

**Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie**

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

KATEDRA AUTOMATYKI I ROBOTYKI



PRACA INŻYNIERSKA

FILIP GACEK

**WIRTUALNY SILNIK SZACHOWY Z WYKORZYSTANIEM
UCZENIA MASZYNOWEGO ZE WZMOCNIENIEM**

PROMOTOR:

dr hab. inż. Joanna Jaworek-
Korjakowska, prof. AGH

Kraków 2023

AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical
Engineering

DEPARTMENT OF AUTOMATIC CONTROL AND ROBOTICS



BACHELOR THESIS

FILIP GACEK

**VIRTUAL CHESS ENGINE WITH REINFORCEMENT
LEARNING**

SUPERVISOR:

Joanna Jaworek-Korjakowska,
PhD, Univ. Prof.

Krakow 2023

Serdecznie dziękuję Promotorowi dr hab. inż. Joannie Jaworek-Korjakowskiej oraz mgr. inż. Aleksandrowi Kostuch za nadzór i pomoc w przygotowaniu pracy, bratu i rodzicom za wsparcie oraz inż. Wojtkowi Woszczkowi i inż. Szymonowi Majewskiemu za wzajemną motywację w procesie tworzenia pracy.

Spis treści

1. Wprowadzenie	6
1.1. Cele pracy	6
2. Silniki szachowe w ujęciu teoretycznym	8
2.1. Szachy i ich historia	8
2.2. Silniki szachowe	10
2.3. Przeszukiwanie drzewa gier metodą Monte Carlo	12
2.4. Uczenie maszynowe	13
2.5. Uczenie ze wzmocnieniem	14
2.5.1. Równania Bellmana	18
2.5.2. Metoda Q-learning	19
2.6. Głębokie uczenie maszynowe ze wzmocnieniem	20
2.6.1. Sieci neuronowe	20
2.6.2. Głębokie sieci neuronowe	21
2.6.3. Głębokie uczenie ze wzmocnieniem	23
2.7. AlphaZero	24
2.8. Miary jakości silnika szachowego	26
2.8.1. Metoda obliczania relatywnej siły gry szachistów	26
2.8.2. Sposoby mierzenia siły gry silników	26
3. Implementacja silnika szachowego	27
3.1. Implementacja	27
3.1.1. Stworzenie logiki gry i implementacja zasad gry	27
3.1.2. Funkcje oceniające pozycję szachową	28
3.1.3. Algorytm poszukujący najlepszy ruch	28
3.1.4. Program szachowy	29
3.2. Architektura	30
3.2.1. Model sieci neuronowej	30
3.2.2. Kodowanie i dekodowanie pozycji szachowej	30
3.3. Proces uczenia silnika	31

3.4.	Aplikacja internetowa.....	36
3.5.	Interfejs graficzny	36
3.6.	Komunikacja z serwerem	36
4.	Badania eksperymentalne	38
4.1.	Porównanie z innymi silnikami	39
4.2.	Wnioski.....	40
5.	Podsumowanie	41
5.1.	Potencjalne możliwości rozwoju pracy	41

1. Wprowadzenie

W 1943 roku, ówczesny prezes firmy IBM Thomas J. Watson wypowiedział następujące słowa: Na światowych rynkach jest, jak sędzę, miejsce dla może pięciu komputerów. Z perspektywy czasu bardzo łatwo osądzić, że całkowicie się pomylił. Obecnie komputery znajdują się w prawie każdym domu, instytucji, czy przedsiębiorstwie. Jednak słowa Thomasa powinny zmusić do refleksji i zastanowienia nad postrzeganiem przyszłości. Aktualnie kluczową rolę w rozwoju technologii odgrywa sztuczna inteligencja. Oddziałuje ona na każdy aspekt współczesnego życia. Translacja tekstu z wykorzystaniem usługi Tłumacz Google, reklamy dobierane w oparciu o nasze zainteresowania, odblokowanie ekranu w telefonie za pomocą funkcji wbudowanej FaceID, czy predykcja wartości akcji spółki giełdowej to tylko garstka zastosowań, które w ciągu ostatnich kilkunastu lat zostały z sukcesem wprowadzone na rynek.

Na początku 2013 roku został opublikowany artykuł "Playing Atari with Deep Reinforcement Learning"[14], który stanowił początek rozwoju nowej gałęzi sztucznej inteligencji. Omawia on zastosowanie głębokiego uczenia przez wzmacnianie w stworzeniu inteligentnych botów w kultowych grach zręcznościowych na konsolę Atari. W publikacji znalazło się szerokie porównanie wyników uzyskiwanych przez komputer i przeciętnego gracza. W wielu grach komputer deklasował człowieka. Wydawać by się mogło, że to bardzo błahy temat, niemniej jednak sprawił on, że świat naukowy dostrzegł potencjał i mnogość zastosowań głębokiego uczenia przez wzmacnianie, które przy obecnych zasobach obliczeniowych jest w stanie rozwiązać większość skomplikowanych problemów, z którymi nie radzi sobie głębokie uczenie maszynowe lub inne algorytmy. Kolejny kamieniem milowym było stworzenie programu komputerowego do gry w Go - popularnej gry planszowej, o stopniu skomplikowania przewyższającym szachy - o nazwie Alpha Go przez angielski start-up Deepmind. W 2015 roku Alpha Go po raz pierwszy pokonał zawodowego gracza, a w marcu 2016 roku rozegrał 5 partii z najlepszym zawodnikiem Go na świecie. Mecz cieszył się ogromną popularnością w Azji i przyciągnął ogromne zainteresowanie mediów, a transmisję obejrzało około 64 mln ludzi. Alpha Go wygrał cztery razy, co spowodowało szok, a zarazem ekscytację. Przed meczem, każdy spisywał go na straty. Od tego momentu powoli zaczynało stawać się coraz bardziej oczywistym, że głębokie uczenie przez wzmacnianie odegra kluczową rolę w rozwoju ludzkości.

1.1. Cele pracy

Celem poniższej pracy jest stworzenie wirtualnego silnika szachowego na podstawie artykułu opisującego AlphaZero [20], potrafiącego rywalizować z ludźmi oraz innymi silnikami, a także stworzenie

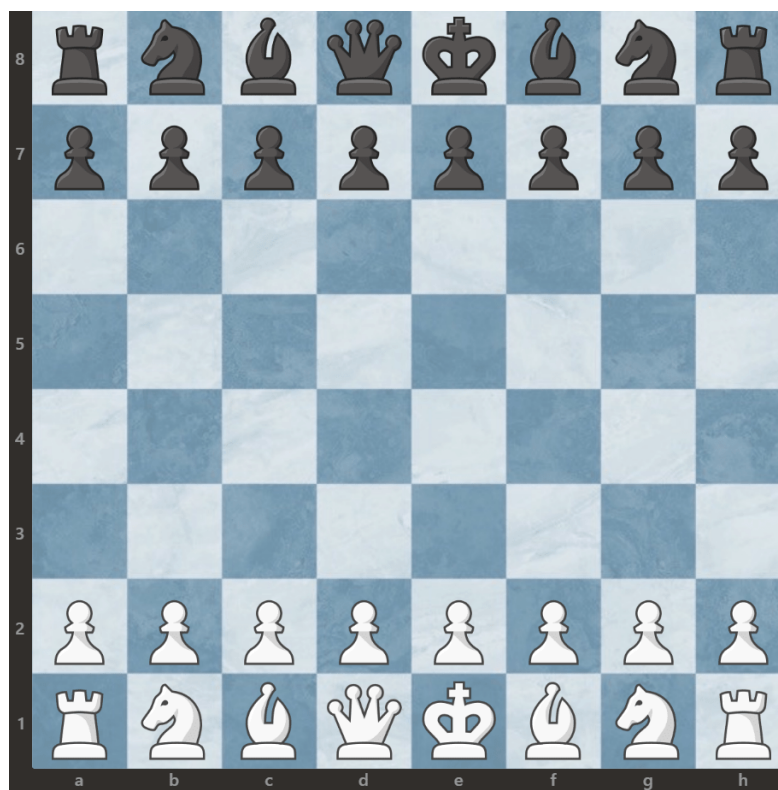
aplikacji internetowej, która umożliwi użytkownikom grę w szachy online. Obecnie istnieje już bardzo wiele różnych implementacji szachowych silników. Wykorzystują one różne rodzaje technik oraz posiadają zróżnicowany stopień zaawansowania metod *sztucznej inteligencji*. Od przeglądu zupełnego (z ang. brute force), przez sieci neuronowe (z ang. neural networks), po głębokie uczenie (ang. deep neural networks) przez wzmacnianie (z ang. deep reinforcement learning). Celem poniższej pracy było wykorzystanie najbardziej aktualnych rozwiązań, czyli właśnie metod głębokiego uczenia przez wzmacnianie. Wybór motywowany był ogromnym rozwojem tej dziedziny i przełomowymi wydarzeniami mającymi miejsce w ostatnich latach, które ożywiły naukowców i inżynierów na całym świecie do tworzenia coraz bardziej wydajniejszych algorytmów uczenia przez wzmacnianie oraz poszerzania obszarów, w jakich można je wykorzystać. Dodatkowo autor pracy jest aktywnym szachistą i posiada wiedzę teoretyczną z dziedziny gry w szachy, co niewątpliwie ułatwi zrozumienie głębokiego uczenia przez wzmacnianie w tej właśnie grze.

2. Silniki szachowe w ujęciu teoretycznym

2.1. Szachy i ich historia

Szachy to jedna z najstarszych planszowych gier strategicznych, ciesząca się popularnością na całym świecie. Historycy uważają, że powstała ok. V-VI wieku w Indiach, skąd miała się przenieść do świata arabskiego, a następnie do Europy. Na przestrzeni lat reguły gry ewoluowały i w swoich początkach znacząco się różniły od znanego nam dzisiaj formatu. Obecne reguły zostały przyjęte w XV-XVI wieku, a od tego czasu wprowadzano tylko subtelne zmiany, głównie w sposobie rozgrywania turniejów.

W grze uczestniczy dwóch zawodników. Na początku wybierany jest kolor bierek, jedna ze stron otrzymuje czarne, a druga białe i to właśnie ona uzyskuje przywilej wykonania pierwszego ruchu. Partia jest rozgrywana na kwadratowej planszy o wymiarach 8x8 zwanej szachownicą.



Rysunek 2.1: Rozpoczynające ustawienie bierek na szachownicy.

Reguły gry

Zawodnicy wykonują ruchy naprzemiennie. Każda z bierek posiada określony zakres ruchów, opisany w tabeli 2.1. Partia może zakończyć się następującymi rezultatami:

Zwycięstwo i przegrana

Zawodnik wygrywa partię w następujących sytuacjach:

- Król przeciwnika jest szachowany i nie jest w stanie wykonać żadnego ruchu sprawiającego, że król uwolni się spod szachu.
- Przeciwnikowi skończy się czas.
- Przeciwnik podda się.

Remis

Partia kończy się remisem w następujących sytuacjach:

- Zawodnicy zgodzą się na remis.
- Dojdzie do trzykrotnego powtórzenia pozycji.
- Na szachownicy znajdują się bierki, którymi nie jesteśmy w stanie doprowadzić do zamatowania jednego z króli.
- Jeśli zawodnicy wykonają 50 ruchów bez pchnięcia pionka do przodu lub zbitcia figury.





Bierka	Hetman	Król	Wieża	Goniec	Skoczek	Pion
Ilość	1	1	2	2	2	8
Wygląd						
Zakres Ruchów	wiele pól przekątne, poziomo, pionowo	jedno pole przekątne, poziomo, pionowo	wiele pól, poziomo, pionowo	wiele pól, przekątne	dwa pola pionowo/poziomo + jedno pole poziomo/pionowo	jedno lub dwa pola do przodu
Wartość	9 (8.8)	3.5 (3.2)	5 (5.1)	3 (3.2)	3 (3.33)	1 (1)

Tabela 2.1: Tabela figur szachowych. Wartości w nawiasach odnoszą się do systemu wartości zaproponowanego przez Hansa Berlinera [3].

Szachy to teoretycznie gra deterministyczna. Z uwagi na fakt, że wszystkie ruchy są jawne, zawodnicy mają pełny wgląd na sytuację na szachownicy więc mają możliwość przeanalizować każdy z możliwych wariantów dalszej rozgrywki. Jednak w praktyce obliczenie wszystkich możliwych wariantów jest niemożliwe przez ich ogromną ilość. Szacuje się, że liczba wszystkich możliwych przebiegów partii szachów wynosi około 10^{120} [18]. Dla porównania szacowana liczba atomów we wszechświecie wynosi około 10^{80} (warto tutaj zaznaczyć, że liczba partii, które mają sens tzn. z odrzuceniem ruchów bezsensownych to około 10^{40}). Tak ogromne liczby sprawiają, że w większości przypadków nie jest możliwe

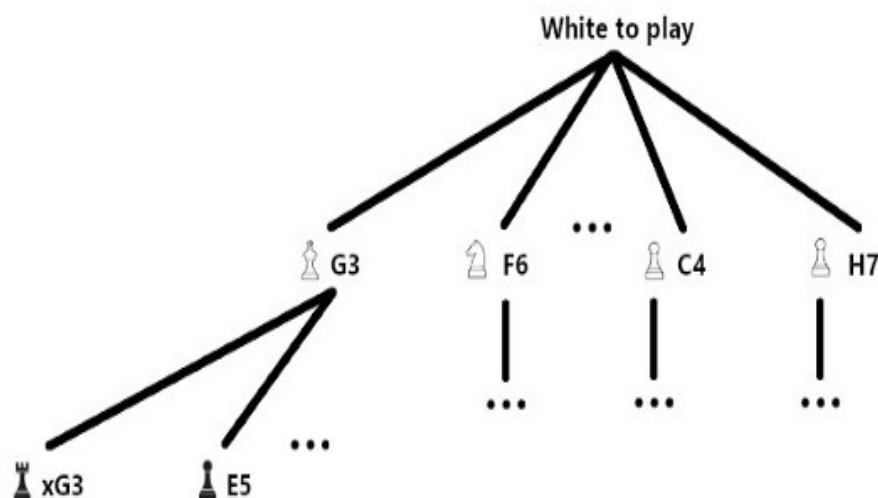
jednoznaczne określenie konkretnej pozycji szachowej, a jedynie jej estymacja. Dlatego też, dzięki swojej złożoności, szachy na przestrzeni lat były środowiskiem, w którym próbowano wykorzystać złożone algorytmy, aby stworzyć program, który osiągnie jak najlepsze wyniki.

2.2. Silniki szachowe

Powstanie silników szachowych zrewolucjonizowało szachy. Obecnie w zwykłym telefonie jesteśmy w stanie zainstalować program komputerowy, który bez problemu pokonałby najsilniejszych szachistów na naszej planecie. Historia silników szachowych zaczyna się w latach 1948, kiedy to Claude Shannon i Alan Turing - jedne z kluczowych postaci w historii rozwoju komputerów - pracowali nad stworzeniem pierwszego programu komputerowego potrafiącego samodzielnie grać w szachy. Pierwszy z nich w 1949 roku opublikował artykuł "Programming a Computer For Playing Chess" [18], natomiast drugi w 1950 roku stworzył "Turbochamp". O geniuszu Turinga może świadczyć fakt, że stworzył swój program 'na papierze', bez dostępu do komputera.

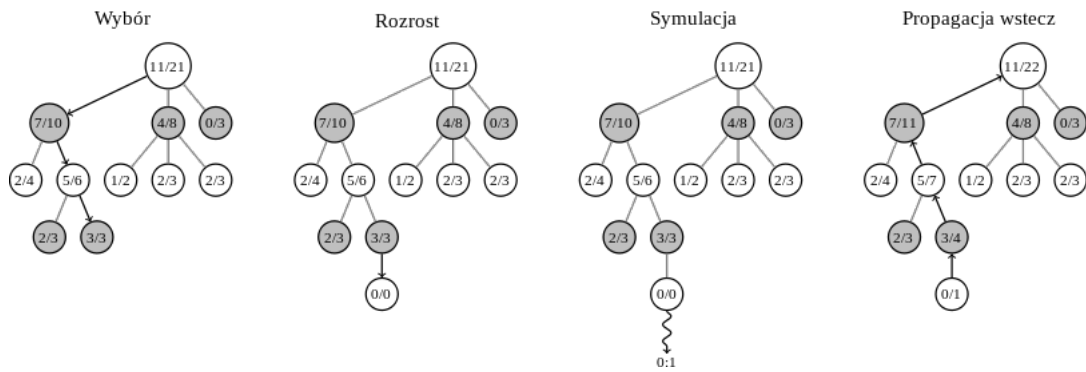
Mimo geniuszu obydwu naukowców, były to początki ery komputerów, a silniki wykorzystywały proste algorytmy i w żadnym wypadku nie były w stanie osiągnąć poziomu topowych szachistów. Kolejny przełom nastąpił w latach 60. i 70. XX wieku za sprawą Johna von Neumanna. Stworzył on algorytm Min-Max, idealnie nadający się do gry w szachy. Min-Max to algorytm rekursywny, używany do znalezienia optymalnego ruchu, zakładając, że przeciwnik też wykonuje optymalne ruchy. Również możliwości obliczeniowe komputerów pozwalały już na użycie bardziej zaawansowanych algorytmów. Tym sposobem w 1970 roku w Nowym Jorku rozegrano pierwsze mistrzostwa świata w szachach dla programów komputerowych. Zwycięzcą okazał się Chess 3.0 autorstwa Larrego Atkina i Davida State'a, zaprogramowany w językach Fortran IV oraz COMPASS. W tym okresie silniki szachowe były już na tyle dobre, aby pokonać amatorów, lecz nadal nie miały szans w starciu z profesjonalnymi graczami. W latach 80. XX wieku programowanie silników szachowych stało się prawdziwą rywalizacją. Profesor informatyki Edward Fredkin, ogłosił nagrody pieniężne: 5000 \$ dla pierwszego silnika, który osiągnie poziom mistrzowski, 10 000 \$ za osiągnięcie poziomu arcymistrzowskiego oraz 100 000\$ za pokonanie mistrza świata. Programiści na całym świecie rozpoczęli walkę o zbudowanie najsilniejszego silnika. Nie chodziło tylko i wyłącznie o pieniądze. Sam fakt stworzenia programu potrafiącego pokonać arcymistrzów był swego rodzaju kamieniem milowym w dziejach ludzkości, ponieważ przez długi okres uważano, że komputery nigdy nie będą w stanie prześcignąć najlepszych szachistów na świecie. W tym momencie coraz bardziej klarownym stawał się fakt, że silniki szachowe w najbliższej przyszłości osiągną wyższy poziom niż ludzkość. Jednak na ten moment trzeba było jeszcze poczekać. W 1985 roku Garry Kasparov, jeden z najwybitniejszych szachistów w całej historii, rozegrał partie z 32 najsilniejszymi silnikami szachowymi i pokonał wszystkie z nich. Moment dominacji komputerów jeszcze nie nadszedł. W latach 90. XX wieku miał dokonać się przełom. Wszystko za sprawą projektu Deep Blue utworzonego w 1989 roku przez IBM. W 1996 roku pierwszy raz komputer wygrał partię z mistrzem świata w szachach klasycznych. Mimo to całe spotkanie zostało wygrane przez Kasparova cztery do dwóch. Przegrana w spotkaniu nie tylko nie zniechęciła programistów z Deep Blue, ale jeszcze bardziej napędziła rozwój projektu. W następnym roku zorganizowano kolejne spotkanie, z Garrym Kasparovem

i jeszcze silniejszą wersją silnika. Tym razem Deep Blue okazał się lepszy i w sześciu partiach pokonał Kasparova $3\frac{1}{2}$ do $2\frac{1}{2}$ i otrzymał nagrodę Friedkina w wysokości 100 000\$. Rok 1997 został symbolicznie zapamiętany w historii ludzkości jako koniec ery dominacji ludzi w dziedzinie szachów. Wydawać by się mogło, że po 1997 roku silniki szachowe całkowicie zdominują ludzi, jednak w latach 1997-2003 nie odnosiły znaczących sukcesów. Dopiero lata 2003 - 2006 całkowicie przesądziły o przegranej ludzkości. Kolejno, w 2004 silnik szachowy Hydra pokonał arcymistrza Evgeniego Vladimirova 3-1 oraz Ruslana Ponomariova 2-0, a w 2006 roku Deep Fritz pokonał ówczesnego mistrza świata Vladimira Kramnika cztery do dwóch. Od tego momentu silniki całkowicie zrewolucjonizowały szachowy świat. W transmisjach rozgrywek szachowych zaczęto oceniać pozycje zgodnie ze wskazaniem silnika, zawodnicy zaczęli trenować i ćwiczyć taktyki w oparciu o komputerowe analizy, co niewątpliwie wpłynęło też na zwiększenie poziomu gry wśród ludzi. Coraz częściej zaczęto organizować turnieje wyłącznie dla silników szachowych. Firmy z całego świata rywalizowały ze sobą w stworzeniu jak najlepszego silnika szachowego. Większość z nich działała (i nadal działa) w podobny sposób. *Tak jak skrzydła samolotów wzbijając się w powietrze nie naśladują ruchu ptaków, tak silniki szachowe nie generują ruchów w sposób w jaki ludzie rozumieją szachy* - Garry Kasparov. Na początku program tworzy abstrakcyjną reprezentację ustawienia bierek na szachownicy. Następnie iterując po dozwolonych ruchach, dla każdego z nich tworzy drzewo decyzyjne do określonej głębokości, by na samym końcu wybrać najlepszy z nich.



Rysunek 2.2: Drzewo decyzyjne silnika szachowego [4].

Jednym z kluczowych graczy na rynku został Stockfish autorstwa Torda Romstada, Marca Costalba i Joona Kiiski, a który obecnie rozwijany jest przez grono pasjonatów z Stockfish Community jako oprogramowanie o otwartym kodzie źródłowym. Do 2017 roku to właśnie on mógł szczycić się mianem najmocniejszego szachowego programu komputerowego. W tym właśnie roku na scenę szachową wkroczyli naukowcy z zespołu DeepMind, którzy stworzyli AlphaZero Chess - pierwszy silnik szachowy wykorzystujący metodę głębokiego uczenia przez wzmacnianie, bazujący na algorytmach AlphaGo Zero



Rysunek 2.3: Zasada działania przeszukiwanie drzewa gier metodą Monte Carlo [1].

[20, 21]. Alpha Zero wyróżniał się kreatywnością i innowacyjnym podejściem do prowadzenia partii. Z uwagi na to, że uczył się grając sam ze sobą, znajdował nieznane dotąd zagrania i wybierał nieszablonowe strategie, które szokowały szachowy świat.

2.3. Przeszukiwanie drzewa gier metodą Monte Carlo

Przeszukiwanie drzewa gier metodą Monte Carlo (ang. Monte Carlo Tree Search, MCTS) to algorytm heurystyczny wykorzystywany w zadaniach sztucznej inteligencji. MCTS tworzy drzewo decyzyjne i skupia się na analizie najbardziej obiecujących ruchów, odrzucając te gorsze. Rozrost drzewa wariantów opiera na losowym próbkowaniu przestrzeni poszukiwań. Pełna metoda MCTS polega na rekurencyjnym wykonywaniu symulacji na wielu poziomach głębokości drzewa wariantów. Każda tura składa się z następujących kroków [1]:

- Wybór: Startując z korzenia drzewa R , wybieraj kolejne węzły potomne, do momentu dojścia do liścia drzewa L .
- Rozrost: Jeśli L nie kończy gry, to utwórz w nim jeden lub więcej węzłów potomnych i wybierz jeden z nich C .
- Symulacja: Rozegraj losową symulację z węzła C .
- Propagacja wstecz: Zgodnie z rozegraną symulacją, zaktualizuj informacje o liczbie zwycięstw lub porażek w wykonanej symulacji.

Każdy z wierzchołków drzewa powinien zawierać informacje o estymowanej wartości na bazie przeprowadzonych symulacji oraz liczbę odwiedzin wierzchołka. Bardziej efektywniejszym rozszerzeniem algorytmu Monte Carlo jest inteligentne próbkowanie. Zamiast losowego wybierania ruchów, faworyzowane są ruchy, które okazały się najlepsze. Oznacza to, że na początku ruchy wykonywane są zupełnie losowo, lecz z czasem wybierane są posunięcia, które okazały się zwycięskie w poprzednich symulacjach. Podejście to opisuje poniższa formuła:

$$UCT = \overline{wynik_i} + c * \sqrt{\frac{n}{n_i}} \quad (2.1)$$

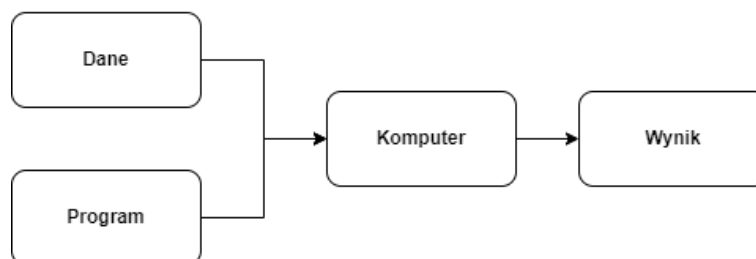
gdzie:

- $\overline{wynik_i}$: dotychczasowy średni wynik i -tego ruchu,
- c : parametr odpowiadający za to, jaką część czasu algorytm poświęci na eksplorację nieznanych ruchów. Domyślna wartość to $\sqrt{2}$,
- n : liczba rozegranych symulacji,
- n_i : liczba symulacji rozegranych na i -tym ruchu.

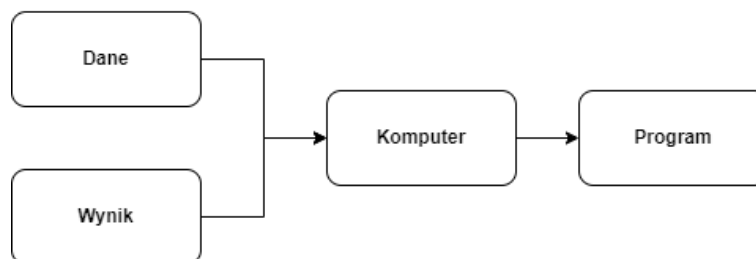
2.4. Uczenie maszynowe

W ciągu ostatnich kilkudziesięciu lat miał miejsce gwałtowny rozwój sztucznej inteligencji, a nasza cywilizacja coraz częściej określana jest mianem społeczeństwa informacyjnego. Mimo różnych definicji tego terminu jego przesłanie pozostaje niezmiennie – świat znajduje się w momencie, w którym informacja i dane są jego najcenniejszym dobrem. To właśnie ogromna ilość dostępnych danych i przyrost mocy obliczeniowej sprawił, że jedną z najprężniej rozkwitających gałęzi sztucznej inteligencji stało się uczenie maszynowe. Wykorzystuje ono dane i algorytmy do naśladowania sposobu w jaki uczą się ludzie, dlatego często pozwala w prosty sposób rozwiązać skomplikowane problemy i jest efektywniejszym rozwiązaniem od tradycyjnego programowania.

Tradycyjne programowanie



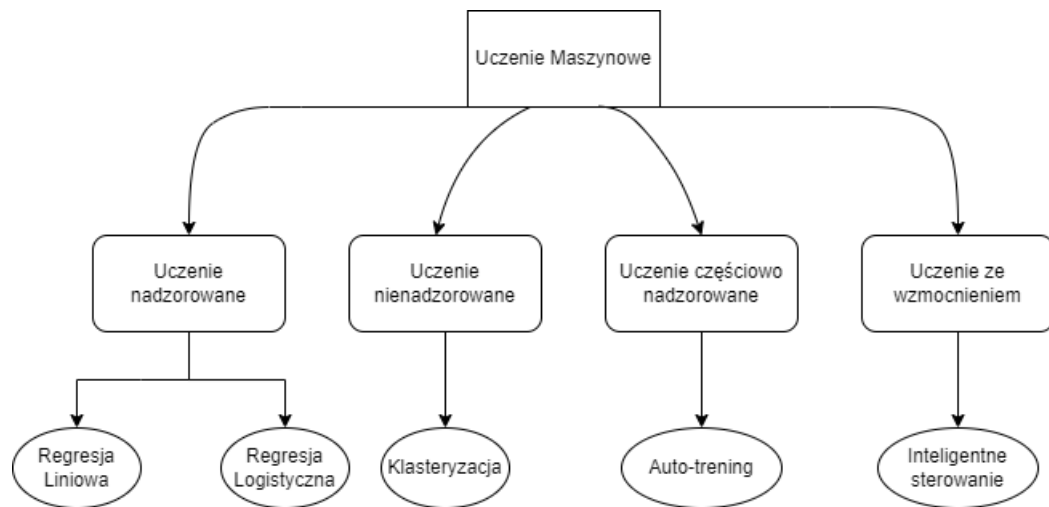
Uczenie maszynowe



Rysunek 2.4: Uczenie maszynowe w porównaniu z tradycyjnym programowaniem.

Dzięki swojej skuteczności techniki uczenia maszynowego są wykorzystywane w niemalże każdej dziedzinie naszego życia. Autokorekta w klawiaturze, systemy rekomendacji produktów na stronach

internetowych, wyznaczanie kursów meczów w zakładach bukmacherskich, predykcja cen akcji na giełdzie, sterowanie ramieniem robota czy detekcja twarzy to tylko drobna garstka możliwych zastosowań.



Rysunek 2.5: Podział metod uczenia maszynowego ze względu na sposób uczenia modelu.

2.5. Uczenie ze wzmocnieniem

Uczenie ze wzmocnieniem to jedna z czterech głównych gałęzi uczenia maszynowego, która zajmuje się sekwencyjnymi procesami podejmowania decyzji. Koncepcja uczenia się przez wzmacnianie opiera się na kumulacji nagród i kar otrzymywanych na podstawie akcji podejmowanych przez agenta w dynamicznym lub statycznym środowisku. Proces uczenia oparty jest na procesie decyzyjnym Markowa, który stanowi matematyczną podstawę modelowania decyzji. W uczeniu przez wzmacnianie nie wykorzystuje się danych historycznych, lecz model optymalizuje swoje parametry poprzez badanie i eksplorację środowiska samemu generując dane. W grach takich jak np. szachy czy Go algorytm dysponuje nieograniczoną ilością danych, ponieważ reguły gry są jasno określone, więc jest w stanie rozgrywać partie sam ze sobą, a następnie uczyć się i doskonalić na ich podstawie. Jedynymi ograniczeniami pozostają czas i moc obliczeniowa [8].

Zagadnienie uczenia ze wzmocnieniem formalnie opisane jest jako dyskretny i stochastyczny proces, gdzie agent dokonuje interakcji ze środowiskiem w następujący sposób: Agent zaczyna w określonym stanie swojego środowiska $s_0 \in S$ zbierając początkowe obserwacje środowiska $\omega_0 \in \Omega$. Z każdym krokiem czasowym t agent musi podjąć decyzję $a_t \in A$. Wynikiem tego agent otrzymuje nagrodę $r_t \in R$, następuje przejście do następnego stanu oraz agent dokonuje obserwacji kolejnego stanu, a proces zaczyna się od nowa.

Własność Markowa

Kontrolowany proces czasu dyskretnego to łańcuch Markowa (zawierający własność Markowa) jeśli [9]:

$$P(\omega_{t+1}|\omega_t, a_t) = P(\omega_{t+1}|\omega_t, a_t, \dots, \omega_0, a_0) \quad (2.2)$$

$$P(r_t|\omega_t, a_t) = P(r_t, \omega_t, a_t, \dots, \omega_0, a_0) \quad (2.3)$$

Jeśli model posiada własność Markowa to znaczy, że jego przyszłość zależy wyłącznie od obecnej obserwacji, a więc agent nie ma potrzeby, aby analizować przeszłość, czyli jak został osiągnięty obecny stan.

Proces Decyzyjny Markowa (ang. Markov Decision Process - MDP)

Proces decyzyjny Markowa to pięcioelementowa krotka (S, A, T, R, γ) , gdzie [9]:

- S jest przestrzenią stanu.
- A jest przestrzenią akcji.
- $T: S \times A \times S \rightarrow [0,1]$ jest przekształceniem funkcji, czyli zbiorem warunkowych przekształceń prawdopodobieństwa pomiędzy stanami.
- $R: S \times A \times S \rightarrow \mathbb{R}$ jest funkcją nagrody, gdzie \mathbb{R} jest zbiorem ciągłym o możliwej nagrodzie w zakresie $R_{max} \in \mathbb{R}^+$.
- $\gamma \in [0, 1)$ jest współczynnikiem dyskontowym.

System jest całkowicie obserwowalny w MDP, co znaczy że obserwacja jest dokładnie taka sama jak stan środowiska: $\omega_t = s_t$. W każdym kroku czasowym prawdopodobieństwo ruchu do stanu s_{t+1} określone jest funkcją $T(s_t, a_t, s_{t+1})$, a nagroda przyjmuje wartość funkcji $R(s_t, a_t, s_{t+1}) \in \mathbb{R}$.

Chcąc porównać uczenie ze wzmocnieniem do uczenia nadzorowanego, można powiedzieć, że to pierwsze uczy się na zdobytym doświadczeniu, a drugie na danych historycznych i przykładach. Uczenie ze wzmocnieniem swoje piękno zawdzięcza przede wszystkim naśladowaniu ludzkiego sposobu nauki. W początkowym etapie nauki popełnia bardzo wiele błędów i wygląda niezdarnie, lecz wraz z kolejnymi próbami, stopniowo dochodzi do perfekcji. Bardzo trafnym porównaniem jest tutaj nauka chodzenia przez małe dziecko, które początkowo wykonuje bardzo nieprzemyślane i nieskoordynowane ruchy. Wraz z kolejnymi upadkami nabiera ono doświadczenia i zapamiętuje również poprawne ruchy. Na podstawie prób i błędów oraz eksploracji otaczającego je świata, po pewnym okresie czasu dochodzi do sukcesu i opanowuje umiejętność poruszania się na dwóch nogach.

Metody uczenia maszynowego			
Przez wzmacnianie	Nadzorowane	Nienadzorowane	
Dynamiczny	Statyczny	Statyczny	Proces uczenia
Dane wejściowe, nagroda	Dane wejściowe, oznaczone dane, dane historyczne	Dane wejściowe, dane historyczne	Dane treningowe
Funkcja nagrody	Funkcja straty	Funkcja dystansu	Funkcja kosztu

Rysunek 2.6: Uczenie przez wzmacnianie na tle innych sposobów uczenia maszynowego.

W uczeniu ze wzmocnieniem wyróżnia się następujące elementy tworzące jego szkielet [9]:

Agent (ang. agent)

Agent posiada zbiór możliwych do wykonania akcji, które podejmuje w każdej iteracji oraz dysponuje wiedzą i doświadczeniem zebrany w procesie uczenia. Każda z podjętych akcji oddziałuje na środowisko, w którym się znajduje. Na przykładzie szachów, agentem jest jeden z graczy.

Akcje (ang. actions)

Akcje to zbiór wszystkich możliwych operacji lub ruchów, jakie może podjąć agent w konkretnym stanie. Jest to zbiór dyskretny, a liczba możliwych akcji zależna jest od rozważanego problemu oraz może zmieniać się w zależności od stanu w jakim znajduje się agent. W szachach akcjami nazywa się ruchy, które podejmuje zawodnik.

Środowisko (ang. environment)

Środowisko to wirtualny świat w jakim znajduje się agent. Oddziałują na nie wszystkie akcje podejmowane przez agenta. Środowisko zawiera aktualny stan agenta i możliwe akcje oraz zwraca informacje o nagrodzie lub karze za wykonanie poprzez agenta poszczególnych akcji. W grze szachowej środowiskiem jest szachownica wraz ze znajdującymi się na niej bierkami.

Stan (ang. state)

Stan to poszczególna sytuacja w jakiej znajduje się agent. W rozgrywce szachowej jest to konkretna pozycja szachowa.

Nagroda (ang. reward)

Nagroda to informacja zwrotna pochodząca od środowiska, w jakim znajduje się agent. Nagroda wyznaczana jest na podstawie nowego stanu, w jakim znajdzie się agent po podjęciu konkretnej akcji. Informuje ona o jakości podjętej decyzji. Jest kluczowym elementem uczenia przez wzmocnianie. Nagroda może być natychmiastowa lub odroczone w czasie. W grze w szachy nagrodą jest wygranie partii przez jednego z zawodników.

Polityka (ang. policy)

Polityka to mapowanie obserwacji obecnego stanu środowiska na probabilistyczny rozkład prawdopodobieństwa potencjalnych akcji. W trakcie uczenia, agent znajduje optymalne parametry aproksymatora w celu zmaksymalizowania długoterminowej nagrody. Polityka pomaga agentowi wybrać najlepszą decyzję w stanie, w którym się znajduje. Można rozróżnić dwie podstawowe rodzaje polityk: deterministyczną i stochastyczną. Pierwszą z nich opisuje równanie $\pi(s) : S \rightarrow A$, natomiast drugą $\pi(s, a) : S \times A \rightarrow [0, 1]$, gdzie $\pi(s, a)$ oznacza prawdopodobieństwo wybrania akcji a w stanie s . Posługując się szachową analogią, polityką można nazwać zbiór możliwych ruchów do wykonania w konkretnej pozycji szachowej wraz z oceną ich jakości.

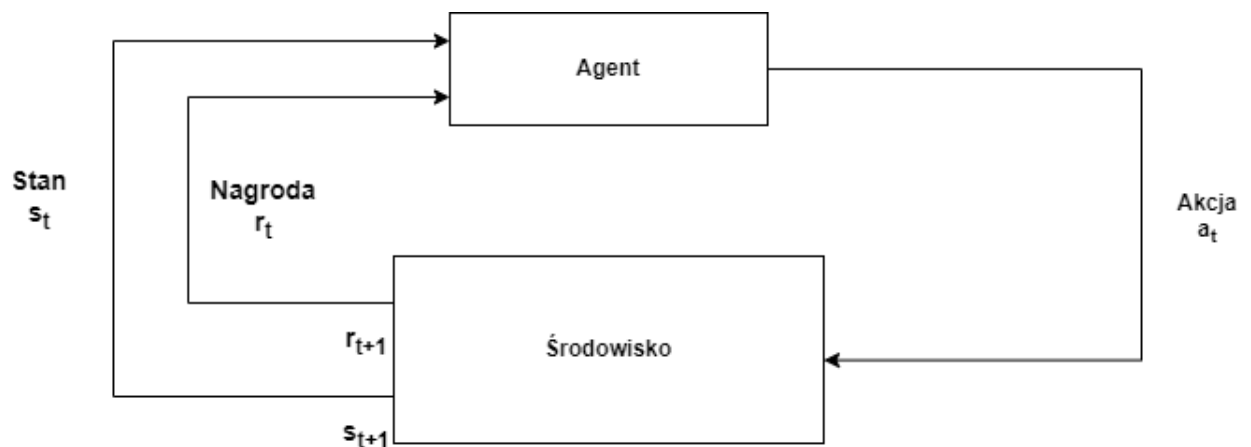
Wartość (ang. value)

Wartość to oczekiwana nagroda jaką agent zbierze kierując się obraną polityką. Jest oceną jakości konkretnego stanu. W rozgrywce szachowej ostateczny wynik może przyjmować trzy wartości: Wygrana - 1.0, remis - 0.0, przegrana -1.0.

Współczynnik dyskontowy (ang. discount factor)

Współczynnik dyskontowy używany jest w celu wprowadzenia przypadkowości w podejmowaniu przyszłych decyzji. Wprowadzenie niepewności jest korzystne dla modelu szczególnie we wczesnych etapach

uczenia, kiedy agent eksploruje nieznaną przestrzeń i zapobiega ugrzęźnięciu w minimum lokalnym.



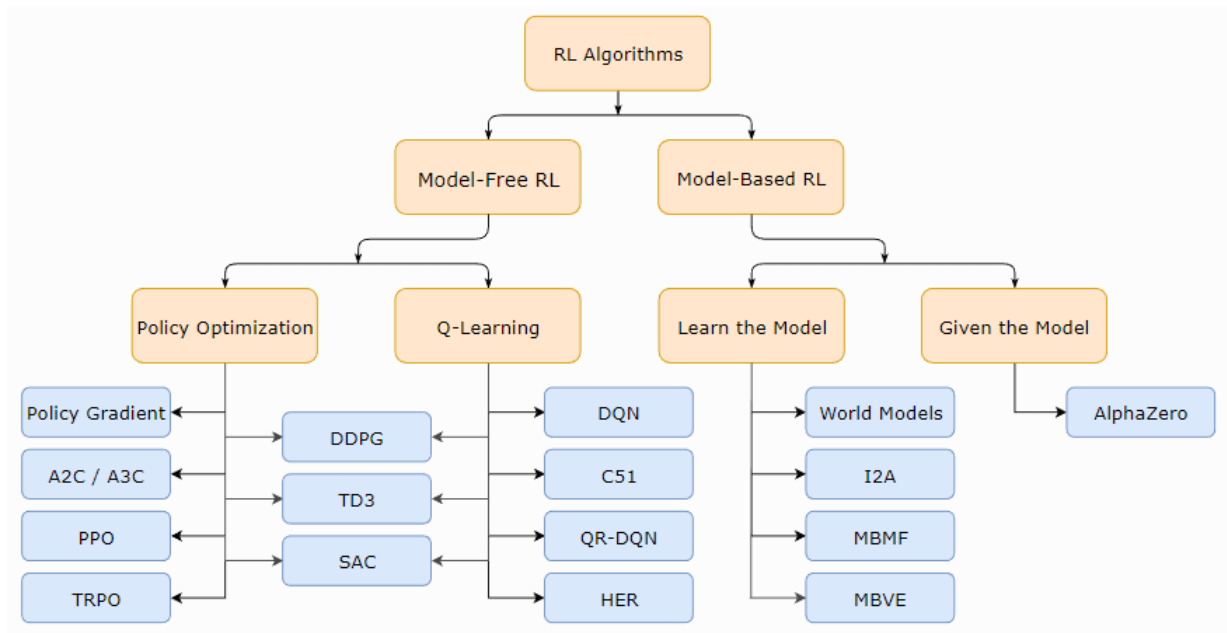
Rysunek 2.7: Proces uczenia ze wzmocnieniem.

Metody modelowe (ang. model-based)

Ten rodzaj algorytmów wykorzystuje przejścia w inne stany i funkcje nagrody w celu znalezienia najbardziej optymalnej polityki. Są używane w problemach, w których mamy kompletną wiedzę na temat środowiska oraz tego jak reaguje na różne akcje. Agent ma w nich dostęp do środowiska, co oznacza, że zna prawdopodobieństwa i nagrody wynikające z przejścia do poszczególnych stanów. Pozwala mu to na wybieganie w przyszłość i dokładną kalkulację kilku posunięć do przodu. Algorytmy wykorzystujące model są używane w statycznych i zamkniętych środowiskach, w których reguły są jasno określone [2].

Metody bezmodelowe (ang. model-free)

Algorytmy bezmodelowe znajdują optymalną politykę posiadając niewielką wiedzę o środowisku, które jest zmienne i dynamiczne. Sprawia to, że bardzo łatwo adaptują się do nowych środowisk i dobrze radzą sobie w nieznanach wcześniej stanach. Nie posiadają funkcji nagrody lub przekształcenia pozwalającego ocenić jakość polityki. Oszacowują one optymalną politykę bezpośrednio z interakcji agenta ze środowiskiem. Algorytmy te nadają się idealnie do scenariuszów, w których nie posiadamy dokładnej wiedzy na temat środowiska. Dlatego doskonale radzą sobie w środowisku, które jest światem rzeczywistym i deklasują pozostałe techniki [2].



Rysunek 2.8: Podział algorytmów uczenia ze wzmocnieniem [15].

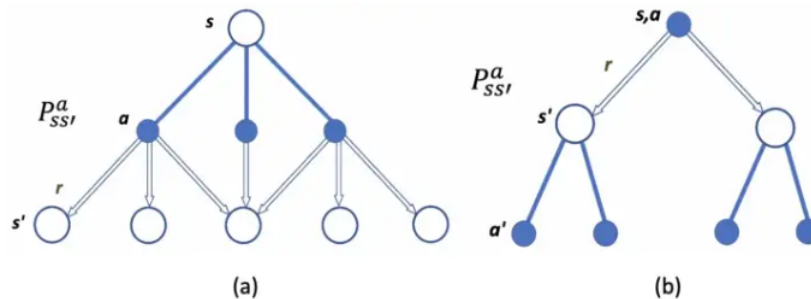
2.5.1. Równania Bellmana

Zadaniem agenta w uczeniu ze wzmocnieniem jest znalezienie sekwencji akcji maksymalizujących oczekiwaną nagrodę w zadanym epizodzie lub w przeciągu całego okresu działania. Do znalezienia optymalnej sekwencji akcji wykorzystuje się równanie Bellmana [23].

$$V_{\pi}(s) = \sum_a \pi(a|s) \cdot \sum_{s'} P_{ss'}^a (r(s, a) + \gamma V_{\pi}(s')) \quad (2.4)$$

$$Q_{\pi}(s, a) = \sum_{s'} P_{ss'}^a (r(s, a) + \gamma V_{\pi}(s')) \quad (2.5)$$

Równania określają jak znaleźć wartość stanu s z odpowiadającą mu polityką π . Są one ważne z tego



Rysunek 2.9: V-funkcja oraz Q-funkcja [23].

względu, że pozwalają na określenie wartości stanu s za pomocą wartości stanu przyszłego s' . Dzięki temu, wykorzystując podejście rekursywne, możliwe jest obliczenie lub aproksymacja wartości wszystkich możliwych stanów środowiska.

2.5.2. Metoda Q-learning

Metoda Q-learning należy do algorytmów bezmodelowych oparty na oczekiwanych wartościach poszczególnych akcji, który skupia się na optymalizacji funkcji wartości. Model przechowuje wszystkie wartości stanów w tabeli nazywanej *Q-table* - wiersze to poszczególne stany, a kolumny to akcje. W tabeli przechowywane są wartości funkcji Q, które odpowiadają przejściom pomiędzy stanami.

		actions			
states		a_0	a_1	a_2	\dots
	s_0	$Q(s_0, a_0)$	$Q(s_0, a_1)$	$Q(s_0, a_2)$	\dots
	s_1	$Q(s_1, a_0)$	$Q(s_1, a_1)$	$Q(s_1, a_2)$	\dots
	s_2	$Q(s_2, a_0)$	$Q(s_2, a_1)$	$Q(s_2, a_2)$	\dots
	\vdots	\vdots	\vdots	\vdots	\vdots

Rysunek 2.10: Ogólna tabela wartości stanów type Q-table [22].

Algorytm można podzielić na następujące etapy:

- Inicjalizacja - polega na stworzeniu Q-table z odpowiednią liczbą kolumn i wypełnienie jej zerami lub losowymi wartościami.
- Eksploracja lub Eksploatacja - tym kroku agent wybierze jedną z dwóch możliwych dróg. Eksploatacja to wybieranie kolejnego ruchu na podstawie utworzonej Q-table. Eksploracja to losowe wybranie kolejnego ruchu w celu zapoznania agenta ze środowiskiem. Jest szczególnie istotna w początkowej fazie nauki.
- Pomiar - w tym kroku następuje pomiar oczekiwanej nagrody, dla każdej z wybranych dróg.
- Aktualizacja Q-table - w tym kroku za pomocą równania Bellmana następuje aktualizacja wartości znajdujących się w tablicy.

2.6. Głębokie uczenie maszynowe ze wzmocnieniem

2.6.1. Sieci neuronowe

Sztuczna sieć neuronowa to system przeznaczony do przetwarzania i klasyfikacji informacji, wzorowany na biologicznym systemie nerwowym. W założeniu ma odzwierciedlać zachowanie ludzkiego mózgu. Wzrost mocy obliczeniowej oraz pamięci w ostatnich latach sprawił, że znalazły one zastosowanie w niemalże każdej dziedzinie życia. Dzięki nim rozwiązywanym jest większość problemów uczenia maszynowego. Sieci neuronowe opierają się na danych szkoleniowych, aby z upływem czasu uczyć się i poprawiać dokładność swoich predykcji. Kiedy proces uczenia zakończy się, a sieć osiągnie przyzwoitą dokładność, wtedy staje się ona potężnym narzędziem, który w bardzo krótkim czasie umożliwia błyskawiczne klasyfikowanie lub podział danych na klastry.

Perceptron

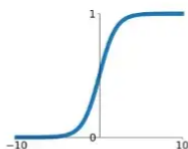
Perceptron to najprostsza i najstarsza sieć neuronowa stworzona przez Franka Rosenblatta w 1958 roku, zawierająca jeden sztuczny neuron, w skład którego wchodzi warstwa wejściowa oraz połączenia prowadzące od jednostek wejściowych do jednej jednostki wyjściowej. Zadaniem perceptronu jest klasyfikacja wzorów pojawiających się na jego wejściu. Jednostka wyjściowa sumuje wartości każdego wejścia x_n pomnożone przez siłę jego połączenia, inaczej nazywaną wagą w_n oraz biasu - numerycznej wartości dodawanej do sumy perceptronu w celu poprawy jakości klasyfikacji. Suma ważona wejść jest porównywana z wartością progową θ i przepuszczana przez funkcję aktywacji, która zwraca na wyjściu wartość 1, jeśli wartość jest większa od wartości progowej lub 0 jeśli wartość jest od niej mniejsza lub równa [17]. Perceptron opisuje równanie:

$$y = f(b + \sum_{i=1}^n w_i \cdot x_i) \quad (2.6)$$

W perceptronach używane są różne funkcje aktywacji, najpopularniejsze z nich zamieszczone zostały na rysunku 2.11.

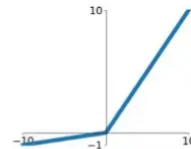
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



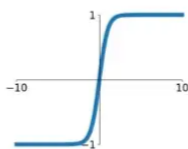
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

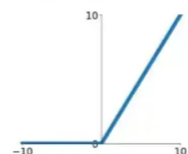


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

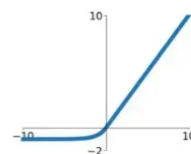
ReLU

$$\max(0, x)$$

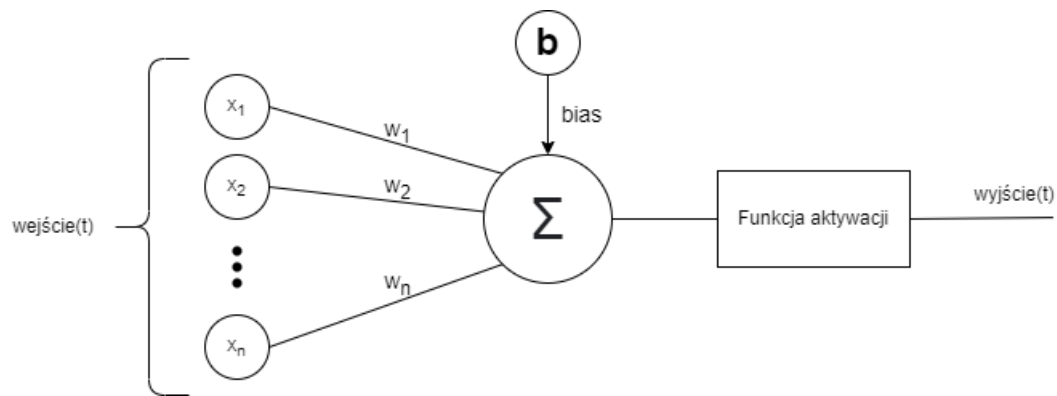


ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Rysunek 2.11: Funkcje aktywacji perceptronu [11].



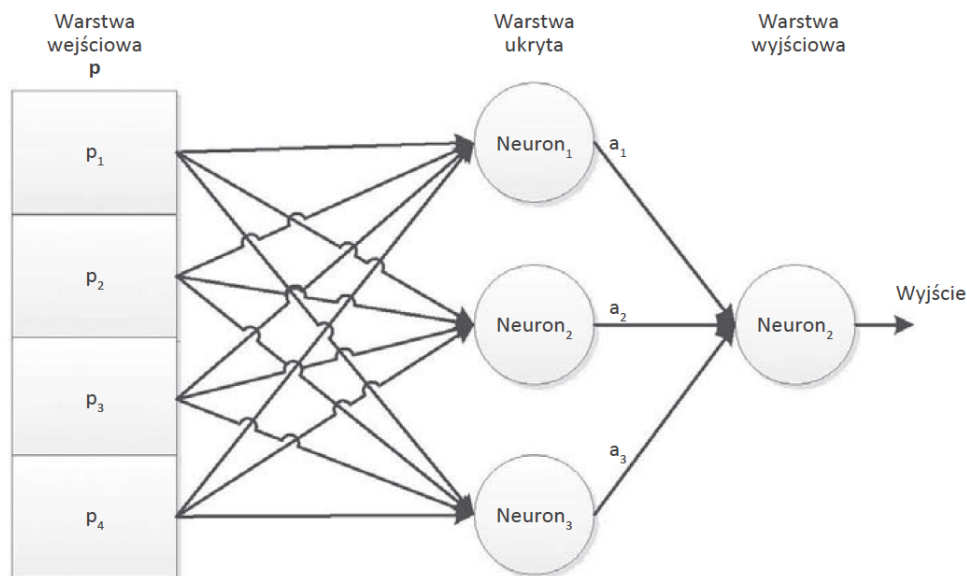
Rysunek 2.12: Schemat perceptronu.

2.6.2. Głębokie sieci neuronowe

Głębokie sieci neuronowe to rodzaj sztucznych sieci neuronowych zaprojektowanych do posiadania dużej ilości warstw ukrytych, połączonych ze sobą za pomocą węzłów. Pomysł ten zainspirowany jest strukturą mózgu i sposobem w jaki człowiek przetwarza informacje i uczy się. Można powiedzieć, że jest to rozszerzenie zwykłych sieci neuronowych. Różnica polega na ilości warstw ukrytych, gdzie w przypadku sieci głębokich występuje ich bardzo duża liczba. Warstwy te realizują nieliniowe transformacje i tworzą model hierarchiczny, gdzie kolejne warstwy odwzorowują kolejne poziomy abstrakcji sygnału wejściowego. Duża ilość warstw pozwala na automatyczną naukę wykrywania cech. Dużą zaletą głębokich sieci neuronowych jest umiejętność nauki z dostępnych danych i podejmowania poprawnych decyzji w sytuacjach, które jeszcze nie były znane modelowi. Potrafią one dokonywać poprawnych predykcji w problemach o wysokim stopniu skomplikowania bez konieczności użycia wiedzy eksperckiej z analizowanej dziedziny, dlatego znalazły szeroki zakres zastosowań w wielu współczesnych problemach.

Wyróżnia się różne rodzaje sieci neuronowych. Najważniejsze z nich to [6]:

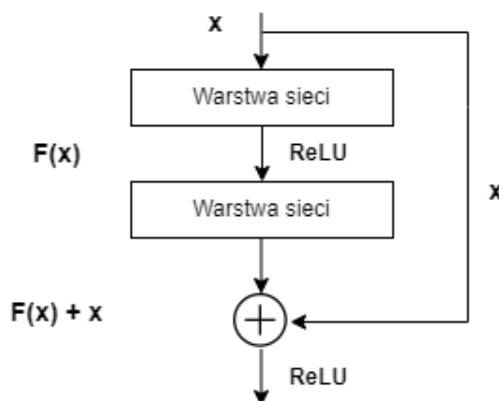
- **Sieci neuronowe jednokierunkowe**, czyli perceptrony wielowarstwowe. Składają się z warstwy wejściowej i wyjściowej oraz warstw ukrytych, znajdujących się pomiędzy wejściem, a wyjściem. Do trenowania tych modeli wykorzystuje się duże ilości danych. Należy tutaj pamiętać, że w rzeczywistości składają się one głównie z neuronów sigmoidalnych, a nie perceptronów, z uwagi na fakt, że większość rzeczywistych problemów ma charakter nieliniowy.
- **Konwolucyjne sieci neuronowe** są podobne do sieci jednokierunkowych, ale są one zazwyczaj wykorzystywane do rozpoznawania obrazów, wzorców lub widzenia komputerowego. Sieci te wykorzystują zasady algebry liniowej, a w szczególności mnożenie splotowe macierzy.
- **Rekurencyjne sieci neuronowe** wyróżnia się w oparciu o pętle informacji zwrotnych. Te algorytmy uczenia się są stosowane głównie przy wykorzystywaniu danych szeregów czasowych do przewidywania przyszłych wyników, takich jak notowania giełdowe czy prognozowanie sprzedaży.
- **Generatywne sieci przeciwstawne** to koncept, w którym tworzy się dwie sieci: generator i dyskryminator. Generator próbuje wygenerować nowe dane podobne do zbioru danych uczących,



Rysunek 2.13: Sieć neuronowa jednokierunkowa [12].

natomiast dyskryminator próbuje rozróżnić dane wygenerowane od zbioru uczącego. Obie sieci uczone są razem, w taki sposób, aby generator oszukał dyskryminator, natomiast celem dyskryminatora jest poprawne rozpoznanie wygenerowanych danych. Sieci te często są wykorzystywane do generowania obrazów i augmentacji danych.

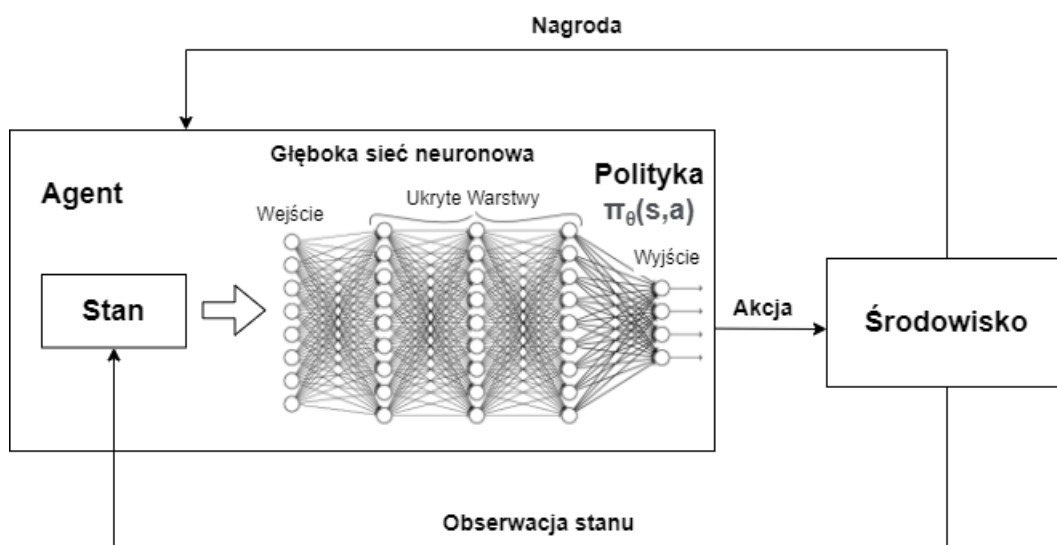
- **Rezydualne sieci neuronowe (ResNet)** to specjalny rodzaj sieci neuronowej, która została zaprojektowana w celu rozwiązania problemu zanikającego gradientu. Zanikający gradient to sytuacja, w której wartości gradientu stają się coraz mniejsze wraz z każdą warstwą sieci neuronowej, co utrudnia uczenie się sieci i prowadzi do gorszych rezultatów. Sieć rezydualna rozwiązuje ten problem, dodając "ścieżkę rezydualną" do każdej warstwy. Ścieżka ta pozwala przenieść część danych z warstwy wejściowej bezpośrednio do warstwy wyjściowej, co zapobiega zanikaniu gradientu i umożliwia sieci uczenie się z większą dokładnością.



Rysunek 2.14: Schemat jednego bloku ResNet [19].

2.6.3. Głębokie uczenie ze wzmocnieniem

Głębokie uczenie ze wzmocnieniem (z ang. deep reinforcement learning, DRL) to rodzaj uczenia maszynowego, w którym algorytm uczy się poprzez interakcję z otoczeniem i otrzymywanie nagród za wykonane akcje. Algorytm jest w stanie samodzielnie zdecydować, które akcje są najlepsze do wykonania w danej sytuacji, aby osiągnąć jak największą nagrodę. Głębokie uczenie ze wzmocnieniem wykorzystuje głębokie sieci neuronowe do przetwarzania informacji o stanie i dobierania odpowiednich akcji. Wykorzystanie sieci neuronowej rozwiązuje problem, który pojawia się w wielu przypadkach tradycyjnego uczenia ze wzmocnieniem - pozwala estymować wartość funkcji Q w sytuacjach gdzie przestrzeń stanów jest zbyt duża, aby wykorzystywać tablice Q-table lub przechowywać wiedzę na temat stanów w innych postaciach. Sieć uczy się poprzez dane pochodzące z wielokrotnych interakcji z otoczeniem oraz otrzymywanych nagród i kar za wykonane akcje. Im więcej interakcji i nagród, tym lepsze będzie uczenie się sieci. Proces treningowy zazwyczaj wymaga dużych nakładów mocy obliczeniowej. DRL jest szczególnie przydatny w sytuacjach, gdzie trudno zdefiniować dokładne reguły lub cele, a uczenie się poprzez próby i błędy jest bardziej skuteczne. Najczęściej wykorzystuje się tą metodę w grach komputerowych, gdzie agent musi uczyć się najlepszych strategii. Warto zaznaczyć, że to właśnie artykuł *Playing Atari with Deep Reinforcement Learning* [14] rozpoczął erę głębokiego uczenia ze wzmocnieniem i sprawił, że zaczęto dostrzegać możliwości jakie niesie ono za sobą. W ostatnich latach DRL coraz częściej znajduje zastosowanie w dziedzinach takich jak logistyka, robotyka, medycyna czy transport.



Rysunek 2.15: Schemat głębokiego uczenia ze wzmocnieniem.

2.7. AlphaZero

AlphaZero to algorytm, który osiągnął nadludzki poziom umiejętności w grach takich jak Szachy, Go czy Shogi. Odwieczną ambicją sztucznej inteligencji było stworzenie algorytmu potrafiącego uczyć się z samego siebie, zamiast z nadanych reguł i można stwierdzić, że AlphaZero jest temu bardzo bliski. Jego ogromną zaletą jest ogólność i wszechstronność co znaczy, że można zaaplikować go do różnych gier bez wiedzy eksperckiej na temat konkretnej gry. Dodatkowo sprawia to, że algorytm nie jest obciążony błędem ludzkim, pozwalając na osiągnięcie poziomu nadludzkiego w szerokim zakresie dziedzin.

AlphaZero zastępuje wiedzę ekspercką z danej dziedziny i historię partii historycznych, używanych w tradycyjnych programach komputerowych, za pomocą głębokich sieci neuronowych oraz uczenia ze wzmocnieniem. W tradycyjnych programach używa się skomplikowanych funkcji ewaluacji pozycji, które w przypadku szachów uwzględniają różne aspekty konkretnej pozycji szachowej [24]:

- Bezpieczeństwo króla
- Przewaga materialna w zależności od etapu gry
- Przewaga przestrzenna
- Struktura pionowa
- Mobilność i *uwięzione* figury

Przewaga AlphaZero nad klasycznymi silnikami polega na tym, że wszystkie wymienione wyżej kryteria zawierają się w postaci głębokiej sieci neuronowej $(p, v) = f_{\theta}(s)$ z parametrami θ . Wejściem sieci neuronowej jest abstrakcyjna reprezentacja aktualnego stanu szachownicy s , a wyjściem wektor prawdopodobieństw ruchów p , który dla każdej z akcji a przyporządkowuje $p_a = P_r(a|s)$ oraz wartość v estymującą wynik z , zadanego stanu szachownicy s , wyrażonego jako $v \approx E[z|s]$. Algorytm uczy się prawdopodobieństw wyłącznie poprzez rozgrywanie kolejnych partii z samym sobą (ang. self-play). Trening AlphaZero w początkowej postaci wyglądał w następujący sposób:

1. Silnik rozgrywa n partii sam ze sobą.
2. Model sieci neuronowej uczy się na podstawie zebranych danych.
3. Silnik ze starym modelem rozgrywa k partii między silnikiem z nowym modelem.
4. Jeśli nowy model okazuje się lepszy, to następuje zamiana, a jeśli gorszy to do kolejnej iteracji używa się starego modelu. Proces powtarza się od nowa.

Podejście to zmieniono w nowszej wersji AlphaZero. Okazało się, że zamiana modelu ze starego na nowy jest zawsze korzystniejsza - nawet jeśli okazał się on słabszy -, ponieważ w długofalowej perspektywie przynosi to lepsze rezultaty. Jest to zjawisko podobne, do ludzkiego sposobu nauki, gdzie przyrost umiejętności nie jest przez cały czas rosnący, a proces nauki przeplatany jest zwycięstwami i porażkami.

W tradycyjnych silnikach do przewidywania kolejnych ruchów i tworzenia przyszłej sytuacji na planszy/szachownicy wykorzystywany jest algorytm *alpha-beta* z modyfikacjami wynikającymi ze specyfiki

gry. AlphaZero wymagało innego podejścia. Zastosowano w nim drzewo decyzyjne Monte-Carlo. Każde z przeszukiwań zawiera serie symulacji MCTS, które przechodzą drzewo od korzenia (aktualna pozycja na szachownicy) s_{root} do liścia (wariant pochodzący z aktualnej pozycji szachowej, który został odwiedzony po raz pierwszy). Każda symulacja przystępuje do selekcji ruchu a wybierając stan s , kierując się niską ilością odwiedzin, dużym prawdopodobieństwem ruchu oraz wysoką estymowaną wartością zwracaną przez obecną sieć neuronową f_θ . Wynikiem przeszukiwania jest wektor π reprezentujący rozkład prawdopodobieństwa poszczególnych ruchów.

Parametry θ głębokiej sieci neuronowej AlphaZero są inicjalizowane losowo, a następnie trenowane za pomocą rozgrywania partii ze samym sobą. Gry rozgrywane są poprzez wybranie ruchów dla obydwu graczy za pomocą MCTS, $a_t \sim \pi_t$. Wynik końcowy z pozycji s_T określony jest zgodnie z punktacją gry w szachy:

- -1 za przegraną.
- 0 za remis.
- 1 za wygraną.

Parametry sieci neuronowej są aktualizowane tak, aby zminimalizować różnicę pomiędzy przewidywanym wynikiem v_t i wynikiem gry z oraz zmaksymalizować podobieństwo wektorów polityki p_t i prawdopodobieństw wyszukiwania π_t . Do modyfikacji parametrów θ wykorzystywana jest metoda gradientu prostego z funkcją kosztu l sumującą różnicę kwadratów błędów oraz *categorical cross-entropy*:

$$(p, v) = f_\theta(s) \qquad l = (z - v)^2 - \pi^T \log p + c \|\theta\|^2 \qquad (2.7)$$

gdzie c to parametr kontrolujący wagi regularyzacji L_2 . Aby zapewnić poprawną eksplorację środowiska, do polityki pochodzącej z sieci neuronowej dodawany jest losowy szum, wyskalowany w zależności od ilości legalnych ruchów w danej pozycji [20].

Firma DeepMind w procesie nauki używała obliczeń zrównoleglonych na pięciu tysiącach jednostek *Tensor Processing Unit* [10]. TPU to zaprojektowane przez Google układy scalone używane głównie do przyspieszenia obliczeń w uczeniu maszynowym. Swoją szybkość zawdzięczają architekturze stworzonej specjalnie do obliczeń w dziedzinie algebry liniowej. W związku z powyższym są to ogromne, a co za tym idzie, bardzo kosztowne nakłady mocy obliczeniowej. W rzeczywistości osiągalne są tylko dla największych firm i projektów o wysokim budżecie.

AlphaZero znalazło zastosowanie nie tylko w świecie gier komputerowych. Firma Google użyła go do optymalizacji chłodzenia centrum przetwarzania danych. Rezultatem było obniżenie kosztów chłodzenia o 40%[5]. Algorytm opłaca się zastosować wszędzie tam, gdzie nie da określić się optymalnego rozwiązania, rozmiar środowiska jest zbyt duży, aby jednoznacznie rozwiązać zagadnienie, a potencjalne zyski płynące z optymalizacji szacuje się większe od wartości mocy obliczeniowej potrzebnej do procesu uczenia.

2.8. Miary jakości silnika szachowego

2.8.1. Metoda obliczania relatywnej siły gry szachistów

Ocenienie siły gry szachisty to zadanie niebanalne z uwagi na fakt, że nie istnieją metody nierelatywnej oceny wartości konkretnych szachowych ruchów w partii szachów. Dlatego jedynym rozwiązaniem jest ocena relatywna tzn. ocenianie rankingu na podstawie wyniku partii pomiędzy dwoma szachistami. Zakłada się, że jeśli gracz wygrał partię z przeciwnikiem to rozegrał ją na wyższym poziomie. Remis oznacza, że obaj szachiści zegrali na równym poziomie. Kierując się powyższymi założeniami amerykański naukowiec Arpad Elo zaproponował model obliczania relatywnej siły gry szachistów oparty o rozkład normalny (obecnie używa się rozkładu logistycznego), w którym o zmianie rankingu decyduje ranking przeciwników i bezpośrednie wyniki pomiędzy nimi. Wynika z tego, że jeśli szachista uzyskał wynik partii wyższy od oczekiwanego to jego ranking wzrasta, natomiast jeśli gorszy to maleje. Ranking Elo zakłada, że każdy z zawodników posiada własną określoną, aktualną siłę, która nie jest znana, a jest estymowana poprzez ranking. Biorąc pod uwagę sytuację dwóch szachistów A i B, odpowiednio z rankingami R_A i R_B , oczekiwane rezultaty zawodników E_A oraz E_B , określoną są wzorami:

$$E_A = \frac{1}{1 + 10^{\frac{-(R_B - R_A)}{400}}} \quad E_B = \frac{1}{1 + 10^{\frac{-(R_A - R_B)}{400}}} \quad (2.8)$$

Przykładowo, jeśli rankingi graczy A i B wynoszą odpowiednio 1500 i 1700 to powyższa formuła zwraca długoterminowy średni wynik gracza A równy 0.2403, a dla gracza B jest to 0.7597. Na podstawie tych oczekiwanych wyników aktualizowany jest ranking graczy. Gracz A posiada mniejszy oczekiwany wynik, dlatego w przypadku zwycięstwa zdobędzie on więcej punktów rankingu niż gracz B. Analogicznie sytuacja wygląda w sytuacji przegranej - gracz A straci mniejszą ilość punktów niż gracz B.

Ranking Elo został przyjęty w 1960 roku przez Federację Szachową Stanów Zjednoczonych (USCF) i w 1970 roku przed Międzynarodową Federację Szachową (FIDE). Warto zaznaczyć, że obie federacje wprowadziły własne modyfikacje Rankingu Elo, co skutkuje różnicą średnio około 100 punktów pomiędzy tym samym szachistą znajdującym się w obu rankingach.

2.8.2. Sposoby mierzenia siły gry silników

Zmierzenie siły silnika szachowego to zadanie, które można wykonać na kilka różnych sposobów. W przypadku silników o poziomie gry nie przekraczającym ludzkich możliwości, najlepszą metodą jest rozegranie kilkunastu partii z szachistą o znanym rankingu Elo i wyliczenie rankingu silnika na podstawie uzyskanych wyników. Sytuacja komplikuje się przy silnikach osiągających wyniki przewyższające poziom ludzki. Mierzenie ich rankingu Elo poprzez rozgrywanie partii z ludźmi nie jest dobrym rozwiązaniem. Wyróżnia się tutaj dwa podejścia. Pierwsze z nich to estymacja siły za pomocą rozkładu normalnego. Drugie to wstępna estymacja rankingu ELO, a następnie określenie rankingu Elo poprzez rozgrywanie partii szachowych pomiędzy dwoma silnikami.

3. Implementacja silnika szachowego

3.1. Implementacja

Stworzenie silnika szachowego to proces skomplikowany i złożony. Istnieje bardzo dużo różnych implementacji tego zadania. Jak wcześniej wspomniano, silniki generalnie można podzielić na klasyczne oraz wykorzystujące sieci neuronowe. Silnik utworzony w tej pracy został zaimplementowany w oparciu o przełomowy artykuł opisujący działanie AlphaZero [20] i wykorzystuje głębokie sieci neuronowe. Z uwagi na powyższe, zalicza się do tej drugiej grupy. Z uwagi na ograniczone zasoby obliczeniowe zostały wprowadzone w nim pewne uproszczenia, które zostaną opisane w dalszej części rozdziału. Do implementacji został wykorzystany język Python z zasadami programowania obiektowego. Język ten został użyty w oparciu o jego ogromne możliwości w obszarze uczenia maszynowego i łatwość utworzenia procesu samouczenia. Język Python pozwala na użycie modeli takich jak Tensorflow oraz PyTorch, które idealnie nadają się do operacji związanych z głębokim uczeniem ze wzmocnieniem i pozwalają w optymalny oraz dokładny sposób utworzyć model głębokiej sieci neuronowej, jednocześnie zapewniając ogrom narzędzi służących do analizy procesu uczenia i zbierania istotnych danych. Model sieci w tej pracy został zbudowany przy użyciu Tensorflow.

3.1.1. Stworzenie logiki gry i implementacja zasad gry

Logika gry i jej zasady zostały stworzone z wykorzystaniem szerokiej i bardzo dobrze udokumentowanej biblioteki *python-chess* [7], udostępniającej funkcje i klasy związane z reprezentacją pozycji szachowej na szachownicy, konwersje zapisów partii szachowych i wiele innych przydatnych narzędzi, służących do poprawnej implementacji rozgrywki. Pozycję szachową można przedstawiać za pomocą różnych sposobów. W tej pracy została użyta notacja Forsytha-Edwardsa (FEN). Jej zamierzeniem jest podanie wszystkich niezbędnych informacji do odtworzenia dokładnie jednej konkretnej pozycji szachowej. Nie zawiera ona żadnych informacji na temat wcześniejszych ruchów, przez co jest uniwersalna, a jej wykorzystywanie ułatwia implementację logiki gry. Warto tutaj zaznaczyć, że nie jest to rozwiązanie optymalne obliczeniowo, istnieją inne sposoby zapisu partii szachowej, które mogłyby poprawić szybkość działania programu. Notacja FEN opisuje początkową konfigurację szachownicy w następujący sposób:

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

Pierwsza część odnosi się do sytuacji na szachownicy, gdzie małymi literami oznacza się figury koloru czarnego, a dużymi figury koloru białego. Zapis zaczyna się od pola A8 i kończy na polu H1. Cyfry określają ilość kolejnych pustych miejsc. Każdy z kolejnych rzędów oddzielony jest znakiem '/'. Natomiast druga część informuje kolejno o kolorze bierki, które teraz mają wykonać ruch, możliwościach roszady, biciu w przelocie (En Passant), liczbie ruchów od ostatniego zbitia bierki oraz całkowitej liczbie ruchów. Biblioteka *python-chess* zapewnia obsługę wyżej wspomnianych niestandardowych ruchów. Jedne z jej najważniejszych funkcji i klas użytych w procesie implementacji to:

- `Board`, klasa reprezentująca aktualny stan szachownicy. Klasa konwertuje zapis w notacji szachowej FEN i przenosi go do abstrakcyjnej reprezentacji w postaci macierzy 8x8 wypełnionej poszczególnymi figurami.
- `Move`, klasa reprezentująca szachowy ruch. Instancja klasy zawiera informacje na temat figury wykonującej ruch, pola na którym znajduje się figura oraz pola na które zmierza figura.
- `generate_legal_moves`, funkcja zwracająca generator zawierający liczbę wszystkich legalnych ruchów w zadanej pozycji szachowej, uwzględniając potencjalne szachy i związania figur.
- `Board.outcome`, funkcja informująca o stanie rozgrywki szachowej, zwracająca informacje czy partia toczy się dalej, czy zakończyła się wygraną jednej ze stron bądź remisem.

3.1.2. Funkcje oceniające pozycję szachową

W silnikach korzystających z sieci neuronowych, funkcje oceniającą pozycję szachową pełni model głębokiej sieci neuronowej. Dzięki temu ocena pozycji trwa krótką chwilę i nie wymaga zaawansowanych obliczeń analizujących różne aspekty pozycji szachowej. Model na wejściu przyjmuje zakodowaną aktualną pozycję szachową, a na jego wyjściu zwracana jest wartość w zakresie $< -y_{eval} : y_{eval} >$, wartości skrajne oznaczają nieuchronnego mata w n następnych ruchach dla białych lub czarnych, a wartość 0.0 oznacza pozycję remisową lub wyrównaną. W tej pracy przyjęto podobną koncepcję, wzorując się na AlphaZero Chess [20] [21]. Stworzony został model, który na wejściu przyjmuje zakodowaną pozycję szachową oraz posiada dwa wyjścia: wartość aproksymująca stan sytuacji na szachownicy - głowa wartości (ang. value head) - oraz rozkład prawdopodobieństwa ruchów w zadanej pozycji - głowa polityki (ang. policy head). Przyjęto wartość maksymalną y_{eval} równą 1.0 dla białych i -1.0 dla czarnych.

3.1.3. Algorytm poszukujący najlepszy ruch

Aby znaleźć najlepszy ruch w zadanej pozycji szachowej zbudowany silnik korzysta z dwóch kluczowych elementów. Pierwszym z nich jest algorytm Monte Carlo Search Tree zaadaptowany do przeszukiwania kolejnych posunięć szachowych. W tym przypadku węzłem jest konkretna pozycja szachowa. Natomiast każdy z dozwolonych ruchów odzwierciedla krawędź wychodzącą z węzła. Algorytm MCTS wykonuje określoną ilość symulacji. Do procesu uczenia wykorzystano wartość $N_{MCTS} = 200$, biorąc pod uwagę ograniczone zasoby obliczeniowe. W wykorzystaniu gotowego modelu do gry używano $N_{MCTS} = 300$, bazując na parametrach opisanych w artykule opisującym AlphaZero. Drugim elemen-

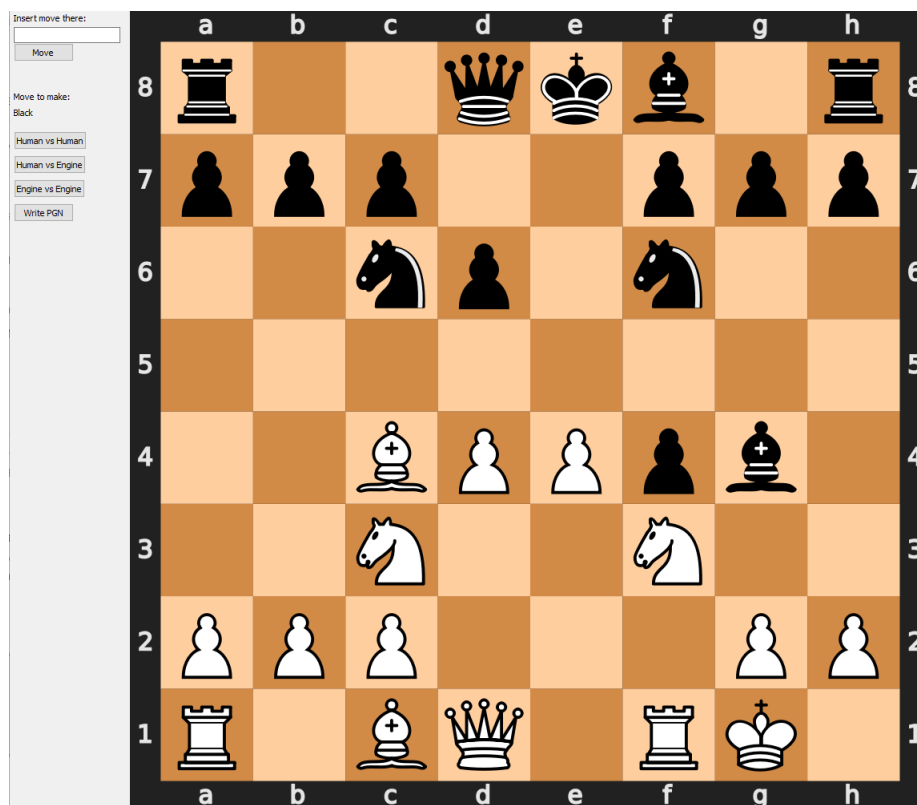
tem jest model sieci neuronowej, który dla każdego węzła dokonuje predykcji, której wynikiem jest ewaluacja pozycji oraz rozkład prawdopodobieństw dozwolonych ruchów. Wartości te wykorzystywane są w węzłach oraz krawędziach symulacji MCTS i odgrywają kluczową rolę w rozszerzaniu drzewa decyzyjnego oraz analizowaniu najbardziej obiecujących wariantów.

3.1.4. Program szachowy

Ostatnim etapem budowy silnika jest zbudowanie programu komputerowego, który ma odpowiadać rozgrywce szachowej. Posiadając algorytmy i model potrafiący wskazywać potencjalnie najlepsze ruchy oraz ewaluować aktualną pozycję na szachownicy, należy zadbać o poprawne zaimplementowanie reguł rozgrywki gry w szachy oraz różne tryby rozgrywki. Program posiada następujące tryby:

1. Człowiek przeciwko człowiekowi
2. Komputer przeciwko komputerowi
3. Człowiek przeciwko komputerowi

W celu pomocniczym stworzono prostą aplikację z interfejsem graficznym z wykorzystaniem biblioteki PyQt5. Pomagała ona w weryfikowaniu i testowaniu działania silnika oraz sprawdzeniu poprawności zaimplementowanych reguł szachowych. Główny program został stworzony jako strona internetowa i został opisana w podrozdziale 3.4 .

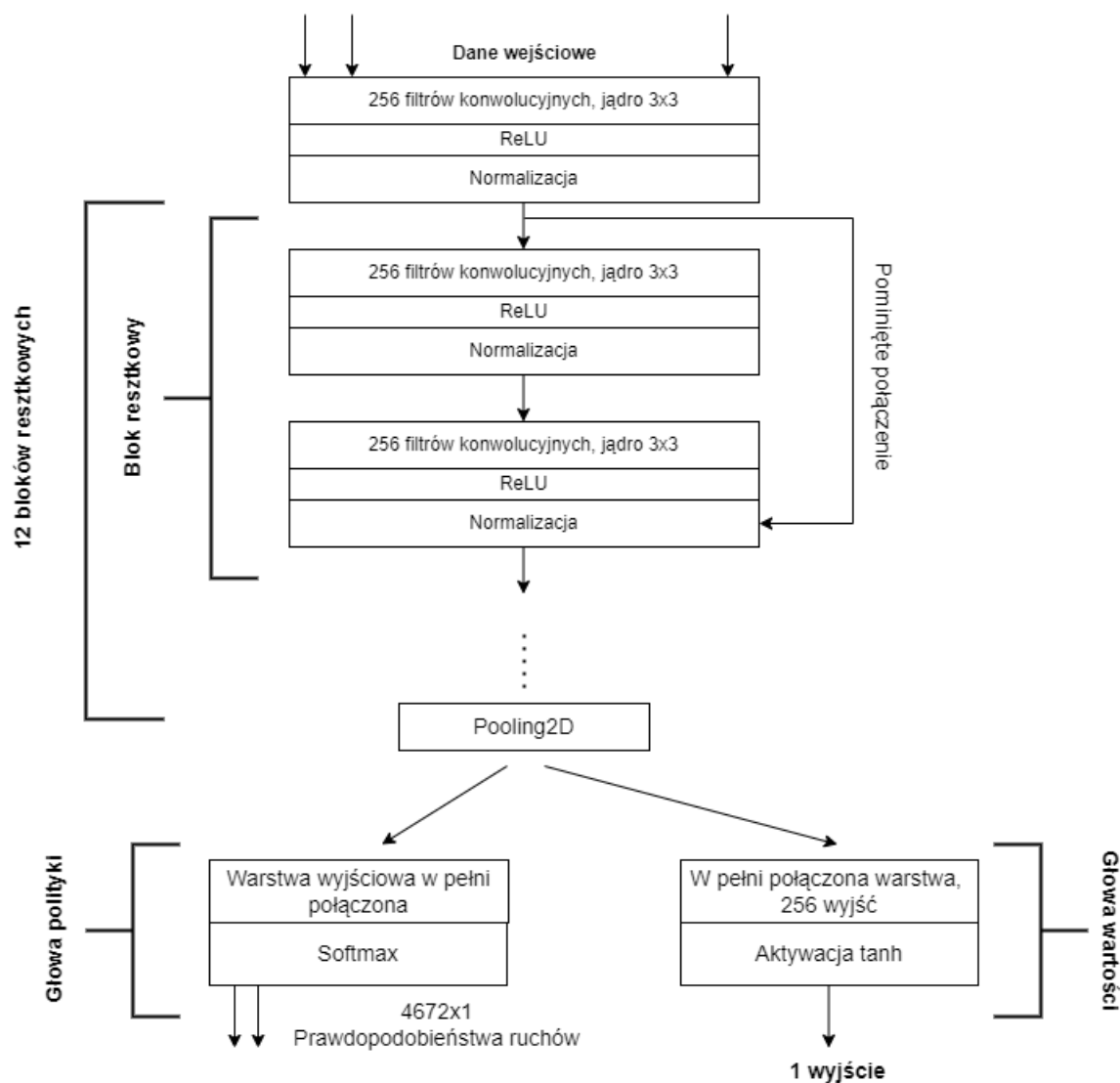


Rysunek 3.1: Interfejs graficzny prostego programu szachowego.

3.2. Architektura

3.2.1. Model sieci neuronowej

Budowa głębokiej sieci neuronowej została skonstruowana w oparciu o model AlphaZero [20]. Jej budowę przedstawia rysunek 3.2. Model składa się z warstwy wejściowej, 12 bloków rezydualnych, operacji łączenia (ang. pooling) oraz dwóch głów, które stanowią wyjścia sieci. Cały model posiada 14,875,969 parametrów i zajmuje 175 MB pamięci, co świadczy o jego dużej złożoności, mimo zastosowanych uproszczeń.



Rysunek 3.2: Schemat budowy głębokiej sieci neuronowej z dwoma głowami.

3.2.2. Kodowanie i dekodowanie pozycji szachowej

Sieć neuronowa na swoim wejściu przyjmuje wektor wartości liczbowych, stąd zapis pozycji szachowej w pozycji FEN nie nadaje się bezpośrednio do użycia. Należy go przekonwertować do innej postaci.

Wejściem sieci neuronowej jest 75 macierzy o rozmiarze 8x8, zawierających informacje o poprzednich stanach szachownicy i jest to drobne odejście od zasad uczenia ze wzmocnieniem oraz własności Markowa (przyszłość modelu zależy tylko od obecnej obserwacji środowiska). Jest ono konieczne w celu efektywniejszego działania sieci neuronowej, która dzięki historii poprzednich pozycji jest w stanie lepiej 'kojarzyć' sytuacje panującą na szachownicy. W skład 75 warstw wchodzi:

- Pięć ostatnich pozycji szachowych, z których każda opisana jest za pomocą sześciu warstw opisujących bierki białe, sześciu warstw opisujących bierki czarne (sposób kodowania został pokazany na rysunku 3.3) oraz dwóch warstw informujących czy dana pozycja powtórzyła się.
- Cztery warstwy opisujące prawa do roszady. Jeśli istnieje możliwość roszady, każdy z elementów wynosi 1, jeśli nie istnieje 0.
- Jednej warstwy opisującej zawodnika, który ma wykonać następny ruch. Jeśli jest to biały, każdy element jest równy 1, jeśli czarny to 0.

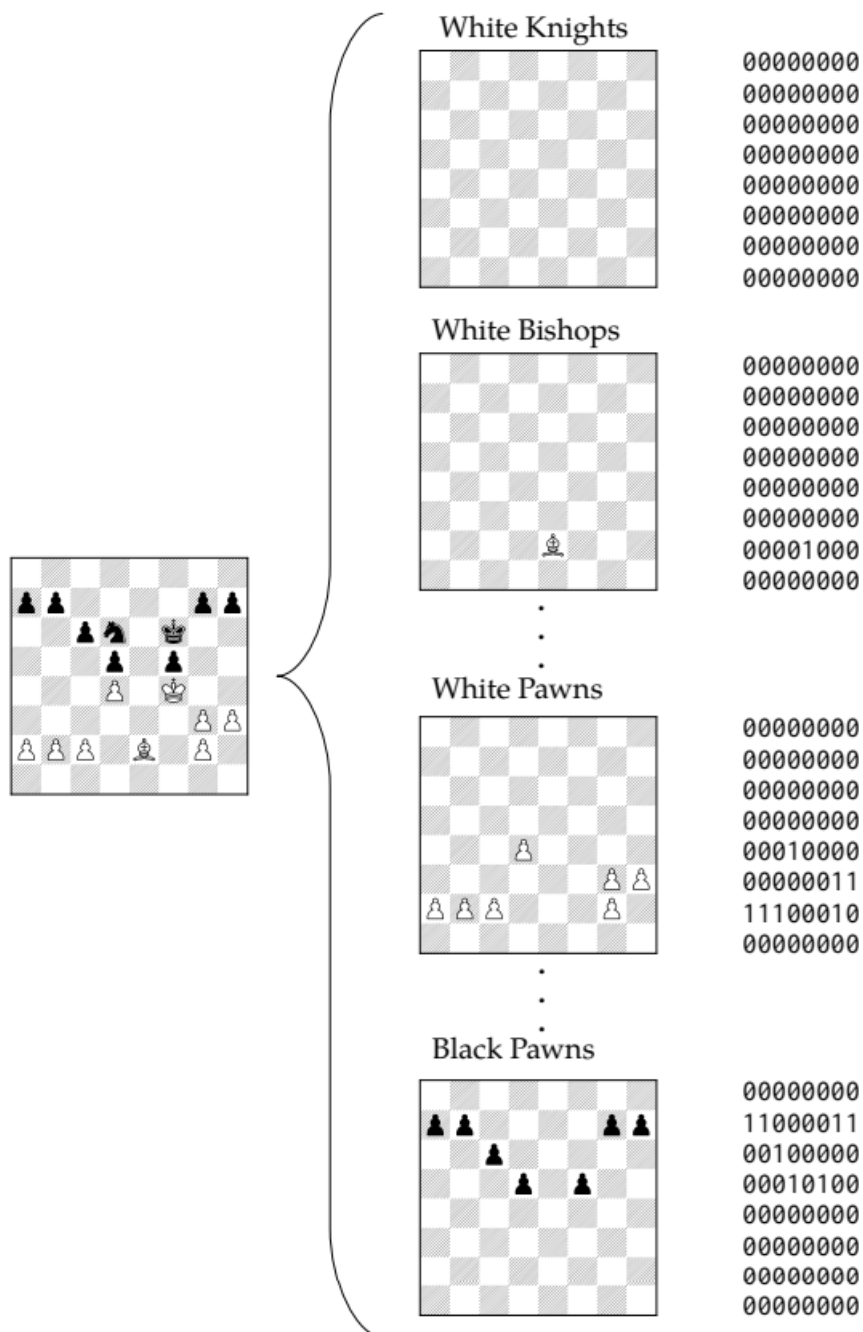
Reprezentacja wejścia posiada mniej warstw niż jest to wymienione w artykule AlphaZero [20]. Uproszczenia zostały wprowadzone z uwagi na ograniczone zasoby obliczeniowe. Główna różnica polega na zmniejszeniu historii pozycji z ośmiu do pięciu oraz usunięciu mało istotnych warstw, nie związanych z pozycją szachową.

Skonstruowanie wyjścia sieci opisującego rozkład prawdopodobieństw legalnych ruchów wymaga stałej ilości wyjść. Jednak w zależności od pozycji szachowej, liczba legalnych ruchów jest zmienna. Rozwiązaniem tego problemu jest stworzenie wyjścia, które obejmuje wszystkie możliwe ruchy, nawet te nielegalne lub całkowicie niemożliwe. Aby to osiągnąć przyjęto koncepcję *Super Figury*, potrafiącej wykonywać ruchy jak Królowa i Skoczek. Do stworzenia wszystkich możliwych ruchów *Super Figury* potrzeba dla każdego pola na szachownicy wyznaczyć wszystkie możliwe ruchy królowej w każdym z ośmiu kierunków, a następnie wszystkie możliwe ruchy Skoczka. Należy również uwzględnić ruchy promocji, czyli ruchy pionów na pole przemiany. Końcowa postać wyjścia to wektor o rozmiarze 4672x1 lub 73x8x8, w zależności od przyjętej koncepcji wyjścia. W tej pracy wybrano wektor jednowymiarowy.

Rozwiązanie to generuje ruchy nielegalne w danej pozycji, bądź całkowicie niemożliwe, dlatego po dokonaniu predykcji i otrzymaniu rozkładu prawdopodobieństw nakłada się maskę, której celem jest zachowanie tylko legalnych ruchów. Suma prawdopodobieństw wszystkich możliwych ruchów wynosi 1.0, dlatego po użyciu maski należy dokonać normalizacji przy uwzględnieniu wyłącznie legalnych ruchów. Po przefiltrowaniu ruchów nielegalnych i normalizacji, ostatnim etapem jest konwersja z wektora 4672x1 do listy krotek, zawierających ruch oraz odpowiadające mu prawdopodobieństwo. W takiej postaci wyjście gotowe jest do użycia w algorytmie MCTS.

3.3. Proces uczenia silnika

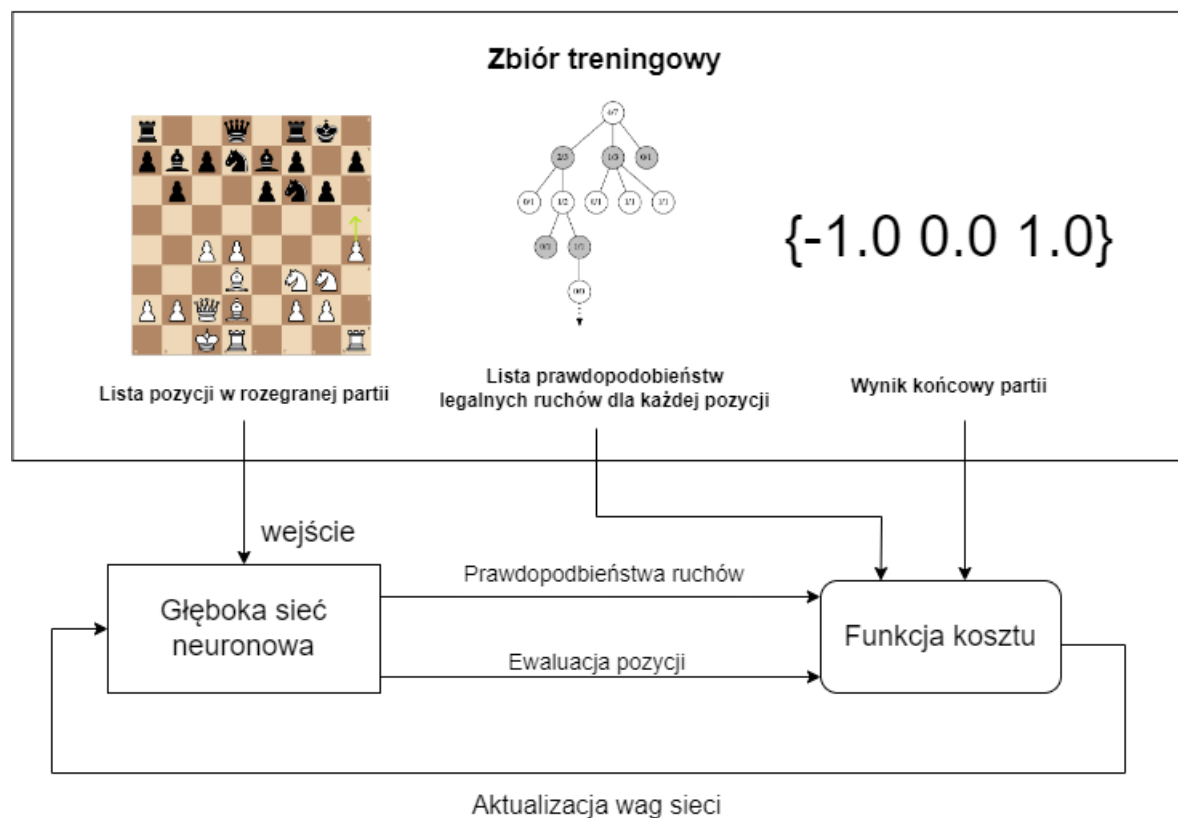
Proces uczenia silnika oparty został o rozgrywanie partii z samym sobą. Umożliwia on osiągnięcie nadludzkich umiejętności szachowych, które potrafią przewyższyć najlepsze silniki klasyczne. Zrezygnowanie z uczenia sieci na podstawie danych historycznych posiada jedną kluczową zaletę - eliminację



Rysunek 3.3: Kodowanie przykładowej pozycji szachowej dla figur [13].

błędu ludzkiego. Szachy to gra, która nie została rozwiązana, tzn. nie istnieją algorytmy lub sposoby, które są w stanie jednoznacznie określić każdą pozycję szachową, podać sekwencje ruchów, która prowadzi to gwarantowanego zwycięstwa, czy remisu. Używając danych historycznych model nie potrafiłby wyjść poza narzucone ramy, opierając się na ewaluacji konkretnych pozycji, które niekoniecznie są prawidłowymi oraz obierając wzorce wykonywania ruchów, które mogą być nieoptymalne. Pozwalając mu uczyć się od początku na podstawie obserwacji, błędów i sukcesów wynikających z rozgrywanych partii i symulacji MCTS, silnik odkrywa samemu najlepsze taktyki i wychodzi poza ludzkie rozumienie gry, mimo że w początkowym etapie nauki wykonuje wręcz losowe ruchy. Analizując partie AlphaZero można zauważyć, że styl gry znacząco odbiega od silników klasycznych, które starają się naśladować ludzki tok rozumowania. Jest on częściej skłonny do poświęceń materiału w celu uzyskania większej aktywności na szachownicy. Jego myślenie jest bardziej strategiczne. Często we wczesnym etapie gry i grze środkowej wykonuje ruchy, których arcymistrzowie i silniki klasyczne nie brałyby pod uwagę, a które okazują się genialne dopiero po przejściu do końcowej fazy gry.

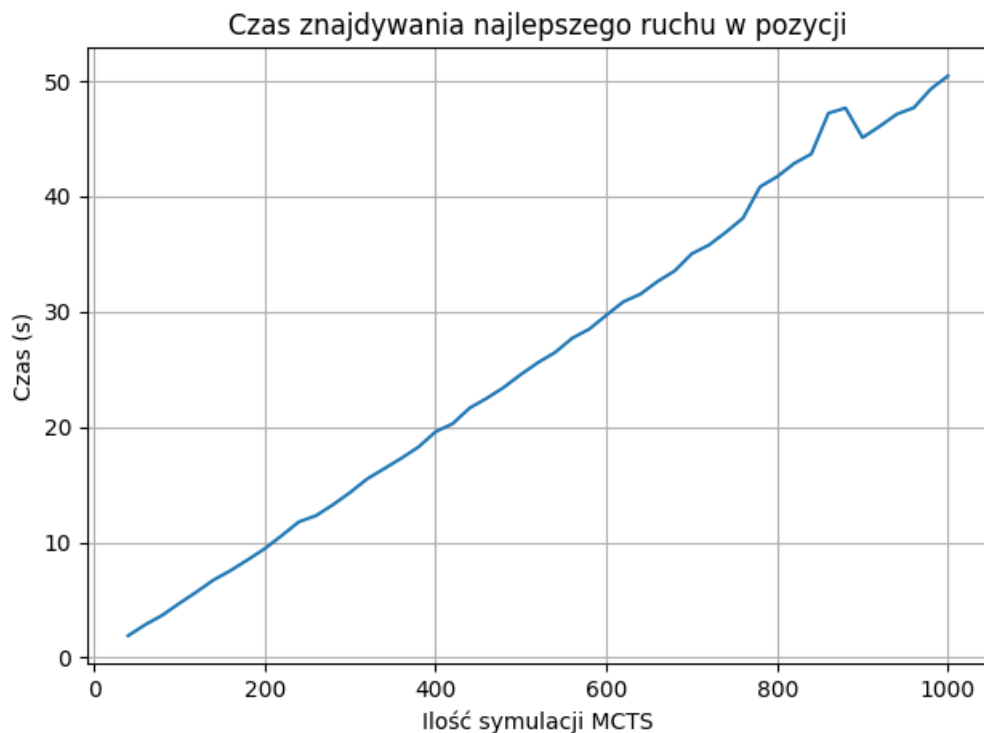
Proces uczenia sieci



Rysunek 3.4: Proces uczenia sieci na podstawie danych z rozegranej partii.

W procesie uczenia wykonywano 200 symulacji MCTS w celu znalezienia jednego ruchu. Wykres 3.5 przedstawia liniową zależność czasu potrzebnego na znalezienie najlepszego ruchu w zależności od ilości symulacji Monte Carlo. Wartość $N_{MCTS} = 200$ wybrano tak, aby umożliwiała ona rozegranie dużej ilości partii w procesie uczenia, a jednocześnie dawała dobre rezultaty i rozszerzała drzewo poszukiwań na kilka posunięć do przodu. Operacje wykonano na sześciordzeniowym procesorze Intel(R)

Core(TM) i7-9750H o częstotliwości taktowania 2.60 GHz oraz karcie graficznej NVIDIA GeForce GTX 1660Ti.



Rysunek 3.5: Wykres obrazujący ilość czasu potrzebną na znalezienie najlepszego ruchu w pozycji.

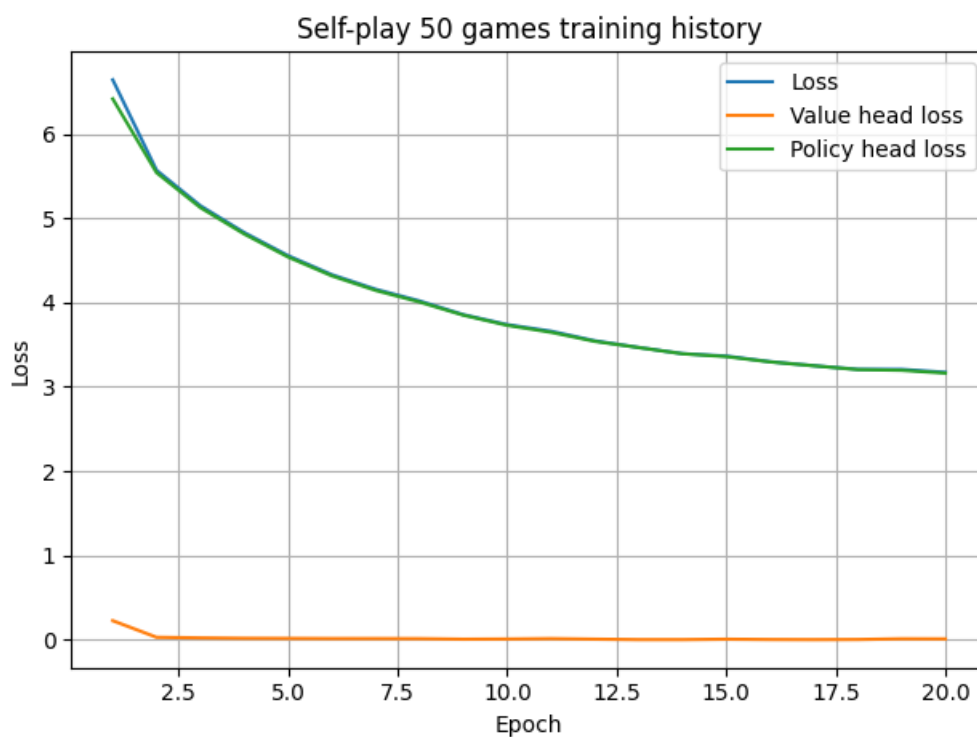
Proces uczenia został podzielony na kilkanaście bloków treningowych, z których każdy składał się z 20 lub 50 partii. Ilość rozegranych partii w jednym bloku powinna być większa, na przykład 1000 lub 2000, jednak z uwagi na brak wystarczającej mocy obliczeniowej została ona zmniejszona. W celu przyspieszenia procesu nauki skorzystano z pomocy silnika Stockfish. Jeśli partia nie zakończyła się w 200-tu ruchach, zwracał on ewaluację pozycji i na tej podstawie określano nagrodę. Rozwiązanie to ma swoje zalety i wady. Pozwala sieci na szybszą naukę kluczowych zasad gry w szachy, jednak dodaje do sieci negatywny bias.

Po każdej z rozegranych partii zapisywano dane zebrane w trakcie rozgrywki. Wykresy 3.6 oraz ?? przedstawiają historie nauki sieci dla jednej i 50 gier. Strata głowy polityki była mierzona za pomocą kategoriycznej entropii krzyżowej (ang. categorical cross-entropy). Natomiast stratę głowy wartości mierzono za pomocą błędu średniokwadratowego (ang. mean squared error).

W początkowym etapie nauki (pierwsze 1000 partii) parametr c odpowiadający za eksplorację i eksploatację w drzewie MCTS został dobrany tak, aby zapewnić dużą eksplorację. Jego wartość wynosiła 3.0. W późniejszym etapie nauki była ona stopniowo zmniejszana, a polityka wybierania ruchów stawała się bardziej zachłanna (z ang. greedy).



Rysunek 3.6: Historia uczenia modelu danymi pochodzącymi z jednej partii.



Rysunek 3.7: Historia uczenia modelu danymi pochodzącymi z 50 partii.

3.4. Aplikacja internetowa

Szachy cieszą się ogromną popularnością na całym świecie. Obecnie na rynku istnieje bardzo dużo stron internetowych umożliwiających użytkownikom grę w szachy przez internet. Dwie wiodące prym to *chess.com* oraz *lichess.org*. Aby umożliwić użytkownikom korzystanie z stworzonego silnika szachowego, w tej pracy również została stworzona strona internetowa. Pozwala ona na zmierzenie się z silnikiem w rozgrywce szachowej. Całość została wdrożona i umieszczona w chmurze Microsoft Azure.

3.5. Interfejs graficzny

Część wizualna strony została stworzona z wykorzystaniem języków Typescript, HTML 5, CSS oraz biblioteki React.js. W rozgrywce zadbano o zapewnienie jak najlepszego doświadczenia użytkownika (z ang. User Experience, UX). Wprowadzono dźwięki informujące o wykonaniu ruchu oraz wyświetlające się podpowiedzi możliwych ruchów na szachownicy. Istnieje możliwość cofnięcia ruchu, a nad planszą wyświetla się aktualna ocena pozycji pochodząca od stworzonego silnika oraz silnika Stockfish. Ciekawym aspektem GUI jest logo aplikacji, które zostało wygenerowane przez sztuczną inteligencję, a konkretnie przez system DALL-E 2 [16], stworzony przez firmę OpenAI. System ten potrafi generować realistyczne obrazy na podstawie tekstowego opisu podanego przez użytkownika. Jednym z jego rzeczywistych zastosowań jest tworzenie obrazów, służących do tworzenia, bądź inspiracji przy tworzeniu interfejsów.

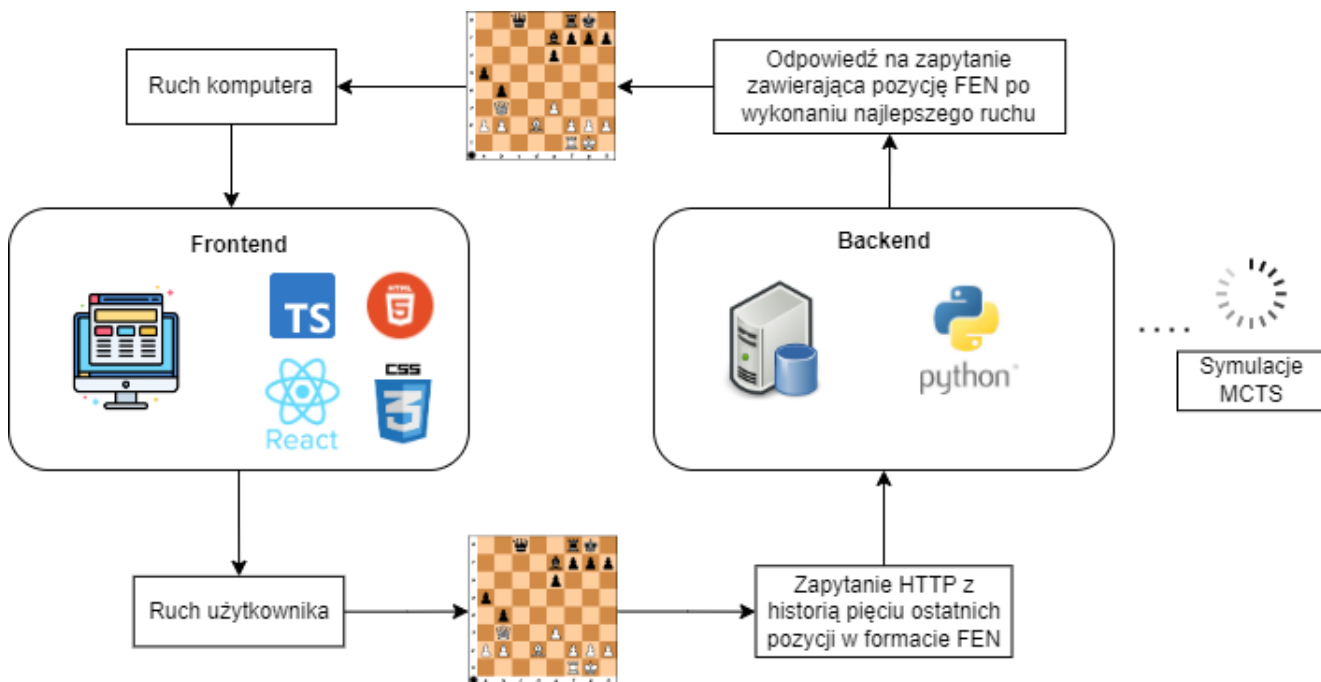


Rysunek 3.8: Interfejs graficzny aplikacji internetowej.

3.6. Komunikacja z serwerem

Aplikacja webowa działa na zasadzie komunikacji interfejsu i serwera, które komunikują się za pomocą REST API i protokołu HTTP. Wykonanie ruchu przez użytkownika, poprzez przesunięcie figury

na szachownicy wywołuje wysłanie zapytania przez interfejs użytkownika. Zawiera ono pięć ostatnich pozycji w notacji FEN, przechowywanych w postaci łańcucha znaków. Serwer aplikacji został napisany w języku Python, odbiera zapytanie i automatycznie zwraca odpowiedź *HTTP 200 OK*, a następnie uruchamia skrypt wywołujący symulację MCTS. Po zakończeniu procesu do interfejsu wysyłana zostaje ewaluacja obecnej pozycji szachowej oraz proponowany najlepszy ruch w pozycji, które następnie wyświetlane są na interfejsie użytkownika.



Rysunek 3.9: Schemat działania aplikacji internetowej.

4. Badania eksperymentalne

W trakcie procesu uczenia zanotowano przebieg rozgrywki w grach treningowych nr 150 oraz nr 500. Partie zostały zapisane w formacie PGN.

Gra nr 150

1. Nf3 d6 2. e3 a5 3. Nc3 Bh3 4. g4 Nc6 5. Nh4 Na7 6. Be2 Nh6 7. Rb1 e5 8. Nb5 Qb8 9. Rg1 c6 10. a3 Bxg4 11. Bc4 Kd8 12. Bf1 f5 13. b4 a4 14. Rh1 Nxb5 15. d4 Bh3 16. c4 Kd7 17. f3 exd4 18. Bxh3 Nc3 19. Qe2 Kc8 20. Qd2 Rg8 21. Bf1 Kd8 22. Qd1 d5 23. Rg1 Ra6 24. Qc2 Bc5 25. f4 Kc8 26. Qd1 Kc7 27. Rxg7+ Be7 28. Kd2 dxe3+ 29. Kc2 b6 30. Rg5 Rh8 31. Bg2 Bf6 32. Qh1 Bd8 33. b5 Rg8 34. Kb2 Re8 35. Rg4 Kb7 36. Qd1 Nxb5 37. h3 Qc7 38. Qg1 Nxa3 39. Nf3 fxg4 40. Qh1 Kc8 41. Bf1 Rg8 42. Ka1 gxh3 43. Nd2 e2 44. Bxa3 Qa7 45. Rc1 Qg7+ 46. Ka2 Kc7 47. Nf3 h2 48. Bb4 Qe7 49. Ne5 Rg5 50. Bc5 Qf8 51. Qxh2 exf1=B 52. Be3 Nf5 53. Bc5 bxc5 54. Kb1 Rg6 55. Qh1 Kb7 56. Qe4 Qd6 57. Qe3 Rb6+ 58. Ka1 Qe6 59. Rd1 Kc8 60. Rxd5 Be2 61. Nd7 Rg2 62. Nf6 Rf2 63. Qd4 Bxc4 64. Qd3 Ne7 65. Qc3 h5 66. Qd4 Qf5 67. Rxd8+ Kb7 68. Rf8 Ra2.

Wygrały bierki koloru czarnego po macie w 68 ruchu. Gra cechowała się dużą losowością, a silnik bardzo rzadko przejawiał inteligentne zachowania.

Gra nr 500

1. h3 Nc6 2. b3 b6 3. e3 Rb8 4. Ba3 a6 5. Ne2 Ra8 6. Nd4 Rb8 7. Nxc6 h5 8. Nxb8 e6 9. Nc6 Qg5 10. Ne7 Qh4 11. Nxg8 e5 12. Nf6+ gxf6 13. c3 Qg3 14. Qc1 f5 15. Bb4 Qg6 16. Bc4 h4 17. Qa3 Rh7 18. Qa4 Qc6 19. a3 Qc5 20. Kd1 Rh6 21. Ba5 Rg6 22. Kc2 Rxg2 23. Bb4 Rg4 24. f3 Qb5 25. Qxa6 Bxa6 26. Rh2 Qxb4 27. Kd3 Qxc3+ 28. Ke2 Kd8 29. Ke1 Bb5 30. Re2 f4 31. exf4 Qb2 32. Bd3 d5 33. Bg6 Qxb3 34. a4 Bxe2 35. Na3 Bf1 36. Bh5 Bh6 37. Kxf1 d4 38. a5 Qa2 39. Nc4 Qxa1+ 40. Kf2 Bg7 41. Ne3 Qc3 42. Kf1 d3 43. axb6 Qa1+ 44. Kf2 Qa4 45. Ke1 Bh6 46. Nd5 Rg5 47. b7 Qa6 48. Kf2 Qe6 49. Bg4 Rg6 50. Kg1 Qc6 51. b8=N Qc4 52. Kf2 Qa6 53. Nxa6 Rb6 54. Ndxc7 Rb4 55. Na8 Rxf4 56. Kg2 Bf8 57. Kh2 Ba3 58. Bd7 f6 59. N6c7 Bb2 60. Ne6+ Ke7 61. Kg1 Re4 62. Nd8 Kxd8 63. Bf5 Re3 64. Kh2 Re2+ 65. Kg1 Re1+ 66. Kf2 e4 67. Bg4 Rc1 68. Nb6 Rc5 69. Na4 Rb5 70. Kg1 exf3 71. Bxf3 Rb4 72. Nc3 Kc7 73. Nb5+ Kb8 74. Bg4 Rxg4+ 75. Kf1 Rg7 76. Nd4 f5 77. Ne6 Rg5 78. Nf4 Ba3 79. Ng2 Kc8 80. Ke1 Kc7 81. Kf1 Bb4 82. Kf2 Rg4 83. Ne1 Kc8 84. Kf1 f4 85. Nxd3 Bc3 86. hxg4 Kc7 87. g5 Kd8 88. Kg1 f3 89. Kh2 Kc7 90. Ne5 h3 91. Ng4 Kc6 92. Ne3 Kd6 93. Kxh3 Kd7 94. g6 Ke8 95. Nc4 f2 96. Kh4 f1=R 97. g7 Kd8 98. Kg4 Kd7 99. g8=Q Re1 100. Qh8 Re8 101. Nb2 Ke6 102. d3 Bxh8 103. Na4 Kd6 104. Kf4 Bc3 105. Kf3 Bd4 106. Kg3 Ke6 107. Nb2 Bf2+ 108. Kg2 Kf7 109. Nd1 Re5 110. Kxf2 Re4 111. Kf1 Rb4 112. Nb2 Ke7 113. d4 Rb6 114. d5 Kd8 115. d6 Ke8 116. d7+ Kf7

117. d8=Q Rb5 118. Qc8 Rb6 119. Qd7+ Kf8 120. Qd6+ Kf7 121. Qf8+ Ke6 122. Qh6+ Ke5 123. Qf6+ Kxf6 124. Kg1 Ke7 125. Kf1 Kd8.

Gra skończyła się bez rozstrzygnięcia. W sposobie gry silnika można było zauważyć pierwsze inteligentne zachowania. Coraz częściej wybierał ruchy, które kończyły się zbiciem figury przeciwnika. W końcowym etapie gry widać też, że program rozumie potrzebę szachowania króla przeciwnika.

Po skończonymi procesie nauki, stworzono skrypt, który rozegrał 100 partii, w których decyzje po obu stronach szachownicy podejmował silnik. Wyniki zaprezentowano w tabeli 4.1. Równomierny rozkład wyników świadczy o tym, że sieć neuronowa w trakcie nauki nie nabrała stronniczości i nie zaczęła faworyzować jednej ze stron gry.

Wygrana białych	Remis	Wygrana czarnych
33	40	27

Tabela 4.1: Wynik 100 partii rozegranych przez silnik pomiędzy samym sobą.

4.1. Porównanie z innymi silnikami

W celu weryfikacji poziomu na jakim znajduje się silnik, sporządzony został skrypt, który rozegrał 100 partii z silnikiem o otwartym kodzie źródłowym - Stockfish. Stockfish nie wykorzystuje sieci neuronowych do ewaluacji pozycji oraz znajdowania najlepszych ruchów w pozycji, jednak od lat gości w ścisłej czołówce i jest najczęściej wykorzystywanym silnikiem szachowym na świecie. Stworzony w tej pracy silnik wykonywał 200 symulacji MCTS na jedno posunięcie. Średni czas rozegrania jednej partii wynosił około 30 minut. Partie rozegrano na 3 różnych poziomach trudności, a wyniki zostały zamiesz-

Poziom	Ranking Elo
1	500 - 800 (Amator)
2	900 - 1000 (Początkujący)
3	1100 - 1300 (Średnio zaawansowany)

Tabela 4.2: Poziomy trudności silnika Stockfish.

czone w tabeli 4.3.

Poziom silnika Stockfish	Wygrane	Remis	Przegrane
Amator	4	1	5
Początkujący	1	3	6
Średnio zaawansowany	0	0	10

Tabela 4.3: Wyniki partii rozegranych z silnikiem Stockfish.

Na podstawie rozegranych partii można określić ranking Elo utworzonego silnika na poziomie około 700 punktów.

4.2. Wnioski

Głębokie uczenie ze wzmocnieniem oraz algorytm AlphaZero to bardzo potężne narzędzia, dzięki którym jesteśmy w stanie rozwiązać bardzo skomplikowane problemy. Należy jednak mieć na uwadze, że osiągnięcie końcowego sukcesu zależne jest od posiadanej mocy obliczeniowej - zwłaszcza przy wykorzystywaniu algorytmu AlphaZero. Co ważne istotną rolę odgrywa zarówno CPU - służące do przeprowadzania symulacji MCTS - jak i GPU, które lepiej radzi sobie w procesie nauki głębokiej sieci neuronowej, dzięki lepszemu przystosowaniu do obliczeń macierzowych. W czasie tworzenia pracy silnik przeszedł około 1500 godzin nauki, co pozwoliło rozegrać około 2000 partii. AlphaZero Chess rozegrał ich 800 000, a dopiero przy liczbie około 20 tysięcy rozegranych gier zaczął przejawiać inteligentne zachowania [20]. Liczby podano dla lepszego zobrazowania skali problemu. Proces uczenia sieci to również trudne zadanie. O jego rezultatach dowiadujemy się po wielu godzinach symulacji, co uniemożliwia dowolne testowanie parametrów modelu i dobieranie ich w sposób empiryczny. W skomplikowanych zadaniach bardzo łatwo o błąd implementacyjny, który może wyjść na jaw dopiero w momencie zorientowania się, że wielogodzinny proces nauki nie przynosi rezultatów. Dlatego bardzo istotne jest dokładne przetestowanie programu oraz stworzenie scenariuszów testowych przed rozpoczęciem procesu uczenia. Warto pamiętać, aby w początkowym etapie zadbać o dużą eksplorację środowiska i stopniowo zmniejszać ją wraz z kolejnymi iteracjami, stawiając na wybór sprawdzonych wariantów. Jeżeli nie dysponujemy ogromną mocą obliczeniową, to zdecydowanie lepszy rezultat osiągniemy implementując tradycyjny silnik szachowy, który nie wykorzystuje sieci neuronowych oraz procesu *self-play*.

5. Podsumowanie

Szachy to jedna z najpiękniejszych i najtrudniejszych istniejących gier planszowych. W wielu krajach, szkoły podstawowe wprowadzają je jako przedmiot obowiązkowy w celu rozwoju wyobraźni, zdolności analitycznych i strategicznego myślenia. Wysoki poziom skomplikowania, przez lata utrzymywał ludzi w przekonaniu, że szachy to gra nierozwiązywalna. Powstanie silników szachowych, wzrost mocy obliczeniowej oraz coraz to wydajniejsze algorytmy sprawiły, że człowiek zrobił duży krok w kierunku pełnego rozwiązania i zrozumienia tej gry. AlphaZero bardzo zbliżył się do osiągnięcia szachowej perfekcji i wskazał drogę, która powinniśmy kierować się w przyszłości, przy rozwiązywaniu podobnych problemów.

Silnik stworzony w tej pracy nie osiągnął nadludzkich umiejętności, jednak efekty są zadowalające, ponieważ już po rozegraniu kilku tysięcy gier zaczął przejawiać inteligentne zachowania. Głębokie uczenie ze wzmocnieniem to gałąź uczenia maszynowego, która pozwala na zbudowanie najlepszych możliwych modeli. Należy pamiętać, że wymaga to dużej mocy obliczeniowej.

Utworzona strona internetowa, może być idealnym początkiem szachowej przygody dla nowych graczy. Poziom silnika odpowiada poziomowi amatorów co sprawia, że początkujący szachiści mogą rozgrywać partie na równym poziomie.

5.1. Potencjalne możliwości rozwoju pracy

Jako, że zagadnienie silników szachowych jest bardzo szerokie, istnieje bardzo dużo potencjalnych kierunków rozwoju pracy. Aby poprawić wydajność programu i zoptymalizować proces uczenia, można wprowadzić obliczenia wielowątkowe, a algorytm MCTS oraz proces *Self-play* mógłby zostać zaimplementowany w języku C lub C++. Niewątpliwie skróciłoby to czas rozgrywania partii w procesie uczenia, a to poskutkowałoby możliwością rozegrania większej ilości gier.

Ciekawym zagadnieniem wydaje się również stworzenie silnika hybrydowego, który korzystałby z sieci neuronowych i symulacji MCTS w środkowym etapie gry, natomiast we wczesnym etapie oraz w końcówkach, korzystałby z biblioteki najsilniejszych otwarć oraz tablic gry końcowej. Kolejnym ważnym zagadnieniem jest wybór algorytmu poszukującego najlepszy ruch w pozycji. Istnieje bardzo dużo innych rozwiązań, na które można zamienić Monte Carlo Search Tree. Na przykład *Alpha-Beta Pruning* oparty na algorytmie *Min-Max*, który bardzo często wykorzystuje się w klasycznych silnikach.

Bardzo ciekawym wydaje się również zaimplementowanie algorytmu AlphaZero w świecie rzeczywistym, gdzie głęboka sieć neuronowa odzwierciedlałaby realny świat, a agent wyposażony w sensory

dostarczałyby nieustannie informacji i uczył się.

Bibliografia

- [1] Drzewo gry - monte carlo tree search. 2019.
- [2] P. Baheti. Deep reinforcement learning guide. 2022.
- [3] H. Berliner. System: A world champion's approach to chess. 1999.
- [4] A. Champion. Dissecting stockfish part 1: In-depth look at a chess engine. 2021.
- [5] J. Devanesan. Has google cracked the data center cooling problem with ai? 2020.
- [6] I. C. Education. Neural networks. 2020.
- [7] N. Fiekas. Python chess library.
- [8] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.
- [9] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau. An introduction to deep reinforcement learning. *CoRR*, abs/1811.12560, 2018.
- [10] Google. Tensor processing units. 2022.
- [11] S. Jadon. Introduction to different activation functions for deep learning. *Medium, Augmenting Humanity*, 16, 2018.
- [12] J. W. Key. Sieci neuronowe w sterowaniu procesami technologicznymi. 2017.
- [13] D. Klein. Neural networks for chess. 2022.
- [14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.
- [15] OpenAI. Introduction to reinforcement learning. 2018.
- [16] OpenAI. Dall-e 2. 2022.
- [17] T. J. Sejnowski. The deep learning revolution. 2018.
- [18] C. E. SHANNON. Programming a computer for playing chess. 1950.
- [19] C. Shorten. Introduction to resnets. 2019.

- [20] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.
- [21] H. A. M. Silver, D. Mastering the game of go with deep neural networks and tree search. 2016.
- [22] N. Software. Tic-tac-toe with tabular q-learning. 2019.
- [23] J. TORRES. The bellman equation. 2020.
- [24] O. Zeigermann. How do chess engines work? looking at stockfish and alphazero. 2019.