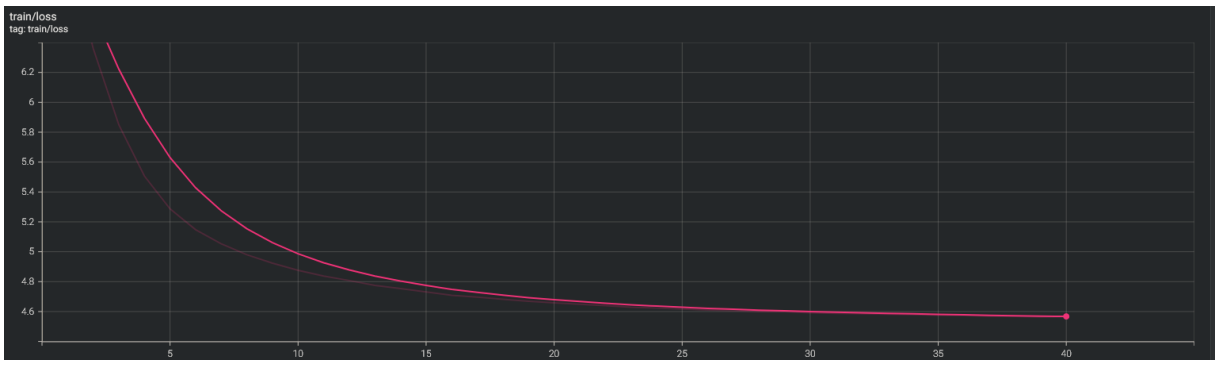
Hiper-parametr	Wartość
Współczynnik uczenia	$5 * 10^{-5}$
Rozmiar partii	64
Liczba bloków modelu	5
Epoki	40

III.3.5. Trening agenta

Krytycznym aspektem procesu szkolenia agentów AlphaZero wymagającym modyfikacji była liczba symulacji MCTS przeprowadzanych podczas samodzielnej gry. Parametr ten dyktuje liczbę symulacji MCTS, które agent wykonuje przed określeniem swojego następnego ruchu. Ma to ogromny wpływ na zakres informacji dostępnych dla agenta dotyczących środowiska w danym stanie, co z kolei zwiększa skuteczność podejmowania decyzji. Jednak podstawową wadą zwiększania tego parametru jest proporcjonalna eskalacja czasu wymaganego do wykonania pojedynczego ruchu, co spowalnia trening.

Aby zachować równowagę między ilością obliczeń a strategiczną siłą rozgrywki, ustawiłem liczbę symulacji MCTS na 200, w przeciwieństwie do oryginalnej implementacji AlphaZero dla szachów, gdzie parametr ten został ustawiony na 800. Ze względu na ograniczenia sprzętowe nie było możliwe podniesienie wartości tego parametru do poziomu oryginalnego ustawienia, co wymagało kompromisu na niższym progu.

W eksperymencie została przeprowadzona jedna iteracja treningowa składająca się z 100 partii samodzielnej gry z liczbą symulacji MCTS ustawioną na 200. Faza gromadzenia danych trwała około 60 godzin. Po zebraniu 5600 próbek danych przeprowadziłem trening modelu z zoptymalizowanymi hiper-parametrami (patrz tabela nr 6). Kolejne sekcje przedstawiają graficzne zobrazowanie wyników treningu:



Rys. 21. Wykres całkowitej funkcji straty w środowisku Blokus $20x20$ **dla Alhpa-Zero**

Źródło: Badania własne

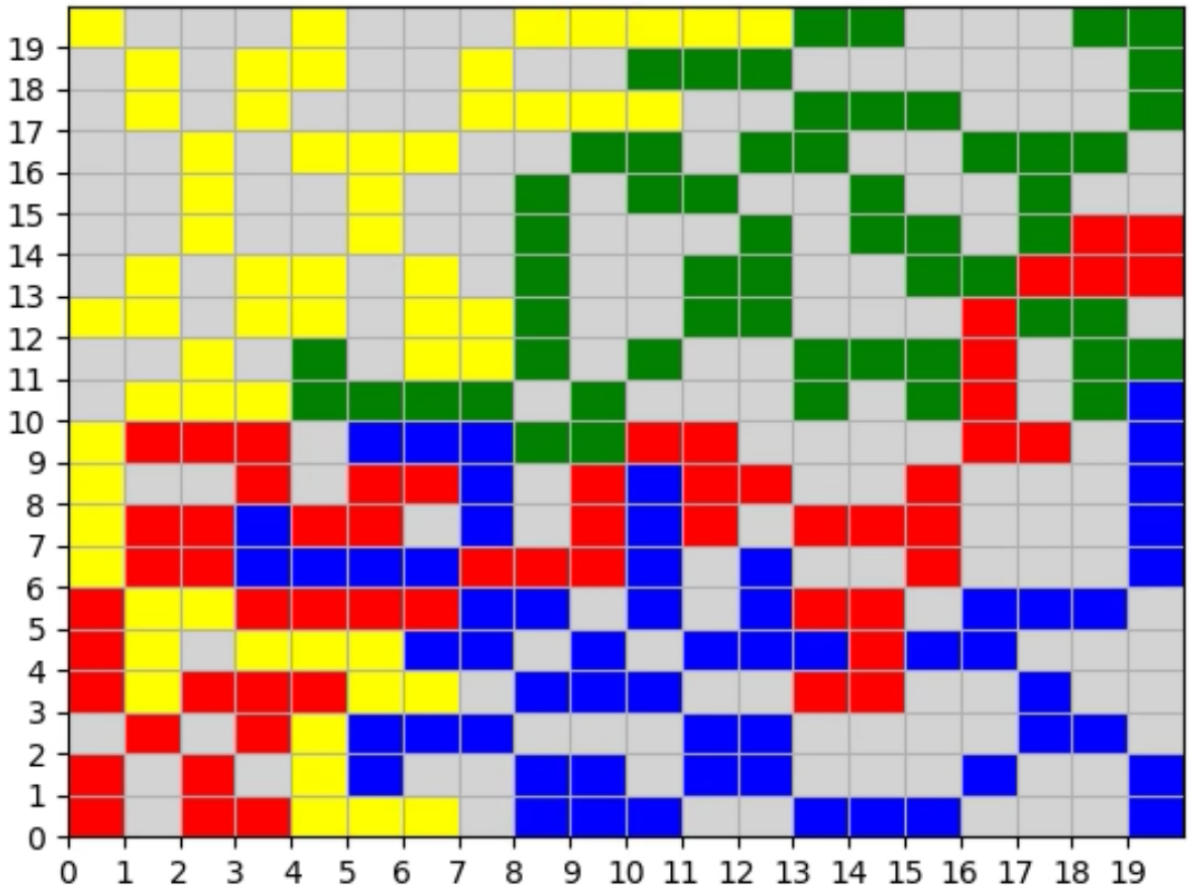
Wykres nr 21 wskazuje na bardzo gładki i stabilny spadek funkcji straty dla tych danych. Możemy wnioskować że wraz ze wzrostem ilości dostępnych danych i wydłużeniem treningu, model stawałby się coraz lepszy i funkcja straty mogłaby jeszcze znacząco spaść.

III.3.6. Ewaluacja agenta

Na koniec w celu ewaluacji wytrenowanego modelu przeprowadziłem 24 areny kontrolne. Przeciwnikiem agenta był klasyczny algorytm MCTS, za którego politykę ruchów odpowiadał model z rozkładem równomiernym z ilością symulacji ustawioną na 200. Wyniki przedstawiam w tabeli nr 7. Znaczna przewaga zdobytych punktów przez agenta wskazuje, że pomimo stosunkowo małej próbki danych pochodzącej tylko ze 100 rozgrywek, agent był w stanie udoskonalić swoją strategię i pokonać model bazowy, który bazuje na symulacji MCTS. Warto zauważyć, że zarówno agent AlhpaZero jak i MCTS z równomiernym rozkładem mają ustawiony parametr symulacji MCTS na 200. Poniżej przedstawiam przykładowy obraz z przebiegu gry:

Tab. 7. Wyniki ewaluacji agenta AlhpaZero dla Blokus $20x20$

Agent	Zdobyte punkty
AlphaZero	24
MCTS 200	-8
MCTS 200	-8
MCTS 200	-8



Rys. 22. Przykład rozgrywki w środowisku Blokus $20x20$

Źródło: Badania własne

III.4. Podsumowanie

Agent osiągnął wysoki wynik 24 punktów przeciwko -8 punktom gdy walczył z przeciwnikiem wykorzystującym wyłącznie Przeszukiwanie Drzew Monte Carlo. Wydajność ta jest bardzo obiecująca, szczególnie biorąc pod uwagę skomplikowaną naturę Blokus'a jako strategicznej, przestrzennej gry planszowej oraz ograniczoną ilość dostępnych danych treningowych. Proces treningu i ewaluacji sugerują znaczne pole do dalszej poprawy. Biorąc pod uwagę, że prawdziwy potencjał AlphaZero uwalnia się

wraz ze wzrostem ilości rozegranych gier i zasobów obliczeniowych.

Kluczową obserwacją z tego eksperymentu jest potwierdzenie, że AlhpaZero wymaga znacznej mocy obliczeniowej aby w pełni wykorzystać swój potencjał. Oryginalny AlhpaZero dla Szachów, Go i Shogi był trenowany przy użyciu 5000 jednostek przetwarzania tensorowego (TPU) do generowania danych przez epizody samodzielnej gry oraz 64 TPUs do trenowania sieci neuronowej. Cały proces treningowy trwał $700k$ kroków jednak już po $300k$ krokach, które zajęły 4 godziny AlphaZero w szachy pokonał ówczesny najlepszy silnik szachowy Stockfish[29]. Jednak tak jak pokazał ten eksperyment nawet wykorzystując tylko jedną graficzną jednostkę obliczeniową można wytrenować agenta przewyższającego model bazowy w skomplikowanej strategicznej grze planszowej.

Podsumowanie

Obiektem pracy było stworzenie algorytmu uczenia ze wzmocnieniem dla gry planszowej Blokus. W tym celu użyłem dwóch algorytmów: Proksymalnej Optymalizacji Polityki[22] i AlphaZero[26].

Swoje eksperymenty zacząłem od implementacji algorytmu PPO. Wybrałem go dlatego, że osiągnął on wysokie wyniki w grach typu Atari wygrywając w 30 na 49 gier przeciwko innym algorytmom opartym na polityce takich jak Zaawansowany Aktor Krytyk[14] czy Aktor-krytyk z Powtórką Doświadczenia[31]. Został również z powodzeniem zastosowany do stworzenia zespołu agentów OpenAI Five, w grze Dota 2 osiągając skuteczność zbliżoną do najlepszych ludzkich graczy[15]. Zdecydowałem się również na przetestowanie tego algorytmu w grze planszowej Blokus przez stosunkowo prostą implementację w porównaniu do innych gradientowych polityk takich jak Tęcza[9]. Przed przystąpieniem do właściwych eksperymentów w środowisku Blokus sprawdziłem działanie swojej implementacji PPO w środowisku CartPole. Algorytm, w krótkim czasie już po około 60 iteracjach zaczął osiągać maksymalną nagrodę 500 punktów co znaczyło że nauczył się odpowiedniej strategii w tym środowisku.

Po upewnieniu się, że implementacja PPO działa poprawnie przystąpiłem do eksperymentów w środowiskiem Blokus. Tutaj też zacząłem od uproszczonej wersji tej gry, to znaczy planszy o wymiarach $7x7$ i dwóch graczy. Skorzystałem z środowiska dostępnego przez Pythonowe API gymnasium. Przestrzeń stanów w tym środowisku wynosi 919, obserwacją jest plansza gry o wymiarach $7x7$, gdzie 1 oznacza klocki gracza, -1 oznaczone są pola zajęte przez przeciwnika, a 0 to puste pole. W tym eksperymencie wykorzystałem dwie architektury sieci neuronowej: perceptron wielowarstwowy (patrz rysunek nr 3) i splotową sieć neuronową (patrz rysunek nr 4), które miały odpowiednio 79k i 388k parametrów. Niestety pomimo wielu prób eksperyment zakończył się niepowodzeniem. Agent nie był w stanie pokonać strategii polegającej na losowym wyborze ruchu. Dodatkowo proces treningu okazał się być bardzo niestabilny i po około 20k iteracjach doszło do zaniku gradientu[4], a funkcja straty stała się $NaN'em$. Nie pomogła nawet zmiana systemu nagród na natychmiastową, gdzie agent otrzymywał za każdy ruch tyle punktów z ilu kwadratów składał się wyłożony klocek. Dodatkowo aby podkreślić znaczenie ostatecznego wyniku, za wygraną przyznawane było 10 punktów, za remis 0 i -10 za przegraną. Algorytm w tym przypadku nie zakończył się wybuchem gradientów pomimo wykonania ponad 250 tysięcy kroków nie wykazał żadnej poprawy grając przeciwko graczowi z równomiernym rozkładem. Jest kilka potencjalnych przyczyn które sprawiły, że PPO nie poradziło sobie w tym środowisku:

- Zmienna ilość dostępnych ruchów: W grach planszowych dostępne akcje różnią się między stanami, gdzie w grach Atari pozostają niezmienne

- Brak symulacji: PPO podejmuje decyzje tylko i wyłącznie na podstawie obecnego stanu gry bez przeprowadzania symulacji tego co może się wydarzyć w przyszłości.

Drugim algorytmem jaki wykorzystałem w swojej pracy jest AlphaZero[26, 25]. Algorytm ten został opracowany i zastosowany w planszowych grach strategicznych takich jak: go, szachy czy shogi, osiągając w nich nadludzki poziom. AlphaZero rozpoczyna naukę od zera, opierając się wyłącznie na zasadach gry i stopniowo doskonali swoje umiejętności poprzez samodzielną grę. Wykorzystuje uogólnioną adaptację algorytmu Przeszukiwania Drzewa Monte Carlo (MCTS), zintegrowaną z głębokimi sieciami neuronowymi. Charakterystyczną cechą AlphaZero, odróżniającą go od swoich poprzedników, jest jego wszechstronność i zdolność do dostosowania się do dowolnej gry dwuosobowej z doskonałą informacją. Takie podejście umożliwia AlphaZero opanowanie różnych złożonych gier strategicznych bez konieczności posiadania specjalistycznej wiedzy w danej dziedzinie. Swoje eksperymenty z AlphaZero zacząłem od wersji dla dwóch graczy i planszy $7x7$. Po wykonaniu 103 pętli treningowych, agent w każdej iteracji rozgrywał 100 epizodów samodzielnej gry z ilością symulacji MCTS ustawioną na 25. Następnie na podstawie zebranych próbek ze środowiska, wagi sieci neuronowej były aktualizowane przy pomocy metody spadku gradientu. Na koniec nowa wersja modelu rozgrywała 40 aren kontrolnych przeciw swojej najlepszej wersji, jeśli nowy model przekroczył próg 60% wygranych to stawał się nowym najlepszym modelem. Aktualizacja modelu nastąpiła 26 razy i najbardziej wydajny model jest z ostatniej 103 iteracji. Całkowita funkcja straty, która jest interpolacją liniową straty funkcji wartości i polityki, minimalizowała wartość przez cały okres treningu uzyskując końcowy poziom poniżej progu 1.8 co może wskazywać, że agent dostosował swoją strategię dla tej gry. Na koniec wykonałem ewaluacje agenta poprzez rozegranie 5 meczów najlepszego modelu przeciwko jego poprzednim wersjom. Każdy mecz składał się z 50 partii, wyniki są przedstawione w tabeli poniżej:

Tab. 8. Wyniki ewaluacji najlepszego agenta AlhpaZero z 103 **iteracji treningowej w środowisku Blokus** $7x7$ **w meczach przeciwko swoim poprzednim wersją**

Przeciwnik	Wygrane najlepszego modelu	Remisy	Wygrane przeciwnika
losowy	40	7	3
iteracja 3	43	4	4
iteracja 27	14	35	1
iteracja 64	8	36	6
iteracja 90	8	37	5

Źródło: Badania własne.

Stała wysoka liczba remisów, szczególnie na późniejszych etapach, podkreśla iteracyjne udoskonalanie procesu decyzyjnego modelu. Wyniki te sugerują konwergencję w kierunku solidnej strategii, ponieważ ostatnia iteracja modelu nie przewyższyła znacząco swoich poprzedników. Wskazuje to na plateau uczenia się, co sugeruje, że zdolności modelu do rozgrywki w środowisku Blokus osiągnął większość swojego potencjału.

Po udanych próbach w mniejszej wersji środowiska Blokus $7x7$ nadszedł czas na przetestowanie AlphaZero w pełnej wersji dla czterech graczy i planszy $20x20$. Przestrzeń akcji w tym środowisku jest ponad 33 razy większa i wynosi 30433. Musiałem więc dostosować implementacje algorytmu AlphaZero do czterech graczy, ponieważ pierwotnie zakładała ona gry dwuosobowe. Zaaplikowałem zmiany w mechanizmie samodzielnej gry, wyjściu głowy funkcji wartości oraz formacie obserwacji jaki był wejściem do modelu, zaproponowane w artykule "Multilingual AlphaZero"[17]. W celu przyśpieszeniu procesu zbierania danych skorzystałem z implementacji środowiska Blokus $20x20$ dostępnej w pakiecie colosseumrl[24]. Wraz ze znacznym wzrostem skomplikowania środowiska potrzebne było zwiększenie możliwości modelu przez zastosowanie bardziej rozbudowanej architektury i amplifikacji parametrów. W tym celu wykorzystałem architekturę podobną do sieci ResNet[8], który składał się z zestawionych bloków dwuwymiarowych warstw konwolucyjnych przeplatanych Normalizacją Baczową[11] i Poprawioną Funkcją Liniową (patrz rysunek nr 19). Model składał się z 5 takich bloków i na wyjściu miał dwie głowy modelujące jedną dla polityki i drugą dla funkcji wartości. Rozmiar warstwy liniowej dla polityki był równy przestrzeni stanów 30433, a dla wartości ilość neuronów wyjściowych była równa 4 co odpowiadało liczbie graczy. Łączna ilość parametrów modelu wynosi $24.28M$. Obserwacją wejściową do modelu była macierz o rozmiarach planszy $20x20$ z 8 kanałami. Pierwsze 4 kanały odpowiadały stanowi zajętych pól przez konkretnego gracza, natomiast warstwy od $5-8$ były wyznacznikami tego kto wykonał ostatni ruch. Przed przystąpieniem do właściwego treningu wykonałem optymalizację hiper-parametrów korzystając z pakietu Optuna[1]. Zebrałem dane ze środowiska wykonując 10 epizodów samodzielnej gry, gdzie każdy z nich składał się z 100 gier, a ilość symulacji MCTS wynosiła 25. Na podstawie zebranych próbek wykonałem 100 iteracji optymalizacji hiper-parametrów otrzymując następujące wartości:

- **Współczynnik uczenia**: $5 * 10^{-5}$
- **Rozmiar partii**: 64
- **Liczba bloków modelu**: 5
- **Epoki**: 40

Następnie przystąpiłem do właściwej części treningu agenta. Rozegrał on 100 epizodów samodzielnej gry, z zwiększoną do 200 ilością symulacji MCTS, gdzie w poprzednim eksperymencie wartość ta była ustawiona na 25. Faza zbierania danych trwała około 60 godzin i zostało zgromadzonych 5600 próbek. Następnie przeprowadziłem trening modelu na tych danych z hiper-parametrami otrzymanymi w procesie optymalizacji. Wartość funkcji straty stabilnie zbiegła do poziomu 4.6 co zostawia jeszcze duże pole do spadków wraz z wzrostem ilości danych treningowych. Na koniec wykonałem ewaluacje agenta przeciwko modelowi bazowemu jakim był agent oparty na Przeszukiwaniu Drzewa Monte Carlo z rozkładem równomiernym jako politykę wyboru ruchu. Model AlphaZero i trzech przeciwników MCTS rozegrali 24 areny kontrolne, gdzie każdy z nich miał ustawioną wartość symulacji MCTS na 200. W ostatecznym rozrachunku AlphaZero zdobył 24 punkty, przeciwko -8 punktom każdego z agentów MCTS, gdzie

pojedyncza wygrana dawała 3 punkty, w przypadku remisu dwóch graczy otrzymywało po 1 punkcie, a przegrani dostawali -1 punkt.

W swojej pracy przeprowadziłem szereg eksperymentów związanych z uczeniem ze wzmocnieniem w szczególności dla planszowej gry strategicznej Blokus. Skupiłem się na dwóch algorytmach: Proksymalnej Optymalizacji Polityki oraz AlphaZero. Wyniki wykazały, że PPO ma problem z dostosowaniem swojej taktyki dla tego środowiska, nie będąc w stanie przewyższyć strategii losowania ruchów. Sam proces treningu był bardzo niestabilny z brakiem wyraźnej konwergencji funkcji wartości do minimum co po kilkudziesięciu tysiącach kroków poskutkowało eksplodującym gradientem i zakończeniem treningu. Przyczyną tego mogła być zbyt mała wartość informacji jaką algorytm otrzymywał od środowiska, dostając tylko epizodyczną nagrodę oraz brak przeprowadzania symulacji przed podjęciem decyzji o kolejnej akcji. Eksperymenty z AlphaZero wykazały, że algorytm ten ma potencjał na osiągnięcie bardzo wysokiego poziomu w grze Blokus. W wersji środowiska dla dwóch graczy AlphaZero wykazał silny iteracyjny przyrost poziomu gry i znacząco pokonał model bazowy z równomiernym rozkładem prawdopodobieństwa ruchów oraz swoje wersje z wcześniejszych iteracji. Nawet w znacznie rozszerzonej wersji środowiska o wielkości planszy $20x20$ i 4 graczami algorytm ten wykazał, że pomimo niewielkiej ilości próbek treningowych jest w stanie zoptymalizować swoją strategię pokonując model bazowy, którym była symulacja MCTS z równomiernym rozkładem prawdopodobieństwa ruchów. Tym samym zrealizowałem założenie pracy jakim było stworzenia algorytmu uczenia ze wzmocnieniem dla gry Blokus, w skali w jakiej pozwalały na to dostępne zasoby obliczeniowe.

Wyniki eksperymentów przeprowadzone z PPO oraz AlphaZero w kontekście gry planszowej Blokus prezentują potencjał do wykorzystania w przyszłych badaniach. Kluczowe obszary do zbadania obejmują:

- **Zwiększone zasoby obliczeniowe**: Wykorzystanie bardziej zaawansowanych zasobów obliczeniowych może znacznie poprawić wydajność modelu, potencjalnie umożliwiając uzyskanie nowych strategicznych spostrzeżeń i wyższego poziomu gry.
- **Doskonalenie algorytmów**: Dalsze badania i udoskonalanie, w szczególności poprzez dostosowywanie hiper-parametrów i badanie alternatywnych architektur sieci neuronowych.
- **Wykorzystanie następnika AlphaZero**: Zaadaptowanie algorytmu MuZero[21] następnika AlhpaZero, którego możliwości wychodzą poza gry z doskonałą informacją.

Odkrycia te mają jednak też implikacje wykraczające poza grę Blokus. Wyzwania które występowały podczas szkolenia PPO i AlphaZero dla Blokusa są zbliżone do tych w różnych rzeczywistych aplikacjach. Na przykład, spostrzeżenia uzyskane z niestabilności treningu PPO i sukcesu AlphaZero w złożonych środowiskach mogą informować o podejściach w takich dziedzinach, jak robotyka, gdzie sztuczna inteligencja musi poruszać się w nieprzewidywalnych środowiskach, lub w modelowaniu finansowym, gdzie

podejmowanie strategicznych decyzji ma kluczowe znaczenie. Co więcej, zdolność AlphaZero do przystosowania się do wieloosobowego środowiska na dużą skalę ma potencjalne zastosowanie w logistyce i zarządzaniu łańcuchem dostaw, gdzie koordynacja wieloma jednostkami jest niezbędna.

Badania podkreślają znaczenie uwzględnienia wartości informacyjnej dostarczanej algorytmom uczącym się i wpływu strategicznego podejmowania decyzji, które są podstawowymi pojęciami w wielu zastosowaniach sztucznej inteligencji. Tak więc, chociaż głównym celem tej pracy było stworzenie algorytmu uczenia ze wzmocnieniem dla Blokusa w ramach ograniczeń dostępnych zasobów obliczeniowych, metodologie i zebrane spostrzeżenia mają szersze zastosowanie, oferując cenny wkład w dziedzinę sztucznej inteligencji i jej praktyczne wykorzystanie w różnych dziedzinach.

Bibliografia

[1] Takuya Akiba i in. “Optuna: A Next-generation Hyperparameter Optimization Framework”. W: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[2] Peter Auer. “Using Confidence Bounds for Exploitation-Exploration Trade-offs.” W: *Journal of Machine Learning Research* 3 (sty. 2002), s. 397–422. DOI: 10.1162/153244303321897663.

[3] Richard Bellman. *Dynamic Programming*. 1 wyd. Princeton, NJ, USA: Princeton University Press, 1957.

[4] Y. Bengio, P. Simard i P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. W: *IEEE Transactions on Neural Networks* 5.2 (1994), s. 157–166. DOI: 10.1109/72.279181.

[5] Merim Bilalic, Peter McLeod i Fernand Gobet. “Does chess need intelligence? — A study with young chess players”. W: *Intelligence* 35 (wrz. 2007), s. 457–470. DOI: 10.1016/j.intell.2006.09.005.

[6] Cameron B. Browne i in. “A Survey of Monte Carlo Tree Search Methods”. W: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), s. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[7] Mogens Dalgaard i in. “Global optimization of quantum dynamics with AlphaZero deep exploration”. W: *npj Quantum Information* 6.1 (sty. 2020). ISSN: 2056-6387. DOI: 10.1038/s41534-019-0241-0. URL: http://dx.doi.org/10.1038/s41534-019-0241-0.

[8] Kaiming He i in. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[9] Matteo Hessel i in. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. 2017. arXiv: 1710.02298 [cs.AI].

[10] Shengyi Huang i in. “CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms”. W: *Journal of Machine Learning Research* 23.274 (2022), s. 1–18. URL: http://jmlr.org/papers/v23/21-1342.html.

[11] Sergey Ioffe i Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[12] Diederik P Kingma i Jimmy Ba. “Adam: A method for stochastic optimization”. W: *arXiv preprint arXiv:1412.6980* (2014).

[13] Y. Lecun i in. “Gradient-based learning applied to document recognition”. W: *Proceedings of the IEEE* 86.11 (1998), s. 2278–2324. DOI: 10.1109/5.726791.

[14] Volodymyr Mnih i in. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. arXiv: 1602.01783 [cs.LG].

[15] OpenAI i in. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG].

[16] Long Ouyang i in. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL].

[17] Nick Petosa i Tucker Balch. *Multiplayer AlphaZero*. 2019. arXiv: 1910.13012 [cs.AI].

[18] Marius-Constantin Popescu i in. "Multilayer perceptron and neural networks". W: *WSEAS Transactions on Circuits and Systems* 8 (lip. 2009).

[19] David E Rumelhart, Geoffrey E Hinton i Ronald J Williams. "Learning representations by back-propagating errors". W: *nature* 323.6088 (1986), s. 533–536.

[20] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". W: *Neural Networks* 61 (sty. 2015), s. 85–117. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2014.09.003. URL: http://dx.doi.org/10.1016/j.neunet.2014.09.003.

[21] Julian Schrittwieser i in. "Mastering Atari, Go, chess and shogi by planning with a learned model". W: *Nature* 588.7839 (grud. 2020), s. 604–609. ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: http://dx.doi.org/10.1038/s41586-020-03051-4.

[22] John Schulman i in. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[23] John Schulman i in. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[24] Alexander Shmakov i in. *ColosseumRL: A Framework for Multiagent Reinforcement Learning in N-Player Games*. 2019. arXiv: 1912.04451 [cs.MA].

[25] David Silver i in. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". W: *Science* 362.6419 (2018), s. 1140–1144. URL: http://science.sciencemag.org/content/362/6419/1140/tab-pdf.

[26] David Silver i in. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[27] Richard S Sutton i in. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". W: *Advances in Neural Information Processing Systems*. Red. S. Solla, T. Leen i K. Müller. T. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

[28] Shantanu Thakoor, Surag Nair i Megha Jhunjhunwala. *Learning to play othello without human knowledge*. 2016.

[29] The Stockfish developers (see AUTHORS file). *Stockfish*. URL: https://github.com/official-stockfish/Stockfish.

[30] Chun-Hung Tzeng i Paul Purdom. “A Theory of Game Trees.” W: sty. 1983, s. 416–419.

[31] Ziyu Wang i in. *Sample Efficient Actor-Critic with Experience Replay*. 2017. arXiv: 1611.01224 [cs.LG].

Spis rysunków

Spis tabel

Spis algorytmów

Załączniki

1. Płyta CD/DVD zawierająca:
   a) Prezentację wyników pracy dyplomowej
2. Repozytorium z kodem źródłowym:
   a) Projekt jest dostępny pod adresem:
   `https://github.com/KubiakJakub01/Blokus-RL`
   b) Instalacja oraz sposób użycia został opisany w pliku README w głównym folderze repozytorium