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:

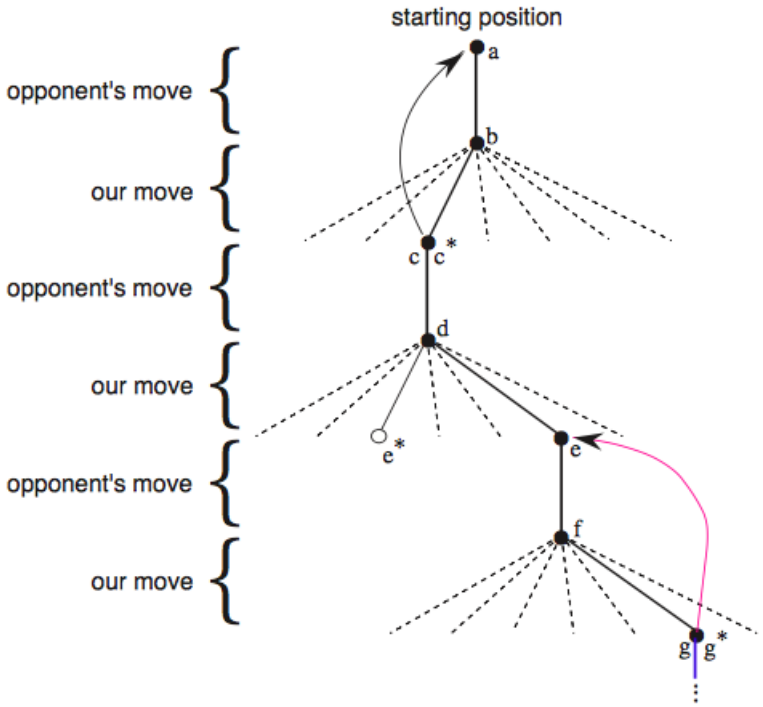
1. **DODATEK A** - Kod z implementacją opisanych algorytmów.
2. **DODATEK B** - aplikacja do gry w kółko i krzyżyk

Rozdział 1

Minimax

1.1. Opis algorytmu

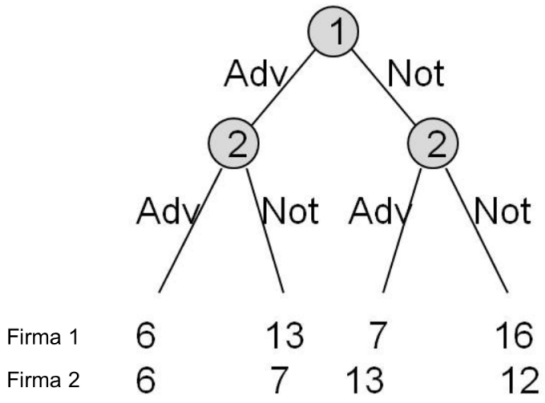
W takich grach jak kółko i krzyżyk, szachy lub warcaby możemy mówić o przestrzeni stanów gry. Każdy stan gry jest reprezentowany przez położenie pionków albo rozmieszczenie kółek i krzyżyków na kwadratowej planszy z 9 polami. Za każdym ruchem, jednego z dwóch graczy, gra przechodzi do innego stanu. Wszystkie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

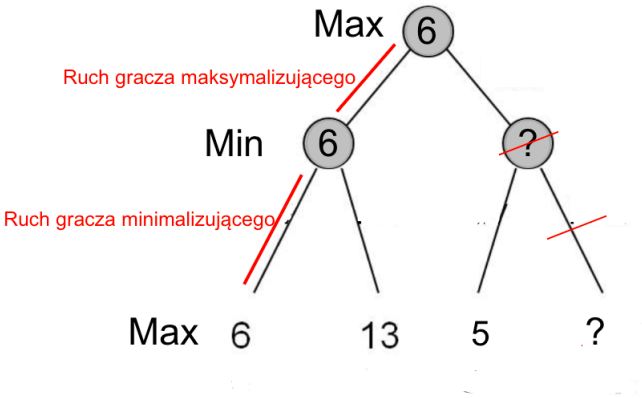
Okazuje się, że jeżeli gracze stosują się do powyższego algorytmu, to znaczy wybierają ruchy, które zapewnią im największą wygraną wyliczoną według powyższego algorytmu, to

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

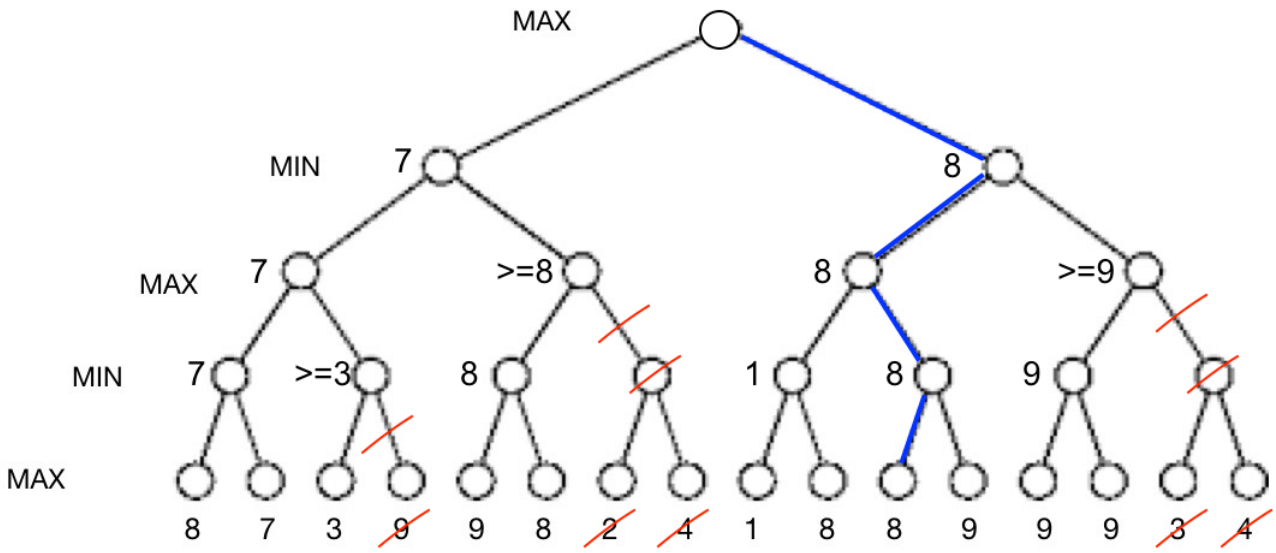
Podstawowym problemem w zastosowaniach algorytmu minimax jest ilość niezbędnych operacji przy dużych drzewach reprezentujących gry. Dla gry w kółko i krzyżyk, górnym ograniczeniem ilości możliwych gier (liści na drzewie gry) jest $9! = 362880$. W rzeczywistości ilość możliwych partii ogranicza się do 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 $\alpha - \beta$.

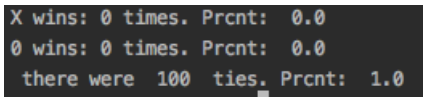
Liczyłem na to, że zastosowanie przycinania zmniejszy czas wykonania algorytmu na początku gry, w pierwszych dwóch ruchach, kiedy konieczne jest przeszukanie największych drzew. Od trzeciego ruchu, wykonanie algorytmu jest rzędu 0.1 sekundy, więc optymalizacje przestają mieć znaczenie. Niestety zastosowanie 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.



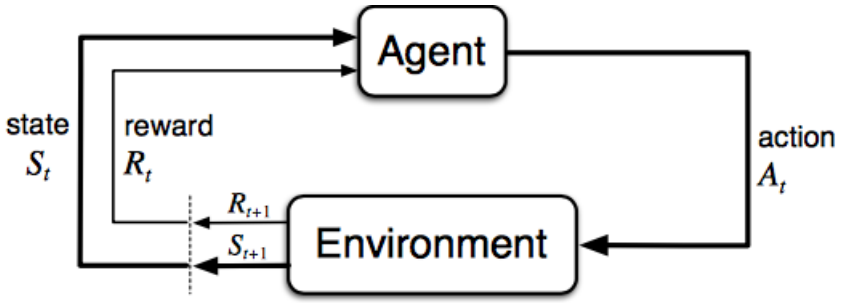
Dwóch graczy stosujących minimax zawsze będą ze sobą remisować.

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:

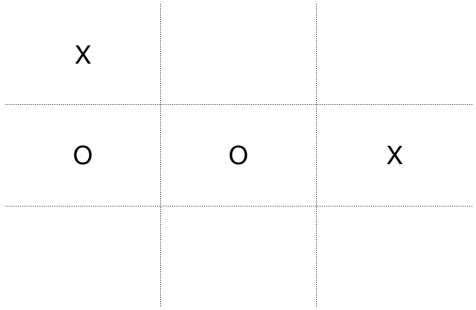
O	X	X
X	O	O
	O	X

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.

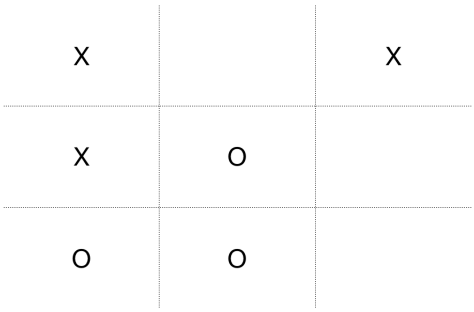
Kolejny przykład, to ciąg $2, 2, 1, 2, 2, 1, 1, 0, 0, -1$, jak łatwo zauważyć przedstawiam sytuację, w której ruch należy do gracza systemowego (kółko lub cyfra 1) i gracz ten może z łatwością wygryać uzyskując trzy „kółka" na prawej skrajnej kolumni. Stąd wartość q w tym przypadku jest równa -1.

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.



Rozdział 3

Deep Q-learning

3.1. Rys historyczny

Jak poprzednio - krótki opis.

3.2. Opis algorytmu

W tej materii jeszcze niczego nie zrobiłem, ale będę walczył :-)

Rozdział 4

Podsumowanie

Tak jak zaznaczyłem we wprowadzeniu, tutaj opiszę - bardzo skrótowo - wrażenie z implementacji oraz podam jakieś informacje, który z algorytmów jest skuteczniejszy, szybszy i dlaczego. Na razie nic specjalnie mądrego nie przewiduję w tym rozdziale, ale wypadałoby, żeby jakieś podsumowanie było. Może po ukończeniu i przetestowaniu kodu, przyjdzie mi do głowy coś wartego napisania.

Dodatek A

Kody

Pytanie, czy warto tu umieszczać wszystkie, kody czy też nie ma sensu i może wystarczy link do gitHub'a.

A.1. Klasa gry

```
return line
```

Dodatek B

Aplikacja

Tutaj zamieszczę krótki opis i ścreen shotź aplikacji do gry w kółko i krzyżyk.

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