Politechnika Wrocławska Wydział Elektroniki Kierunek Teleinformatyki

Inżynierska praca dyplomowa

# Projekt sztucznej inteligencji

**Michał Żarejko**


| | |
|---|---|
| Dyplomant: | **Michał Żarejko** |
| Nr albumu: | **249374** |
| Promotor: | **dr inż. Paweł Zyblewski** |


Wrocław. 2021

# Spis treści

# Rozdział 1

# Wstęp

Uczenie maszynowe jest zagadnieniem rozwijanym od dłuższego czasu. Już w latach 40 powstawały książki oraz programy powiązane ze sztuczną inteligencją [1]. Między innymi w 1940 roku Donald Hebb stworzył podstawy teoretyczne wykorzystane w późniejszych sieciach neuronowych [1]. Po mimo, wielu wczesnych pomysłów omawiany dział nauki, dopiero od niedawna zaczął osiągać wielkie sukcesy. Wynika to z faktu, że wiele algorytmów potrzebuje dużej mocy obliczeniowej i dopiero nowoczesny sprzęt był w stanie spełnić takie wymagania [1].

Dzisiaj można wymienić wiele rozwijanych narzędzi związanych z tym tematem. Między innymi asystenci głosowi, tłumacze językowe, modele wyświetlające elementy na stronach internetowych, gry wideo lub inteligentne samochody. Dodatkowo sztuczna inteligencja jest mocno wykorzystywana w firmach, na halach produkcyjnych, w transporcie, medycynie albo cyberbezpieczeństwie.

Pomimo tylu możliwości i zastosowań AI zyskuje aktualnie największą popularność medialną przez gry rywalizacyjne, gdzie głównym zadaniem jest pokazanie przewagi algorytmów względem ludzi. Dzisiaj można wymienić wiele takich wydarzeń, gdzie mistrzowie świata w danej grze, przegrywali z modelami rozpoznawania.

Między innymi w 2016 roku zorganizowano mecz między Fan Hui, mistrzem Europy w chińskiej grze Go oraz algorytmem AlphaGo [2]. Model utworzony przez zespół DeepMind osiągnął duży sukces przez wygraną z przeciwnikiem. Do tej pory gra była uważana powszechnie za skomplikowaną i trudną do rozwiązania.

W 2019 roku utworzono model zwany OpenAI Five czyli pierwsze na świecie AI, które pokonało zespół Team OG w rywalizacji e-sport w grze Dota 2 [3]. Wyzwanie polegało na rywalizacji 5-osobowych zespołów. Wydarzenie było mocno omawiane w mediach z powodu pierwszego takiego osiągnięcia w tej dziedzinie sportu. Dodatkowo Dota 2 była bardzo skomplikowanym środowiskiem. Przykładowo gra Go rozwiązana pare lat wcześniej, zawierała 150 możliwych ruchów na turę, Dota 2 mogła posiadać ich 20 000 w niecałą godzinę [3].

Istnieje wiele takich wydarzeń. Pokazują one, że w dzisiejszych czasach sztuczna inteligencja może przewyższać myśleniem strategicznym człowieka. Dodatkowo udowadniają one, że temat AI jest dalej rozwijany i zyskuje coraz większe zainteresowanie.

## 1.1 Geneza tematu pracy

Często w tworzeniu programów sztucznej inteligencji dużym wyzwaniem jest poziom skomplikowania gry. Zależy to między innymi od typu środowiska np. deterministycznego lub stochastyczne, od poziomu dynamiki gry lub od tego, czy przestrzeń wymiarowa jest dyskretna, lub nieskończona. Aktualnie jednak jednym z większych problemów takich programów jest niedostateczny zakres dostępnych informacji o środowisku [4].

Sztuczna inteligencja, aby zwyciężać, musi zostać nauczona grać, więc potrzebuje dużej ilości danych wejściowych, które są rozróżnialne. Przykładem gry, która jest pozbawiona tego problemu, są szachy. AI wykonuje ruchy, bazując na informacjach jak np. ułożenie pionków w danej turze. Widoczna zmiana stanu środowiska przeciwnika jest zauważalna przez gracza, przez co może on łatwiej powiązać obserwacje z wykonywanymi akcjami.

Przykładem gry ciężkiej do rozwiązania w której, występuje niepełny zestaw informacji, jest Poker Texas Hold'em, pomimo wiedzy o kartach w ręce i na stole, gracz nie posiada wiedzy o kartach przeciwników. W takim przypadku dwa pozornie identyczne stany środowiska w rzeczywistości mogą się różnić. Z powodu takich cech większość popularnych algorytmów jak DQN (*Deep Q Learning*), Actor-Critic lub AlphaZero staje się bezużyteczna i nie daje dobrych rezultatów.

W niniejszej pracy przedstawiono sposób możliwego rozwiązania takiego problemu przy pomocy algorytmu Deep CFR [5]. Jest to popularna metoda do tworzenia modeli rozpoznawania w grach karcianych.

## 1.2 Cel i zakres pracy

Głównym celem pracy jest implementacji algorytmu Deep CFR, który stworzy 5 modeli rozpoznawania w grze HULH (*Heads Up Limit Texas Poker Hold'em*). Jest to popularna wersja rozgrywki 2-osobowej, gdzie uczestnicy nie mogą wybrać samodzielnie kwoty podbicia stawki. Jest ona ograniczona przez ustaloną wartość. Takie środowisko minimalizuje możliwe ruchy do 3 akcji co czyni go prostszą bazą do uczenia maszynowego. Uzyskane modele zostaną następnie wykorzystane do stworzenia rozgrywek składających się na wszystkie kombinacje dwóch modeli, gdzie każda gra zostanie powtórzona 100 razy. Taki proces pozwoli określić, który modela posiada najwięcej wygranych meczów i jest najbardziej dopracowany.

Praca została podzielona w tym celu na rozdziały opisujące każdy z etapów projektu. Pierwszy przedstawia możliwe rozwiązania problemu, analizę gry Poker Texas Hold'em oraz wymagana teorię do zrozumienia algorytmu. Trzecia część przedstawia implementacje Deep CFR wraz z użytymi technologiami. Ostatnia część przedstawia wyniki rozgrywek modeli oraz podsumowanie algorytmu.

# Rozdział 2

# Analiza istniejących rozwiązań

W ciągu ostatnich 10 lat powstało wiele algorytmów rozwiązujących różne wersje gry Poker. Między innymi CFR (*Counterfactual Regret Minimization*) [7], XFP (*Extensive-Form Fictitious Play*) [6] lub NFSP (*Neural Fictitious Self-Play*) [8]. Pierwszy z wymienionych, CFR powstał w 2007 roku. Był pomyślną prób rozwiązania abstrakcyjnego środowiska Texas Hold'em [7]. Na jego podstawie utworzono wiele nowoczesnych algorytmów, które dają szanse rozwiązać takie środowiska jak HULH [7].

Z wymienionych metod zaimplementowanym rozwiązaniem w niniejszej pracy jest CFR rozszerzony o sieci neuronowe, czyli Deep CFR z grą HULH. Pozwala on na szybsze trenowanie modeli, dodatkowo jest prostszy niż bazowy CFR [5].

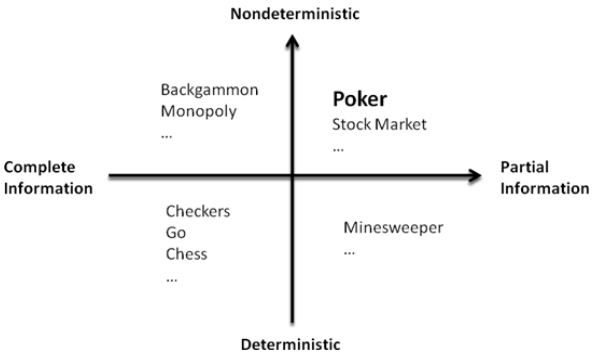
W tym rozdziale zostanie przedstawiona argument, dlaczego gra Poker Texas Hold'em nie bazuje tylko na szczęściu i utworzenie dla niej modelu rozpoznawania jest możliwe. Następnie zostanie opisany algorytm CFR oraz jego następca, czyli MCCFR. Ostatnim etapem będzie przedstawienie Deep CFR.

## 2.1 Analiza Texas hold'em Poker

Omawiana gra jest jedną z najpopularniejszych gier rywalizacyjnych w kasynach, dodatkowo jest to dominująca gra hazardowa. Można ją scharakteryzować brakiem stanów deterministycznych oraz częściową obserwowalnością, rys. 2.1 [9].

Przez takie cechy gra była od zawsze tematem sporów, czy na jej wynik ma większy wpływ losowość, czy umiejętności. Jak wynika z badań dużym aspektem pomagającym w osiągnięciu zwycięstwa, jest panowanie nad emocjami, dokładna analiza stanu gry oraz umiejętność opóźnienia natychmiastowej nagrody [10].

Dodatkowo ważnym elementem jest umiejętności obserwacji gry oraz wybór prawidłowej strategii. Na podstawie tego, można sklasyfikować graczy na cztery kategorie.




Rysunek 2.1: Charakterystyka gier [9].

### Loose Passive

Osoba, która bardzo często wchodzi do gry niezależnie czy karty, które posiada, dają jej wysokie szanse na wygraną. Ten typ gry charakteryzuje się częstym wykonywanie akcji 'call' oraz małą ilością przebić stawki [11].

### Loose Aggressive

Ten typ gry określa osobę, która często wykonuje akcję 'raise' i 're-raise' pod warunkiem, że ma silne karty [11]. W innym przypadku wyrównuje żetony do aktualnej stawki. Taka strategia okazuje się nieefektywna w przypadku gry z osobami, które często wchodzą do gry niezależnie od kart jak na przykład typ 'Loose-Passive' [11].

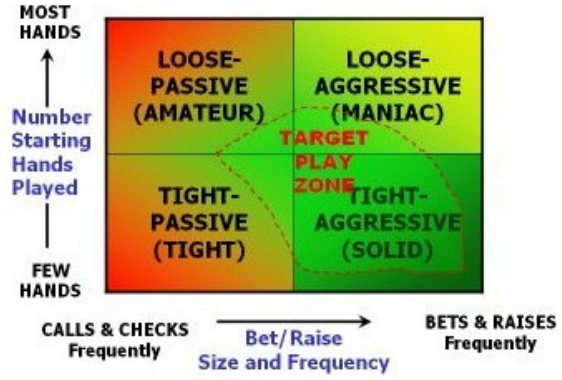
### Tight Passive

Można poznać tę kategorię gry przez to, że uczestnik wchodzi tylko z dobrymi kartami i często pasuje przy spotkaniu z graczem agresywnym, który przebija. Taki gracz traci wiele okazji, kiedy mógłby wygrać [11].

### Tight Aggressive

Typ gry popularny wśród profesjonalnych graczy i najtrudniejszy do opanowania [11].

Jak wynika z powyższych kategorii, gra 'Poker Texas Hold'em' zawiera wiele elementów niezwiązanych z losowością, gdzie obserwacje i dobieranie odpowiedniej strategi do typu gracza pełni kluczową funkcję. W takim wypadku jest uzasadnione wysunięcie tezy, że można utworzyć sztuczną inteligencję, która będzie bazować na podobnych założeniach, które mogłyby być charakterystyczne dla profesjonalnych graczy.

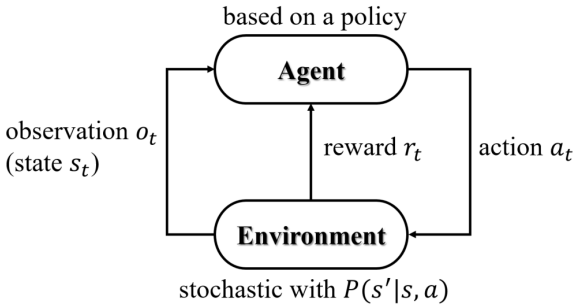



Rysunek 2.2: Podział graczy [11].

## 2.2 Uczenie przez wzmacnianie

Jest wiele sposobów na tworzenie sztucznej inteligencji do gier, między innymi można użyć technik uczenia nadzorowanego pod warunkiem, jeśli przygotuje się odpowiednie zbiory danych. W pracy jednak zdecydowano się na uczenie przez wzmacnianie. Wynika to z faktu, że jest mało publicznych zapisów gry profesjonalnych graczy, które mogłyby posłużyć jako zbiory uczące. Algorytmy należące do wybranego działu uczenia powinny być w stanie polepszać swoje wyniki na podstawie interakcji ze środowiskiem bez korzystania z zewnętrznych materiałów.

Wiele istniejących algorytmów należących do wybranej techniki zakłada, że środowisko można opisać przez model matematyczny MDP (*Markov Decision Process*). Określa ona sekwencyjnie podejmowane decyzje w niepewnym środowisku [12]. W każdym z nowych stanów, w jakich znajduje się *Agent*, wykonuje on pojedyńczą akcję, zyskując od środowiska informacje o nowym stanie oraz nagrodzie. Elementem wyjściowym zasady powinien być zbiór strategii w postaci modelu, rys. 2.3.




Rysunek 2.3: Schemat interakcji ze środowiskiem [12].

MDP może opisywać jedynie środowiska z pełnym zakresem informacji, w przypadku algorytmu będącego tematem pracy, głównym zadaniem jest rozwiązanie środowiska z niepełnym zestawem danych. Wtedy należy rozpatrywać zasadę Partially Observable Markov Process, POMDP [12].

W przeciwieństwie do poprzedniej zasady tutaj agent nie zna aktualnego stanu, w którym się znajduje [12]. Przez takie okoliczności musi połączyć zależnością wykonywane akcie i obserwacje, a nie stany. Większość gier karcianych można zakwalifikować do tego typu problemów [12]. Dodatkowo gry karciane można powiązać z terminem "Teoria Gier".

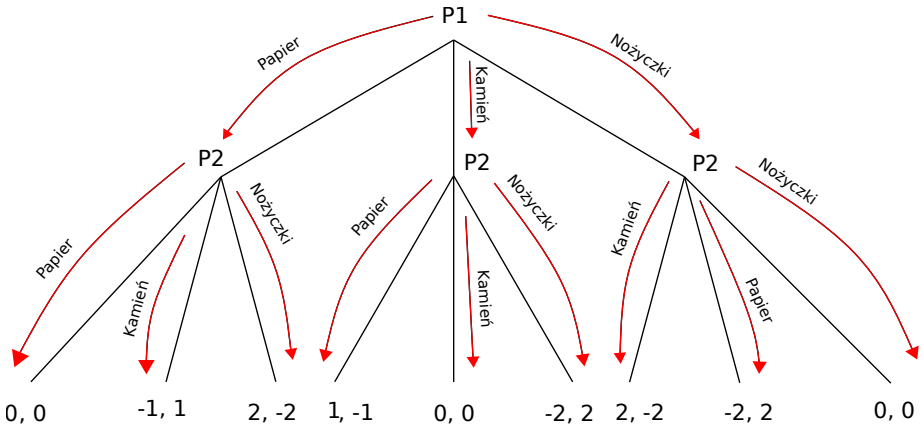
## 2.3 Teoria Gier

Aby zrozumieć działanie algorytmów CFR, Deep CFR, MCCFR itd. należy zapoznać się z podstawami działu matematyki o nazwie Teoria Gier. Bada on optymalne zachowanie w środowiskach gdzie występuję konflikt [14]. W przypadku gry Poker Texas Hold'em zostaną wytłumaczone tylko takie terminy jak Równości Nasha lub gra w postaci ekstensywnej.

### Gra w postaci ekstensywnej

Gry w formie ekstensywnej można przedstawić jako drzewo decyzyjne, gdzie każdy węzeł rozgałęzia się na możliwe akcje oraz identyfikuje aktualny stan gracza przez zestaw informacji, ostatnie węzły to stany końcowe gdzie określony gracz zyskuje nagrodę lub ją traci [14]. Jest to sposób na uproszczenie opisu gry.

Na rys. 1.4 przedstawiono przykład gry 'Papier-Kamień-Nożyce' w formie ekstensywnej, gdzie gracze P1 i P2 eksplorują 3 akcie w swoich węzłach.




Rysunek 2.4: przykład gry w postaci ekstensywnej.

Każdy z graczy eksploruje wyniki swoich akcji oraz zapamiętuje dotychczasową historię, co może zostać potem wykorzystane do znalezienia najbardziej opłacalnych ścieżek.

**Równowaga Nasha**

W grach to twierdzenie określa perfekcyjny stan gry, gdzie wszyscy gracze wykorzystują najlepszy zestaw strategii, którego zmiana przyniesie tylko straty. Oznacza to, że nie jest możliwa zmiana ruchów oraz zwiększenie uzyskanej nagrody [14].

Dobrym przykładem prezentującym taki stan jest "Dylemat Więźnia" [13]. To środowisko zawiera dwóch przestępców, którzy są przesłuchiwani w odseparowanych pomieszczeniach. Każdy z nich ma dwie opcje, przyznać się do zarzutów lub tego nie robić. Każda z kombinacji akcji uczestników jest zaprezentowana w tab. 2.1, gdzie wartości określają lata spędzone w więzieniu po danym ruchu.

Posługując się Równowagą Nasha, można stwierdzić, że najlepszą opcją dla obu uczestników będzie przyznawanie się za każdym razem [13]. Wynika to z faktu, że wyniki przegranej są tam małe wraz z brakiem ryzykowania porażką, czyli 5 latami w więzienia.

Tabela 2.1: Wyniki akcji środowiska "Dylemat więźnia".

| | przyznanie się więźnia A | więzień A kłamie |
|---|---|---|
| przyznanie się więźnia B | 1 / 1 | 5 / 0.5 |
| więzień B kłamie | 0.5 / 5 | 0 / 0 |

## 2.4 Historia modeli Texas Hold'em Poker

Bazując na teorii gier oraz różnych algorytmach powstało wiele rozwiązań różnych wersji gry Poker. Pierwsze dokumenty naukowe omawiały bardzo proste środowiska jak Poker Kuhn. Dopiero w 2015 roku utworzono pierwszą znaną sztuczną inteligencję "Cepheus" rozwiązującą problem HULH przez algorytm CFR+ [15]. Po tym osiągnięciu rozpoczęto prace nad algorytmem mogącym rozwiązać problem gry HUNH (*Heads Up No-limit Texas Hold'em*). Powstały model nazwano "DeepStack". Mieszał on sieci neuronowe z technikami algorytmu CFR. Przetestowano go na 33 profesjonalnych graczach w wielu iteracjach gry. Algorytm w większości przypadków wygrał [16]. Była to pierwsza wygrana AI z człowiekiem w normalnej wersji gry Poker Texas Hold'em.

Jak pokazują dotychczasowe osiągnięcia Poker Texas Hold'em jest bardzo skomplikowanym środowiskiem do uczenia maszynowego. Sposoby na jego rozwiązanie zaczęły powstawać od niedawna, a pierwsze duże osiągnięcie w grze HUNH miało miejsce dopiero w 2017 roku.

## 2.5 Counterfactual Regret Minimization

### 2.5.1 Regret Matching

Jest to nieodłączna metoda uczenia AI w grach karcianych. Polega ona na liczeniu najlepszej strategii pod warunkiem, że znany jest wektor żalu w węźle. Taki wektor opisuje się jako tablicę wag o długości równej liczbie możliwych akcji gracza. Każda z tych wag opisuje stopień opłacalności danego ruchu.

Poniżej przedstawiono wzór wynikający z tej metody, gdzie $R^t(I,a)$ jest omawianym wektor [7]. Następnie aby uzyskać nową strategię, usuwa się wartości ujemne (formuła nr 1.2) i sprawdza, czy ich suma jest większa od zera. W zależności od tego warunku wybierany jest rozkład, wzór nr 1.1.

$$p_i^t(a) = \begin{cases} \frac{R^{T,+}(a)}{\Sigma_{a' \in A} R^{T,+}(a')} & \text{if } \Sigma_{a' \in A} R^{T,+}(a') > 0; \\ \frac{1}{|A|} & otherwise. \end{cases} \tag{2.1}$$

$$R^{t,+}(a) = max(R^t(a), 0) \tag{2.2}$$

Proces ten jest powtarzany wielokrotnie, tak, aby przy każdej iteracji rozkłady prawdopodobieństwa ruchów były stopniowo poprawiane.

W przypadku algorytmu Deep CFR, zachodzi modyfikacja formuły nr 1.1. Strategia jest liczona na dodatnich wartościach żalu podzielonego przez prawdopodobieństwo dostania się do tego stanu $D^T(I,a)$ [5]. Jeśli suma jest ujemna to zostaje wybrana akcja z najwyższą wartością $D^T(I,a)$ [5].

$$\sigma_i^{t+1}(I,a) = \begin{cases} \frac{D^{T,+}(a)}{\Sigma_{a' \in A} D^{T,+}(a')} & \text{if } \Sigma_{a' \in A} D^{T,+}(a') > 0; \\ \text{argmax}(D^T(I,a)) & otherwise. \end{cases} \tag{2.3}$$

### 2.5.2 Counterfactual Regret

Algorytm CFR do znanych wcześniej metod dodał termin 'Immediate Counterfactual Regret' oznaczany przez $R^T_{i,imm}(I)$, czyli żal przydzielony do węzła I. Do obliczenia takiego parametru została zdefiniowana wartość "counterfactual utility" $u_i(\sigma, I)$. Oznacza ona przewidywany wynik nagrody dla stanu I gdzie wszyscy gracze używają strategi $\sigma$, [7]. Dodatkowo $\pi^\sigma(h,h')$ oznacza prawdopodobieństwo dostania się z historii h do nowego stanu h' przy strategii $\sigma$ [7].

$$u_i(\sigma, I) = \frac{\sum_{h \in I, h' \in Z} \pi^\sigma_{-i}(h) \pi^\sigma(h,h') u_i(h')}{\pi^\sigma_{-i}(I)} \tag{2.4}$$

Na podstawie równania 2.5 można wyliczyć końcową wartość żalu w algorytmie CFR.

$$R^T_{i,\text{imm}}(I,a) = \frac{1}{T}\sum_{t=1}^{T} \pi^{\sigma^t}_{-i}(I)(u_i(\sigma^t|_{I\to a}, I) - u_i(\sigma^t, I)) \tag{2.5}$$

Powyższe 2 równania można doprowadzic do formuły nr 2.6. Wartość $\pi^\sigma(h,h')$ została zastąpiona przez 1, ponieważ CFR zakłada, że dla $u_i(\sigma^t|_{I\to a}, I)$, gracz wykonuje zawsze akcję $a$ [7].

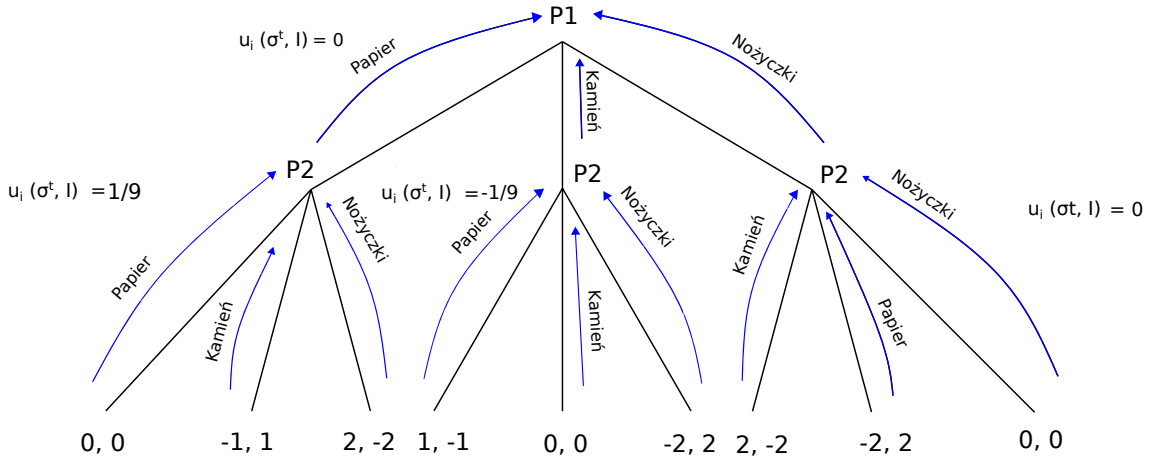
$$R^T_{i,\text{imm}}(I,a) = \frac{1}{T}\sum_{t=1}^{T} \pi^{\sigma}_{-i}(h) \sum_{h\in I, h'\in Z} (1 * u_i(h') - \pi^\sigma(h,h')u_i(h')) \tag{2.6}$$

Po uzyskaniu $R^T_{i,\text{imm}}(I,a)$ można wykorzystać metodę "Regret Matchning" i zaktualizować strategie. Poniżej przedstawiono przykład obliczeń pojedyńczego węzła oraz wyniki dla gry 'Papier-Kamień-Nożyce' przy ustawionych nagrodach i karach w stanach końcowych jak na rys. 2.5, 2.6, 2.7.

$$\pi^\sigma(h,h')u_i(h') = (\frac{1}{3}\cdot 0) + (\frac{1}{3}\cdot -1) + (\frac{1}{3}\cdot 2) = \frac{1}{3}$$

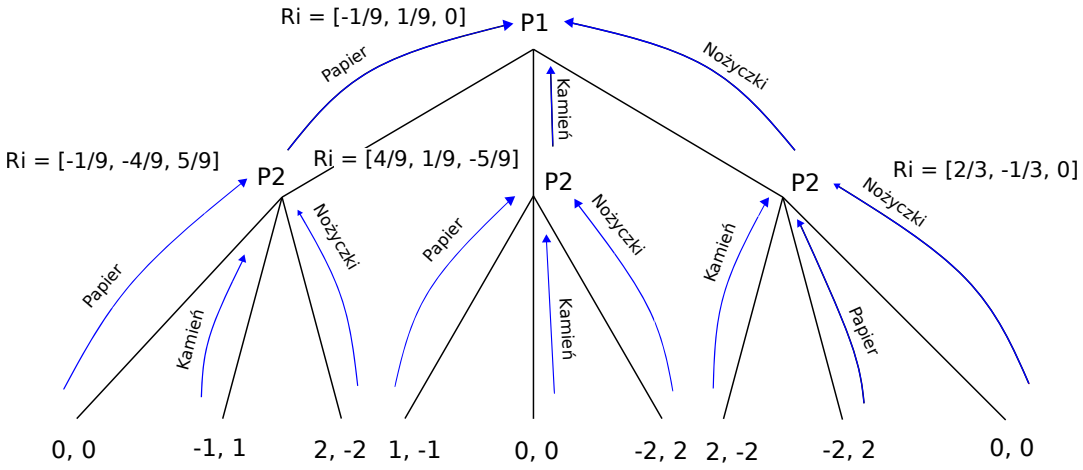
$$T * R^T_{i,\text{imm}}(I,a) = ((0-\frac{1}{3}),(-1-\frac{1}{3}),(2-\frac{1}{3}))\cdot\frac{1}{3} = (-\frac{1}{9}, -\frac{4}{9}, \frac{5}{9})$$

$$\sigma_i^{t+1}(I,a) = (0,0,1)$$



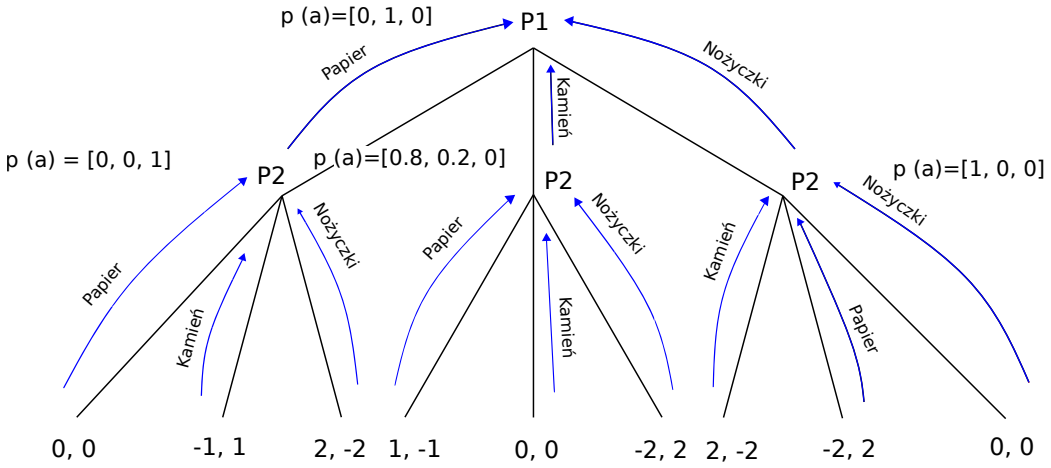

Rysunek 2.5: Przykład 'counterfactual utility'.

Na podstawie rys. 2.5 widać, że gracz P2 będzie miał stan o najwyższej wartości "counterfactual utility" w węźle $u_i(\sigma^t, I) = \frac{1}{9}$, a najniższej dla $u_i(\sigma^t, I) = -\frac{1}{9}$.




Rysunek 2.6: Przykład 'Immediate Counterfactual Regret'.

Powyżej zaprezentowano wektory $R^T_{i,\text{imm}}(I, a)$. Gracz P1 grając, najbardziej będze żałował nie wykonania akcji 'Kamień', a najmniej 'Papier'.




Rysunek 2.7: Przykład strategii.

Rys. 2.7 przedstawia wektory określające jakimi rozkładami akcji powinni się kierować gracze, aby osiągnąć najlepsze wyniki. Są to wektory wyliczone po pierwszej iteracji, w praktyce eksploracja drzewa i powyższe obliczenia są powtarzane wielokrotnie. Końcowym etapem algorytmu CFR jest policzenie średniej strategii [7].

### 2.5.3 Monte Carlo Conterfactual Regret Minimization

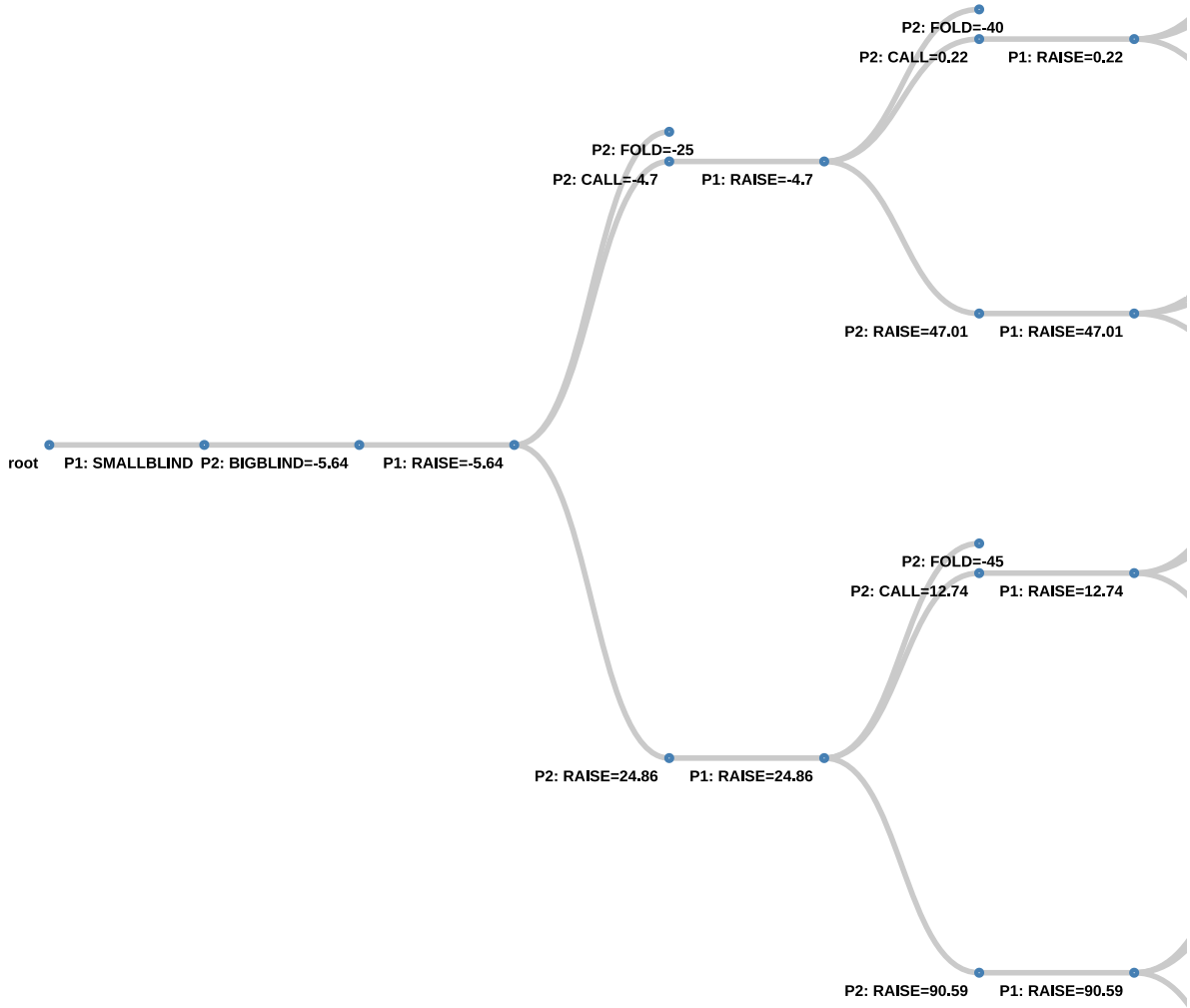
Algorytm CFR eksploruje całe drzewa decyzyjnego w jednej iteracji, co tworzy wymagania na dużą moc obliczeniową i długi czas uczenia. Dla małych gier takie rozwiązanie jest akceptowalne, ale w przypadku większych środowisk jest nieefektywny. Spowodowa-

ło to powstanie nowszej wersji algorytmu, MCCFR ('*Monte Carlo Conterfactual Regret Minimization*') , który w każdą iterację eksploruje tylko część drzewa [18].

Metodę można podzielić na dwie odmiany, Outcome-Sampling oraz External-Sampling [18].

W pracy zostanie przedstawiony sposób MCCFR ES ('*Monte Carlo Conterfactual Regret Minimization External Sampling*'), ponieważ taki został zaimplementowany w algorytmie Deep CFR. MCCFR ES przed eksploracją drzewa wybiera kolejno spośród graczy jednego uczestnika, którego oznacza się jako 'traverser'. Eksploruje on wszystkie odpowiedzi ze swoich akcji w danym węźle. W międzyczasie inni uczestnicy wykonują pojedynczy ruch na podstawie wybranej strategii [18].

Na rys. 2.8 przedstawiono przykład część drzewa HULH, gdzie gracz 'P2' został wybrany jako 'traverser'. Jak można zauważyć, tylko jego węzły rozgałęziają się na wszystkie możliwe ścieżki. Dodatkowo w drodze powrotnej obliczono $u_i(\sigma, I)$ dla wszystkich stanów używając wzoru 1.5.




Rysunek 2.8: Część drzewa decyzyjnego MCCFR ES z graczem P2 jako traverser.

Jest to przykład z jednej eksploracji gry, w praktyce powtarza się ten proces wielokrotnie, uzyskując różne wersje gry.

Mając takie drzewo, można przystąpić do liczenia $R^T_{i,imm}(I)$ oraz poprawiania wyników przez Regret Matching. MCCFR ES spełnia swoją funkcję, ale wymaga wielu iteracji, aby uzyskać dobre wyniki. Z tego powodu w dalszym rozdziale zostanie przedstawiona metoda Deep CFR, która przyspiesza proces uczenia przez użycie sieci neuronowych.

## 2.6 Deep CFR

Algorytm Deep CFR rozwija podstawową wersję metody CFR o sieci neuronowe. Taka modyfikacja była wymagana, aby utworzyć algorytm, który może rozwiązać nie tylko proste gry, ale też i duże jak HULH. Liczy on wektory w drzewie decyzyjnym przez algorytm MCCFR ES. Dodatkowo Deep CFR zbiega się do Równości Nasha szybciej niż popularny algorytm NFSP z 2016 roku [5].

Deep CFR wykorzystuje sieci neuronowe do przewidzenia wartości $D^T$(I, a) w podobnych obserwacjach, potem na podstawie predykcji liczy strategię ze wzoru nr 1.3. Następnie liczona jest zaktualizowana wersja wektora żalu, formuła nr 1.5. Obliczone wyniki są dodawane do buforów $B_1$, $B_2$, a strategia do zbioru $B_s$.

Po wielu iteracjach rozpoczyna się nauka sieci z zebranej bazy. Poniżej przedstawiono dokładny opis Deep CFR.

**Algorithm 1:** Deep CFR

**Wejście:** $B_p$, $B_s$, $\theta_s$, $\theta_p$, $P$, $N$, $K$
**Wyjście:** $\theta_s$

**for** $n$ ***in*** $N$ **do**
  $h$, $t \leftarrow$ Nowa gra
  **for** $p$ ***in*** $P$ **do**
    **for** $k$ ***in*** $K$ **do**
      MCCFRES($\theta_p$, $\theta_{p-1}$, $p$, $t$, $h$, $B_p$, $B_s$)
    $\theta_p \leftarrow$ TRAIN($B_p$, $\theta_p$, $t$) ▷ loss: $\frac{1}{N_{batch}} \sum (y_i - \widehat{y_i})^2 \cdot t_i$
$\theta_s \leftarrow$ TRAIN($B_s$, $\theta_s$, $t$) ▷ loss: $\frac{1}{N_{batch}} \sum (y_i - \widehat{y_i})^2 \cdot \mathrm{t}_i$
**return** $\theta_s$

Algorytm na wejściu dostaje argumenty przystosowane do gry 2-osobowej. Pierwszymi elementami są bufory graczy $B_p$ ($B_1$, $B_2$) oraz kontener na strategie $B_s$. Dodatkowo metoda potrzebuje listę uczestników P, z których będzie wybierać 'traversera'. Ostatnimi argumentami są iteracje gry N oraz liczba eksploracji drzewa K.

W metodzie należy ustawić trzy pętle wraz z nową rundą uzyskując początkową historię oraz krok gry t. Kolejnym etapem jest wykonanie funkcji *MCCFRES*. Przyjmuje ona na wejściu sieci neuronowe obu graczy, listę uczestników p, numer kroku t, historię rundy h

oraz bufory. Po k powtórzeniach następuje uczenie sieci neuronowej wybranego wcześniej gracza jako 'traversera'.

Model używa zmodyfikowanej wersji funckcji MSE do liczenia błędu predykcji. Każdy wynik $(y_i - \hat{y}_i)^2$ mnożony jest przez krok $t_i$ w, którym uzyskano $y_i$ [5]. Dodatkowo jak wynika z badań algorytm osiąga lepsze wyniki jeśli sieci neuronowe są trenowane od początku [5]. Dlatego należy przed funkcją *TRAIN* wyczyścić model.

Po wszystkich powtórzeniach i zebraniu całej bazy bufora $B_s$ rozpoczyna się trenowanie sieci neuronowej $\theta_s$ w ten sam sposób jak inne modele. Elementem wyjściowym Deep CFR jest $\theta_s$.

**Algorithm 2:** Implementacja MCCFRES kożystając z sieci neuronowych

**Function** `MCCFRES(`$\theta_p$, $\theta_{p-1}$, $p$, $t$, $h$, $B_p$, $B_s$`):`
- $t_r \leftarrow h$ ▷ sprawdzenie czyja jest aktualnie tura
- **if** *h jest stanem końcowym Z* **then**
  - **return** $u_p(h)$
- **else if** $p = t_r$ **then**
  - $\hat{D}(I) \leftarrow$ obliczenie wektora $\hat{D}(I)$ używając h
  - $\sigma(I) \leftarrow$ Obliczenie strategi $\sigma_{t_r}(I)$, kożystając z $\hat{D}(I)$ i wzoru 1.1
  - **for** *a* ***in*** *A(h)* **do**
    - $u_{t_r}$(h) $\leftarrow$ `MCCFRES(`$\theta_p$, $\theta_{p-1}$, $p$, $t+1$, $h+a$, $B_p$,$B_s$`)`
  - $u_{r_r}(\sigma, I) \leftarrow \sum(u_{t_r}(h) \cdot \sigma(I))$
  - $R^T_{t_r,imm}$(I) $\leftarrow$ ($u_{t_r}(h) - u_{t_r}(\sigma, I)$)
  - $B_{t_r} \leftarrow$ dodanie próbki do bufora [$R^T_{t_r,imm}$(I), h, t]
- **else**
  - $\hat{D}(I) \leftarrow$ obliczenie wektora $\hat{D}(I)$ używając $\theta_p$ i stanu h
  - $\sigma(I) \leftarrow$ Obliczenie strategi $\sigma_p(I)$, kożystając z $\hat{D}(I)$ i wzoru 1.1
  - $B_s \leftarrow$ dodanie próbki do bufora $[\sigma(I), h, t]$
  - a $\leftarrow \sigma(I)$
  - **return** `MCCFRES(`$\theta_p$, $\theta_{p-1}$, $p$, $t+1$, $h+a$, $B_p$, $B_s$`)`

Implementacja MCCFR ES przedstawiona powyżej przy zadanych argumentach rozpoczyna się od sprawdzenia, który gracz rozpoczyna daną turę. Na podstawie tej informacji będzie wykonywana dalsza część algorytmu.

Pierwszym krokiem jest sprawdzenie, czy stan gry jest końcem gry. W przypadku prawdziwego warunku zwracana jest wygrana lub przegraną wartość stawki. Jeśli powyższy etap jest fałszywy, algorytm sprawdza, czy gracz jest oznaczony jako 'traverser'. Wtedy program używając sieci neuronowej gracza, otrzymuje wektor $\hat{D}(I)$ przez wprowadzenie do modelu informacji o widocznych kartach oraz dotychczasowej historii gry. Korzystając ze wzoru 1.3, liczy strategie, odczytuje $u_{t_r}$ (h) wykonując rekurencję. Ostatnim krokiem jest wyliczenie wektora żalu i dodanie go do bufora.

Jeśli powyższy warunek był fałszywy, gracz liczy tak samo jak wcześniej $\hat{D}(I)$, nowy

rozkład akcji i dodaje ją do zbioru strategii. Następnie korzystając z tej sieci neuronowej i otrzymanej dystrybucji wykonuje nową akcję.

## 2.7 Podsumowanie

Środowiska z niepełnym zestawem informacji i brakiem deterministyczności są trudne do rozwiązania. Algorytmy tworzące modele dla takich gier są często skomplikowane i obciążające obliczeniowo. Przez takie cechy dopiero od niedawna zaczęły powstają algorytmy zdolne pokonać ludzi w dużych grach karcianych jak 'Cepheus' lub 'DeepStack'. Metody zdolne tworzyć takie AI dalej są rozwijane i aktualizowane z roku na rok. W taki sposób w 2014 roku powstał CFR i MCCFR, po para latch zastąpiono je przez CFR+, a następnie Deep CFR bedący tematyką pracy.

Rozdział dokładnie opisał działanie omawianego algorytmu przez przedstawienie terminów należących do działu matematyki 'Teoria Gier'. Między innymi pokazał, że opis środowiska przez drzewa decyzyjnych pozwala na wiele uproszczeń i możliwości śledzenia gry. Dodatkowo przedstawiono problem poszukiwania stanu Równości Nasha, którego znalezienie jest celem większości algorytmów sztucznej inteligencji gier karcianych.

Dobrzez zaimplementowany algorytm Deep CFR przy prawdidlowej parametryzacji i odpowiednio dużej ilości iteracji powinien zbiec się do pókktu bliskiego takiego stanu.

# Rozdział 3

# Implementacja algorytmu

Program Deep CFR został napisany w niniejszej pracy, korzystając z narzędzi pozwalających na zredukowanie nadmiarowości kodu oraz na prostą implementację. W tym rozdziale skupiono się na opisaniu wybranych technologii, parametrów oraz funkcji, które znalazły się w implementacji algorytm.

Bazą do napisanego kodu jest język programowania, Python 3.8. Zawiera on wiele technologii wspomagających uczenie maszynowe i rozległą społeczność wspierającą jego rozwój. Poniżej wy listowano i opisano główne narzędzia użyte w pracy.

**TensorFlow** Jest to wysokopoziomowe API dostępne dla takich języków jak Python, JavaScript, C++ lub Java [21]. Wykorzystuje je się głównie do zadań głębokiego uczenia maszynowego. Przez swoją prostotę, dostępność i dobrą dokumentację stał się jednym z najpopularniejszych narzędzi wykorzystywanych do tworzenia sztucznych inteligencji. Dodatkowo od niedawna biblioteki technologi Keras stały się częścią Tensorflow. Daje to możliwości znacznego zredukowania kodu przy prostych problemach, które często są rozwiązane przez fynkcje w tym module.

W przypadku niniejszej pracy głównie korzystano z funkcji zawartych w bibliotekach Keras. Wyjątkiem są nieliczne wiersze w kodzie gdzie np. było wymagane wykonanie obliczeń na tensorach.

**Numpy** Duża biblioteka do naukowych obliczeń na wielowymiarowych tablicach [20]. Jest nieodłącznym elementem przy pisaniu programów uczenia maszynowego, zwłaszcza jeśli korzysta się z bibliotek Tensorflow. Wynika to z faktu, że wiele funkcji tego API, jako argumenty przyjmuje typy danych powiązane z Numpy [21].

**Tqdm** Małe narzędzie w języku Python pozwalające na wyświetlenie postępu procesów w działającym programie. Przydatne narzędzie w celach testowych. W przypadku pracy zostały użyte do śledzenia iteracji drzew decyzyjnych algorytmu Deep CFR.

**TensorBoard** Moduł należący do API Tensorflow. Wizualizuje postępy uczenia sieci neuronownych oraz ich jakość przez przedstawienie odpowiednich wykresów.

**PyPokerEngine** Biblioteka wspomagająca symulację gry Poker Texas Hold'em. Użyto jej jako podstawę do napisania środowiska HULH do interakcji z metodą Deep CFR.

## 3.1 Implementacja sieci neuronowych

Algorytm Deep CFR do działania wymaga dwóch sieci neuronowych, jedna ma rozpoznawać strategie $\sigma_p^t$, a druga przypisana do określonego gracza przewiduje opłacalność akcji $D_p^t$. Dodatkowo każdy z tych elementów jest trenowany na podstawie cyklicznie aktualizowanych buforów $B_s$ i $B_p$. W tym rozdziale zostanie przedstawiona dokładna implementacja modeli, proces ich uczenia, struktura zbiorów danych oraz budowa środowiska, z którym jest wykonywana interakcja. W tab. 3.1 i 3.2 zamieszczono podstawowe informacje o parametrach sieci. Dalsza część rozdziału dokładnie opisuje dodatkowe elementy, ważne podczas uczenia modelu.

Tabela 3.1: Podstawowe parametry sieci neuronowej $\theta_s$.

| parametr | użyte wartości |
|---|---|
| rozmiar danych wejściowych | (3, 52) |
| rozmiar danych wyjściowych | (None, 3) |
| prędkość uczenia | 0.0001 |
| końcowa funkcja aktywacyjna | *softmax* |
| rozmiar bufora | 200 000 |
| maksymalna liczba iteracji | 16 000 |
| rozmiar *minibatch* | 500 |
| parametr *patience* | 10 |

Tabela 3.2: Podstawowe parametry sieci neuronowej $\theta_p$.

| parametr | użyte wartości |
|---|---|
| rozmiar danych wejściowych | (3, 52) |
| rozmiar danych wyjściowych | (None, 3) |
| prędkość uczenia | 0.0001 |
| końcowa funkcja aktywacyjna | *linear* |
| rozmiar bufora | 100 000 |
| maksymalna liczba iteracji | 16 000 |
| rozmiar *minibatch* | 500 |
| parametr *patience* | 10 |

### 3.1.1 Architektura modelu

W algorytmie zaimplementowano trzy sieci neuronowe $\theta_1$, $\theta_2$, $\theta_s$ o nieskomplikowanej architekturze. Z tego powodu wykorzystano bibliotekę *Keras*, która do takich przypadków jest dobrym rozwiązaniem. Dodatkowo założono, że wszystkie modele będą miały podobną budowę poza ostatnią warstwą z innymi funkcjami aktywacyjnymi. Wynika to z faktu, że algorytm Deep CFR korzysta z sieci, które dostają dane wejściowe o tej samej strukturze, ale zwracają $D_p^t$ lub $\sigma_p^t$.

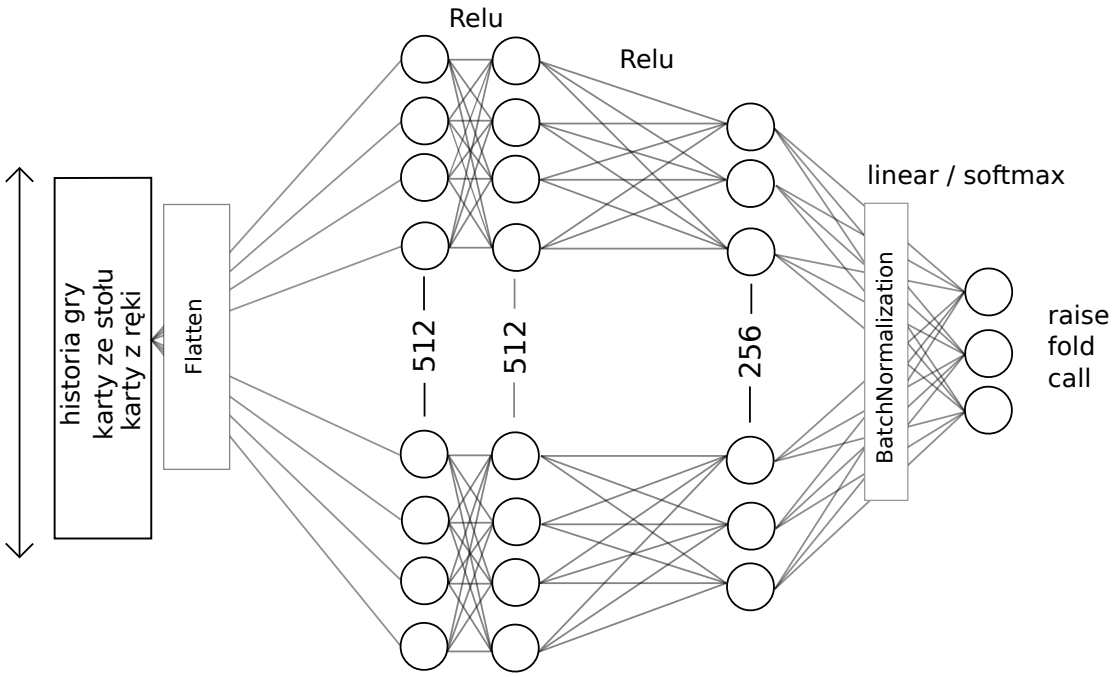
Architektura sieci neuronowych składa się z 7 elementów, wejścia, wyjścia, bloku o nazwie *Flatten* [21], trzech warstw ukrytych zakończonych normalizacją i wyjściem w postaci wektora o trzech polach.

Początkowo sieci dostają tablicę o wymiarach (3, 52), czyli kolumnę kart ze stołu, z ręki oraz historię dotychczasowej gry. Model w dalszych obliczeniach musi przekształcić takie dane do formy jedno-wymiarowej (None, 156), co robi warstwa *Flatten*. Kolejne dwa elementy składają się z 512 wag, trzecia warstwa ukryta posiada ich 256. Na trzech wymienionych elementach ustawiono funkcję *Relu*. Całość kończy się wyjściem wektora reprezentującego możliwe akcje gry HULH. Dodatkowo dane są poddawane normalizacji przez warstwę o nazwie *BatchNormalization* [21].

Wyjście modeli, aby zwracało prawidłowe liczby, używa innej funkcji aktywacyjnej niż poprzednie elementy. W przypadku predykcji strategii $\sigma_p^t$ wybrano *softmax*. Sieci neuronowe $\theta_1$, $\theta_2$ używają funkcji *linear*. Dokładna architektura sieci jest zaprezentowana na rys. 3.1.

W trakcie trenowania sieci korzystają z funkcji optymalizującej Adam. Prędkość uczenia jest równa 0,0001, co spowoduje powolne uczenie, ale zminimalizuje szanse na ominięcie minimum globalnego.

Jak wynika z dokumentu prezentującego algorytm Deep CFR, uzyskuje on lepsze wyniki, jeśli sieci neuronowe są uczone za każdym razem od początku przy losowo ustawionych zerach w wagach [5]. W tym celu ustawiono w każdej warstwie funkcję, która tworzy losowo wartości w przedziałach od -0,005 do 0,005.




Rysunek 3.1: Architektura sieci neuronowych wykorzystana w programie.

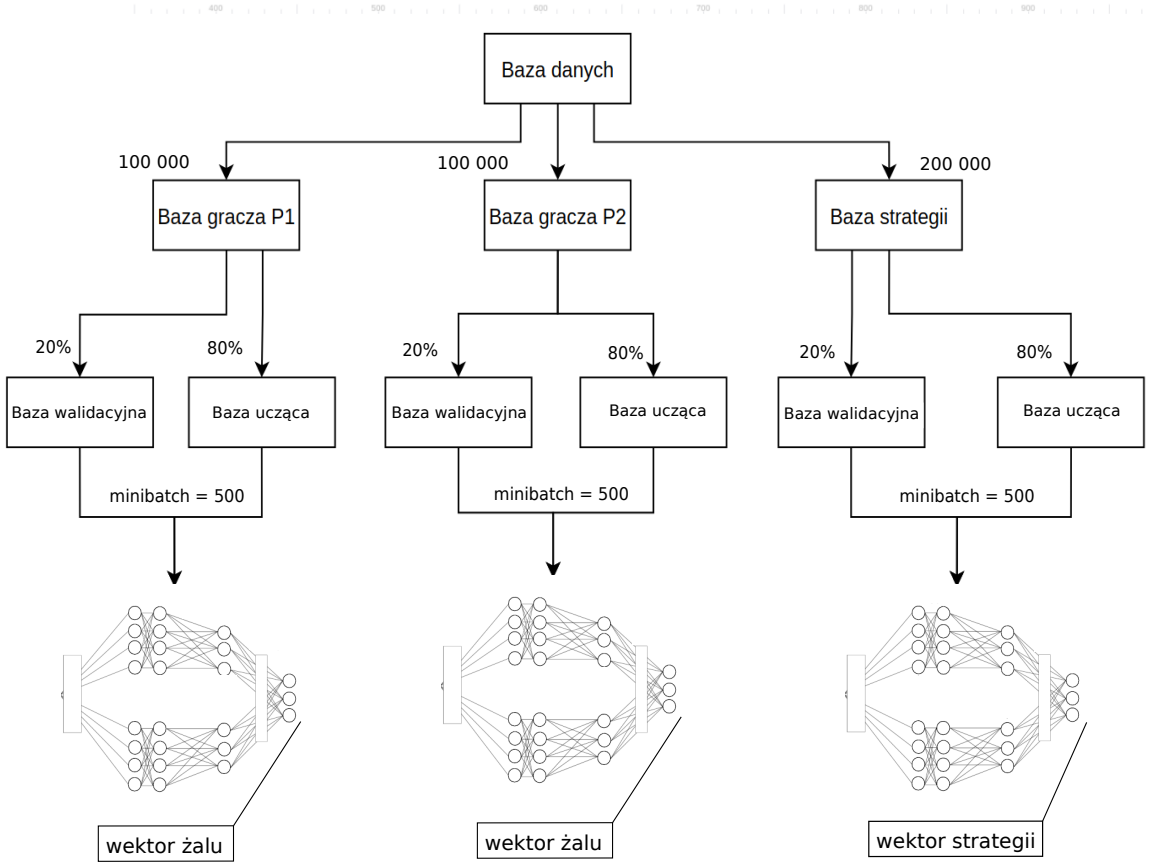
Ostatnim elementem sieci jest funkcja licząca błąd predykcji w trakcie uczenia. Jak wynika z opisu algorytmu Deep CFR, wymaga on zmodyfikowanej wersji MSE (*Mean Square Error*). Zostało to wykonane przy pomocy funkcji matematycznych na tensorach, jakie udostępnia Tensorflow [21].

### 3.1.2 Budowa zbiorów danych

Jak wynika z algorytmu Deep CFR, w programie muszą być zawarte trzy bufory. Maksymalna pojemność kontenerów $B_1$ i $B_2$ jest równa 100 000 próbek. Bufor $B_s$ może posiadać ich 200 000. Każdy z dodatnych elementów do bufora składa się z trzech połączonych wektorów o rozmiarach 52.

Zaimplementowano klasę *Memory* zarządzającą tymi buforami, które zachowują się jak kolejki, w przypadku przepełnienia jest usuwany najstarszy wpis.

Mając uzupełnione dane w tablicach, program rozpoczyna przygotowanie zbiorów danych do uczenia wybranych sieci neuronowych. Każdy z buforów zostaje losowo przetasowany i podzielony na dwa podzbiory. Pierwszy z nich to zbiór uczący. Wykorzystuje się go do poprawiania wag modelu sekwencyjnie. Drugim zbiorem są dane walidacyjne. Dokonanie takiego podziału było wymagane, aby przeciwdziałać stanowi przetrenowania modelu. Zbiór walidacyjny jest nadzorowany i na jego podstawie można określić moment od, którego model przestaje dobrze działać. Schemat tego podziału przedstawiono na rys. 3.2.



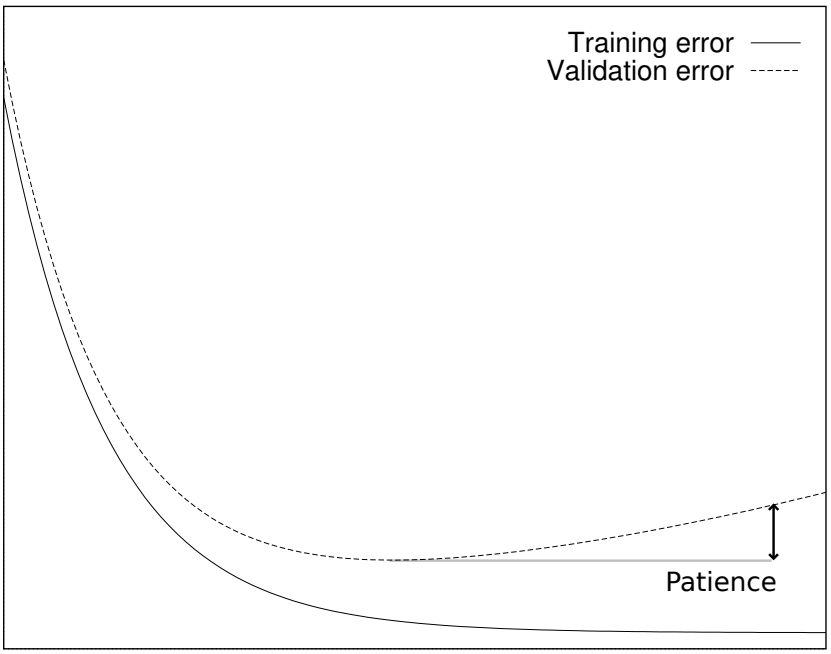

Rysunek 3.2: Podział danych.

W trakcie nauki sieć neuronowa aktualizuje swoje wagi w każdym kroku przez użycie elementu o nazwie *minibatch* będący parametrem określającym mały podzbiór bazy użyty do trenowania sieci w pojedynczym kroku. Jego liczebność jest równa 500 próbek.

### 3.1.3 Proces uczenia

Algorytm Deep CFR do eksploracji wykorzystuje metodę MCCFR ES, która eksploruje w jednej iteracji 250 razy drzewo dezyzyjne gromadząc przy tym próbki w buforach. Po zakończeniu wszystkich powtórzeń zachodzi etap uczenia sieci neuronowych $\theta_1$, $\theta_2$ wykonując maksymalnie 16 000 aktualizacji wag. Taki schemat działania jest wykonywany dla obu graczy i powtarza się 50 razy. Końcowym zadaniem jest wytrenowanie sieci neuronowej $\theta_s$. Wszystkie te elementy zostały połączone przez klasę *Player* posiadającą obiekty *Memory* oraz *Network*, rys. 3.4.

Dodatkowo w celu poprawienia wyników uczenia użyto funkcji EarlyStopping [21]. Zatrzymuje ona iteracje modelu w przypadku kiedy błąd predykcji na zbiorze walidacyjnym wzrośnie odpowiednio wysoko. Jest to sprawdzane na podstawie argumentu *patience*, który określa próg, po którym nauka się kończy. Taka procedura została zastosowana, aby zminimalizować szanse na przetrenowanie modeli, a wraz z tym ich gorszą jakość [19]. Rys. 3.3 prezentuje przykładowy punkt zakończenia nauki modelu.




Rysunek 3.3: Przykładowy punkt zatrzymania się uczenia po zastosowaniu *EarlyStopping* [19].

## 3.2 Implementacja środowiska

Do symulacji gry HULH z którym model będzie się komunikował, użyto blblioteki PyPokerEngine. Klasa *HULH_Emulator* zaimplementowana w programie pozwala na utworzenie obiektu zarządzającego grą HULH. Przy tworzeniu obiektu zostają zdefiniowane nazwy graczy P1 i P2, które w dalszej części będą używane do śledzenia histori gry. W tabeli nr 3.3 wylistowano parametry ustawione w programie, związane z zasadami grą.

Tabela 3.3: Parametry gry HULH.

| parametr | wartość |
|---|---|
| ante | 0 |
| mała w ciemno | 5 |
| duża w ciemno | 10 |
| liczba rund po których resetuje się środowisko | 1 |
| udział każdego z graczy (stock) | 80 |
| liczba graczy | 2 |

Zaimplementowane środowisko opiera się na PyPokerEngine, co oznacza, że każda akcja zwraca dwa elementy, obiekt określający stan gry oraz historię gry w formie json. Obie wartości są używane do śledzenia stanu środowiska w drzewie decyzyjnym. Informacje jak

karty na stole, historia gry lub wygrana gracza jest zarządzane przez użytą bibliotekę. Napisane funkcje w programie pełnia tylko zadanie interfejsu dla drzewa.

## 3.3 Implementacja Deep CFR

Klasa *DCFR* zawiera implementację algorytmu Deep CFR. Przy tworzeniu obiektu powstają w konstruktorze trzy elementy gracz, oponent oraz strategia $\sigma$ z klasy *Brain*. Do tego ustawiane jest środowisko *HULH_Emulator* przez referencje.

Cały proces powstawania modelu rozpoczyna się od funkcji *iterate*, która wykonuje trzy pętle *for*, dla iteracji algorytmu i powtórzeń eksploracji drzewa dla każdego z graczy. W pierwszej z nich środowisko tworzy nową grę, w ostatniej pętli jest wykonywana funkcja *__traverse*, działająca według metody MCCFR ES. Dostaje ona argumenty *state* czyli obiekt określający stan gry, *events* - dotychczasową historię oraz *timestep*. Ostatni argument *verbose* jest opcialny i służy tylko do wizualizacji powstałego drzewa decyzyjnego. Funckja działa po przez rekurencję. Po zakończeniu działania MCCFR ES, rozpoczyna się uczenie sieci neuronowej $\theta_p$. I powtażanie powyrzszego procesu. Ostatnim etapem jest trenowanie sieci $\theta_s$ oraz zapisanie jej do pliku. Tab. 3.4 prezentuje podstawowe parametry zaimplementowanego algorytmu.

Tabela 3.4: Parametry algorytmu Deep CFR.

| parametr | wartość |
|---|---|
| liczba powtórzeń eksploracj drzewa | 250 |
| liczba iteracji algorytmu | 50 |
| co ile iteracji wytrenować i zapisać model | 10 |

## 3.4 Podsumowanie


Rysunek 3.4: Diagram UML projektu.

# Rozdział 4

# Wyniki

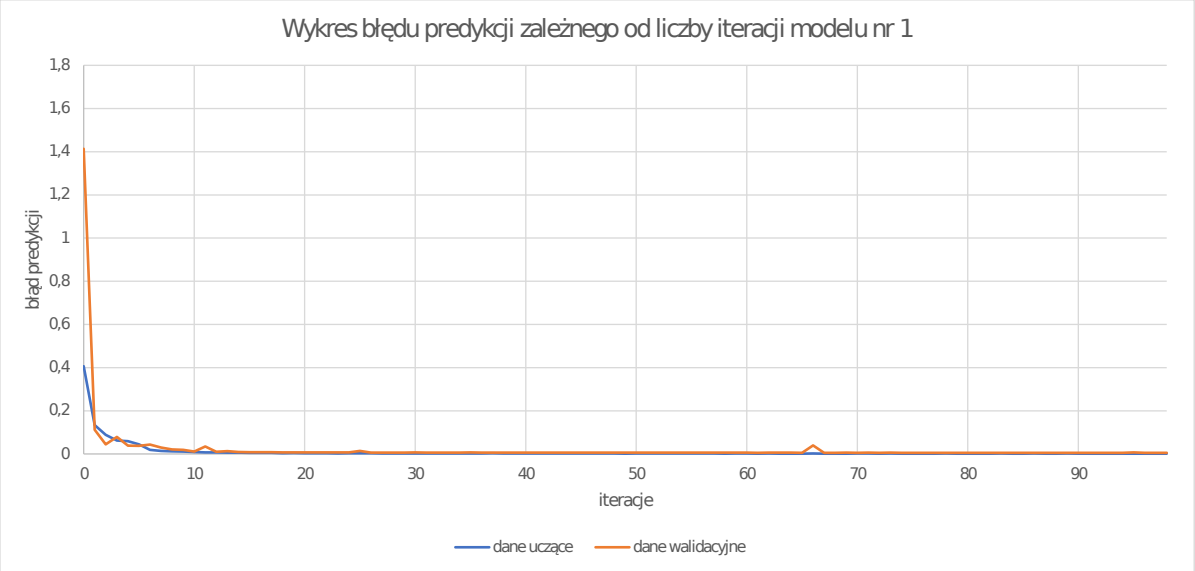
Rozdział przedstawia różnice między utworzonymi modelami z algorytmu Deep CFR. Zostały tutaj zaprezentowane wykresy jakości uczenia oraz wyniki rozegranych gier między nimi wraz z zwycięscą.

## 4.1 Proces uczenia modelów rozpoznawania

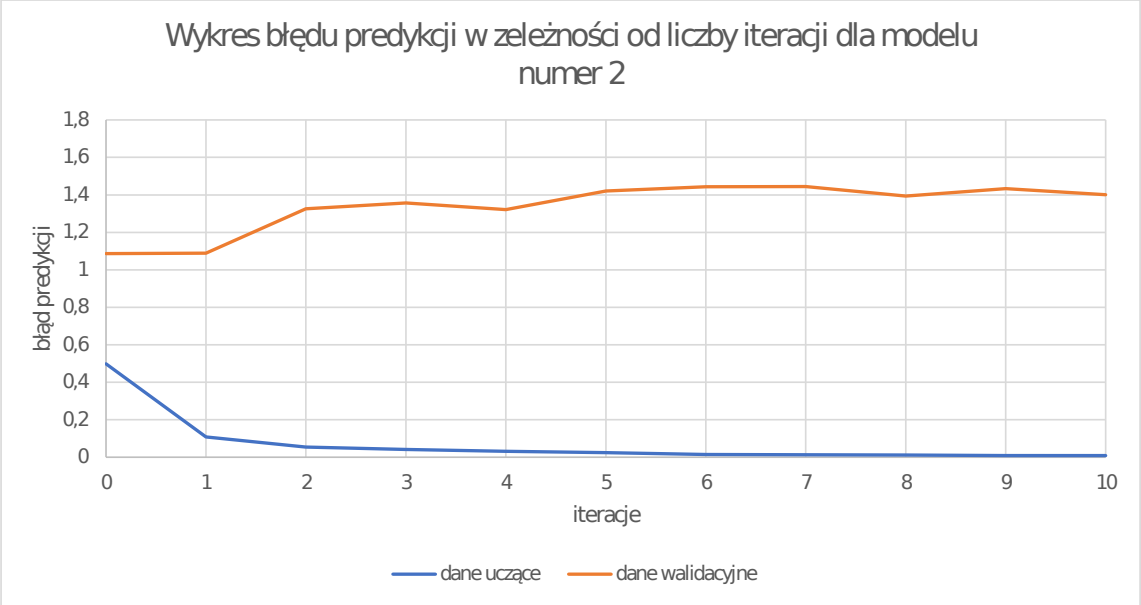
Algorytm Deep CFR pracując przez trzy dni zdołał utworzyć pięć modeli, gdzie każda nauka była śledzona przez bibliotekę *TensorBoard*. Uzyskane dane następnei zostały zaimportowane do Excela w celu utworzenia czytelnych wykresów ilustrujących zalezność błędu predykcji od liczby iteracji.

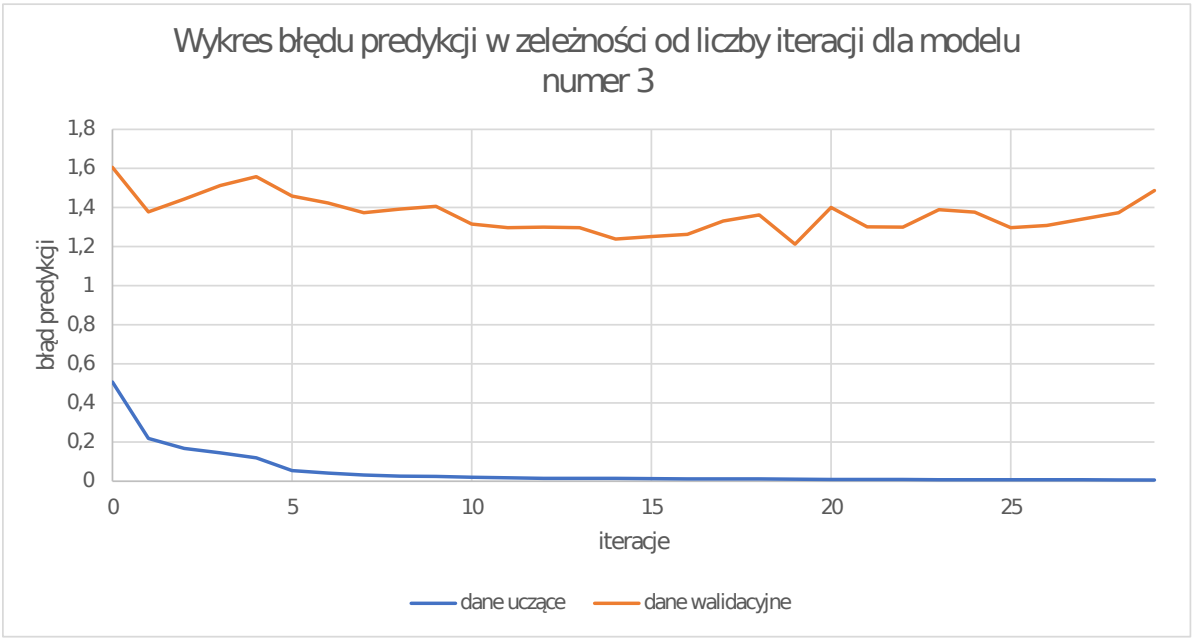
Pierwszy utworzony model powstał po 10 powtórzeniach algorytmu. Jak można zobaczyć na rys. 4.1 wartość błędu zbiegła się bardzo szybko do wartości bliskiej 0 dla zbioru uczącego i walidacyjnego. Wynika to z małej liczby zebranych próbek w buforze $B_s$ o małej różnorodności. Po mimo tak dobrych efektów można zauwazyć na rys. 4.2, że model nr 1 nie wygrał tylko z modelem nr 3 z nie wielką przewagą.

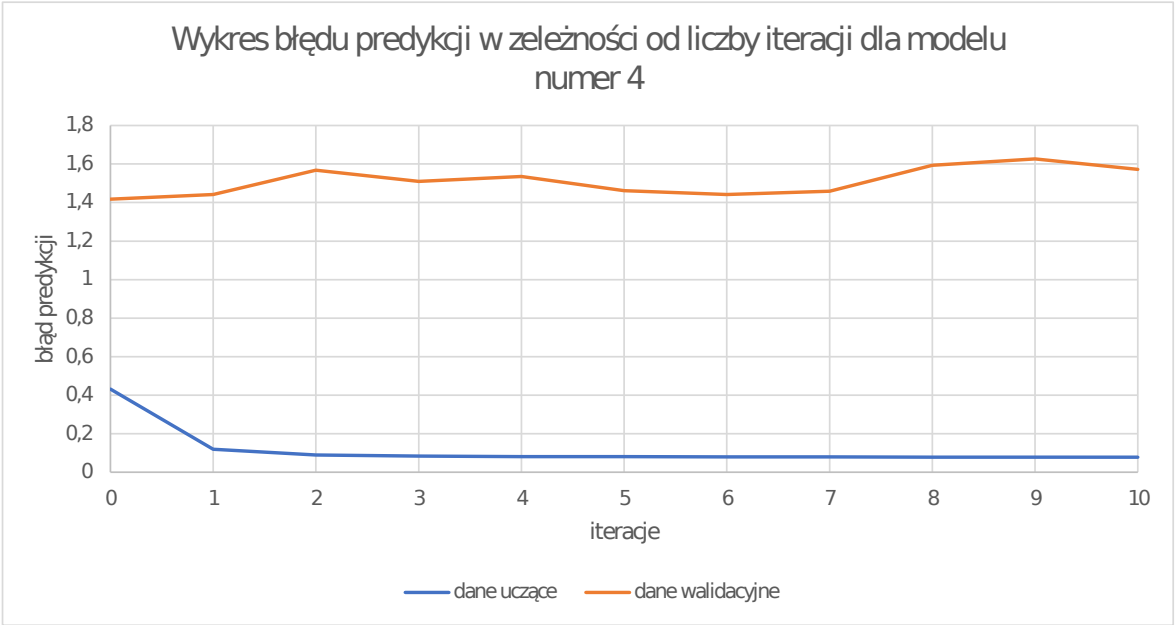
## 4.2 Wyniki rozgrywek modeli

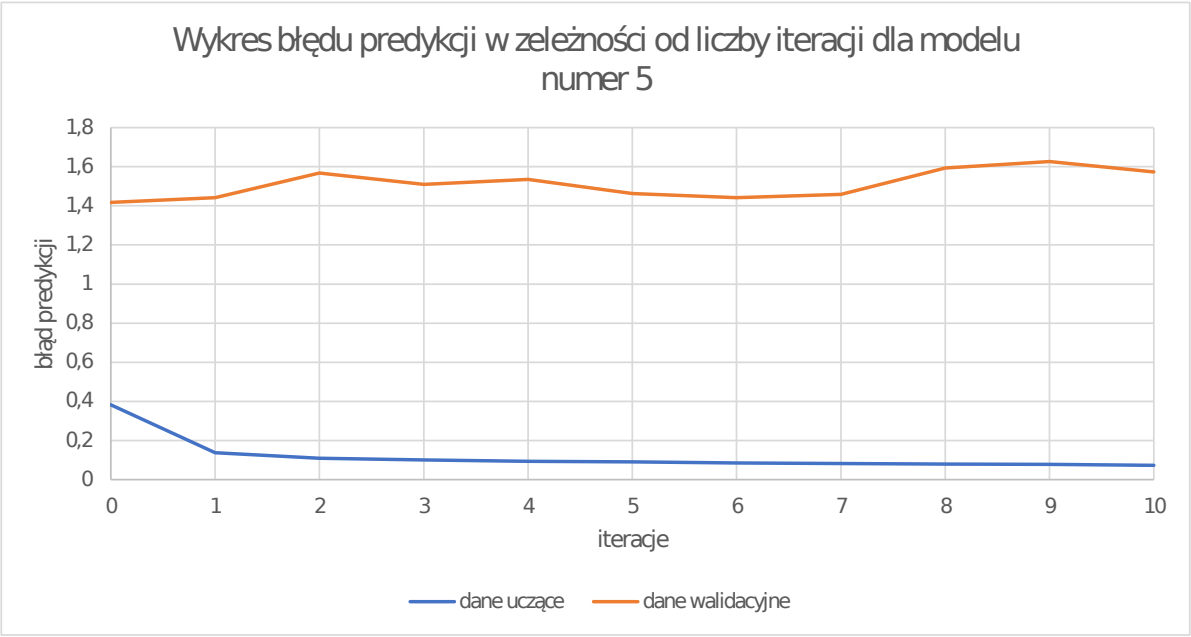


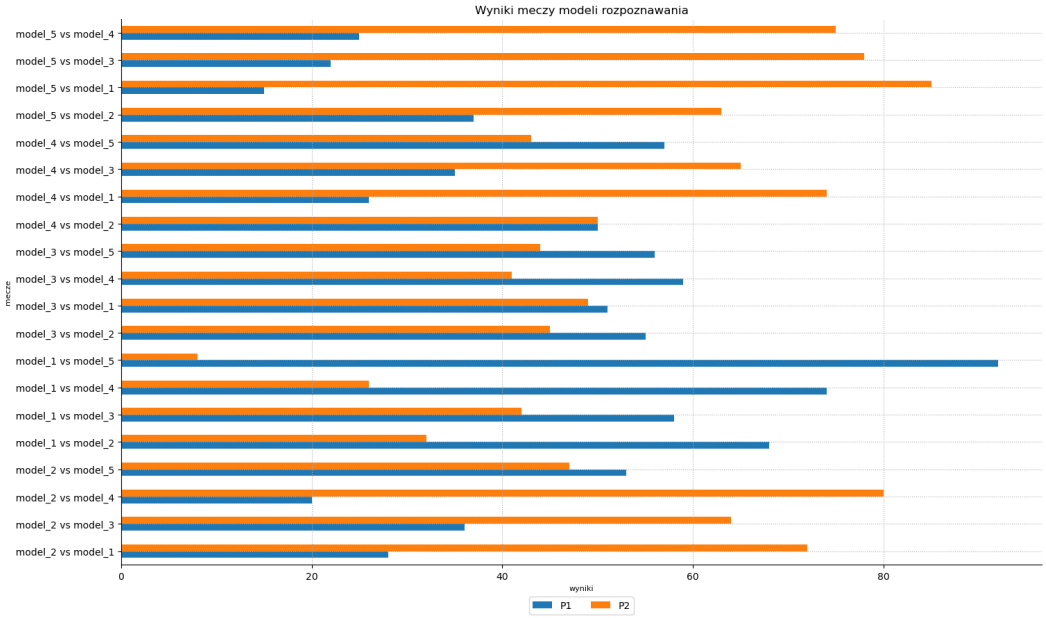

Rysunek 4.1: Wyniki uczenia modelu nr 1.




Wykres błędu predykcji w zależności od liczby iteracji dla modelu numer 3
błąd predykcji
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0
5
10
15
20
25
iteracje
dane uczące
dane walidacyjne


Wykres błędu predykcji w zależności od liczby iteracji dla modelu numer 4
błąd predykcji
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0
1
2
3
4
5
6
7
8
9
10
iteracje
dane uczące
dane walidacyjne




Rysunek 4.2: Diagram UML projektu.

# Bibliografia


[1] Haenlein, Michael, and Andreas Kaplan. "A brief history of artificialintelligence: On the past, present, and future of artificial intelligence." California management review 61.4 (2019): 5-14.

[2] Gibney, Elizabeth. "Google AI algorithm masters ancient game of Go." Nature News 529.7587 (2016): 445.

[3] Berner, Christopher, et al. "Dota 2 with large scale deep reinforcement learning." arXiv preprint arXiv:1912.06680 (2019).

[4] Brown, Noam, et al. "Combining deep reinforcement learning and search for imperfect-information games." arXiv preprint arXiv:2007.13544 (2020).

[5] Brown, Noam, et al. "Deep counterfactual regret minimization." International conference on machine learning. PMLR, 2019.

[6] Heinrich, Johannes, Marc Lanctot, and David Silver. "Fictitious self-play in extensive-form games." International conference on machine learning. PMLR, 2015.

[7] Zinkevich, Martin, et al. "Regret minimization in games with incomplete information." Advances in neural information processing systems 20 (2007): 1729-1736.

[8] Heinrich, Johannes, and David Silver. "Deep reinforcement learning from self-play in imperfect-information games." arXiv preprint arXiv:1603.01121 (2016).

[9] Teófilo, Luís Filipe Guimarães. "Building a poker playing agent based on game logs using supervised learning." (2010).

[10] Bouju, Gaëlle, et al. "Texas hold'em poker: a qualitative analysis of gamblers' perceptions." Journal of Gambling Issues 28 (2013): 1-28.

[11] Félix, Dinis Alexandre Marialva. "Artificial intelligence techniques in games with incomplete information: opponent modelling in Texas Hold'em." (2008).

[12] Xiang, Xuanchen, and Simon Foo. "Recent Advances in Deep Reinforcement Learning Applications for Solving Partially Observable Markov Decision Processes (POMDP) Problems: Part 1—Fundamentals and Applications in Games, Robotics and Natural Language Processing." Machine Learning and Knowledge Extraction 3.3 (2021): 554-581.

[13] Nogal-Meger, P. (2012). Dylemat więźnia jako przykład wykorzystania teorii gier. Prace i Materiały Wydziału Zarządzania Uniwersytetu Gdańskiego, 10(4, cz. 2), 87–95.

[14] Myerson, Roger B. Game theory. Harvard university press, 2013.

[15] Bowling, Michael, et al. "Heads-up limit hold'em poker is solved." Communications of the ACM 60.11 (2017): 81-88.

[16] Moravčík, Matej, et al. "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker." Science 356.6337 (2017): 508-513.

[17] Davis, Trevor, Neil Burch, and Michael Bowling. "Using response functions to measure strategy strength." Twenty-Eighth AAAI Conference on Artificial Intelligence. 2014.

[18] Lanctot, Marc, et al. "Monte Carlo sampling for regret minimization in extensive games." Advances in neural information processing systems 22 (2009): 1078-1086.

[19] Prechelt, Lutz. "Early stopping-but when?." Neural Networks: Tricks of the trade. Springer, Berlin, Heidelberg, 1998. 55-69.

[20] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020)

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.