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 α - β. Następnie wykorzystano algorytm sieci neuronowych oraz algorytm genetyczny. Praca zawiera, opis porównanie działania algorytmów oraz aplikację do gry z komputerem wykorzystującym wybrany algorytm.

Słowa kluczowe

mimimax,teoria gier, uczenie maszynowe, sieci neuronowe, reinformcent learning

Spis treści

Wprowadzenie

W trakcie zajęć na studiach podyplomowych wiele uwagi poświęcone było budowaniu modeli predykcyjnych. Pracując z programem R i RStudio tworzyliśmy modele i szacowaliśmy 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. Wykorzystywaliśmy 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 budowaliśmy na zajęciach - miały skuteczność mierzoną średnim odchyleniem kwadratowy rzędy $70\% - 85\%$ Trudno jednoznacznie powiedzieć na ile satysfakcjonujące są takie wyniki. Można powiedzieć, że model „działa", można też w ten sposób porównywać modele, ale trudno powiedzieć na ile „uchwycono", prawa natury rządzące poziomem przychodu.

Przymierzając się do napisania tej pracy chciałem skoncentrować się bardziej na algorytmach uczących komputery, aniżeli na zbieraniu, czyszczeniu i doborze danych, czyli na tzw. „feature engineering"'. Chcę podkreślić że w żadnym wypadku nie pomniejszam wagi ani znaczenia tej sztuki. Po prostu bardziej interesuje mnie sam proces uczenia się komputerów, aniżeli proces dostarczania odpowiednich danych, żeby to uczenie było skuteczne. Szukałem więc problemów, w których dane byłyby niejako naturalnie dostępne i tak moja uwaga skupiła 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. Rozważałem gry karciane, ale wolałem skupić się na grze w pełni deterministycznej, w której losowość nie miałaby wpływu na wynik. Dzięki temu wiadomo byłoby 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. Dla mnie idealne pole do eksperymentowania z uczeniem komputera.

W pierwszym rozdziale opisuję 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 Inteligenceńa MIT [4], to chciałem, żeby pojawiło się t tej pracy.

W kolejnych dwóch rozdziałach prezentuję 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 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 koncentruje się na algorytmach, i implementacja tych algorytmów jest również częścią pracy, to chciałem wykorzystać uniwersalny obiektowy język programowania. Zdecydowałem się na Python 2.7, który w moim odczuciu łatwiej umożliwia tworzenie i obsługę klas obiektów od R. Do pracy dołączam:

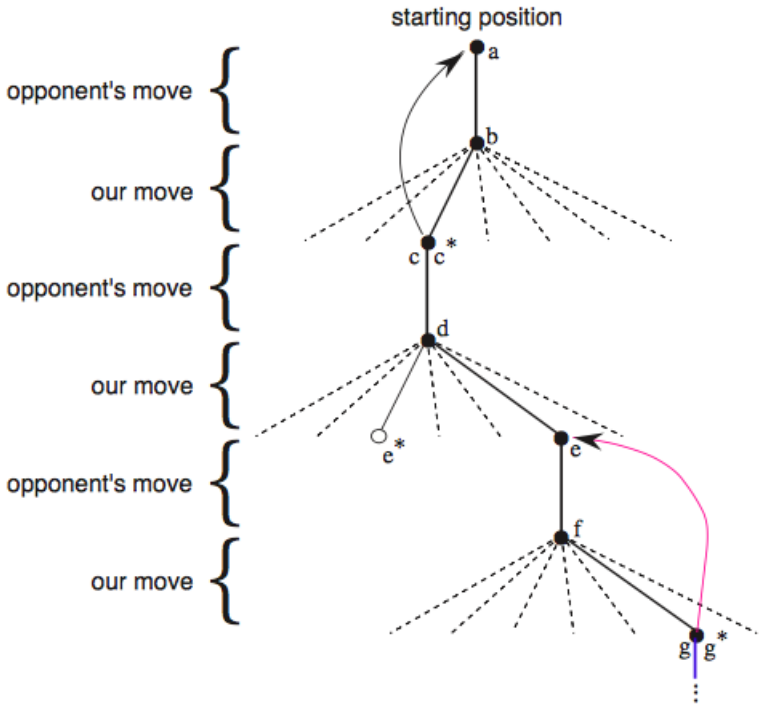
1. **DODATEK A** - Kod z implementacją opisanych algorytmów.
2. **DODATEK B** - aplikacja do gry w kółko i krzyżyk

Rozdział 1

Minimax

1.1. Opis algorytmu

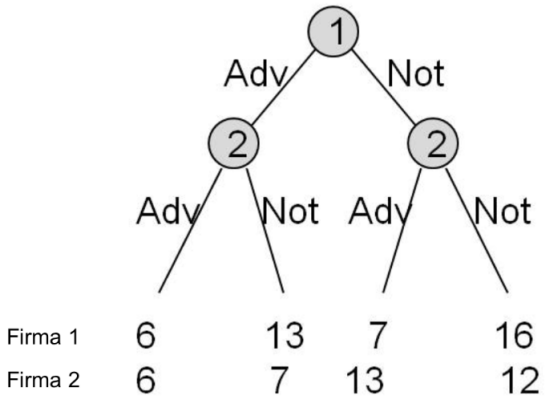
W takich grach jak kółko i krzyżyk, szachy lub warcaby możemy mówić o przestrzeni stanów gry. Każdy stan gry jest reprezentowany przez położenie pionków albo rozmieszczenie kółek i krzyżyków na kwadratowej planszy z 9 polami. Za każdym ruchem, jednego z dwóch graczy, gra przechodzi do innego stanu. Wszystkie 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 zaczerpniętym z książki [3]



Przypuśćmy dalej, że z każdym stanem końcowym gry (liściem) może 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. Możemy 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 liście są różne dla rywalizujących ze sobą graczy. W naszym 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 wybranie takiego ruchu, który „otworzy" nam ścieżkę do liści z największą możliwą ilością punktów. Zobaczymy na poniższym przykładzie wziętym ze strony [5], 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 mamy cztery możliwe sytuacje (cztery różne rozgrywki) i odpowiadające im punkty (można je interpretować np. jako zysk).



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 $((28 - 16)/2)$.

Załóżmy, że jesteśmy prezesem firmy 1 i chcemy podjąć decyzję o tym, czy inwestować w reklamę czy też nie. Załóżmy 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 usiłuję 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 słuszna 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 stablinym. Teoretycznie prezesi firm mogli 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. Załóżmy, że reprezentujmy 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 liśc (tutaj wiem jakie są wygrane) biorąc pod uwagę dla każdego pokolenia czy jest nasz ruch, czyli ruch, czy też ruch naszego oponenta, który ma na celu minimalizowanie naszej wygranej. Przedstawiając powyższe rozumowanie w punktach dochodzimy do algorytmu minimax, który dla każdego węzła zwraca wygraną wartość gracza .

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(wartość, 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(wartość, wartość algorytmu minimax na poddrzewie, którego korzeniem jest rozpatrywany ruch dla nas
   (d) zwróć wartość

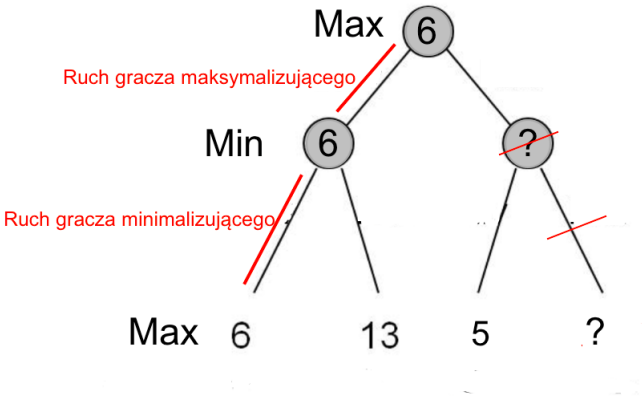
Okazuje się, że jeżeli gracze stosują się do powyższego algorytmu, to znaczy wybierają ruchy, które zapewnią im największą wygraną wyliczoną według powyższego algorytmu, to

takie strategie będą w równowadze Nash'a. Mówiąc obrazowo są to najlepsza strategie, w tym sensie, że żadnemu z graczy nie opłaca się ich zmieniać. Jeżeli wiem, że mój oponent będzie grał zgodnie z algorytmem minimax, to moją najlepszą strategią będzie robić to samo. Jeżeli mój oponent będzie grał inną strategią, to być może istnieje lepsza odpowiedź na jego strategie niż minimax. Jeżeli ja gram według algorytmu minimax, a mój oponent nie, to gra strategią suboptymalną.

1.2. Przycinanie α - β

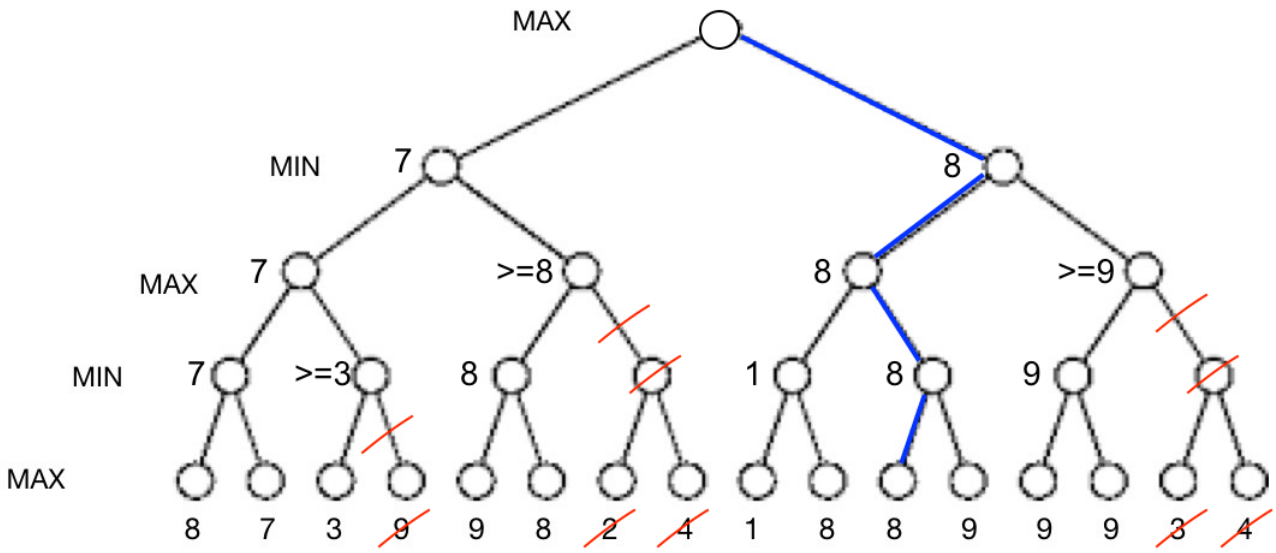
Podstawowym problemem w zastosowaniach algorytmu minimax jest ilość niezbędnych operacji przy dużych drzewach reprezentujących gry. Dla gry w kółko i krzyżyk, górnym ograniczeniem ilości możliwych gier (liści na drzewie gry) jest $9! = 362880$. W rzeczywistości ilość możliwych partii ogranicza się do 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 α - β. Polega to na tym, że jeżeli wiemy, że sprawdzenie i porównanie wartości dla kolejnego ruch/węzła na drzewie gry nie zmieni wyboru gracza , to pomijamy takie obliczenie. Najlepiej zobaczyć to na prostym przykładzie.



Spójrzmy raz jeszcze na poprzedni przykład, ale nieco zmieniony. Jesteśmy graczem rozpoczynającym grę i chcemy wybrać optymalny ruch za pomocą algorytmu minimax. Minimax jest algorytmem rekurencyjnym; żeby policzyć wartości wygranej dla każdego z możliwych ruchów, trzeba "wywołać"minimax dla gracza minimalizującego na odpowiednich poddrzewach. To z kolei sprowadza się do porównywania wszystkich wartości liści, czyli ostatecznej wygranej w grze i następnie sukcesywnego podążania w górę drzewa. Idąc od lewego dolnego liścia widzimy, że gracz minimalizujący będzie miał wygraną 6, bo na pewno nie wybierze liścia z wartością 13 (minimalizuje naszą wygraną). Czyli jeżeli wybierzemy ruch "w lewo", to nasza wygrana, to będzie 6. Jeżeli zaczniemy kalkulować wygraną w przypadku ruch "w prawo", to widzimy, że nasz oponent wybierze 5 albo jeszcze mniej gdyby w skrajnym prawym liściu była mniejsza wartość. Ale ten fakt jest dla nas, dla gracza maksymalizującego, już bez żadnego znaczenia. Decydując się na ruch "w lewo"wygramy 6, a decydując się na ruch "w

prawo"wygramy co najwyżej 5. Możemy zatem w ogóle nie sprawdzać jaka jest wartość prawego skrajnego liścia i nie wyliczać wartości wygranej gdybyśmy wykonali ruch "w prawo". Tym sposobem zaoszczędziliśmy trochę obliczeń.



Załóżmy, że mamy do czynienia z takim drzewem gry jak na rysunku. Ostatni wiersz przedstawia możliwe wygrane gracza maksymalizującego, rozpoczynającego grę. Na rysunku zaznaczono węzły i całe podrzewa, których gracz maksymalizujący nie potrzebuje rozpatrywać. Po lewej stronie węzłów zapisana jest maksymalna możliwa wygrana gracza maksymalizującego o ile gra znajdzie się w tym weźle (czyli wartość jaką zwraca algorytm minimax). Pogrubioną linią widać optymalną ścieżkę gry, która kończy się zdobyciem 8 punktów. Dla zwięzłości pracy nie opisuję całego toku rozumowania. 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 ustawiamy parametry α i β na $+\infty$ i $-\infty$ odpowiednio, i interpretujemy je jako najgorszy możliwy wynik gracza minimalizującego $(+\infty)$ i najgorszy możliwy wynik gracza maksymalizującego $(-\infty)$. Przechodząc rekurencyjnie przez drzewo sprawdzamy aktualizujemy wartości α i β i jeżeli dla danego poddrzewa nie możemy poprawić wyniku α lub β (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 α - β pomoże 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 zaszyty parametr, który każe przerwać przeszukiwanie drzewa na danym poziomie lub po przekroczeniu określonej ilości czasu. Następnie ze wszystkich możliwych ruchów (liści z przeszukanego poddrzewa do momentu zastopowania algorytmu) wybierany jest ten dla którego, funkcja wartości przyjmuje maksymalną wartość. 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

Szczegóły dotyczące implemetacji, jak również kod można znaleźć w **DODATEK B**. W tej sekcji chciałbym ograniczyć się do kilku spostrzeżeń związanych z implementacją algorytmu minimax z przycinaniem $\alpha - \beta$.

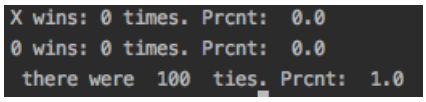
Liczyłem na to, że zastosowanie przycinania zmniejszy czas wykonania algorytmu na początku gry, w pierwszych dwóch ruchach, kiedy konieczne jest przeszukanie największych drzew. Od trzeciego ruchu, wykonanie algorytmu jest rzędu 0.1 sekundy, więc optymalizacje przestają mieć znaczenie. Niestety zastosowanie przycinania $\alpha - \beta$ nie poprawiło w widoczny sposób szybkości działania algorytmu. Żeby ograniczyć czas oczekiwania na odpowiedź komputera dodałem instrukcję, powodującą, że jeżeli algorytm zaczyna grę, to zawsze wybiera środkowe pole.

Nie udało mi 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 mnie do napisania reguły jak grać w kółko i krzyżyk. Wówczas minimax przestaje być potrzebny i można się ograniczyć do do policzenia funkcji wartości dla możliwych ruchów i wybrania tego z największą wartością. Niestety nie udało mi się znaleźć „złotego środka" i w końcowej implementacji nie korzystam z funkcji wartości.

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

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

Gracz stosujący minimax nie przegrał ani razu, za to zremisował z graczem losowym 13 razy na 100 gier.



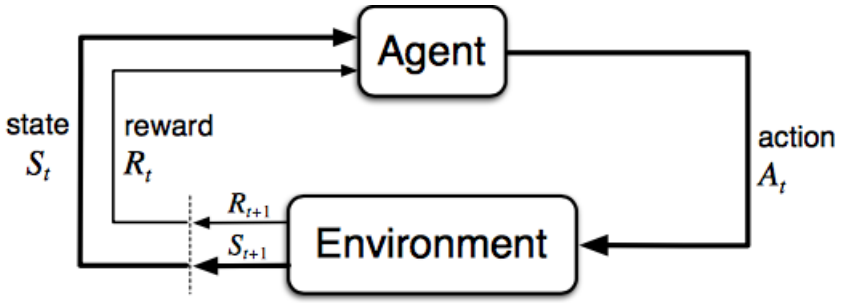
Dwóch graczy stosujących minimax zawsze będą ze sobą remisować.

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 skoncentruję się na innym modelu, a mianowicie na procesach decyzyjnych Markowa. Przed formalnymi definicjami kilka chciałbym przedstawić heurystyczny opis modelu, posiłkując się rysunkiem wziętym z [3].



Załóżmy, że mamy agenta (gracza w naszym przypadku), który jest w interakcji z pewnym systemem (*environment* na rysunku). 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 odpowie 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. Podobnie przy przejściu ze stanu s do s' system może losowo zwracać różne nagrody r. Natomiast ta losowość musi zależeć tylko i wyłącznie od pary (s, a) i nie może w żaden sposób zależeć od poprzednich stanów w jakich system się znajdował. Na tym polega istota „markowości", że historia nie ma znaczenia. W tym miejscu chciałbym poczynić jeszcze jedno założenie, że zbiór możliwych stanów systemu jest skończony. To założenie nie jest konieczne w ogólnej teorii, ale pozwala na daleko idące uproszczenia, z których skrzętnie korzystam w tej, jakby nie było, ilustratywnej pracy. Interakcja agenta z systemem przez kilka kolejnych kroków począwszy od $t = 0$ można przedstawić za pomocą takiej trajektorii:

$$s_o, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, ...$$

Formalizując powyższe rozważmy czwórkę postaci $< \mathcal{S}, \mathcal{A}, \mathcal{R}, p >$ 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 systemy dostępne są różne akcje. Dla uproszczenie będę jednak pomijali ten niuans notacyjny i w dalszej części mówił o możliwych akcjach $\mathcal{A}$ pamiętając, że ten zbiór może być ograniczony konkretnym stanem.

3. $\mathcal{R} \subseteq \mathbf{R}$ jest zbiorem wartości nagród, jakie agent może otrzymać

4. $p : \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times \mathcal{S} \to [0, 1]$ 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 staram 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 osiągania średnio wysokich nagród, a nie wygrywanie.

Zanim przejdę do opisu algorytmu potrzebne będzie jeszcze kilka pojęć. **Strategią** nazwiemy dowolne mapowanie

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

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 w matematyce finansowej) a po drugie, interakcja agenta z system może nie być ograniczona ilością ruchów, a wtedy suma przyszłych nagród niezależnie od wyboru strategii π 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

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

Dla uproszczenia zapisu wprowadźmy oznaczenie na sumę przyszłych zdyskontowanych nagród:

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

Wartością stanu s przy strategii π jest z definicji

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

Jest to wartość oczekiwana sumy dyskontowanych przyszłych nagród pod warunkiem, że stosujemy strategię π począwszy od czasu t. w którym to czasie system znajdował się w

stanie s. Podobnie zdefiniujmy jeszcze funkcję określającą wartość akcji a podjętą w stanie s jako zdyskontowaną wartość przyszłych nagród przy założeniu, że stosujemy strategię π, a w stanie s wybierzemy akurat akcję a (strategia π zadaje rozkład prawdopodobieństwa na $\mathcal{A}$, i w tej definicji zakładamy, że akurat wylosowano 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

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

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

$$v_{\pi_*}(s) = \max_\pi v_\pi(s)$$

$$q_{\pi_*}(a,s) = \max_\pi q_\pi(a,s)$$

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

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

Rozsupłanie powyższej „rekurencji" i znalezienie optymalnej strategii można sprowadzić do rozwiązania $|\mathcal{S}|$ równań liniowych i policzenia wartości $v_\pi(s)$ dla każdego stanu s dla arbitralnie wybranej deterministycznej (nie losowej)strategii π. Następnie można „poprawiać" π 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 poprawianie strategii $\pi \to \pi' \to \pi'' \to ...$ prowadzi do optymalnej strategii π_*. Takie podejście w [3] nazywane jest programowaniem dynamicznym (z ang. *dynamic programming*). Jest to skuteczne podejście jeżeli mamy pełną wiedzę o funkcji p Najczęściej tak jest, jeżeli modeluje się jakieś rzeczywisty system za pomocą procesu decyzyjnego Markowa.

Nas jednak bardziej interesuje sytuacja, w której nie znamy funkcji p i będziemy chcieli stworzyć algorytm, które znajduje optymalną strategię doświadczalnie, czyli algorytm, który będzie uczył się systemu, bez uprzednich założeń odnośnie funkcji p. Najczęściej zaczyna się od losowej strategii i z każdym cyklem interakcji z systemem (albo po 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 π w wielu epizodach iterakcji agenta z system, 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 możemy w każdy kroku, po każdej interakcji agenta z systemem, poprawiać stosowaną przez niego strategię π zgodnie ze wzorem

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

W tym wypadku nie potrzebujemy 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 α jest hiperparametrem algorytmu, często nazywany z angielskiego *learning rate*.

2.2. Opis algorytmu

Strategię zdefiniowaliśmy jako mapowanie, które każdemu stanowi $s \in \mathcal{S}$ przypisuje rozkład prawdopodobieństwa $\mathbf{P}_s$ na przestrzeni stanów $\mathcal{A}$. Teraz jednak chcemy przejść od ogólnej teorii do konkretnego algorytmu grającego w kółko i krzyżyk i w tym celu dokonamy kilku uproszczeń. W szczególności ograniczmy się do strategii nie losowych, tzn, do mapowania

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

Jest to szczególny przypadek opisanej w poprzedniej sekcji teorii, gdzie rozważamy wyłącznie atomowe rozkłady prawdopodobieństwa na $\mathcal{A}$. Mając danych stan $s \in \mathcal{S}$ chcemy znać 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 nagród wartość $q(a, s)$. Teraz chcielibyśmy włożyć te uproszczenia do (2.1) i na tej podstawie wypracować algorytm nadający się do implementacji. Na początek możemy położyć dla każdego stanu $s \in \mathcal{S}$ i dla każdej akcji a, $q(a, s) = 0$. Dalej załóżmy, że wykonujemy pierwszy cykl interakcji z systemem. W przypadku gry w kółko i krzyżyk nasz 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 gracz wybiera je losowo. W każdym razie gracz wybiera $\tilde{a}$, które maksymalizuje wszytskie $q(a, s)$. 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:

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

Możemy 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, albo poprawianie wartości $q(a, s)$ po każdym ruchu, zapewni zbieżność do rzeczywistych wartości $q(a, s)$. Następnie stosując (2.1) otrzymujemy optymalną startegię gry w kółko o krzyżyk.

2.3. Implementacja

Z poprzedniej sekcji wynika, że powinniśmy zainicjować dla wszystkich możliwych stanów gry s i możliwych ruchów a wartości $q(a, s) := 0$ i następnie aktualizować. Wypisanie do listy lub tabeli wszystkich możliwych kombinacji (a, s) dla gry w kółko i krzyżyk jest oczywiście możliwe, aczkolwiek trochę kłopotliwe. Zauważmy również, 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 ustalenia wartości nagród $\mathcal{R}$. Wartości q_s będą w procesie „uczenia" algorytmu dążyć do oczekiwanej nagrody agenta, pod warunkiem, że swoim ruchem doprowadził do stanu s. Wybór jest poniekąd arbitralny, ale warto zauważyć, że w grze w „kółko i krzyżykżemis 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 grając z minimax 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. Poniżej ilustruję kilka pozycji z tego pliku, już po przetrenowaniu algorytmu (około 20 000 gier rozegranych z graczem losowym) z omówieniem jego 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
```

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 rząd planszy idąc od lewej do prawej. Ostatnie 3 cyfry, opisują idąc od lewej do prawej dolny rząd planszy. Cyfry 1 oznaczają ruchy systemu, czyli w tym wypadku człowieka, 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 prze pierwsze 9 cyfr.

Rozważmy pierwszy wiersz z powyżej ilustracji. Dla ustalenia uwagi załóżmy, że system gra „kółkiem", a agent „krzyżykiem". Ciąg $2, 0, 1, 2, 1, 2, 2, 1, 1.0$ Przedstawia od następujący stan gry po ruchu agenta.

X		O
X	O	O
X	O	X

To jest sytuacja w której agent wygrywa, stąd wartość q dla tego stanu jest równa 1. Kolejny wpis w tabeli, to ciąg $1, 2, 2, 2, 1, 1, 0, 1, 2, 0.75$. Pierwsze 9 cyfr przedstawiam następujący stan gry:

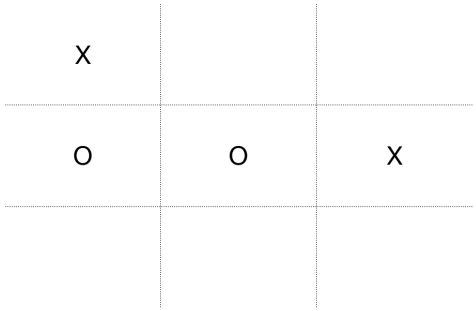
O	X	X
X	O	O
	O	X

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.

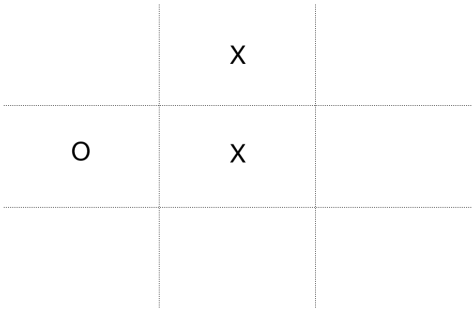
Kolejny przykład, to ciąg $2, 2, 1, 2, 2, 1, 1, 0, 0, -1$, jak łatwo zauważyć przedstawiam sytuację, w której ruch należy do gracza systemowego (kółko lub cyfra 1) i gracz ten może z łatwością wygryać uzyskując trzy „kółka" na prawej skrajnej kolumni. Stąd wartość q w tym przypadku jest równa -1.

X	X	O
X	X	O
O		

Czwarty przykład odpowiadający ciągowi $2, 0, 0, 1, 2, 2, 0, 0, 0, 0.418681257165$ 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".



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.



Mając wszystkie niezbędne elementy możemy przejść do bezpośredniego zdefiniowania algorytmu:

Zaczynając od pustego zbioru wartości q_s, po 20 000 rozegranych grach z graczem losowym mamy zapisanych

1. Inicjuj pusty zbiór z wartościami q_s
2. Ustaw parametry α i γ z przedziału $[0, 1]$
3. Graj n razy z systemem w kółko i krzyzyk:
   (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 takiego elementu w zbiorze 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 $q_{s-1} = q_s + \alpha(R(\tilde{s}) + \gamma q_{\tilde{s}})$

iv. Zapisz nową wartość q_{s-1} w zbiorze. Jeżeli wartość już istniała w zbiorze, to nadpisz.

Popatrzmy teraz na wyniki uczenia. Zaczynam od pustego zbioru q_s i rozgrywam po 100 partii z graczem losowym i z graczem stosującym minmax. Dodatkowo po każdej grze „opróżniamźbiór q_s tak żeby algorytm na razie niczego nie uczył się. Robię to po to, żeby mieć punkt odniesienia przy sprawdzaniu na ile dobrze algorytm nauczył 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 poniżej:

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

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

Natomiast 100 gier rozegranych z graczem minimax dało 10 remisów i 90 przegranych agenta:

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

Po przeuczeniu algorytmu na 20 000 grach z graczem losowym, zbiór q_s ma 775 rekordów (tyle różnych stanów nasz agent napotkał), natomiast wyniki dla 100 gier teraz wyglądają już znacznie lepiej:

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

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.

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

Nie ma natomiast prawie żadnej poprawy jeżeli chodzi o liczbę remisów (bo wygrać się nie da) z systemem grającym strategią minimax. Nasuwa się oczywiście pytanie, czy przetrenowanie algorytmu na grach z systemem grającym strategią minimax poprawiłoby tą statystykę?. Poniżej wyniki algorytmu po dodatkowym 10 000 grach z systemem grającym strategią minimax:

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

A więc trenowanie algorytmu na systemie grającym strategią minimax odniosło skutki. Warto jeszcze sprawdzić, czy po tym trenowania algorytm nie odnotuje gorszych wyników grając z graczem losowym:

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

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 dodam jeszcze, 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.

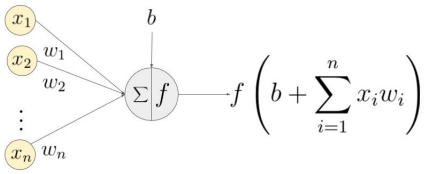
Rozdział 3

Sieć neuronowa grająca w kółko i krzyżyk

3.1. Sieci neuronowe

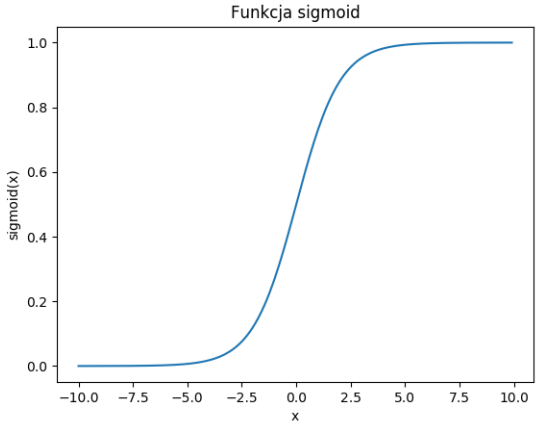
Metoda opisana w poprzednim rozdziale ma jedną zasadniczą wadę. Dla gier o znacznie większej liczbie możliwych kombinacji plansz, zbiory z wartościami q_s stają się ogromne. O ile jeszcze samo zapisanie takiego zbioru, zważywszy na szybki rozwój elektronicznych nośników danych, mógłby być możliwy, to już przeszukiwanie przed każdym ruchem zbioru w celu wyłonienie 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 $s \to q_s$

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

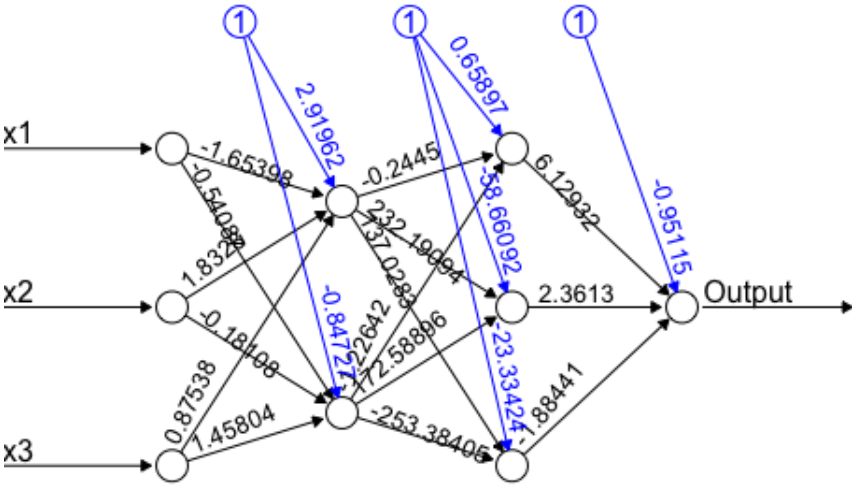


Funkcje aktywacji może być w zasadzie dowolna. Ze względu łatwość „uczeniaśieci neuronowej, o czym za chwilę, warto stosować funkcje różniczkowalne (ewentualnie różniczkowalne poza kilkoma punktmai) i monotoniczne. Najczęściej stosuje się takie funkcje jak sigmoid, tanges hiporbeloczny, relu, czy softplus. Poniżej definicja i wykres funkcji sigmoid - najbardziej klasycznego przykładu funkcji aktywacji.

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



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 służy 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 powyżej przedstawia przykładową sieć, która ma trzy wartości wejściowe, a następnie warstwę 2 neuronów, warstwę 3 neuronów i ostatnią warstwę wyjściową składającą się z jednego neuronu. Wartości na strzałką pokazuję 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 $x_1, ..., x_n$ oraz zbiór wszystkich wag $\mathbf{W}$ i

zbiór wszystkich wyrazów stałych $\mathbf{b}$, która zwraca wektor wartości $y_1, ..., y_m$, którego wymiar jest równy ilości neuronów w ostatniej warstwie.

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

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

$$(x_1, ..., x_n) \overset{\Phi}{\to} (y_1, ..., y_m)$$

Skoro chcemy przybliżać funkcję Φ siecią neuronową $\mathcal{N}$, to potrzebna jest jakaś miara skuteczności takiego przybliżenia. Poniższe zarys dotyczący tego jak trenować sieci neuronowa zaczerpnięty został z [1]. Załóżmy, że mam k-elementowy zbiór danych $\mathbf{X}$ i dla każdego $x_1, ..., x_n \in \mathbf{X}$ obserwujemy $\Phi(x_1, ..., x_n)$ oraz $\mathcal{N}(x_1, ..., x_n, \mathbf{W}, \mathbf{b})$. Jak w tym przypadku określić skuteczność przybliżania Φ 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 $\mathbf{b}$.

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

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 Φ sprowadza się do znalezienia minimum funkcji $C(\mathbf{W}, \mathbf{b}) : \mathbf{R^d} \to \mathbf{R}$, dla pewnej liczby naturalnej d. Jeżeli liczba d jest duża, to analityczne szukanie punktów, w których zerują się wszystkie pochodne cząstkowe byłoby niewykonalne. W praktyce przyjmuje się jednak inne podejście. Dla uproszczenia zapisu przyjmijmy, żę $v := (\mathbf{W}, \mathbf{b}$. Czyli funkcja kosztu C będzie funkcją zmiennej $v \in \mathbf{R^d}$. Z rachunku różniczkowego wiadomo, że:

$$\Delta C = \nabla C \cdot \Delta v \tag{3.2}$$

W powyższym równaniu symbol $(\cdot)$ oznacza iloczyn skalarny wektorów. Natomiast ∇C to jest gradient funkcji C, czyli $\nabla C = (\frac{\delta C}{\delta v_1}, ..., \frac{\delta C}{\delta v_d})$. Zwróćmy uwagę, że ΔC jest zwykłym skalarem. 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 ΔC, 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 η, często nazywana w angielskojęzycznej literaturze *learning rate*. Jeżeli η 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 η może skutkować koniecznością wykonania bardzo wielu kroków w celu odnalezienia minimum. Odmianą tej metody jest „*stochastic gradient descent*" (dalej SGD), która dodatkowo różni się tym, 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 podzbioru, obliczać ∇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 ∇C dla funkcji kosztu, to mamy przepis na to jak zmieniać wagi $\mathbf{W}$ i wartosci stałe $\mathbf{b}$, czyli parametry sieci $\mathcal{N}$, w sposób, który gwarantuje nam zmniejszanie nam 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 trenowania sieci. W wielkim skrócie, algorytm propagacji wstecznej sprawdza się do wliczenia wartości aktywacji na każdym neuronie, a następnie - idąc od ostatniej warstwy wyjściowej aż d pierwszej warstwy (stąd zapewne nazwa propagacji wstecznej) obliczenia błędów sieci i modyfikowanie wag na każdym poziomie. Dokładny opis algorytm, 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 zostanie pominięty.

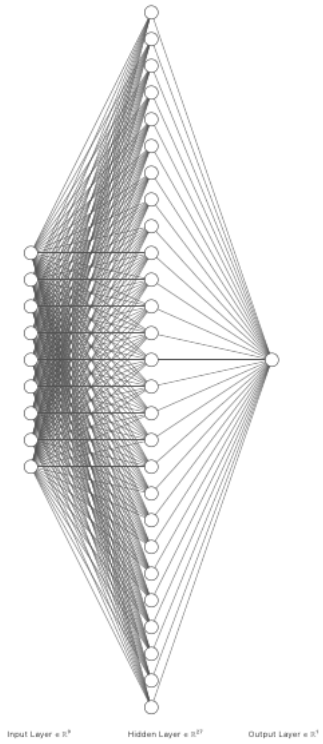
3.2. Podejście I - nieudane

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 niezmienionych. W rezultacie mamy taki algorytm:

1. Inicjuj sieć neuronową, która przyjmuje na wejściu 9 wartości i zwraca na wyjściu 1 wartość - q_s
2. Ustaw parametry α i γ z przedziału $[0, 1]$
3. Graj n razy z systemem w kółko i krzyzyk:
   (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.

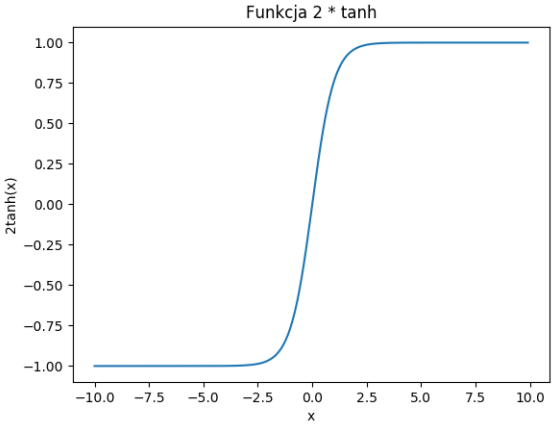
Warstwa wyjściowa, to jedna pozycja zwracającą q_s dla stanu reprezentowanego przez dane wejściowe. Przy czym podobnie jak w poprzednim rozdziale, dane wejściowe reprezentują stan gry po ruchu agenta. Ilość neuronów w warstwie ukrytej wybrałem poniekąd arbitralnie na 27. Poniekąd dlatego, że dla symetrii chciałem, żeby to była wielokrotność ilości danych wejściowych.



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 możliwe wartości przyszłych nagród. W poprzednim rozdziale, przyjąłem 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 hiberboliczny, którego zbiór wartości to właśnie przedział $[-1, 1]$. Problem jednak jest taki, że jeżeli funkcja aktywacji „wypłaszczaśię 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 nie aktualizuje wag $\mathbf{W}, \mathbf{b}$. Chcąc uniknąć takiego efektu „saturacji", za funkcję aktywacji przyjąłem $f(x) = 2\tanh(x)$. Dla porządku, podaję jak jak policzyć funkcję tangens hiperboliczny i jak wygląda jej wykres.



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

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 grać zdecydowanie gorzej od człowieka. Poniżej 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
```

Eksperymentowałem z większą liczbą neuronów w warstwie ukrytej, jak również z dwiema warstwami ukrytymi, oraz z innymi funkcjami aktywacji, ale zawsze z podobnym, niezadawalającym efektem. Najlepsze wyniki jakie mi się udało uzyskać, to około 75% wygranych gier z graczem losowym. Wyniki takie uzyskałem używając funkcji aktywacji $softplus$, która definiuje się jako $f(x) = ln(1 + e^x)$ Ta funkcja aktywacji przyjmuje wartości z przedziału $[0, +\infty]$ (est 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 wynik rozegranych 100 gier z graczem losowym po przetrenowaniu na 20 000 gier.

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

Jak widać, sieć gra statystycznie nieco lepiej od gracza losowego, czyli algorytm „czegośśię nauczył, ale przy tak prostej grze jak kółko i krzyżyk można by się spodziewać znacznie lepszych rezutlatów. Dodam jeszcze, 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? Nie jestem oczywiście odpowiedzieć na to pytanie z całą pewnością, ale stawiam 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, spróbujemy zastosować takie podejście. Pomysł, na którym się wzoruję zaczerpnąłem z pracy magisterskiej złożonej na uniwersytecie technicznym w Monachium [6].

Chcąc mieć pewność, że sieć nie jest za mała, zmieniłem jej topologię dodając jedną warstwę ukrytą. Ostatecznie sieć ma strukturę typu 9-54-27-1, czyli zwiera dwie warstwy ukryte po 54 i 27 neuronów odpowiednio. Za funkcję aktywacji przyjmuję tangens hiperboliczny, który przyjmuje wartości z przedziału $[-1, 1$. Nagrody, czy odpowiedź systemu, ustalam z uwaga, żeby być daleko od ekstremalnych wartości funkcji aktywacji:

1. -0.8 w przypadku przegranej

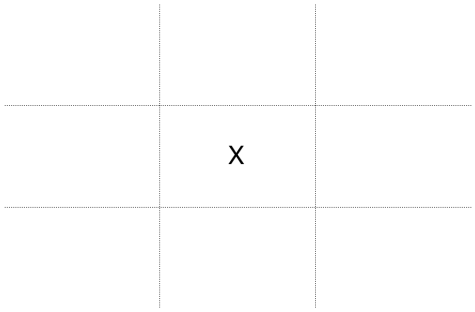
2. 0.5 w przypadku remisu

3. 0.8 w przypadku wygranej

4. 0 dla każdego innego stanu gry

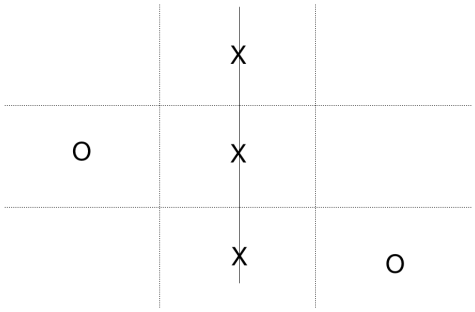
Sam proces uczenia sieci wygląda teraz następująco. Ustalam arbitralnie wielkość zbioru do uczenia (w literaturze odpowiednik pojęcia *batch size*) np. na 2500. Następnie rozgrywam tyle gier, żeby zebrać 2500 obserwacji. Przy czym jedną obserwacją będzie teraz wektor przechowujący stan planszy po ruchu 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. Dla rozjaśnienia zobaczmy 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 naszym przykładzie zaczyna agent, stawia krzyżyk (dwójkę) na środkowym polu. Na 10 pozycji - wartość 0.642 - widzimy oczekiwana wartość nagrody zwróconą przez sieć. Następnie czekamy 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.



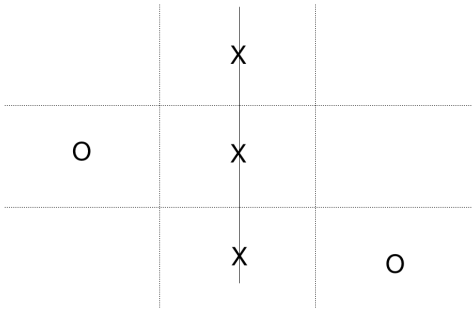
Obserwacja po pierwszym ruchu: $[0, 0, 0, 0, 2, 0, 0, 0, 0, 0.642, 0]$

Następnie system stawia swoje kółko w na pozycji $2, 1$, czyli w pierwszej kolumnie i w drugim rzędzie. Tego stanu planszy nie zapisujemy, bo interesują nas tylko stany po ruchu agenta. Warto jednak nadmienić, że po ruchu systemu sprawdzam 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 nagroda q_s) pozycję $1, 2$:



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 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 chcemy zastąpić prognozy q_s jakie zwracała sieć „poprawnymi" wartościami $\tilde{q}_s$. Ustalamy parametr $\alpha \in (0, 1)$ i postępujemy jak następuje:

1. Jeżeli dana obserwacja jest ostatnią obserwacją w grze, to $\tilde{q}_s = R$
2. W przeciwnym przypadku mamy bierzemy N obserwacji pozyskanych z rozpatrywanej gry i w dla $i = N - 1, N - 2, ..., 1$:
   (a) $\tilde{q}_{s-i} = (1 - \alpha^i)q_{s-i} + \alpha^i q_s$

W skrajnym przypadku, jeżeli $\alpha = 0$, to zmiana q_s na $\tilde{q}_s$ sprowadzała by się wyłącznie do przypisywania $\tilde{q}_s$ wartości nagrody z końca gry. 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.

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 tyle epok ile chcemy. Poniżej przedstawiam wyniki jakie udało mi się uzyskać po 8 epokach, czyli bazując na 20 000 obserwacji.

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

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 procent wygranych gier placuje się w okolicach 90% przypadków. Poniżej przykładowy wynik po 1000 grach.

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

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. Oto wyniki dla 100 rozegranych partii pomiędzy dwoma algorytmami:

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

Warto jeszcze podkreślić, że algorytm z drugiego rozdziału stał się najbardziej efektywny po dodatkowych „sparingachź 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 neruonową już tylko w okolicach 40%-50% przypadków, co ilustruje poniższy wynik rozegranych 100 gier.

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

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 podjąłem 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, modyfikowałem 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 α to parametr sterujący szybkością uczenia się algorytmu (*learning rate*), γ 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) nie 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 opisuję 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 do widać podczas trenowania innych algorytmów poprzez grę z graczem grającym minimax. Np. rozegranie na komputerze autora1000 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. Nieznaczną poprawę szybkości działania przyniosło w przycinanie $\alpha - \beta$. W pojedynczej grzez z komputerem nie daje się tego odczuć, ale przy „masowymżozgrywaniu 1000 jest to nieznacznie szybciej. Podobno najlepsze algorytmy grające w szachy przeszukują drzew gry do 12-13 poziomów, więc pewnie poprawiając struktury danych na bardziej optymalne dało by się odciążyć i przyspieszyć tą implementację.

Największym pozytywnym zaskoczeniem był algorytm *Q- table* bazujący na algorytmie *Off-policy Temporal Difference(0)* opisanym w rozdizale 6.5 w [3]. Zaczynając z pustą tabelą wartości q_s już po 5000 gier mamy tabela 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 słwonik Q-table, i na końcu jest zapisywany do pliku.

Po sukcesach z algorytmem Q-table, wydawało się, że łatwo uda się wytrenować pro-

stą sieć neuronową, która zamiast posługiwać się danymi tabelarycznymi (zapisywanymi do pliku), skompresuje tą informację w macierzach z wagami sieci. Niestety rzeczywistość nie potwierdziła tych optymistycznych założeń i trenując sieć na różne sposoby miałem wrażenie dosięgania„szklanego sufitu", którego nie udało się przebić. Istotnym jest pytania, co jeszcze można by potencjalnie zrobić, że nauczyć sieć dobrze grać w kółko i krzyżyk. Na myśl przychodzi więcej warstw ukrytych oraz inne sieciowo-neuronowe metody 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 wymagała tak zaawansowanych metod. Może to kwestia dobrania odpowiednich hiper parametrów? Chociaż nie próbowałem stosować żadnych formalnych metod przeszukiwania przestrzeni hiper parametrów, to raczej wątpię, żeby ich dobór mógł zaowocować prawdziwym przełomem. W internecie można znaleść 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 (dopasaować o danych). Znacznie bardziej ciekawym pytaniem jest jak wykorzystać sieć neuronową do nauczenia 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 takiem graczu (ponad 90%), co 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 π_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 $\pi_{\tilde{i}}$ mutuje z pewnym prawdopodobieństwem. W efekcie otrzymujemy nowy zbiór strategii zawierający zwycięską strategię π_i oraz jej potomstwo $\pi_{\tilde{i}}$. 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 strategie gry w kółko i krzyżyk, albo jak zapisać kod DNA danej strategii za pomocą parametrów będących obiektami matematycznymi, przerósł możliwości autora.

Dodatek A

Kody

W tej części opisuję podstawowe informacje dotyczące kodów napisanych w ramach niniejszej pracy oraz modułów (bibliotek) innych autorów, których korzystałem. Nie opisuję natomiast poszczególnych elementów kodu i ich działania. Sam kod źródłowy, napisany w języku Python w wersji 2.17, stanowi załącznik do niniejszej pracy i mam nadzieję, że można się w nim z grubsza zorientować 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 Zole
- **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

Jądrem 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 wyłączymy grafikę, co jest niezbędne w przypadku rozgrywania tysięcy gier przy uczeniu algorytmów. 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 ja (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 argument - 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. Cracz 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 sytujację 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 Zello. Podobno ten moduł został stworzony nie tyle z myślą o poważnych zastosowaniach graficznych, ale z myślą żeby uczyć programowania obiektowego na bazie prostych graficzyncyh obiektów typu punkt, linia, koło itp. Ogólnodostępny kod źródłowy tego modułu graficznego można znaleźć tutaj.

A.4. Tensorflow

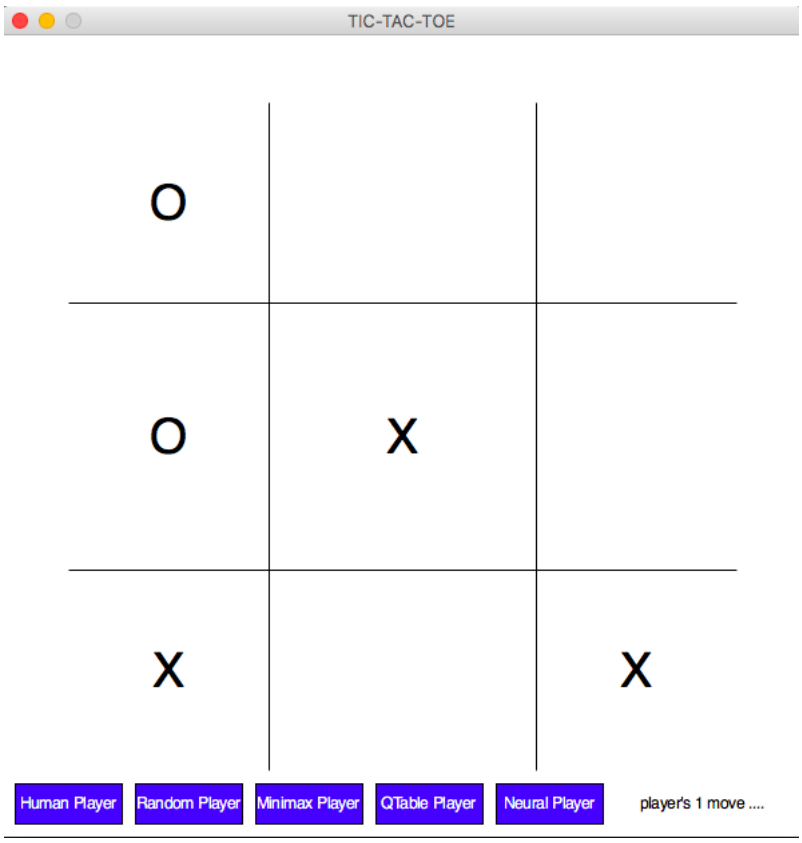
Moim pierwotnym zamysłem była implementacja sieci neuronowej od początku do końca za pomocą podstawowych modułów python'a. I chociaż miałem na tym polu pewne sukcesy, to jednak zwyciężył pragmatyzm i ostatecznie sieć neuronowa opisana w rozdziale 3, w sekcji 3.3 powstała z wykorzystaniem bibliotek *Tensorflow*. 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 o module *Tensorflow* oraz o dostępnym w python'ie API *Tensorflow*do można się dowiedzieć na oficjalnej stronie tutaj.

Dodatek B

Aplikacja

Niezależnie od statystycznych wyników, jakie poszczególne algorytmy osiągają grając w kółko i krzyżyk, ciekawie jest samemu przekonać sie 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 zostałą opisana w tym dodatku.

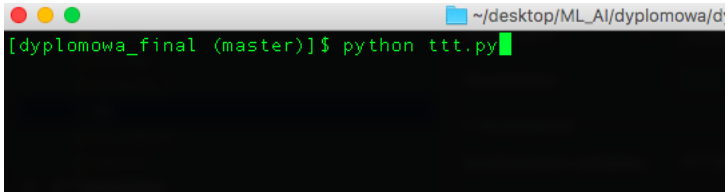
Aplikacja ma tylko jeden ekran (zobacz poniższy rysunku), przedstawiający planszę do gry w kółko i krzyżyk. Po uruchomieniu aplikacji należy za pomocą niebieskich przycisków umieszczonych na dole planszy wybrać sobie przeciwnika. Można do wyboru grać z drugą osobą (człowiekiem), z graczem grającym w 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.



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 działa, ale gracze stosujący te algorytmy będą grali tak jak gracz losowy. Samo uruchomienie aplikacji odbywa się, przez otworzenie pliku ttt.py za pomocą programu Python.



Moduł ttt.py może również służyć do trenowania algorytmów QTable i sieci neuronowej, ale do tego nie ma GUI.

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, Artifical Inteligence*, https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/index.htm

[5] Matthew O. Jackson , A Brief Introduction to the Basics of Game Theory, https://www.ethz.ch/content/dam/ethz/special-interest/gess/chair-of-sociology-dam/documents/education/spieltheorie/literatur/Einführung/Jackson%20Basics%20of%20Game%20Theory%20SSRN-id1968579.pdf

[6] Michael Heining, *Dynamic Learning: A case study on Tic-Tac-Toe*, https://www-m15.ma.tum.de/foswiki/pub/M15/Allgemeines/PublicationsEN/MasterThesisHeining.pdf