

Uniwersytet Warszawski  
Wydział Nauk Ekonomicznych

**Marcin Basiuk**

Nr albumu: mbp-18217

**Przegląd wybranych algorytmów  
uczenia maszynowego i sztucznej  
inteligencji na przykładzie gry w kółko  
i krzyżyk**

**Praca dyplomowa**

**Data Science w zastosowaniach biznesowych. Warsztaty z wykorzystaniem  
programu R.**

Praca wykonana pod kierunkiem  
**dr. Piotr Wójcik**  
Zakład Finansów Ilościowych

Czerwiec 2018

## **Oświadczenie kierującego pracą**

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data

Podpis kierującego pracą

## **Oświadczenie autora (autorów) pracy**

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data

Podpis autora pracy

## Streszczenie

W pracy zaimplementowano trzy różne algorytmy uczące komputer gry w kółko i krzyżyk. Pierwszym algorytmem jest Minimax połączony z przycinaniem  $\alpha - \beta$ . Następny algorytm, to znany pod angielską nazwą *Q-learning* w połączeniu z klasyczną tablicą. Trzecie podejście wykorzystuje sieć neuronową. Praca zawiera opis, porównanie działania algorytmów oraz opis aplikacji do gry z komputerem wykorzystującym wybrane algorytmy.

## Słowa kluczowe

minimax, teoria gier, uczenie maszynowe, sieci neuronowe, reinforcement learning, procesy markowa



# Spis treści

<b>Wprowadzenie</b>	5
<b>1. Minimax</b>	7
1.1. Opis algorytmu	7
1.2. Przycinanie $\alpha - \beta$	10
1.3. Funkcja wartości	12
1.4. Implementacja	12
<b>2. Q-learning</b>	15
2.1. Procesy decyzyjne Markowa	15
2.2. Opis algorytmu	18
2.3. Implementacja	19
<b>3. Sieć neuronowa grająca w kółko i krzyżyk</b>	25
3.1. Sieci neuronowe	25
3.2. Podejście I - nieudane	28
3.3. Podejście II - bardziej udane	31
<b>4. Podsumowanie</b>	35
<b>A. Dodatek - opis kodu</b>	37
A.1. Główny program	37
A.2. Klasy graczy	37
A.3. Grafika	38
A.4. Tensorflow	38
<b>B. Dodatek - opis aplikacji</b>	39
<b>Bibliografia</b>	41



# Wprowadzenie

W trakcie zajęć na studiach podyplomowych wiele uwagi poświęcone było budowaniu modeli predykcyjnych. Pracując z programem R i RStudio tworzone modele i szacowano parametry tych modeli, tak żeby „na końcu” mieć narzędzie, albo posługując się programistyczną nomenklaturą funkcję, która przyjmuje dane wejściowe i zwraca predykcję na podstawie tych danych. Wykorzystywano w tym celu szereg metod, zaczynając od regresji liniowej lub regresji logistycznej jeżeli zmienna objaśniana miała charakter nominalny, drzewa decyzyjne, algorytmy k - najbliższych sąsiadów, analizę głównych składowych, czy w końcu sieci neuronowe.

Doświadczenie z zajęć jednoznacznie wskazywało na dwa kluczowe aspekty, które decydowały o skuteczności modeli predykcyjnych. Po pierwsze jakość danych. Zaszumienie danych, bądź niepoprawne skalowanie prowadziło najczęściej do wyników poniżej oczekiwań. Druga sprawa dotyczyła tego na ile zmienne objaśniające rzeczywiście determinują zmienną (lub zmienne) objaśniane. Intuicyjnie, mało kto uwierzy, że dane ankietowe dotyczące średniej ilości godzin spędzonych w pracy połączonych z danymi demograficznymi takimi jak wiek, poziom wykształcenia, miejsce zamieszkania itd. będzie determinować poziom przychodu. Sukces finansowy najprawdopodobniej zależy również od szeregu cech osobowości takich jak odwaga do podążania za swoimi celami, umiejętność budowania swojej pozycji w systemach społecznych itp. Modele bazujące na danych ankietowych i demograficznych - takie jak budowano na zajęciach - miały skuteczność mierzoną średnim odchyleniem kwadratowym rzędu 70% – 85%. Trudno jednoznacznie powiedzieć na ile satysfakcjonujące są takie wyniki. Można uznać, że model „działa”, można też w ten sposób porównywać modele, ale trudno powiedzieć na ile „uchwycono”, prawa natury rządzące poziomem przychodu.

Podczas przymiarek do napisania tej pracy celem było skoncentrowanie się bardziej na algorytmach uczących komputery, aniżeli na zbieraniu, czyszczeniu i doborze danych, czyli na tzw. „feature engineering”’. Należy podkreślić, że w żadnym wypadku nie pomniejsza się wagi ani znaczenia tej sztuki. Po prostu sam proces tworzenia algorytmów uczących się wydawał się ciekawszy od zbierania i obróbki odpowiednich danych, żeby to uczenie było skuteczne. W poszukiwaniu problemów, w których dane byłyby niejako naturalnie dostępne uwaga autora skupiała się na grach. Jak nauczyć komputer grać w prostą grę? W tym wypadku dane są zbierane poprzez grę komputera z człowiekiem, z graczem losowym lub z samym sobą (komputer vs. komputer) i oczywiście nie ma mowy o żadnym zaszumieniu. Szachy i warcaby wydawały się zbyt ambitnym wyzwaniem. W obszarze rozważań znalazły się gry karciane, ale lepiej skupiać się na grze w pełni deterministycznej, w której losowość nie miałaby wpływu na wynik. Dzięki temu wiadomo na ile komputer dobrze nauczył się grać, bez względu na to jak dobra trafiła mu się karta. Ostatecznie wybór padł na kółko i krzyżyk - prosta i w pełni deterministyczna gra. Idealne pole do eksperymentowania z algorytmami uczącymi się.

W pierwszym rozdziale opisano zaimplementowany algorytm minimax - jest to klasyczne podejście stosowane do uczenia komputerów grania w gry turowe, gdzie jest dwóch graczy i jeżeli jeden wygrywa, to drugi przegrywa; ewentualnie może być remis. Można polemizować na ile minimax wpisuje się w uczenie maszynowe. Jednak skoro uczy się takiego podejścia na przedmiocie Artificial Intelligence MIT [4], to wydaje się, żeby pasuje przedmiotowo do niniejszej pracy.

W kolejnych dwóch rozdziałach zaprezentowano algorytmy typowe dla „reinforcement learning” (RL). Kiedy komputer uczy się grać w kółko i krzyżyk, to po wykonaniu ruchu, o ile ruch nie kończy gry, nie ma natychmiastowej informacji zwrotnej, czy to był dobry ruch, czy też zły ruch. W tym sensie, uczenia komputera gry w kółko i krzyżyk (oraz w inne gry) nie wpisuje się w schemat uczenia z nadzorem. Nie wpisuje się również w uczenie bez nadzoru, bo w przypadku gry, możemy na końcu powiedzieć kto wygrał, a kto przegrał. Z pomocą przychodzi RL, o którym Richard S. Sutton i Andrew Barto w swojej książce [3] piszą, że jest innym paradygmatem uczenia maszynowego, wychodzącym poza uczenie z nadzorem i bez nadzoru. W rozdziale drugim komputer ucząc się grać w kółko i krzyżyk będzie szacował prawdopodobieństwo wygrania z danego stanu gry i będzie te wartości zapisywał te dane w pliku tekstowym, do którego również odwołuje się w kolejnych grach. Takie podejście staje się niemożliwe w przypadku gier z bardzo dużą przestrzenią stanów (np. szachy, czy GO). Plik byłby zbyt duży i przeszukiwanie pliku zajmowałoby za dużo czasu. W takim wypadku można zaimplementować sieć neuronową, które będzie szacować prawdopodobieństwa wygrania. To ostatnie podejście opisane zostało w rozdziale trzecim.

Ponieważ uwaga skupia się na algorytmach i ich implementacji, postanowiono wykorzystać uniwersalny obiektowy język programowania. Decyzja padła na Python 2.7, który w odczuciu autora łatwiej umożliwia tworzenie i obsługę klas obiektów od R. Do pracy dołączono również kody źródłowe napisane w języku Python 2.7.

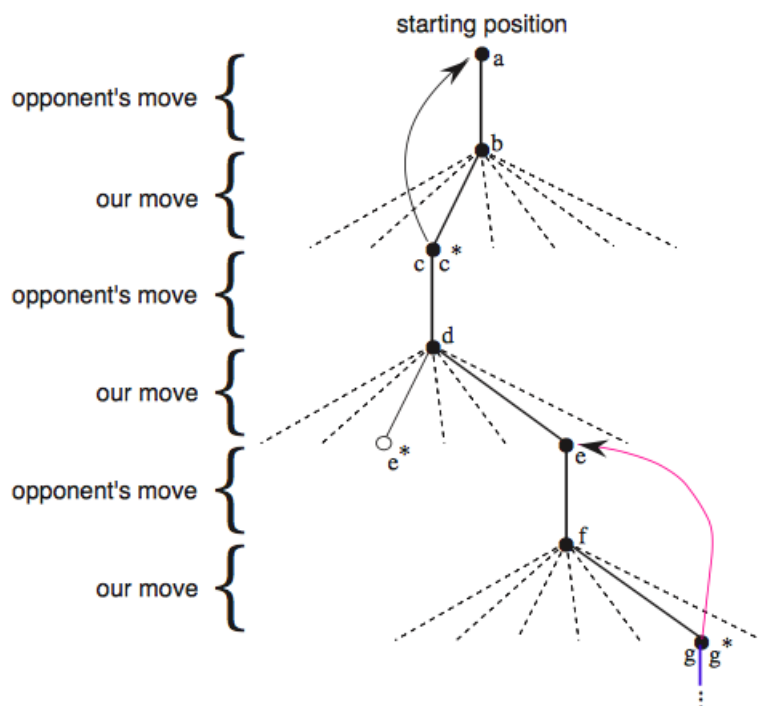


# Rozdział 1

## Minimax

### 1.1. Opis algorytmu

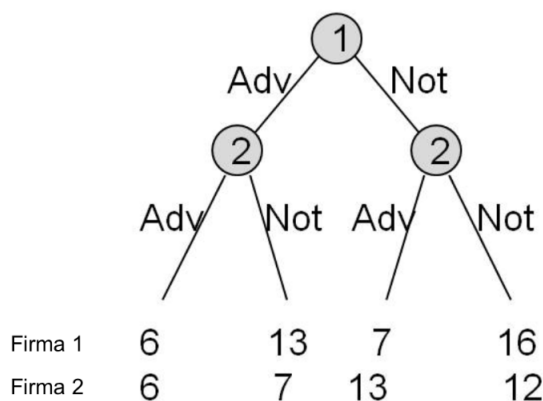
W takich grach jak kółko i krzyżyk, szachy lub warcaby można mówić o przestrzeni stanów gry. Każdy stan gry jest reprezentowany przez położenie pionków albo rozmieszczenie kółek i krzyżyków na kwadratowej planszy z 9 polami. Za każdym ruchem, jednego z dwóch graczy, gra przechodzi do innego stanu. Wszystkie możliwe gry są reprezentowane przez drzewo (drzewo w sensie teorii grafów), którego korzeniem jest początkowy stan gry, a kolejne pokolenia, reprezentują na przemian, możliwe ruchy graczy. Każda ścieżka od korzenia do wybranego liścia odpowiada jednej w pełni rozegranej grze. I na odwrót, każda możliwa realizacja gry ma odpowiadającą jej ścieżkę na drzewie gry. Koncepcja ta jest zilustrowana na poniższym rysunku 1.1 zaczerpniętym z książki [3]



Rysunek 1.1: Drzewo gry

Przypuśćmy dalej, że z każdym stanem końcowym gry (liściem) można powiązać liczbę wygranych punktów. W przypadku gier takich jak kółko i krzyżyk, może to być jedna z trzech wartości  $-1, 0, 1$  odpowiadająca przegranej, remisowi i wygranej odpowiednio. Można jednak wyjść poza ten schemat i rozważać dla stanów końcowych wiele innych wartości. Ważne jest to, że celem gracza jest maksymalizacja zdobytych punktów. Warto jeszcze zaznaczyć, że punkty przypisane do liści są różne dla rywalizujących ze sobą graczy. W rozpatrywanych przypadku gry w kółko i krzyżyk, suma punktów zdobytych przez jednego rywala, będzie równa sumie punktów zdobytych przez drugiego rywala, ale z przeciwnym znakiem. W ogólnym przypadku jednak tak nie musi być.

Można by naiwnie przypuszczać, że najlepszą strategią będzie wybieranie takiego ruchu, który „otworzy” graczowi ścieżkę do liści z największą możliwą ilością punktów. Na poniższym przykładzie wziętym ze strony [6] widać, że taka strategia nie jest optymalna. Wyobraźmy sobie dwie konkurujące ze sobą firmy, które rozważają poniesienie wydatków na promocję oferowanego przez siebie produktu. Firma 1 pierwsza będzie podejmować decyzję, czy inwestować w reklamę, czy też nie. Ta decyzja odpowiada pierwszemu rozgałęzieniu na poniższym drzewie. Kolejna decyzja należy do firmy drugiej - inwestować w reklamę lub nie. W sumie są cztery możliwe sytuacje (cztery różne rozgrywki) i odpowiadające im punkty (można je interpretować np. jako zysk).



Rysunek 1.2: Drzewo decyzyjne dla inwestowania w reklamę

Jeżeli obie firmy nie zdecydują się nie inwestować w reklamę, to firma 1 zdobędzie 16 punktów, a firma 2 - 12 punktów. Przyjmijmy, że koszt reklamy to 8 punktów. Jeżeli jedna firma zdecyduje się na reklamę, a druga nie, to firma reklamująca uzyska 13 punktów, a firma która nie reklamowała swojego produktu 7 punktów (firma reklamująca się zdobywa 21 punktów z 28 dostępnych, ale ponosi 8 punktowy koszt reklamy). Natomiast jeżeli obie firmy zdecydują się na reklamę, to podzielą między siebie rynek po równo, ale ponieważ każda z nich poniesie koszt 8 punktów, to finalnie każda zarobi po 6 punktów  $((28 - 16)/2)$ .

Stawiamy się w pozycji prezesem firmy 1, który chce podjąć decyzję o tym, czy inwestować w reklamę czy też nie. Założmy dalej, że drzewo przedstawiające możliwe wyniki tej gry jest znane wszystkim zainteresowanym stronom. Wówczas logicznym będzie następujące rozumowanie. Jeżeli firma 1 zdecyduje się na reklamę, to firma 2 chcąc maksymalizować liczbę zdobytych punktów nie będzie się reklamować. Wówczas firma 1 zdobędzie 13, a firma 2 - 7 punktów. Z drugiej strony, jeżeli firma 1 zdecyduje się nie reklamować, to firma 2 znowu maksymalizując swoją wygraną zainwestuje w reklamę i końcowy wynik będzie 7 i 13 punktów dla firmy 1 i firmy 2 odpowiednio. Ponieważ pierwszy scenariusz jest korzystniejszy, to firma 1 decyduje się na inwestycję w reklamę.

Z powyższego, bardzo prostego rozumowania, wynika kilka wniosków. Po pierwsze, przy założeniu, że oponent (firma 2 w tym przypadku) racjonalnie usiłuje maksymalizować liczbę zdobytych punktów, to wybór „otwierający” drogę do liści z największą liczbą punktów nie musi być wyborem optymalnym. Firma 1 mogłaby zdecydować nie reklamować się, bo wówczas jej potencjalna wygrana, to 7 lub 16 punktów versus 6 i 13 przy reklamowaniu się. Taka strategia byłaby słuszną gdyby firma 2 losowo podejmowała decyzje, ale nie jeżeli firma 2 ma racjonalnych decydentów. Druga obserwacja, często podnoszona w teorii gier, że najlepsze rozwiązanie dla obydwu firm - nie reklamować swoich produktów - jest rozwiązaniem niemożliwym lub przynajmniej nie stabilnym. Teoretycznie prezesi firm mogliby się zmówić, że nie będą inwestować w reklamę. Ale w takiej sytuacji, o ile firma 1 rzeczywiście zdecyduje nie reklamować się, to firma 2, chcąc maksymalizować swoje punkty, powinna zdecydować się na reklamę i tym samym złamać umowę. Stąd wspomniana wyżej niestabilność.

Na powyższym przykładzie można też zrozumieć w jaki sposób wybrać optymalny ruch. Założmy, że reprezentujemy firmę 1. Do nas należy otwierający ruch. Racjonalnie będzie założyć, że po naszym ruchu firma 2 wykona ruch, który pozwoli w ostatecznym rachunku na maksymalizację jej wygranej, co przy stałej puli możliwych do wygrania punktów, będzie tożsame z takim ruchem, który zminimalizuje naszą wygraną. Zatem należy wybrać taki ruch, który maksymalizuje minimum z naszych wygranych do jakich może doprowadzić firma 2 kolejnym ruchem. To rozumowanie naturalnie przekłada się na gry, których drzewa mają więcej pokoleń. W każdym przypadku konieczne jest przejście wszystkich węzłów drzewa idąc od liść (tutaj wiadomo jakie są wygrane) biorąc pod uwagę dla każdego pokolenia czy to ruch gracza („nasz ruch”), czy też ruch oponenta, który ma na celu minimalizowanie naszej wygranej. Przedstawiając powyższe rozumowanie w punktach dochodzi się do algorytmu minimax, który dla każdego węzła zwraca wygraną wartość gracza (wygraną pod warunkiem stosowania strategii minimax).

1. Jeżeli to jest ruch końcowy, to zwróć wartość wygranej i skończ
2. Jeżeli ruch należy do nas, a nie do oponenta to:
  - (a) ustawa wartość =  $-\infty$
  - (b) Dla każdego możliwego dostępnego ruchu:
  - (c) wartość =  $\max(\text{wartość}, \text{wartość algorytmu minimax na poddrzewie, którego korzeniem jest rozpatrywany ruch oponenta})$
  - (d) zwróć wartość
3. Jeżeli ruch należy do oponenta to:

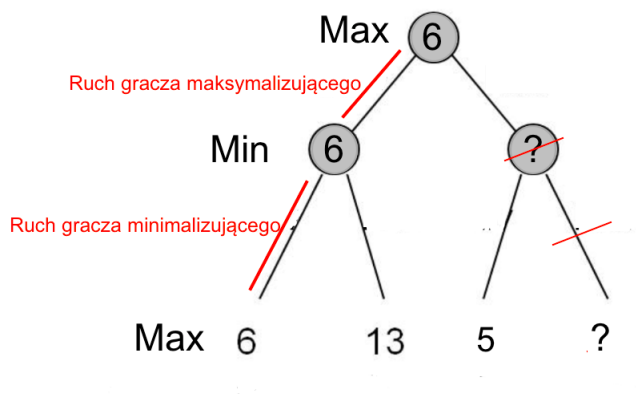
- (a) ustawa wartość  $= +\infty$
- (b) Dla każdego możliwego dostępnego ruchu:
- (c) wartość  $= \min(\text{wartość, wartość algorytmu minimax na poddrzewie, którego korzeniem jest rozpatrywany ruch dla nas}$
- (d) zwróć wartość

Okazuje się, że jeżeli gracze stosują się do powyższego algorytmu, to znaczy wybierają ruchy, które zapewnią im największą wygraną wyliczoną według powyższego algorytmu, to takie strategie będą w równowadze Nash'a. Mówiąc obrazowo są to najlepsze strategie, w tym sensie, że żadnemu z graczy nie opłaca się ich zmieniać. Jeżeli wiadomo, że oponent będzie grał zgodnie z algorytmem minimax, to moją najlepszą strategią jest robić to samo. Jeżeli oponent będzie grał inną strategią, to być może istnieje lepsza odpowiedź na jego strategię niż minimax. Jeżeli gracz gra według algorytmu minimax, a jego oponent nie, to oponent stosuje strategię suboptymalną.

## 1.2. Przycinanie $\alpha - \beta$

Podstawowym problemem w zastosowaniach algorytmu minimax jest ilość niezbędnych operacji przy dużych drzewach reprezentujących gry. Dla gry w kółko i krzyżyk, górnym ograniczeniem ilości możliwych gier (liści na drzewie gry) jest  $9! = 362880$ . W rzeczywistości ilość możliwych partii ogranicza się do kilku tysięcy, ale i tak przeliczenie wszystkich węzłów takiego drzewa na współczesnym komputerze średniej klasy zajmuje kilka sekund (przynajmniej przy amatorskiej implementacji autora niniejszej pracy). Jeżeli pomyśleć o grach takich jak szachy, w których ilość możliwych ruchów i łączna ilość ruchów w grze jest ogromna, to jasnym się staje, że algorytm minimax ma poważne ograniczenia.

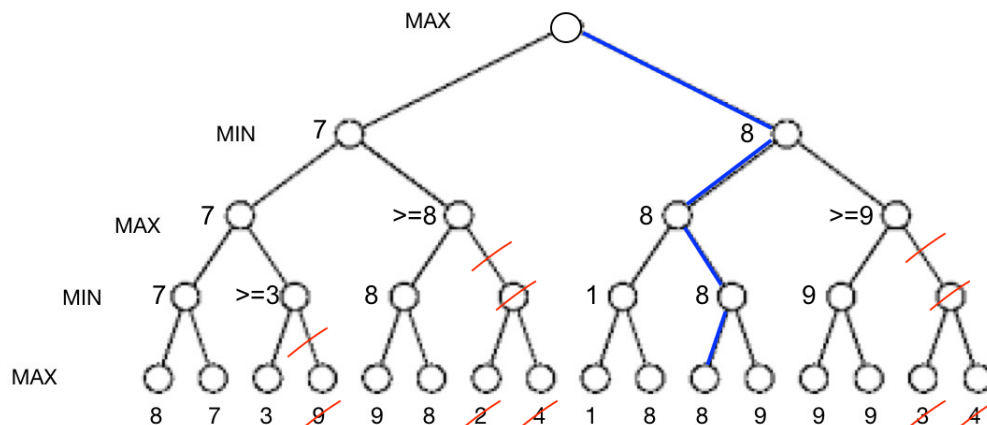
W praktyce, żeby ograniczyć ilość wykonywanych operacji stosuje się tzw. przycinanie  $\alpha - \beta$ . Polega to na tym, że jeżeli wiadomo, że sprawdzenie i porównanie wartości dla kolejnego ruchu/węzła na drzewie gry nie zmieni wyboru gracza, to pomija się takie obliczenie. Najlepiej zobaczyć to (patrz rysunek 1.3) na prostym przykładzie.



Rysunek 1.3: Przycinanie  $\alpha - \beta$

W tym przykładzie gracz rozpoczynający grę chce maksymalizować swoją wygraną. W pierwszym ruchu ma dwie możliwości. Może pójść w lewo i wówczas jego oponent z pewno-

Załóżmy, że mamy do czynienia z takim drzewem gry jak na rysunku B.1. Ostatni wiersz przedstawia możliwe wygrane gracza maksymalizującego, rozpoczynającego grę. Na rysunku zaznaczono węzły i całe podzewa, których gracz maksymalizujący nie potrzebuje rozpatrywać. Po lewej stronie węzłów zapisana jest maksymalna możliwa wygrana gracza maksymalizującego o ile gra znajdzie się w tym węźle (czyli wartość jaką zwraca algorytm minimax). Pogrubioną linią widać optymalną ścieżkę gry, która kończy się zdobyciem 8 punktów. Opis rozumowania prowadzący do tego rysunku byłby długi i nie wiele wnoszący do całości pracy, stąd zostaje pominięty. Przykład ten wzięty jest z 6 wykładu profesora Winstona z [4].



Jak widać, w niektórych przypadkach udaje się sporo zaoszczędzić w ilości obliczeń. Warto jeszcze wyjaśnić skąd  $\alpha - \beta$  w nazwie. Zaczynając przeszukiwanie drzewa ustawia się parametry  $\alpha$  i  $\beta$  na  $+\infty$  i  $-\infty$  odpowiednio, i interpretuje się je jako najgorszy możliwy wynik gracza minimalizującego ( $+\infty$ ) i najgorszy możliwy wynik gracza maksymalizującego ( $-\infty$ ). Przechodząc rekurencyjnie przez drzewo sprawdza się aktualizuje wartości  $\alpha$  i  $\beta$  i jeżeli dla danego poddrzewa nie możemy poprawić wyniku  $\alpha$  lub  $\beta$  (w zależności czy dla tego poddrzewa pierwszy ruch należy do gracza minimalizującego lub maksymalizującego), to nie przeszukujemy dalej tego poddrzewa.

### 1.3. Funkcja wartości

Nietrudno wyobrazić sobie, że dla gier z olbrzymimi drzewami gry, przycinanie  $\alpha - \beta$  może nam zejść o o kilka pokoleń w dół drzewa, ale i tak nawet najszybszy komputer nie jest w stanie przeszukać całego drzewa w takiej grze jak szachy. W związku z tym, algorytmy grające w takie gry wykorzystują **funkcję wartości**. Jest to funkcja zwracająca pewną wartość (scoring) dla każdego stanu gry. Czym wyższa wartość, tym bardziej korzystna sytuacja dla gracza.

Algorytm grający będzie z reguły miał ustawiony parametr, który każe przerwać przeszukiwanie drzewa na danym poziomie lub po przekroczeniu określonej ilości czasu. Następnie dla wszystkich liści z przeszukanego poddrzewa do momentu zastopowania algorytmu oblicza się funkcję wartości i w ten sposób estymuje się optymalny ruch. Dla szachów funkcja wartości może np. zliczać ilość punktów (każda figura ma przypisaną liczbę punktów - pionek 1, koń 5, wieża 8 itd) gracza i oponenta figur będących na planszy.

### 1.4. Implementacja

W tej sekcji opisanych zostało kilka spostrzeżeń związanych z implementacją algorytmu minimax z przycinaniem  $\alpha - \beta$ .

Zastosowanie przycinania  $\alpha - \beta$ , wbrew pokładnych nadziejom, nie zmniejszyło drastycznie czasu wykonania algorytmu. Najdłuższy czas oczekiwania ma miejsce na początku gry, w pierwszych dwóch ruchach, kiedy konieczne jest przeszukanie największych drzew. Od trzeciego ruchu, wykonanie algorytmu trwa poniżej 0.1 sekundy, więc optymalizacje przestają mieć znaczenie. Żeby ograniczyć czas oczekiwania na odpowiedź komputera dodałem instrukcję, powodującą, że jeżeli algorytm zaczyna grę, to zawsze wybiera środkowe pole.

Nie udało się też zaimplementować algorytmu z ograniczeniem głębokości przeszukiwania drzewa i wykorzystującego funkcję wartości. Banalne funkcje wartości typu sprawdzenie, czy mam dwa krzyżyki(kółka) w linii nie dawały dobrych rezultatów. Tak skalibrowany algorytm popełniał oczywiste błędy i łatwo z nim było wygrać. Natomiast uwzględnienie większej ilości niuansów, w zasadzie prowadziło do napisania reguły jak grać w kółko i krzyżyk. Wówczas minimax przestaje być potrzebny i można się ograniczyć do policzenia funkcji wartości dla możliwych ruchów i wybrania tego z największą wartością. Niestety nie udało się znaleźć „złotego środka” i w końcowa implementacja nie korzysta z funkcji wartości.

Algorytm minimax jest graczem optymalnym, nie można grać lepiej od algorytmu minimax. W szczególności, można co najwyżej z nim zremisować. W dalszej części pracy algorytm minimax będzie obok gracza losowego (takiego co losuje z jednostajnym prawdopodobieństwem kolejny ruch z puli możliwych) punktem odniesienia skuteczności algorytmu grającego w kółko i krzyżyk. Skuteczne algorytmy powinny często remisować i rzadko przegrywać z minimax, oraz często wygrywać i rzadko remisować z graczem losowym. Poniżej przedstawiano wyniki partii 100 gier rozegranych pomiędzy graczem stosującym minimax oraz graczem losowym oraz 100 gier rozegranych pomiędzy dwoma graczami stosującymi algorytm minimax.

Gracz stosujący minimax nie przegrał ani razu, za to zremisował z graczem losowym 13 razy na 100 gier co ilustruje rysunek 1.5. Natomiast dwóch graczy stosujących minimax

```
X wins: 0 times. Prcnt: 0.0
O wins: 87 times. Prcnt: 0.87
there were 13 ties. Prcnt: 0.13
```

Rysunek 1.5: Gracz losowy vs minimax

zawsze będą ze sobą remisować (rysunek 1.6).

```
X wins: 0 times. Prcnt: 0.0
O wins: 0 times. Prcnt: 0.0
there were 100 ties. Prcnt: 1.0
```

Rysunek 1.6: Gracz minimax vs minimax



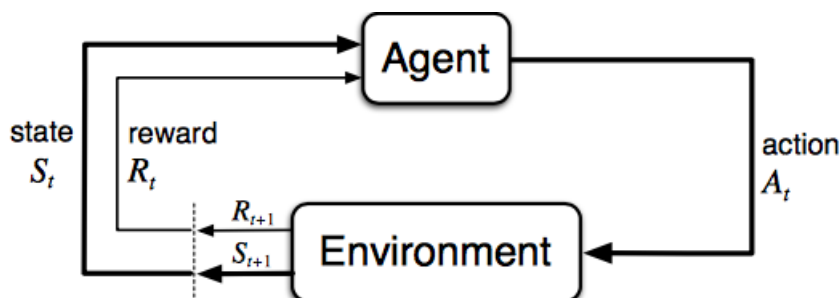


## Rozdział 2

# Q-learning

### 2.1. Procesy decyzyjne Markowa

W rozdziale pierwszym gry były reprezentowane (a w zasadzie wszystkie możliwe rozgrywki) za pomocą drzew, gdzie węzły należały do możliwych stanów gry, a krawędź pomiędzy dwoma węzłami oznaczała przejście na skutek ruchu jednego z graczy do kolejnego stanu. W tym rozdziale zaprezentowany zostanie inny model, oparty na procesach decyzyjnych Markowa. Przed formalnymi definicjami pomocny może okazać się poglądowy rysunkiem wziętym z [3] oraz heurystyczny opis modelu.



Rysunek 2.1: Wzajemne oddziaływanie środowiska i agent w RL

Założmy, że mamy agenta (gracza), który jest w interakcji z pewnym systemem (z angielskiego *environment* na rysunku 2.1). Agent znajduje się w pewnym stanie  $s$ , w dyskretnym czasie  $t$  i ma do wyboru podjęcie jednej z dostępnych akcji  $a \in \mathcal{A}$ . Na skutek podjęcia akcji  $a$  system odpowiada przejściem do nowego stanu  $s'$  oraz nagrodą  $r$  (z ang. *reward*) w czasie  $t + 1$ . Nie jest przy tym powiedziane, że dla konkretnej pary  $(s, a)$  system zawsze przejdzie do tego samego stanu  $s'$ . System może wybierać swoją odpowiedź losowo (czy ogólnie wedle pewnej niedeterministycznej reguły). Podobnie przy przejściu ze stanu  $s$  do  $s'$  system może zwracać różne nagrody  $r$ . Natomiast ważne, jest że przejście do kolejnego stanu  $s'$  oraz powiązana z przejściem nagroda  $r$  zależą tylko i wyłącznie od pary  $(s, a)$  i nie mogą w żaden sposób zależeć od poprzednich stanów w jakich system się znajdował. Na tym polega istota „markowość”, że historia (poza ostatnim ruchem) nie ma znaczenia. W tym miejscu dodaje się jeszcze jedno założenie dotyczące skończoności zbioru możliwych stanów systemu. To założenie nie jest konieczne w ogólnej teorii, ale pozwala na daleko idące uproszczenia wykorzystywane w niniejszej pracy.

Interakcja agenta z systemem przez kilka kolejnych kroków poczynawszy od  $t = 0$  można przedstawić za pomocą takiej trajektorii:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \dots$$

Formalizując pojęcie procedu decyzyjnego Markowa rozważmy czwórkę postaci  $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$  gdzie:

1.  $\mathcal{S}$  jest skończonym zbiorem możliwych stanów systemu
2.  $\mathcal{A}$  jest zbiorem możliwych akcji. Formalnie  $\mathcal{A}$  jest kolekcją zbiorów  $\mathcal{A}_s$  indeksowaną stanami  $s \in \mathcal{S}$ . Chodzi o to, że dla różnych stanów systemu dostępne są różne akcje. Dla uproszczenia pomija się jednak pomijać ten niuans notacyjny i w dalszej części mowa będzie o możliwych akcjach  $\mathcal{A}$  pamiętając, że ten zbiór może być ograniczony w zależności od konkretnego stanu.
3.  $\mathcal{R} \subseteq \mathbf{R}$  jest zbiorem wartości nagród, jakie agent może otrzymać
4.  $p : \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times \mathcal{S} \rightarrow [0, 1]$  jest funkcją determinującą dynamikę procesu decyzyjnego Markowa. Dla każdej pary stanu i wyboru akcji w czasie  $t$  funkcja  $p$  określa rozkład prawdopodobieństwa na  $\mathcal{R} \times \mathcal{S}$  w czasie  $t + 1$ :

$$p(s, a, s', r) = \mathbf{P}(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$$

Opisany model z powodzeniem można stosować do opisu gry w kółko i krzyżyk (jak również innych turowych gier). Przestrzeń stanów  $\mathcal{S}$  w tym wypadku to będą wszystkie możliwe stany gry. Akcje  $\mathcal{A}$  to będą wszystkie możliwe ruchy do wykonania przy danym stanie gry. Agentem jest gracz - algorytm, którego usiłuje się nauczyć grać w kółko i krzyży. Systemem natomiast jest drugi gracz - może to być gracz losowy lub gracz stosujący algorytm minimax. Pozostaje jeszcze określenie nagród  $\mathcal{R}$ . Wybór jest poniekąd arbitralny, ale ponieważ celem będzie maksymalizowanie wartości oczekiwanej nagród, to należy pamiętać, że algorytm wyuczy się osiągnięcia średnio wysokich nagród, a nie wygrywania.

Przed przejściem do opisu algorytmu potrzebne będzie jeszcze kilka pojęć. **Strategią** nazwiemy dowolne mapowanie

$$\pi : \mathcal{S} \rightarrow \mathbf{P}_{\mathcal{A}}, \text{ gdzie } \forall s \in \mathcal{S} \ \pi(s) \text{ jest rozkładem prawdopodobieństwa na } \mathcal{A}$$

Innym słowy, strategia  $\pi(s)$  mówi o tym jak gracz ma losować (wybierać) akcję  $a$  o ile znajduje się w stanie  $s$ .

Celem gracza będzie dążenie do takiej strategii, która maksymalizuje wartość nagród otrzymanych podczas gry. W procesach decyzyjnych Markowa często stosuje się dyskontowanie przyszłych nagród wybranym czynnikiem  $\gamma \leq 1$ . Wynika to z dwóch powodów. Po pierwsze w zastosowaniach często przyszłe nagrody mają rzeczywiście niższą wartość niż obecnie (jak strumień pieniądza w czasie w matematyce finansowej) a po drugie, interakcja agenta z systemem może nie być ograniczona ilością ruchów, a wtedy suma przyszłych nagród niezależnie od wyboru strategii  $\pi$  może być nieskończona, co uniemożliwia wybranie optymalnej strategii. Jeżeli natomiast maksymalna nagroda jest ograniczona, powiedźmy liczbą  $M$ , oraz  $\gamma < 1$ , to suma nagród jest z góry ograniczona przez

$$M + \gamma M + \gamma^2 M + \gamma^3 M + \dots = \frac{M}{1 - \gamma}$$

Dla uproszczenia zapisu wprowadza się oznaczenie na sumę przyszłych zdyskontowanych nagród:

$$G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

Wartością stanu  $s$  przy strategii  $\pi$  jest z definicji

$$v_\pi(s) \stackrel{\text{def}}{=} \mathbb{E}_\pi(G_t | S_t = s)$$

Jest to wartość oczekiwana sumy zdyskontowanych przyszłych nagród pod warunkiem, że stosujemy strategię  $\pi$  począwszy od czasu  $t$ . w którym to czasie system znajdował się w stanie  $s$ .

Podobnie zdefiniuje się jeszcze funkcję określającą wartość akcji  $a$  podjętą w stanie  $s$  jako zdyskontowaną wartość przyszłych nagród przy założeniu, że stosujemy strategię  $\pi$ , w stanie  $s$  i przy wyborze akcji  $a$  (strategia  $\pi$  zadaje rozkład prawdopodobieństwa na  $\mathcal{A}$ , i w tej definicji zakładamy, że akurat wylosowano konkretną akcję  $a$ ):

$$q_\pi(a, s) \stackrel{\text{def}}{=} \mathbb{E}_\pi(G_t | S_t = s, a_t = a)$$

Co zatem oznacza znalezienie optymalnej strategii? Intuicyjnie jest to oczywiste. Chcemy mieć strategię, która będzie maksymalizować wartość oczekiwaną sumy przyszłych zdyskontowanych nagród. Formalnie  $\pi_1 \prec \pi_2$  jeżeli dla każdego  $s \in \mathcal{S}$  zachodzi

$$v_{\pi_1}(s) \leq v_{\pi_2}(s)$$

Powiemy, że  $\pi_*$  jest optymalna, jeżeli dla każdej strategii  $\pi$  zachodzi  $\pi \prec \pi_*$ . Nietrudno przy tym zauważyć, że jeżeli  $\pi_*$  jest optymalną strategią, to spełnione będą równania:

$$\forall s \in \mathcal{S} : v_{\pi_*}(s) = \max_{\pi} v_{\pi}(s)$$

$$\forall s \in \mathcal{S} \wedge \forall a \in \mathcal{A} : q_{\pi_*}(a, s) = \max_{\pi} q_{\pi}(a, s)$$

Gdyby na przykład  $v_{\pi_*}(s) < v_{\tilde{\pi}}(s)$  dla pewnego stanu  $s$  i pewnej  $\tilde{\pi}$ , to strategię  $\pi_*$  można by poprawić grając strategią  $\tilde{\pi}$  po napotkaniu stanu  $s$  co jest sprzeczne z tym, że  $\tilde{\pi} \prec \pi_*$ . Poddobnie ma się rzecz z równaniem  $q_{\pi_*}(a, s) = \max_{\pi} q_{\pi}(a, s)$ .

Dopiero teraz przechodzimy do pytania jak znaleźć optymalną strategię? Jeżeli zbiory  $\mathcal{S}, \mathcal{A}, \mathcal{R}$  są skończone, to znajomość funkcji  $p$  daje „pełną wiedzę” o systemie. W szczególności można dokładnie wyliczyć optymalną strategię, bazując na tzw. równaniach Bellman’a, które wiążą  $v_\pi(s)$  z kolejnym stanem systemu  $v_\pi(s')$

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s', r} p(s, a, s', r) [r + \gamma v_\pi(s')]$$

Jeżeli wybierze się arbitralną nie losową strategię  $\pi$ , to bazując na równaniu Bellman’a można policzyć  $v_\pi(s)$  dla wszystkich stanów  $s$ . Sprowadza się to do rozsupłanie powyższej „rekurencji” przez rozwiązanie  $|\mathcal{S}|$  (tutaj  $|\cdot|$  oznacza liczebność zbioru) równań liniowych. Następnie można „poprawiać” deterministyczną strategię  $\pi$  poprzez sprawdzanie, czy dla poszczególnych stanów  $s$  i  $a$  (takich, że  $\pi(s) \neq a$ ) będzie  $q_\pi(s, a) \geq v_\pi(s)$ . Jeżeli znajdziemy „lepszą” akcję  $a$  dla stanu  $s$  aniżeli  $\pi(s)$ , to mamy poprawioną strategię

$$\pi'(s) = \begin{cases} a, & s = a, \\ \pi(s), & s \neq a \end{cases}$$

Okazuje się, że takie sukcesywne poprawianie strategii  $\pi \rightarrow \pi' \rightarrow \pi'' \rightarrow \dots$  prowadzi do optymalnej strategii  $\pi_*$ . Takie podejście w [3] nazywane jest programowaniem dynamicznym (z ang. *dynamic programming*). Jest to skuteczne podejście jeżeli dostępna jest pełna wiedza o funkcji  $p$  oraz jeżeli liczba możliwych stanów nie jest bardzo duża. Najczęściej to podejście sprawdza się przy modelowaniu rzeczywistych systemów za pomocą procesu decyzyjnego Markowa.

Inną interesującą sytuacją jest brak znajomości funkcji  $p$  i wówczas celem jest stworzenie algorytmu, który znajduje optymalną strategię doświadczalnie, ucząc się systemu, bez uprzednich założeń odnośnie jego dynamiki zadanej przez funkcję  $p$ . Najczęściej zaczyna się od losowej strategii i z każdym cyklem interakcji z systemem (albo po ustalonej liczbie  $n$  cyklach) dostosowujemy strategię na podstawie dotychczasowych obserwacji. Ponieważ  $v_\pi(s)$  jest definiowana jako wartość oczekiwana, to naturalnym odruchem może być wyliczanie przyszłych nagród startując ze stanu  $s$  i podążając za strategią  $\pi$  w wielu epizodach iteracji agenta z systemem, a następnie uśrednienia otrzymanych wartości. Takie podejście, typowo w stylu monte-carlo, ma poważną wadę w przypadku systemów z dużą ilością stanów, bo potrzeba czasami nierealistycznie dużej próbki danych (epizodów), żeby sensownie policzyć  $v_\pi(s)$ .

Metodą, poniekąd łączącą, programowanie dynamiczne z metodami w stylu monte carlo, nie wymagających znajomości funkcji  $p$ , są metody Q-learning (albo ogólnej z ang. *temporal difference*). Idea tych metod sprowadza się do tego, że można w każdy kroku, po każdej interakcji agenta z systemem, poprawiać stosowaną przez niego strategię  $\pi$  zgodnie ze wzorem

$$v_\pi(s_t) := v_\pi(s_t) + \alpha[R_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)] \quad (2.1)$$

W tym wypadku nie trzeba czekać do końca epizodu (np. do końca gry), a najlepiej wielokrotnie, żeby zaktualizować  $v_\pi(s_t)$  jak w metodach monte carlo. Możemy bazować tylko na otrzymanej nagrodzie i estymacji  $v_\pi(s_{t+1})$  dla kolejnego stanu. Tutaj  $\alpha$  jest hiperparametrem algorytmu, często nazywany z angielskiego *learning rate*, a  $\gamma$  jest, już wcześniej wspomnianym, czynnikiem dyskontującym wartość przyszłych nagród. W kolejnym rozdziale opisana jest zastosowanie tego rodzaju algorytmu.

## 2.2. Opis algorytmu

Strategia została zdefiniowana jako mapowanie, które każdemu stanowi  $s \in \mathcal{S}$  przypisuje rozkład prawdopodobieństwa  $\mathbf{P}_s$  na przestrzeni stanów  $\mathcal{A}$ . Teraz jednak celem jest przejście od ogólnej teorii do konkretnego algorytmu grającego w kółko i krzyżyk i w tym celu potrzebnych będzie kilka uproszczeń. W szczególności ograniczenie się do strategii nie losowych, tzn. do mapowania

$$\pi : \mathcal{S} \rightarrow \mathcal{A}$$

Jest to szczególny przypadek opisanej w poprzedniej sekcji teorii, gdzie rozważamy wyłącznie atomowe rozkłady prawdopodobieństwa na  $\mathcal{A}$ . Mając dany stan  $s \in \mathcal{S}$  potrzeba znaleźć optymalny ruch  $\pi(s) \in \mathcal{A}$ . Przy takim uproszczeniu:

$$\pi_*(s) = \arg \max_{a \in \mathcal{A}} q(a, s)$$

oraz

$$v_{\pi_*}(s) = \max_{a \in \mathcal{A}} q(a, s)$$

Optymalna strategia, przy założeniu że znaleźliśmy się w stanie  $s$  będzie maksymalizować oczekiwaną nagrodę wartość  $q(a, s)$ . Kolejnym celem jest włożenie tych uproszczeń do (2.1) i na tej podstawie wypracowanie algorytm nadającego się do implementacji. Na początek można położyć dla każdego stanu  $s \in \mathcal{S}$  i dla każdej akcji  $a$ ,  $q(a, s) = 0$ . Dalej założymy, że wykonano pierwszy cykl interakcji z systemem. W przypadku gry w kółko i krzyżyk oponent wykonał ruch i gra jest w stanie  $s$ . Teraz kolej na agenta, który dąży do tego, żeby nauczyć się optymalnej strategii. Zgodnie z (2.2) agent powinien rozważyć wszystkie możliwe ruchy  $a$  i wybrać ten, który maksymalizuje  $q(a, s)$ . Jeżeli jest więcej niż jedno takie  $a$ , to agent wybiera je losowo. W każdym razie agent wybiera  $\tilde{a}$ , które maksymalizuje wszystkie  $q(a, s)$ . System znowu odpowiada na ruch  $\tilde{a}$  i przechodzi do stanu  $\tilde{s}$  i zwraca nagrodę  $R$ . Teraz podstawiając  $q(\tilde{a}, s)$  za  $v_\pi(s)$  do (2.1) otrzymujemy przypis jak je zmienić:

$$q(\tilde{a}, s) := q(\tilde{a}, s) + \alpha[R + \gamma \cdot \max_{a \in \mathcal{A}} q(a, \tilde{s})] \quad (2.2)$$

Można zainicjować wszystkie  $q(a, s) := 0$ , a następnie podczas wielokrotnie rozgrywanych gier z systemem aktualizować te wartości po każdym ruchu zgodnie z (2.2). Okazuje się, że takie aktualizacje wartości  $q(a, s)$  po każdym ruchu, zapewniają zbieżność do rzeczywistych wartości  $q(a, s)$ . Następnie stosując (2.1) otrzymujemy optymalną strategię gry w kółko o krzyżyk.

## 2.3. Implementacja

Z poprzedniej sekcji wynika, że dla wszystkich możliwych stanów gry  $s$  i możliwych ruchów  $a$  można zainicjować wartości  $q(a, s) := 0$  i następnie je aktualizować. Wypisanie do listy lub tabeli wszystkich możliwych kombinacji  $(a, s)$  dla gry w kółko i krzyżyk jest oczywiście możliwe, aczkolwiek trochę kłopotliwe. Warto również zauważyć, że jeżeli po ruchu agenta mamy jakiś rozkład kółek i krzyżyków na planszy, to nie ma znaczenia jaki był poprzedni stan i jaka akcja agenta doprowadziła do tegoż rozkładu kółek i krzyżyków. Innymi słowy, dla dowolnych dwóch par  $(a', s')$  i  $(a'', s'')$  jeżeli zarówno wybór akcji  $a'$  w stanie  $s'$  jaki i wybór  $a''$  w stanie  $s''$  prowadzi do takiego samego stanu gry  $s$ , to  $q(a', s') = q(a'', s'') = q_s$ . Ta obserwacja skłania do przypisywaniu wartości  $q_s$  do stanów gry po ruchu agenta.

W samej implementacji konieczne jest jeszcze ustalenie wartości nagród  $\mathcal{R}$ . Wybór jest poniekąd arbitralny, ale warto zauważyć, że w grze w „kółko i krzyżyk” remis jest dobrym wynikiem. Grając z graczem stosującym algorytm minimax można co najwyżej zremisować. Stąd potrzeba rozróżnienia przegranej, remisu i wygranej. Gdyby na przykład przypisać nagrodę 1 za wygraną, -1 za przegraną i 0 za każdy inny stan planszy, to algorytm uczyłby się tylko nie przegrywać. Nie byłby bowiem w stanie odróżnić wygranej od remisów. W tej implementacji przyjąłem wartości nagród równe:

1. -1 w przypadku przegranej
2. 0.75 w przypadku remisu
3. 1 w przypadku wygranej
4. 0 dla każdego innego stanu gry

W implementacji wykorzystuję plik tekstowy, przechowujący wartości  $q_s$ , gdzie  $q_s$ , zgodnie z powyższą obserwacją jest równa wszystkim  $q(a', s')$ , które prowadzą do stanu gry  $s$  po

podjęciu akcji  $a'$  w stanie  $s'$  przez agenta. Na rysunku 2.2 widać kilka pozycji z tego pliku, już po przetrenowaniu algorytmu (około 20 000 gier rozegranych z graczem losowym), a poniżej znajduje się szczegółowe omówieniem struktury.

```
2,0,1,2,1,2,2,1,1,1.0
1,2,2,2,1,1,0,1,2,0.75
2,2,1,2,2,1,1,0,0,-1.0
2,0,0,1,1,2,0,0,0,0.418681257165
2,0,2,2,1,0,1,1,0,-0.0186631984604
```

Rysunek 2.2: Przykładowe dane z tablicy QTable

Pierwsze 9 cyfr rozdzielonych przecinkami przedstawia stan gry. Przy czym pierwsza cyfra opisuje pole w lewym górnym rogu planszy. Druga cyfra opisuje środkowe pole w górnym wierszu planszy. Trzecia cyfra pole w prawym górnym rogu. Kolejne trzy cyfry opisują środkowym rzęd planszy idąc od lewej do prawej. Ostatnie 3 cyfry, opisują idąc od lewej do prawej dolny rząd planszy. Cyfry 1 oznaczają ruchy systemu, który gra z algorytmem. Cyfry 2 przedstawiają ruchy agenta, czyli grającego algorytmu. Cyfra 0 oznacza puste pole. Ostatnia liczba rzeczywista oznacza wartość  $q$  dla każdego ruchu agenta, który doprowadzi do stanu planszy reprezentowanego przez pierwsze 9 cyfr.

Rozważmy pierwszy wiersz z rysunku 2.2. Dla ustalenia uwagi założmy, że system gra „kółkiem”, a agent „krzyżykiem”. Ciąg 2,0,1,2,1,2,2,1,1.0 przedstawia stan gry po ruchu agenta jak na rysunku 2.3. To jest sytuacja w której agent wygrywa, stąd wartość  $q$  dla tego stanu jest równa 1.

X		O
X	O	O
X	O	X

Rysunek 2.3: Stan gry dla ciągu (2, 0, 1, 2, 1, 2, 2, 1, 1.0)

Kolejny wpis w tabeli, to ciąg 1,2,2,2,1,1,0,1,2,0.75. Pierwsze 9 cyfr przedstawia stan gry zobrazowany na rysunku 2.4. Jak widać, niezależnie od ruchu gracza reprezentującego system (gracz posługujący się kółkiem), gra zakończy się remisem. Stąd wartość  $q$  w tym przypadku jest równa 0.75.

Kolejny wiersz, to ciąg 2,2,1,2,2,1,1,0,0,-1 (rysunek 2.5), jak łatwo zauważyć przedstawia sytuację, w której ruch należy do gracza systemowego (kółko lub cyfra 1) i gracz ten może z łatwością wygrać uzyskując trzy „kółka” na prawej skrajnej kolumnie. Stąd wartość  $q$  w tym przypadku jest równa -1.

O	X	X
X	O	O
	O	X

Rysunek 2.4: Stan gry dla ciągu (1, 2, 2, 2, 1, 1, 0, 1, 2, 0.75)

X	X	O
X	X	O
O		

Rysunek 2.5: Stan gry dla ciągu (2, 2, 1, 2, 2, 1, 1, 0, 0, -1)

Czwarty przykład odpowiadający ciągowi 2, 0, 0, 1, 2, 2, 0, 0, 0, 0.418681257165 zilustrowany na rysunku 2.6 to gra w stanie jeszcze mało zaawansowanym, w którym obaj gracze mają szansę wygrać lub przegrać. W tym przypadku wyczuona wartość  $q$  wynosi około 0.42, co wskazywałoby, że jest to relatywnie korzystna sytuacja dla agenta grającego „krzyżykiem”.

X		
O	O	X

Rysunek 2.6: Stan gry dla ciągu (2, 0, 0, 1, 2, 2, 0, 0, 0, 0.418681257165)

Ostatni przykład, to ciąg 2, 0, 2, 2, 1, 0, 1, 1, 0, -0.0186631984604. Wartość  $q$  lekko ujemna wskazuje na neutralność tego stanu. Ten przykład jest ciekawy i pokazuje słabość algorytmu.

System może w kolejnym ruchu wygrać grę. Pamiętajmy jednak, że algorytm uczył się grając z graczem losowym, który nie koniecznie wykorzysta możliwość wygrania i z tego stanu możliwe jest nadal zwycięstwo agenta (ale remis już nie). Przypuszczalnie dlatego wartość oczekiwana przyszłych nagród jest bliska zero. Gdyby uczyć algorytm poprzez grę tylko i wyłącznie z idealnym graczem, to najprawdopodobniej ten stan (po przetrenowaniu) miałby wartość bliższą  $-1$ .

		X	
O		X	

Rysunek 2.7: Stan gry dla ciągu (2, 0, 2, 2, 2, 1, 0, 1, 1, 0, -0.0186631984604)

Mając wszystkie niezbędne elementy można przejść do bezpośredniego zdefiniowania algorytmu:

1. Inicjuj pusty zbiór z wartościami  $q_s$
2. Ustaw parametry  $\alpha$  i  $\gamma$  z przedziału  $[0, 1]$
3. Graj  $n$  razy z systemem w kółko i krzyżyk:
  - (a) Jeżeli jest ruch agenta, a plansza jest w stanie  $s - 1$ :
    - i. Dla każdego legalnego ruchu prowadzącego z  $s - 1$  do  $s$  sprawdź wartość  $q_s$  w zbiorze, a jeżeli nie ma zbiorze wartości  $q_s$ , to przyjmij  $q_s = 0$ .
    - ii. Wybierz ten ruch  $\tilde{s}$ , dla którego  $q_{\tilde{s}} = \max_s q_s$ . Jeżeli jest kilka takich ruchów, to wybierz jeden losowo.
    - iii. Połóż  $q_{s-1} = q_{s-1} + \alpha(R(\tilde{s}) + \gamma q_{\tilde{s}} - q_{s-1})$
    - iv. Zapisz nową wartość  $q_{s-1}$  w zbiorze. Jeżeli wartość już istniała w zbiorze, to nadpisz.

Popatrzmy teraz na wyniki uczenia. Zaczynając od pustego zbioru  $q_s$  rozegrano po 100 partii z graczem losowym i z graczem stosującym minmax. Dodatkowo po każdej grze „opróżniany” jest zbiór  $q_s$  tak żeby algorytm na razie niczego nie uczył się, po to, żeby mieć później punkt odniesienia przy sprawdzaniu postępów w uczeniu się gry. W poniższych przykładach agent jest zawsze graczem korzystającym z „kółka”, a system jest reprezentowany przez „krzyżyk”.

W 100 grach rozegranych z graczem losowym otrzymałem wyniki jak na rysunku 2.8.

Jak widać agent jest nieznacznie lepszy od systemu, ale praktycznie w granicach losowego wahania.



```
X wins: 40 times. Prcnt: 0.4
O wins: 53 times. Prcnt: 0.53
there were 7 ties. Prcnt: 0.07
```

Rysunek 2.8: Wynik 100 gier z graczem losowym(X) przed uczeniem algorytmu

Natomiast 100 gier rozegranych z graczem minimax (zobacz rysunek 2.9) dało 10 remisów i 90 przegranych agenta

```
X wins: 0 times. Prcnt: 0.0
O wins: 90 times. Prcnt: 0.9
there were 10 ties. Prcnt: 0.1
```

Rysunek 2.9: Wynik 100 gier z graczem minimax (X) przed uczeniem algorytmu

Po przeuczeniu algorytmu na 20 000 grach z graczem losowym, zbiór  $q_s$  ma 775 rekordów (tyle różnych stanów agent napotkał podczas rozegranych gier), natomiast wyniki dla 100 gier, przedstawione na rysunku 2.10 teraz wyglądają już znacznie lepiej.

```
X wins: 3 times. Prcnt: 0.03
O wins: 76 times. Prcnt: 0.76
there were 21 ties. Prcnt: 0.21
```

Rysunek 2.10: Wynik 100 gier z graczem losowym(X) po przetrenowaniu algorytmu

Z graczem losowym, wytrenowany agent przegrywa tylko w 3%, ma 76% wygranych i 21% remisów. Grając z tak wytrenowanym agentem można odczuć, że gra całkiem dobrze, ale jeszcze czasami potrafi wykonać poważny błąd. Nie ma natomiast prawie żadnej poprawy jeżeli chodzi o liczbę remisów (bo wygrać się nie da) z systemem grającym strategią minimax, co widać na rysunku 2.11.

```
X wins: 0 times. Prcnt: 0.0
O wins: 88 times. Prcnt: 0.88
there were 12 ties. Prcnt: 0.12
```

Rysunek 2.11: Wynik 100 gier z graczem minimax (O) po przetrenowaniu algorytmu

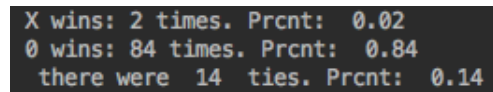
Nasuwa się oczywiście pytanie, czy przetrenowanie algorytmu na grach z systemem grającym strategią minimax poprawiłoby tę statystykę?. Na rysunku 2.12 przedstawione zostały wyniki algorytmu po dodatkowych 10 000 grach z systemem grającym strategią minimax.

```
X wins: 0 times. Prcnt: 0.0
O wins: 0 times. Prcnt: 0.0
there were 100 ties. Prcnt: 1.0
```

Rysunek 2.12: Wynik 100 gier z graczem minimax (O) po przetrenowaniu algorytmu

A więc trenowanie algorytmu na systemie grającym strategią minimax odniosło skutki. Agentowi udało się 100 razy na 100 gier zremisować z graczem grającym według minimax. Warto jeszcze sprawdzić, czy po tym trenowaniu algorytm nie odnotuje gorszych wyników

grając z graczem losowym. Wyniki zamieszczono na rysunku 2.13.



```
X wins: 2 times. Prcnt: 0.02
O wins: 84 times. Prcnt: 0.84
there were 14 ties. Prcnt: 0.14
```

Rysunek 2.13: Wynik 100 gier z graczem losowym(X) po dodatkowym przetrenowaniu algorytmu z graczem minimax

Jak widać wyniki uległy nawet poprawie. Agent po dodatkowym trenowaniu wygrywa w 84%, przegrywa w 2% i w 14% remisuje. Należy jednak zaznaczyć, że takie trenowanie algorytmu z systemem, który gra zgodnie ze strategią minimax, nie jest zupełnie zgodne z pierwotną ideą, żeby algorytm nauczył się grać w kółko i krzyżyk bez żadnych dodatkowych informacji, w tym bez możliwości uczenia się przez grę z optymalną strategią.

Na koniec warto jeszcze dodać, że po takim podwójnym trenowaniu, czysto subiektywne odczucie jest takie, że algorytm gra w sposób zbliżony do człowieka. Rzadko popełnia oczywiste błędy, a chwila nie uwagi kończy się zazwyczaj przegraną.

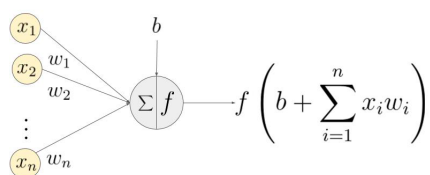
## Rozdział 3

# Sieć neuronowa grająca w kółko i krzyżyk

### 3.1. Sieci neuronowe

Metoda opisana w poprzednim rozdziale ma jedną zasadniczą wadę. Dla gier o znacznie większej liczbie możliwych kombinacji plansz, zbiory z wartościami  $q_s$  stają się ogromne. O ile jeszcze samo zapisanie takiego zbioru, zważywszy na szybki rozwój elektronicznych nośników danych, mógłby być możliwy, to już przeszukiwanie przed każdym ruchem zbioru w celu wyłonienia tego, który maksymalizuje  $q_s$  jest absolutnie nie wykonalne. Alternatywą do przechowywania wartości  $q_s$  w tabeli lub pliku, może być wytrenowanie sieci neuronowej w taki sposób, żeby estymowała funkcję  $s \rightarrow q_s$

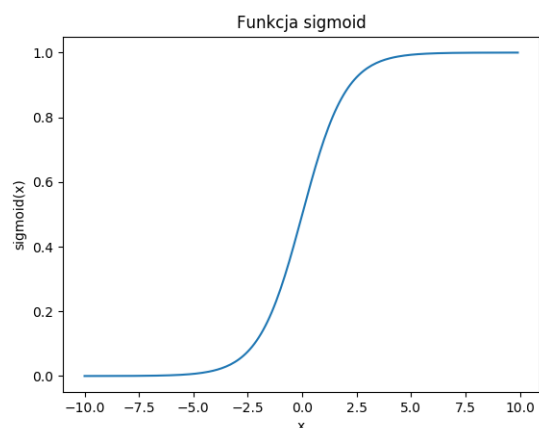
Sieć neuronowa składa się z neuronów. Natomiast jeden neuron ma określoną liczbę  $n$  danych wejściowych, odpowiadającą tej ilości liczbę wag  $w_i, 1 < i \leq n+1$ , (o jeden więcej niż ilość danych wejściowych) przy czym wprowadza się oznaczenie  $b = w_{n+1}$  (wyraz stały), oraz zadaną funkcję aktywacji  $f : \mathbf{R} \rightarrow \mathbf{R}$ . Załóżmy, że dane wejściowe to  $x_1, x_2, \dots, x_n$ . Neuron wylicza sumę ważoną tych danych powiększoną o wyraz stały  $w_{n+1}$  i zwraca wartość funkcji aktywacji z tej sumy :



Rysunek 3.1: Zasada działania neuronu

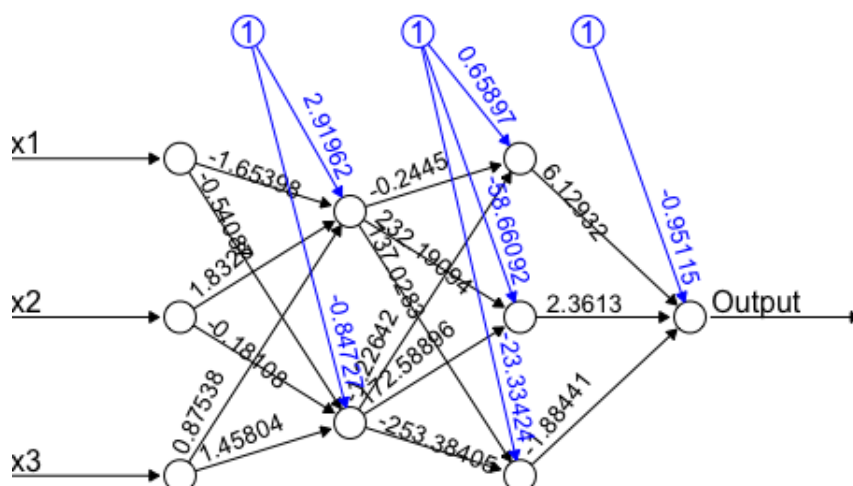
Funkcja aktywacji może być w zasadzie dowolna. Ze względu na łatwość „uczenia” sieci neuronowej, o czym za chwilę, warto stosować funkcje różniczkowalne (ewentualnie różniczkowalne poza kilkoma punktami) i monotoniczne. Najczęściej stosuje się takie funkcje jak sigmoid, tangens hiporbeliczny, relu, czy softplus. Poniżej definicja i wykres (rysunek 3.2) funkcji sigmoid - najbardziej klasycznego przykładu funkcji aktywacji.

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



Rysunek 3.2: Wykres funkcji aktywacji - sigmoid

Sieć neuronowa składa się, jak nazwa wskazuje z wielu neuronów, ułożonych w warstwy. Pierwsza warstwa, to warstwa wejściowa danych. Dane z wejściowe są przekazywane neuronom w w drugiej warstwie z neuronami. Każdy neuron ma swój zestaw wag, które służą do utworzenia kombinacji liniowej z danych wejściowych, po czym neuron zwraca wartości funkcji aktywacji na tej kombinacji liniowej. Wartości zwracane przez neurony w drugiej warstwie są z kolei przekazywane do trzeciej warstwy z neuronami, tak jak dane wejściowe zostały przekazane do drugiej warstwy (pierwszej z neuronami). Wartości obliczone na ostatniej warstwie z neuronami, to są wartości ostatecznie zwracane przez sieć.



Rysunek 3.3: Przykład sieci neuronowej wygenerowany w R

Powyżej, rysunek 3.3 przedstawia przykładową sieć, która ma trzy wartości wejściowe, a następnie warstwę 2 neuronów, warstwę 3 neuronów i ostatnią warstwę wyjściową składającą się z jednego neurona. Wartości na strzałkach pokazują wartości wag neuronu, do którego prowadzi strzałka. Dodatkowo, na niebiesko zaznaczono wartości stałe ( $b$ ) dla poszczególnych neuronów. Przy tworzeniu sieci neuronowej, ważne jest, żeby ilość wag dla każdego neuronu w  $k$ -tej warstwie, była równa ilości neuronów (lub danych wejściowych) w  $k-1$ -ej warstwie (plus dodatkowo wyraz stały). W przeciwnym razie nie uda się uwzględnić wszystkich wartości z

k-1 warstwy w kombinacji liniowej.

Podsumowując można powiedzieć, że siecią neuronową, o zadanej strukturze, jest funkcja  $\mathcal{N}$ , której argumenty, to zbiór wartości wejściowych  $x_1, \dots, x_n$  oraz zbiór wszystkich wag  $\mathbf{W}$  i zbiór wszystkich wyrazów stałych  $\mathbf{b}$ , która zwraca wektor wartości  $y_1, \dots, y_m$  o wymiarze równym ilości neuronów w ostatniej warstwie.

$$\mathcal{N}(x_1, \dots, x_n, \mathbf{W}, \mathbf{b}) = (y_1, \dots, y_m)$$

Argumenty  $\mathbf{W}$  i  $\mathbf{b}$  można traktować jak parametry funkcji  $\mathcal{N}$ . Zasadniczym celem jest tak zmieniać te parametry, żeby sieć neuronowa  $\mathcal{N}$  przybliżała nam jakąś rzeczywistą, obserwowalną, ale nieznaną funkcję  $\Phi$ :

$$(x_1, \dots, x_n) \xrightarrow{\Phi} (y_1, \dots, y_m)$$

Poniżej przedstawiony został zarys klasycznej metody uczenia sieci neuronowych (czyli de facto dostrajania wag  $\mathbf{W}$  i  $\mathbf{b}$  na podstawie danych) bazujący na pierwszych rozdziałach [1].

Po pierwsze, skoro celem jest przybliżanie funkcji  $\Phi$  siecią neuronową  $\mathcal{N}$ , to potrzebna jest jakaś miara skuteczności takiego przybliżenia. Załóżmy, że mam k-elementowy zbiór danych (treningowych)  $\mathbf{X}$  i dla każdego  $x_1, \dots, x_n \in \mathbf{X}$  obserwujemy  $\Phi(x_1, \dots, x_n)$  oraz  $\mathcal{N}(x_1, \dots, x_n, \mathbf{W}, \mathbf{b})$ . Jak w tym przypadku określić skuteczność przybliżania  $\Phi$  za pomocą sieci  $\mathbf{W}, \mathbf{b}$  na zbiorze  $\mathbf{X}$ ? W literaturze można spotkać wiele metod, ale chyba najbardziej naturalną jest średnio kwadratowa funkcja straty (kosztu), którą zapiszemy jako funkcję parametrów  $\mathbf{W}$  i  $\mathbf{b}$ .

$$C(\mathbf{W}, \mathbf{b}) = \frac{1}{2k} \sum_{(x_1, \dots, x_n) \in \mathbf{X}} \|\Phi(x_1, \dots, x_n) - \mathcal{N}(x_1, \dots, x_n, \mathbf{W}, \mathbf{b})\|^2 \quad (3.1)$$

Norma występującą w powyższym wyrażeniu, to zwykła odległość euklidesowa w przestrzeni  $\mathbf{R}^d$ . Zauważmy, że jeżeli ustalimy zbiór testowy  $\mathbf{X}$  i zbiór obserwacji  $\Phi(\mathbf{X})$ , to powyższą funkcję kosztu możemy rozpatrywać jako funkcję parametrów sieci neuronowej  $\mathcal{N}$ . Przy takim podejściu, znalezienie wag  $\mathbf{W}$  i  $\mathbf{b}$ , dla których sieć  $\mathcal{N}$  będzie najlepiej przybliżać funkcję  $\Phi$  sprowadza się do znalezienia minimum funkcji  $C(\mathbf{W}, \mathbf{b}) : \mathbf{R}^d \rightarrow \mathbf{R}$ , dla pewnej liczby naturalnej  $d$ . Dla uproszczenia zapisu przyjmijmy, że  $v := (\mathbf{W}, \mathbf{b})$ . Czyli funkcja kosztu  $C$  będzie funkcją zmiennej  $v \in \mathbf{R}^d$ . Z rachunku różniczkowego wiadomo, że:

$$\Delta C = \nabla C \cdot \Delta v \quad (3.2)$$

W powyższym równaniu symbol  $(\cdot)$  oznacza iloczyn skalarny wektorów. Natomiast  $\nabla C$  to jest gradient funkcji  $C$ , czyli  $\nabla C = (\frac{\delta C}{\delta v_1}, \dots, \frac{\delta C}{\delta v_d})$ . Zwróćmy uwagę, że  $\Delta C$  jest zwykłym skalarą. Jeżeli ustalimy sobie  $v$  i pewną małą wartość  $\eta > 0$ , to możemy położyć:

$$v' = v + \eta \nabla C(v),$$

$$\Delta v = v - v'$$

i wstawimy do (3.2), to otrzymamy:

$$\Delta C = \nabla C \cdot (v - v') = -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2 < 0$$

Jeżeli więc będziemy zmieniać komplet wag zgodnie ze wzorem:

$$v' := v + \eta \nabla C(v),$$

to będziemy zmniejszać wartość  $\Delta C$ , co oznacza, że będziemy podążali w kierunku minimum funkcji  $C$ . Ta metoda znana jest w literaturze (np. [1]) jako „*gradient descent*”. Ważnym parametrem w tej metodzie jest  $\eta$ , często nazywana w angielskojęzycznej literaturze *learning rate*. Jeżeli  $\eta$  jest duże, to potrzeba będzie mniej kroków na odnalezienie lokalnego minimum, ale zagrożenie jest takie, że zmieniając wagi będziemy „przeskakiwać” nad minimum niewiele zbliżając się do niego. Z drugiej strony, zbyt małe  $\eta$  może skutkować koniecznością wykonania bardzo wielu kroków w celu odnalezienia minimum.

Jeżeli liczba  $d$  jest duża, to analityczne szukanie punktów, w których zerują się wszystkie pochodne cząstkowe byłoby kosztowne obliczeniowo jeżeli nie niewykonalne. W praktyce często przyjmuje się inne podejście; odmianę tej metody znaną jako „*stochastic gradient descent*” (dalej SGD), która różni się tym, że funkcję kosztu  $C$  (3.1) wyliczamy na podstawie losowo wybranego podzbioru ze zbioru testowego  $\mathbf{X}$  i odpowiadających obserwacji  $\Phi(\mathbf{X})$ . Dla jednego zbioru obserwacji  $\mathbf{X}$  wielokrotnie losować podzbiór, obliczać  $\nabla C$  zdefiniowanego dla wylosowanego podzbioru i aktualizować wagi  $v$ . W literaturze można przeczytać, że taka metoda pozwala na skuteczne dobranie wag. Skrajny przypadek metody STD, to branie tylko jednej obserwacji do wyliczenia funkcji kosztu.

Reasumując, jeżeli potrafilibyśmy policzyć gradient  $\nabla C$  dla funkcji kosztu, to mamy przepis na to jak zmieniać wagi  $\mathbf{W}$  i wartości stałe  $\mathbf{b}$ , czyli parametry sieci  $\mathcal{N}$ , w sposób, który gwarantuje nam zmniejszanie wartości funkcji kosztu, a więc poprawiający przybliżenie. Tutaj z pomocą przychodzi algorytm propagacji wstecznej (z ang. „*backpropagation algorithm*”). Algorytm propagacji wstecznej, jest podstawowym algorytmem, który w połączeniu z powyższymi rozważaniami o tym jak zmieniać wagi sieci, pozwala na skuteczne trenowanie sieci. W wielkim skrócie, algorytm propagacji wstecznej sprawdza się do wliczenia wartości aktywacji na każdym neuronie, a następnie - idąc od ostatniej warstwy wyjściowej aż do pierwszej warstwy (stąd zapewne nazwa propagacji wstecznej) obliczenia błędów sieci i modyfikowanie wag na każdym poziomie. Dokładny opis algorytmu, jak również dowód poprawności algorytmu, jest dostępny w niemal każdym opracowaniu traktującym o podstawach sieci neuronowych. Ponieważ niniejsza praca raczej ma się koncentrować na praktycznych aspektach uczenia sieci, to opis algorytmu zostaje pominięty.

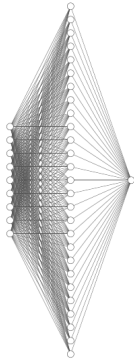
## 3.2. Podejście I - nieudane

Biorąc pod uwagę, że trenowanie algorytmu opisanego w poprzednim rozdziale dało całkiem dobre rezultaty, to naturalnym pomysłem było zastąpienie tabeli przypisujących stanom  $s$  wartości  $q_s$  siecią neuronową i pozostawienie pozostałych elementów niezmiennych. W rezultacie otrzymuje się taki algorytm:

1. Inicjuj sieć neuronową, która przyjmuje na wejściu 9 wartości i zwraca na wyjściu 1 wartość -  $q_s$
2. Ustaw parametry  $\alpha$  i  $\gamma$  z przedziału  $[0, 1]$
3. Graj  $n$  razy z systemem w kółko i krzyżyk:
  - (a) Jeżeli jest ruch agenta, a plansza jest w stanie  $s - 1$ :
    - i. Dla każdego legalnego ruchu prowadzącego z  $s - 1$  do  $s$  sprawdź zwracaną przez sieć neruonową wartość  $q_s$

- ii. Wybierz ten ruch  $\tilde{s}$ , dla którego  $q_{\tilde{s}} = \max_s q_s$ . Jeżeli jest kilka takich ruchów, to wybierz jeden losowo.
- iii. Połóż  $q_{s-1} = q_s + \alpha(R(\tilde{s}) + \gamma q_{\tilde{s}})$ , gdzie  $R(\tilde{s})$  jest nagrodą po ruchu  $\tilde{s}$ .
- iv. Przetrenuj sieć na jednoelementowym zbiorze  $\{s-1, q_{s-1}\}$ .

Pozostaje jeszcze kwestia ustalenia topologii sieci i tego jaka będzie funkcja aktywacji. Kółko i krzyżyk nie jest skomplikowaną grą, stąd decyzja o sieci z jedną warstwą ukrytą. Warstwa wejściowa, to oczywiście 9 pozycji odpowiadających dziewięciu polom na planszy. Warstwa wyjściowa, to jedna pozycja zwracającą  $q_s$  dla stanu reprezentowanego przez dane wejściowe. Przy czym podobnie jak w poprzednim rozdziale, dane wejściowe reprezentują stan gry po ruchu agenta. Ilość neuronów w warstwie ukrytej wybrano poniekąd arbitralnie na 27. Ze względu na symetrię miała to być wielokrotność (jednak nie za duża) ilości danych wejściowych.

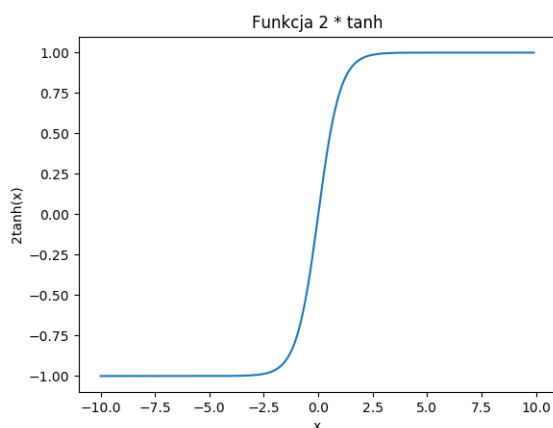


Rysunek 3.4: Topologia sieci w I podejściu

Na koniec jeszcze pozostaje ustalenie funkcja aktywacji. Ponieważ  $q_s$  reprezentuje wartość oczekiwaną przyszłych nagród, to ważne jest, żeby funkcja aktywacji miała zbiór wartości obejmujący zakres możliwych wartości przyszłych nagród. W poprzednim rozdziale, przyjęli takie wartości:

1. -1 w przypadku przegranej
2. 0.75 w przypadku remisu
3. 1 w przypadku wygranej
4. 0 dla każdego innego stanu gry

Ponieważ wygrana, remis, lub przegrana w grze może się zdarzyć tylko raz, to wartość oczekiwana przyszłych nagród, będzie w zakresie  $[-1, 1]$  i stąd dobrze mieć funkcję aktywacji obejmującą ten przedział. Dobrym kandydatem byłaby funkcja  $\tanh$ , czyli tangens hiperboliczny, którego zbiór wartości to właśnie przedział  $[-1, 1]$ . Problem jednak jest taki, że jeżeli funkcja aktywacji „wyplaszcza się” dla dużych argumentów, to jej pochodna jest bliska zeru, co z kolei przekłada się na to, że algorytm propagacji wstecznej (czyli ten, który liczy gradient funkcji kosztu) szacuje pochodne na bliskie zeru i cały algorytm uczenia sieci neuronowej prawie nie aktualizuje wag  $\mathbf{W}$ ,  $\mathbf{b}$ . Chcąc uniknąć takiego efektu „saturacji”, za funkcję aktywacji przyjęto  $f(x) = 2 \tanh(x)$ . Dla porządku, podano poniżej definicję funkcję tangens hiperboliczny i joraz przedstawiono jej wykres na rysunku 3.5.



Rysunek 3.5: Tangens hiperboliczny przemnożony przez 2

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \quad (3.3)$$

Niestety takie proste zastąpienie tabeli przechowującej wartości  $q_s$  przez sieć neuronową, która miałaby w podobny sposób, po każdym ruchu poprawić swoje oszacowanie  $q_s$  nie zdało egzaminu. Po długotrwałym trenowaniu (20 000 gier z graczem losowym) taka sieć potrafi wygrywać z graczem losowym w około 60%-70% przypadków, i w subiektywnym odczuciu gra zdecydowanie gorzej od człowieka. Poniżej (rysunek 3.6) wynik 100 rozegranych gier na przetrenowanej sieci:

```
X wins: 24 times. Prcnt: 0.24
0 wins: 61 times. Prcnt: 0.61
there were 15 ties. Prcnt: 0.15
```

Rysunek 3.6

Eksperymenty z większą liczbą neuronów w warstwie ukrytej, jak również z dwiema warstwami ukrytymi, oraz z innymi funkcjami aktywacji, kończyły się ale zawsze z podobnym, niezadawalającym efektem. Najlepsze wyniki jakie udało się uzyskać, to około 75% wygranych gier z graczem losowym. Wyniki takie uzyskano używając funkcji aktywacji *softplus*, która definiuje się jako  $f(x) = \ln(1 + e^x)$ . Ta funkcja aktywacji przyjmuje wartości z przedziału  $[0, +\infty]$  (jest to wygładzona wersja innej popularnej funkcji aktywacji - ReLu), stąd konieczna była zmiana wartości nagród. W tym przypadku przyjąłem:

1. 0.5 w przypadku przegranej
2. 75 w przypadku remisu
3. 100 w przypadku wygranej
4. 1 dla każdego innego stanu gry

Poniżej (rysunek 3.7) wynik rozegranych 100 gier z graczem losowym po przetrenowaniu na 20 000 gier.

Jak widać, sieć gra statystycznie nieco lepiej od gracza losowego, czyli algorytm „czegoś się nauczył, ale przy tak prostej grze jak kółko i krzyżyk można by się spodziewać znacznie



```
X wins: 30 times. Prcnt: 0.3
0 wins: 67 times. Prcnt: 0.67
there were 3 ties. Prcnt: 0.03
```

Rysunek 3.7

lepszych rezultatów. Należy jeszcze dodać, że dla wszystkich wypróbowanych kombinacji sieci, po przetrenowaniu, miały procent remisów z graczem minimax na poziomie od 0% do 5%.

### 3.3. Podejście II - bardziej udane

Dlaczego sieci z poprzedniej sekcji nie udało się dobrze nauczyć grać w kółko i krzyżyk? Trudno oczywiście o kategorię odpowiedź na to pytanie, ale można postawić hipotezę, że problem leży w tym, że sieć uczy się po każdym ruchu na jedno-elementowym zbiorze treningowym. Takie rozwiązanie ma zasadniczą wadę. Większość ruchów skutkuje nagrodą równą zero. Dopiero ostatni ruch w grze kończy się inną wartością nagrody. Sieć natomiast modyfikuje wagi po każdym ruchu i używa tych zmodyfikowanych wag do dalszej gry i dalszego uczenia. To może może powodować, że wielokrotnie modyfikując wagi dla ruchów z zerową nagrodą sieć „zapomina” wagi, które pozwalały wygrać. Lepszym podejściem byłoby rozegranie wielu gier (wielu ruchów) i przekazanie sieci do trenowania jednego dużego zbioru danych. W tej części takie podejście zostało wypróbowane. Pomysł został zaczerpnięty z pracy magisterskiej złożonej na uniwersytecie technicznym w Monachium [5].

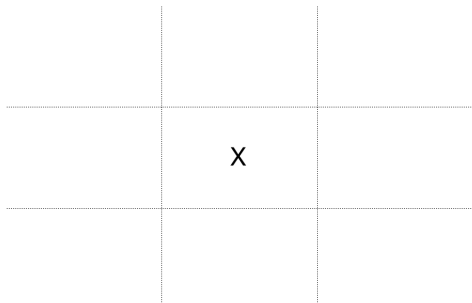
Dodatkowo, żeby wyeliminować ewentualny problem zbyt małej sieci, zmieniono jej topologię dodając jedną warstwę ukrytą. Ostatecznie sieć ma strukturę typu 9-54-27-1, czyli zawiera dwie warstwy ukryte po 54 i 27 neuronów odpowiednio. Za funkcję aktywacji przyjęto tangens hiperboliczny, który przyjmuje wartości z przedziału  $[-1, 1]$ . Nagrody, czy odpowiedzi systemu, ustalono z uwagą żeby być daleko od ekstremalnych wartości funkcji aktywacji:

1. -0.8 w przypadku przegranej
2. 0.5 w przypadku remisu
3. 0.8 w przypadku wygranej
4. 0 dla każdego innego stanu gry

Sam proces uczenia sieci wygląda teraz następująco. Ustala się arbitralnie wielkość zbioru do uczenia (w literaturze odpowiednik pojęcia *batch size*) np. na 2500. Następnie rozgrywane jest tyle gier, żeby zebrać 2500 obserwacji. Przy czym jedną obserwacją będzie teraz wektor przechowujący stan planszy po ruchu agenta (gracza odpowiadającego sieci neuronowej), wartość  $q_s$  czyli zwróconą przez sieć przyszłą oczekiwaną wartość nagród oraz rzeczywistą nagrodę  $R$  jaką system zwrócił po swoim ruchu lub po zakończeniu gry, jeżeli ruch agenta był ostatnim ruchem. Najlepiej zobaczyć jak to wygląda na przykładzie.

Pierwsze 9 pozycji odwzorowuje planszę gry. Zero odpowiada pustemu polu, jedynka to pole zajęte przez system (w tym wypadku gracz losowy), a dwójka to pole zajęte przez agenta. W poniższym (rysunek 3.8) przykładzie zaczyna agent, stawia krzyżyk (dwójkę) na środkowym polu. Na 10 pozycji - wartość 0.642 - znajduje się oczekiwana wartość przyszłych

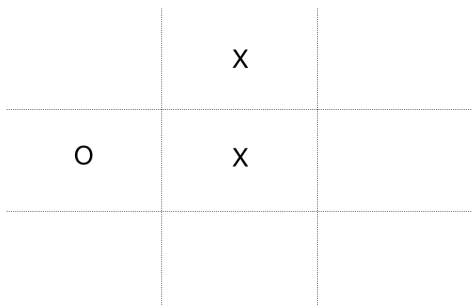
nagród zwrócona przez sieć. Następnie jest oczekiwanie na ruch systemu, chyba że to jest ostatni ruch w grze, i dopiero po tym ruchu systemu dopisujemy nagrodę jaką otrzyma agent. W tym wypadku będzie to zero, bo gra jeszcze się nie skończy.



Rysunek 3.8

Obserwacja po pierwszym ruchu:  $[0, 0, 0, 0, 2, 0, 0, 0, 0, 0.642, 0]$

Następnie system stawia swoje kółko w na pozycji 2,1, czyli w pierwszej kolumnie i w drugim rzędzie. Ten stan planszy nie zostanie zapisany, bo interesujące są tylko stany po ruchu agenta. Jednak po ruchu systemu sprawdza się jaka jest nagroda agenta i tym samym, dopiero po tym ruchu systemu mamy w pełni uzupełnioną pierwszą obserwację. Kolejny, trzeci ruch w grze, należy do agenta i ten wybiera (sprawdzając jaki ruch będzie skutkował maksymalną oczekiwaną nagrodą  $q_s$ ) pozycję 1,2:



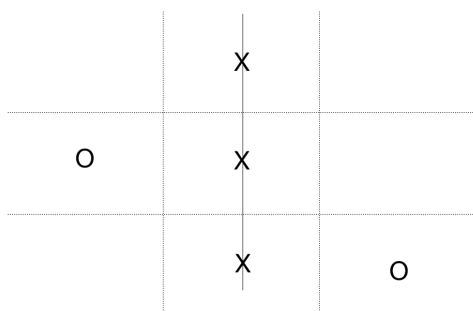
Rysunek 3.9

Obserwacja po drugim ruchu agenta:  $[0, 2, 0, 1, 2, 0, 0, 0, 0, 0.723, 0]$

System - gracz losowy - w kolejnym ruchu nie blokuje agenta, tylko stawia kółko na pozycji 3,3. W odpowiedzi na to, agent wybiera 3,2 i wygrywa grę, stąd nagroda jest teraz równa 0.8.

Obserwacja po trzecim i ostatnim ruchu agenta:  $[0, 2, 0, 1, 2, 0, 0, 2, 1, 0.843, 0.8]$

Powyższa gra dała w wyniku trzy obserwacje. Podczas jednej *epoki* rozgrywamy tyle gier, żeby otrzymać 2500 obserwacji (albo tyle na ile ustalimy *batch size*). Kolejnym krokiem jest zbudowanie na bazie tych obserwacji zbioru referencyjnego, na podstawie którego będziemy uczyć sieć. W tym celu zastąpimy prognozy  $q_s$  jakie zwracała sieć „lepszymi” wartościami  $\tilde{q}_s$  wedle następującej reguły. Ustalamy parametr  $\alpha \in (0, 1)$  i postępujemy jak następuje:



Rysunek 3.10

1. Jeżeli dana obserwacja jest ostatnią obserwacją w grze, to  $\tilde{q}_s = R$
2. W przeciwnym przypadku mamy  $N$  obserwacji pozyskanych w ramach aktualnie rozpatrywanej gry, gdzie  $N$  jest ostatnią obserwacją. Dla  $i = N - 1, N - 2, \dots, 1$  kładziemy:

$$(a) \tilde{q}_{N-i} = (1 - \alpha^i)q_{N-i} + \alpha^i q_N$$

W skrajnym przypadku, jeżeli  $\alpha = 0$ , to zmiana  $\tilde{q}_s = q_s$  dla wszystkich stanów  $s$ , czyli nie się nie zmienia. W drugim skrajnym przypadku, jeżeli  $\alpha = 1$ , to wszystkie  $q_s$  z danej gry przyjmują wartość końcowej nagrody. Pomiędzy tymi skrajnościami modyfikujemy  $q_s$  w zależności od końcowego wyniku gry, przy czym stany z początku gry są w mniejszym stopniu modyfikowane, aniżeli te bezpośrednio poprzedzające koniec gry.

Stosując powyższą regułę do ciągu 2500 obserwacji otrzymujemy zbiór referencyjny, który po służy do uczenia sieci. Po przetrenowaniu sieci i aktualizacji wag, całą procedurę powtarzamy przez z góry ustaloną ilość epok. Poniżej przedstawiam wyniki jakie udało się uzyskać po 8 epokach, czyli bazując na 20 000 obserwacji.

```
X wins: 95 times. Prcnt: 0.95
O wins: 5 times. Prcnt: 0.05
there were 0 ties. Prcnt: 0.0
```

Rysunek 3.11: Wynik trenowania sieci na 20 000 obserwacji w 8 epokach

Jak widać agentowi na 100 gier udało się wygrać z graczem losowym w 95% przypadków. Można uznać, że to jest całkiem niezły wyniki, aczkolwiek w tym wypadku pomogła trochę losowość. Wydaje się, że średni procent wygranych gier plasuje się w okolicach 90% przypadków. Poniżej (rysunek 3.12) przykładowy wynik po 1000 grach.

```
X wins: 55 times. Prcnt: 0.055
O wins: 903 times. Prcnt: 0.903
there were 42 ties. Prcnt: 0.042
```

Rysunek 3.12

A więc teraz agent gra dużo lepiej niż gracz losowy i ma lepsze wyniki niż sieć z I podejścia. Patrząc na wyniki gracza stosującego algorytm z 2 rozdziału (wygrane na poziomie 75%

-85%) można by przypuszczać, że sieć sobie radzi co najmniej nie gorzej. Nie jest to jednak poprawny wniosek. Algorytm z 2 rozdziału wygrywa rzadziej, ale też znacznie rzadziej przegrywa (w około 2%) w stosunku do średnio 5% przegranych w przypadku gracza bazującego na przetrenowanej sieci. Okazuje się, że ta różnica w ilości przegranych gier ma kluczowe znaczenie. Gracz „neuronowy” prawie zawsze przegrywa z graczem stosującym algorytm Q-learning z drugiego rozdziału. Na rysunku 3.13 zaprezentowano wyniki dla 100 rozegranych partii pomiędzy dwoma algorytmami:

```
X wins: 95 times. Prcnt: 0.95
O wins: 5 times. Prcnt: 0.05
there were 0 ties. Prcnt: 0.0
```

Rysunek 3.13: Wynik 100 gier rozegranych pomiędzy graczem stosującym Q-learning (X) i sieć neuronową (O)

Warto jeszcze podkreślić, że algorytm z drugiego rozdziału stał się najbardziej efektywny po dodatkowych „sparingach” z graczem stosującym minimax. W wypadku sieci neuronowej dodatkowe trenowanie na bazie obserwacji uzyskanych podczas gier z graczem stosującym minimax nie przyniosło żadnej poprawy. Algorytm z rozdziału drugiego, przetrenowany dodatkowo na algorytmie minimax, prawie zawsze remisuje z graczem stosującym algorytm minimax, natomiast gracz wykorzystujący sieć neuronową już tylko w okolicach 40%-50% przypadków, co ilustruje (rysunek 3.13) poniższy wynik rozegranych 100 gier.

```
X wins: 58 times. Prcnt: 0.58
O wins: 0 times. Prcnt: 0.0
there were 42 ties. Prcnt: 0.42
```

Rysunek 3.14: Wynik 100 gier rozegranych pomiędzy graczem minimax (X) a graczem stosującym sieć neuronową (O)

Reasumując, sieć nauczyła się skutecznie wygrywać z graczem losowym, ale niekoniecznie dobrze grać w kółko i krzyżyk, stąd pozostaje nieco słodko-gorzki smak po wykorzystaniu sieci neuronowych do gry w kółko i krzyżyk.

Na koniec podjęto jeszcze jedną próbę wytrenowania sieci bazując na zasadzie modyfikacji  $q_s$  jak w I podejściu, ale tym razem nie trenując sieci w locie, po każdym ruchu, ale znowu rozgrywając tyle gier ile potrzeba do uzyskania 2500 obserwacji i trenowanie sieci dopiero na pełnym zbiorze obserwacji. Dokładniej,  $q_s$  czyli oczekiwaną wartość przyszłych nagród dla planszy w stanie  $s$  po ruchu agenta z  $s - 1$  do  $s$ , modyfikowano zgodnie z regułą (zobacz [3] rozdział 6.5):

$$q_s := q_s + \alpha(R_s + \gamma \max_{s+1} q_{s+1} - q_s),$$

gdzie podobnie jak w poprzednich rozdziałach  $\alpha$  to parametr sterujący szybkością uczenia się algorytmu (*learning rate*),  $\gamma$  to parametr dyskontujący przyszłe nagrody,  $R_s$  to nagroda po odpowiedzi systemu na ruch agenta z  $s - 1$  do  $s$ . Niestety, tak jak w poprzednim podejściu, trenując sieci na 20 000 obserwacji (8 epizodów) udało się uzyskać wyniki rzędu 70% wygranych, 20% przegranych i 10% remisów z graczem losowym. A więc wyniki porównywalne z tymi uzyskanymi w podejściu I. Nieznaczna poprawa może wynikać z dodatkowej warstwy ukrytej.

## Rozdział 4

# Podsumowanie

Celem pracy było zaimplementowanie i opisanie kilku wybranych algorytmów grających w grę „kółko i krzyżyk” i ten cel został osiągnięty, aczkolwiek w przypadku algorytmu bazującego na sieciach neuronowych, poziom gry pozostawia wiele do życzenia. W poniższych akapitach opisano po krótce, każdy z trzech algorytmów wraz z refleksjami odnośnie ich skuteczności, możliwych ulepszeń w samym algorytmie lub w sposobie implementacji.

Minimax jest najbardziej klasycznym podejściem w przypadku gier z dwoma graczami o zerowej sumie nagród, czyli gier, w których maksymalizacja wygranej jednego z graczy jest równoważna z minimalizacją wygranej oponenta. Mimo, że kółko i krzyżyk składa się maksymalnie z 9 sekwencji ruchów, a więc drzewo gry ma 9 poziomów, to algorytm wydaje się ciężki i gra zajmuje dosyć długo. Grając z komputerem trudno to odczuć, ale ewidentnie dostrzec podczas trenowania innych algorytmów poprzez grę z graczem grającym minimax. Np. rozegranie na domowym komputerze autora 1000 gier pomiędzy dwoma graczami grającymi losowo zajmuje kilka sekund, a rozegranie 1000 gier pomiędzy graczem grającym losowo z graczem grającym minimax, to już są minuty. Gdyby dodać jeszcze jeden poziom do gry (takie kółko i krzyżyk na planszy 4 razy 4), to oczekiwanie na ruch komputera byłoby nie do zaakceptowania. W takim przypadku trzeba by koniecznie kończyć przeszukiwanie drzewa na jakimś wcześniejszym poziomie, tak jak to robią algorytmy grające w szachy. Nieznaczna poprawa szybkości działania przyniosło w przycinanie  $\alpha - \beta$ . W pojedynczej grze z komputerem nie daje się tego odczuć, ale przy „masowym” zrywaniu 1000 jest to nieznacznie szybciej. Podobno najlepsze algorytmy grające w szachy przeszukują drzewo gry do 12-13 poziomów, więc pewnie poprawiając struktury danych na bardziej optymalne dało by się odciążyć i przyspieszyć tę implementację.

Największym pozytywnym zaskoczeniem był algorytm *Q-table* bazujący na algorytmie *Off-policy Temporal Difference(0)* opisanym w rozdziale 6.5 w [3]. Zaczynając z pustą tabelą wartości  $q_s$  już po 5000 gier mamy tabelę wypełnia się około 600-700 stanami z odpowiadającymi im  $q_s$  i algorytm wygrywa z graczem losowym w ponad 70% przypadków. Trenowanie algorytmu na 20 000 grach skutkuje graczem na zbliżonym do człowieka. W zakresie implementacji kluczową rzeczą było, żeby nie otwierać i nie zamykać pliku z tabelą ze stanami i wartościami  $q_s$  ponieważ jest to czasochłonne i opóźnia działanie programu. Czy to przy jednorazowej grze (człowieka z graczem posługującym się algorytmem Q-table), czy przy trenowaniu algorytmu podczas wielu grach, plik jest otwierany na początku, następnie przez całą sesję przetrzymywany w pamięci jest słownik Q-table, i na końcu jest zapisywany do pliku.

Po sukcesach z algorytmem Q-table, wydawało się, że łatwo uda się wytrenować prostą sieć neuronową, która zamiast posługiwać się danymi tabelarycznymi (zapisywanymi do pliku), skompresuje tę informację w macierzach z wagami sieci. Niestety rzeczywistość nie potwierdziła tych optymistycznych założeń i trenując sieć na różne sposoby nie sposób było oprzeć się wrażeniu dosięgania „szklanego sufitu”, którego nie udało się przebić. Istotnym jest pytanie, co jeszcze można by potencjalnie zrobić, że nauczyć sieć dobrze grać w kółko i krzyżyk. Na myśl przychodzi więcej warstw ukrytych oraz bardziej zaawansowane metody sieciowo-neuronowe typu . Wydaje się jednak, że to idzie w kierunku strzelania z armaty do muchy. Sieć z jedną warstwą ukrytą jest w stanie całkiem dobrze rozpoznawać odręcznie pisane cyfry [1]. Trudno przypuszczać, żeby bądź co bądź banalna, gra w kółko i krzyżyk wymagałaby bardziej zaawansowanych metod. Może to kwestia doboru odpowiednich hiperparametrów? Chociaż nie podjęto formalnych prób przeszukiwania przestrzeni hiperparametrów, to raczej wątpliwym wydaje się być, żeby ich dobór mógł zaowocować prawdziwym przełomem. W internecie można znaleźć kilka ciekawych podejść, jednak większość nie jest satysfakcjonujących. Jednym z takich podejść jest klasyfikowanie ruchów (dobry lub zły) za pomocą algorytmu minimax. Jeżeli gracz grający algorytmem minimax wybrałby ten ruch, to jest to dobry ruch, a w przeciwnym razie uznajemy ruch za zły. Następnie generując  $n$  ruchów podczas gry z graczem losowym, możemy łatwo zbudować zbiór do trenowania sieci. Inne osoby z powodzeniem trenowały sieć korzystając z wcześniej „wytrenowanej” tabeli Q-table. Takie drogi zupełnie mnie nie interesowała, jako że ogólnie wiadomo, że sieć można dobrze wytrenować (dopasować o danych). Znacznie bardziej ciekawym pytaniem jest jak wykorzystać sieć neuronową do nauczania się jak grać w kółko i krzyżyk bez wspierania się już istniejącymi algorytmami; bez żadnej a priori wiedzy, poza samymi regułami gry. Stąd mój nacisk na trenowanie tylko i wyłącznie z graczem losowym, co w rezultacie dało całkiem niezłe wyniki przeciwko takiemu graczowi (ponad 90%). Jednak nie przełożyło się na umiejętność grania w potocznym, ludzkim, sensie.

Pierwotnie zamierzałem zająć się jeszcze jednym podejściem, a mianowicie algorytmem genetycznym. Zasadę działania takiego algorytmu można łatwo zobrazować. Na początku potrzebny jest relatywnie duży zbiór strategii  $\pi_i$  indeksowany jakimś parametrem  $i$ . Następnie, losujemy pary strategii, której grają między sobą w kółko i krzyżyk. Pokonana strategia znika (ginie), a zwycięska może się reprodukować, ale potomstwo  $\pi_i$  mutuje z pewnym prawdopodobieństwem. W efekcie otrzymujemy nowy zbiór strategii zawierający zwycięską strategię  $\pi_i$  oraz jej potomstwo  $\pi_i$ . Sekwencje rywalizacji a następnie reprodukcji z możliwością mutacji powtarzamy wielokrotnie, najlepiej aż do pozostania jednej, ewolucyjnie najlepszej strategii. Jak widać opis wydaje się prosty i zachęcający. Niestety trudności związane z wymyśleniem jak w ogóle charakteryzować strategię gry w kółko i krzyżyk, albo jak zapisać kod DNA danej strategii za pomocą parametrów będących obiektami matematycznymi, przerósł możliwości autora.

## Dodatek A

# Dodatek - opis kodu

W tej części opisano podstawowe informacje dotyczące kodów napisanych w ramach niniejszej pracy oraz modułów (bibliotek) innych autorów, z których korzystano. Nie ma natomiast opisu poszczególnych elementów kodu i ich działania. Sam kod źródłowy, napisany w języku Python w wersji 2.17, stanowi załącznik do niniejszej pracy i można zorientować się za co odpowiadają poszczególne fragmenty czytając komentarze. Całość kodu składa się z kilku plików:

- **ttt.py** - zawiera implementację głównego programu i klas graczy
- **graphics.py** - moduł obsługujący grafikę autorstwa Johna Zelle [7]
- **neural.py** - moduł z implementacją sieci neuronowej w narzędziu *Tensorflow*
- **QTable.txt** - plik tekstowy z wyuczonymi danymi dla algorytmu Q-Table
- **ttt\_model.index**, **ttt\_model.meta**, **ttt\_model.data-000000-of-000001** - dane z wagami sieci neruonowej

### A.1. Główny program

Jądem programu jest funkcja *main()*, która jest odpowiedzialna za przeprowadzenie pojedynczej gry w kółko i krzyżyk. Funkcja przygotowuje planszę i zaznacza graficznie ruchy poszczególnych graczy, chyba że podając odpowiedni parametr wyłączy się grafikę, co jest niezbędne w przypadku rozgrywania tysięcy gier przy uczeniu algorytmów. Funkcja decyduje na podstawie podanych parametrów (domyślnie losowo) o tym, do którego gracza należy pierwszy ruch i wymaga wskazania dwóch graczy, przy czym mogą to być gracze stosujący różne algorytmy lub ludzie (w przypadku wybrania opcji włączonej grafiki). Funkcja *main()* korzysta też z kilku funkcji pomocniczych, które sprawdzają, na przykład, czy po danym ruchu jest remis, czy gra się zakończyła wygraną lub obsługują aktualizację grafiki. Program korzysta z elementarnych modułów *random*, *sys*, *copy* oraz ze znanego modułu *numpy* ułatwiającego obliczenia numeryczne, zwłaszcza w przypadku wielowymiarowych struktur danych, takich jak (n-wymiarowe) macierze.

### A.2. Klasy graczy

Opisana wyżej funkcja *main()* wymaga przekazania jako argumentów dwóch instancji obiektów typu *Player*, czyli obiektów reprezentujących graczy. Ogólnie, klasa typu *Player* ma

tylko jeden atrybut - symbol, który wskazuje czy gracz będzie grał kółkiem czy krzyżykiem. Natomiast każda „właściwy” typ gracza jest definiowany w osobnej podklasie:

1. Człowiek: klasa *playerHuman*
2. Gracz grający losowo: klasa *playerRandom*
3. Gracz grający algorytmem minimax: klasa *playerMinimax*
4. Gracz grający algorytmem Q-Table: klasa *playerQTable*
5. Gracz grający z użyciem sieci neuronowej: klasa *playerNeural*

Każda z tych podklas klasy *Player* ma jedną główną metodę *move()*, która przyjmuje stan gry jako argument i wybiera najlepszy według siebie ruch. W przypadku klasy *playerHuman* metoda *move()* śledzi pozycję kursora i sprawdza gdzie na planszy nastąpiło kliknięcie.

Gdyby była potrzeba dodania kolejnej wersji algorytmu grającego w kółko i krzyżyk, np. algorytmu genetycznego, to poza dodaniem odpowiedniej opcji w GUI gry, „wystarczyłoby” dodać nową podklasę *playerGenetic* ze stosowną metodą *move()*, która przyjmuje głównie 9 elementowy ciąg 0,1 i 2 obrazujący sytuację na planszy i zwraca cyfrę od 1 do 9, która oznacza wybrane pole do ruchu.

### A.3. Grafika

Istnieje wiele modułów pozwalających tworzenie obiektów graficznych w Pythonie. Jako, że gra w kółko i krzyżyk nie jest graficznie wymagająca, to skorzystałem z bodajże najprostszego narzędzia napisanego przez Johna Zelle [7]. Podobno ten moduł został stworzony nie tyle z myślą o poważnych zastosowaniach graficznych, ale z myślą żeby uczyć programowania obiektowego na bazie prostych graficznych obiektów typu punkt, linia, koło itp. Ogólnodostępny kod źródłowy tego modułu graficznego można znaleźć na stronie [7].

### A.4. Tensorflow

Pierwotnym zamysłem była implementacja sieci neuronowej od początku do końca za pomocą podstawowych modułów python’a. I mimo pewnych sukcesów na tym polu, ostatecznie zwyciężył pragmatyzm i sieci neuronowe opisana w rozdziale 3 powstały z wykorzystaniem bibliotek *Tensorflow*. Jest to narzędzie stworzone przez potentata Google, i służy ogólnie do budowania grafów, w których węzłach mają miejsce obliczenia (w tym różnego typu optymalizacje), natomiast krawędzie ilustrują przepływ danych pomiędzy obliczeniami. Praca z tym narzędzie polega na zbudowaniu lub wczytaniu grafu, następnie zapewnienie niezbędnych danych wejściowych, przeprowadzeniu obliczeń po kolei zgodnie ze strukturą grafu i odczytaniu wyniku. Więcej o module *Tensorflow* oraz o dostępnym w python’ie API *Tensorflow* do można się dowiedzieć na oficjalnej stronie [8].

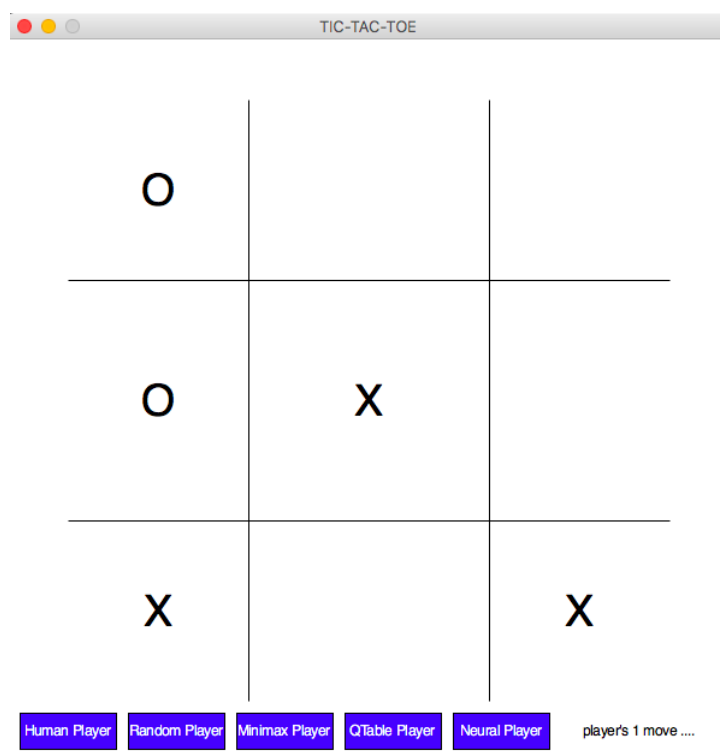


## Dodatek B

# Dodatek - opis aplikacji

Niezależnie od statystycznych wyników, jakie poszczególne algorytmy osiągają grając w kółko i krzyżyk, ciekawie jest samemu przekonać się o skuteczności algorytmu grając z nim „osobiście”. Taka ciekawość była główną motywacją do przygotowania w ramach niniejszej pracy bardzo prostej aplikacji gry w kółko o krzyżyk, która została opisana w tym dodatku.

Aplikacja ma tylko jeden ekran (zobacz rysunek B.1), przedstawiający planszę do gry w kółko i krzyżyk. Po uruchomieniu aplikacji należy za pomocą niebieskich przycisków umieszczonych na dole planszy wybrać przeciwnika. Do wyboru można grać z drugą osobą (człowiekiem), z graczem grającym w sposób losowy, graczem stosującym algorytm minimax, stosującym algorytm Q-Table lub graczem wykorzystującym sieć neuronową z 3 rodziału. Dodatkowo w prawym dolnym rogu aplikacja informuje o tym, którego gracza jest obecnie ruch.

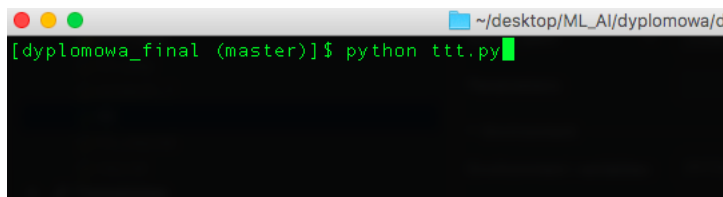


Rysunek B.1: Plansza do gry w kółko i krzyżyk

W celu uruchomienia aplikacji trzeba mieć na komputerze zainstalowanego Pythona 2. Dodatkowo należy w jednym wybranym katalogu umieścić następujące pliki:

1. ttt.py
2. graphics.py
3. neural.py
4. Qtable.txt
5. ttt\_model.meta
6. ttt\_model.index
7. ttt\_model.data-00000-of-00001

Pliki 4 - 7 przechowują parametry wytrenowanych algorytmów Qtable i sieci neuronowej. Jeżeli tych plików zabraknie, to aplikacja będzie działała, ale gracze stosujący Q-Table lub sieć neruronową będą grali tak jak gracz losowy. Samo uruchomienie aplikacji odbywa się przez uruchomienie (otworzenie) pliku ttt.py za pomocą programu Python. Jeżeli dysponujemy rozbudowanym środowiskiem programistycznym (jak np. PyCharm) to można otworzyć otw-  
przyć plik korzystając ze stosownego okna dialogowego. W przeciwnym razie można posłużyć się wierszem poleceń tak jak na rysunku B.2.



Rysunek B.2: Uruchomienie aplikacji z wiersza poleceń

Moduł ttt.py może również służyć do trenowania algorytmów QTable i sieci neuronowej, ale do tego nie ma GUI i konieczna jest zmiana wybranych parametrów w kodzie.

# Bibliografia

- [1] Michael Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015
- [2] James Gareth, *Introduction to Statistical Learning*, Springer, 2013
- [3] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 2018
- [4] Prof. Patrick Henry Winston, *MIT course number 6.043, Artificial Intelligence*, <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/index.htm>
- [5] Michael Heining, *Dynamic Learning: A case study on Tic-Tac-Toe*, <https://www-m15.ma.tum.de/foswiki/pub/M15/Allgemeines/PublicationsEN/MasterThesisHeining.pdf>
- [6] Matthew O. Jackson, *A Brief Introduction to the Basics of Game Theory*, <https://ethz.ch/content/dam/ethz/special-interest/gess/chair-of-sociology-dam/documents/education/spieltheorie/literatur/Einf%c3%bchrung/Jackson%20Basics%20of%20Game%20Theory%20SSRN-id1968579.pdf>
- [7] John Zelle, *Pyhton Programming: An introduction to Computer Science*, <https://mcsp.wartburg.edu/zelle/python/>
- [8] TensorFlow, <https://www.tensorflow.org/>