Podstawy sztucznej inteligencji

Projekt

Temat projektu: ***Nauczyć sieć neuronową grać w kółko i krzyżyk.***

Autorzy:
Jan Kumor
Rafał Wądołowski
Michał Witanowski

Cel projektu

Celem projektu jest stworzenie interaktywnej aplikacji pozwalającej użytkownikowi grać w kółko i krzyżyk ze sztuczną siecią neuronową. Interfejs powinien pozwalać na przeprowadzenie procesu uczenia sieci oraz ich porównania.

Decyzje projektowe

Głównymi decyzjami projektowymi, jakie należało podjąć przed przystąpieniem do implementacji aplikacji, były:

- sposób odwzorowania stanu planszy i następnego ruchu w wejściach i wyjściach sieci neuronowej,
- metodę uczenia sieci (offline/online, algorytm uczenia).

Zdecydowano się, że danymi wejściami do sieci będzie aktualny stan planszy (9 neuronów), gdzie każdemu polu planszy odpowiada jeden neuron:

- kółko – wartość **-1**,
- krzyżyk – wartość **1**,
- puste pole – wartość **0**.

Przewidziano dwa podejścia do reprezentacji danych wyjściowych sieci:

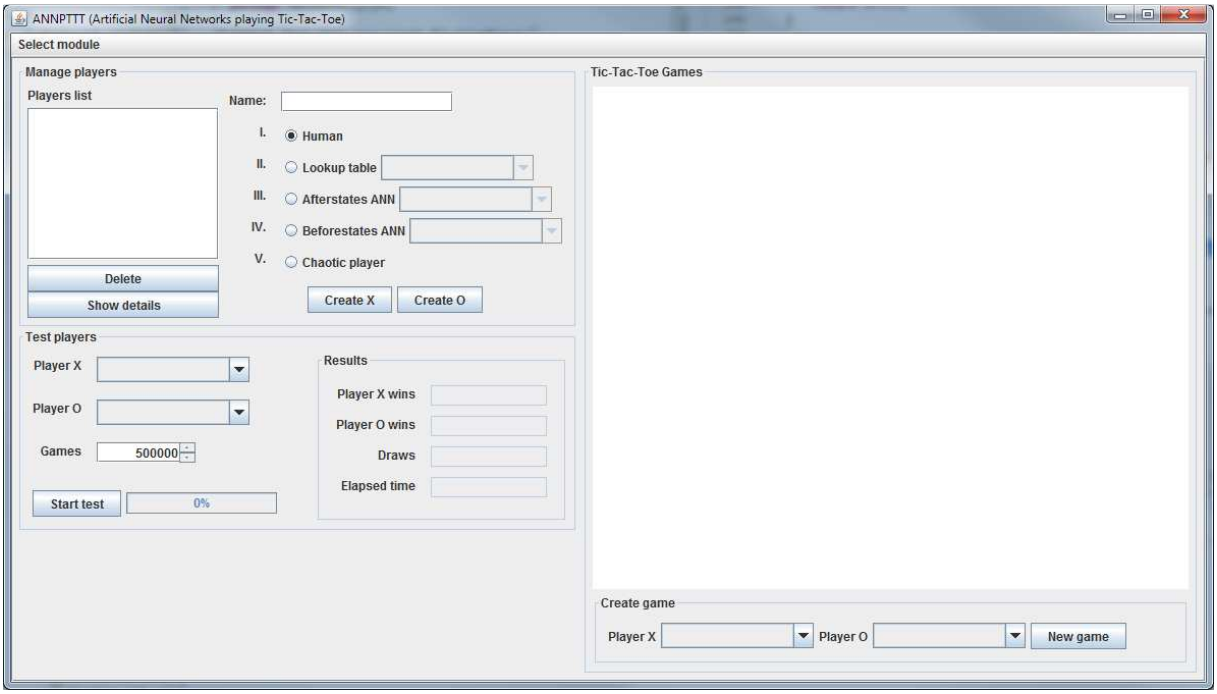
- *Afterstate*: Jedno wyjście, określające prawdopodobieństwo wygrania przy planszy w danym stanie. Do podjęcia ruchu wymagane jest uruchomienie sieci tyle razy, ile zostało wolnych pól aby wybrać najlepiej rokujące.
- *Beforestate*: 9 wyjść – tablica wskazująca prawdopodobieństwo wygranej przy podjęciu danego ruchu. Dzięki temu, sieć będzie wykonywała taki ruch, gdzie największą wartość będzie na wyjściu sieci. Ponadto w tym podejściu wektor danych uczących będzie mniejszy, ponieważ nie będzie posiadać stanów końcowych gry.

Jako metodę uczenia sieci wybrano algorytm wstecznej propagacji błędu. Jedynym problemem był sposób pozyskania danych uczących. Uczenie sieci na podstawie rozgrywanych z użytkownikiem gier zajęłoby zbyt dużo czasu, dlatego zdecydowano się na wygenerowanie danych automatycznie. Tabelę afterstates można stworzyć rozgrywając wszystkie możliwe gry. Wykorzystano algorytm Q-learning, przy pomocy którego

określono prawdopodobieństwo wygranej przy danych układzie planszy. W przypadku podejścia beforestates wymagana jest konwersja tablicy.

Instrukcja

Menu gry (*Tic-Tac-Toe Game Module*)



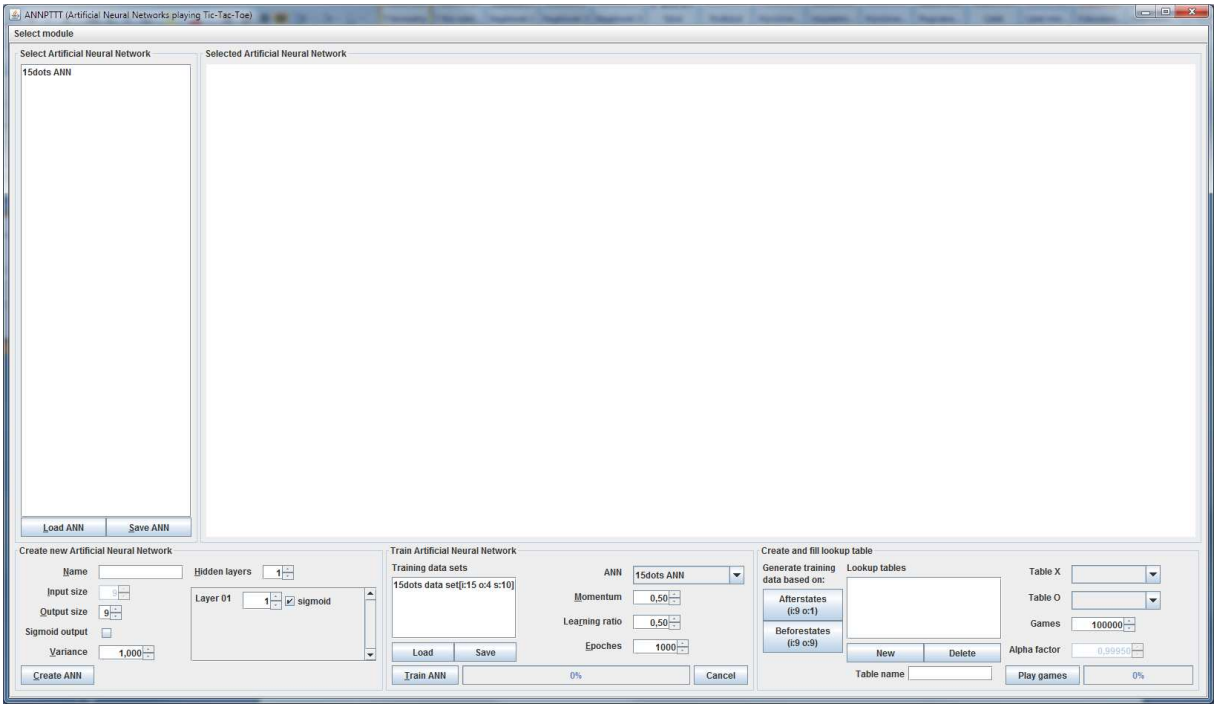
W menu gry można zarządzać listą graczy, porównywać ich i odbywać grę. Przewidziano następujących typów graczy:

- *Human* (użytkownik),
- *Lookup table* (gracz podejmujący ruch na podstawie tabeli z prawdopodobieństwami wygrania),
- *Afterstates ANN* (sieć neuronowa z 1 wyjściem określającym prawdopodobieństwo wygrania w danym układzie planszy),
- *Beforestates ANN* (sieć neuronowa z 9 wyjściami prawdopodobieństwo wygrania przy podjęciu danego ruchu),
- *Chaotic* (gracz podejmujący losowe ruch, używany głównie do testowania – dobrze nauczona sieć nie powinna nigdy przegrywać).

W polu *Test players* można porównać dwóch komputerowych graczy. Po rozegraniu *N* gier wyświetlane są statystyki (ile razy wygrał krzyżyk, ile razy wygrało kółko, ile było remisów, itp.).

Pole *Create game* służy do tworzenia nowej gry. Wybierając dwóch komputerowych graczy można obejrzeć przebieg gry.

Menu sieci neuronowych (*Artificial Neural Networks Module*)

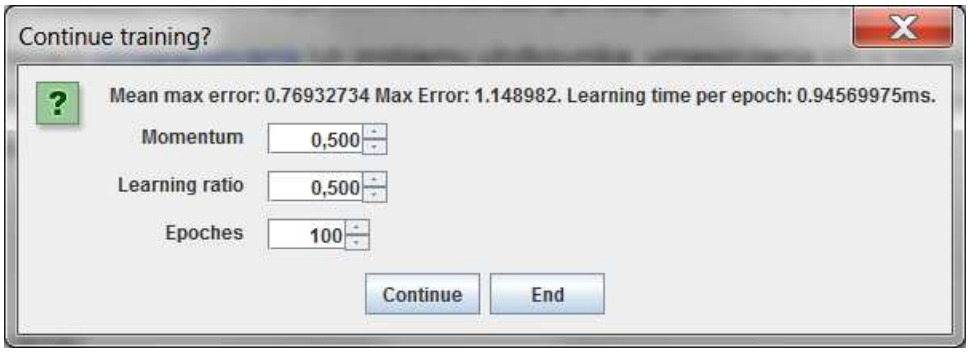


W menu sieci neuronowych można tworzyć, uczyć sieci oraz generować dla nich wektory testowe.

Aby stworzyć i nauczyć sieć należy kolejno:

1. Rozegrać gry budujące tablicę LUT w polu *Create and fill lookup tables*.
2. Wygenerować wektory uczące przy pomocy przycisku *Afterstates* lub *Beforestates.*
3. W polu *Create new Artificial Neural Network* stworzyć nową sieć. Należy pamiętać, że w przypadku wybrania danych typu Afterstates sieć musi mieć 1 wyjście, a dla danych Beforestates – 9. Można tu skonfigurować ilość warstw ukrytych oraz liczbę neuronów w każdej warstwie.
4. W polu *Train Artificial Neural Network* należy wybrać sieć w polu *ANN*, dane uczące (*Training data sets*) oraz ustawić parametry procesu uczenia:
    a. *Momentum* – bezwładność,
    b. *Learning ratio* – skalowanie korekcji wag neuronów,
    c. *Epoches* – liczba epok (rund) uczenia.

Po każdym cyklu treningowym program wyświetla zapytanie o kontynuację treningu, wraz z informacją o stanie zbieżności oraz czasie treningu. W oknie zapytania możliwe jest zmiana parametrów procesu uczenia dla kolejnego cyklu. Zakończenie treningu powoduje dodanie wytrenowanej sieci do listy sieci.



Budowa programu

Aplikacja została napisana z użyciem języka Java 1.7 z zastosowaniem wzorca projektowego MVC (Model-View-Controller). Graficzny interfejs użytkownika został zrealizowany za pomocą natywnej dla Javy technologii Swing.

Program składa się z dwóch modułów:

- Menu gry (*Tic-Tac-Toe Game Module*)
- Menu sieci neuronowych (*Artificial Neural Networks Module*)

Możliwości modułów zostały opisane pokrótce w poprzednim rozdziale.

I. Najważniejsze klasy warstwy Model

- **NeuralNetwork** - implementacja sztucznej sieci neuronowej typu perceptron wielowarstwowy (ang. Multilayer perceptron - MLP).
    - metoda ***train***() - umożliwia uczenie sieci z zastosowaniem **algorytmu propagacji wstecznej**
    - metoda ***run()*** - oblicza wyjście sieci na podstawie wektora wejść
- **LookupTable** - implementacja tablicy stanów algorytmu uczenie ze wzmocnieniem Q-learning. Przechowuje wartości oczekiwanej nagrody w zależności od zakodowanego stanu gry.
    - metoda ***updateProcedure()*** - automatyczna aktualizacja tabeli algorytmem **Q-Learning** na podstawie historii rozegranej gry
    - metoda ***getAfterstateTrainingData()*** - generacja danych do nauki sieci typu afterstates (dla sieci 9 wejść-1 wyjście)
    - metoda ***getBeforestateTrainingData()*** - generacja danych do nauki sieci typu beforestates(dla sieci 9 wejść-9 wyjść)
- **GameModel** - reprezentacja gry w kółko i krzyżyk. Przechowuje aktualny stan planszy oraz historię gry. Zawiera implementację zasad gry oraz szereg metod pomocniczych w tym:
    - metoda ***fastPlay()*** - automatyczne rozegranie gry pomiędzy dwoma graczami komputerowymi
    - metoda ***hash()*** - wyznaczenie kodu jednoznacznie reprezentujący dany stan planszy
    - metoda ***unhash()*** - wyznaczenie stanu planszy na podstawie kodu

II. Najważniejsze klasy warstwy View

- **View** - kontener przechowujący instancje okna aplikacji oraz modułów. Klasa opakowująca umożliwiająca komunikację kontrolera z elementami widoku
- **AppFrame** - główne okno aplikacji, umożliwiające przełączanie między modułami
- **GameModule** - klasa reprezentująca *Menu gry (Tic-Tac-Toe Game Module)*
- **ANNModule** - klasa reprezentująca *Menu sieci neuronowych (Artificial Neural Networks Module)*
- **GameWindow** - okno rozgrywania gry dwuosobowej

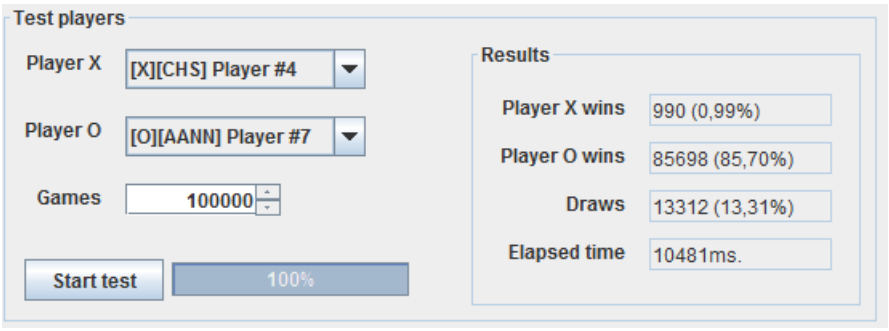
III. Najważniejsze klasy warstwy Controller

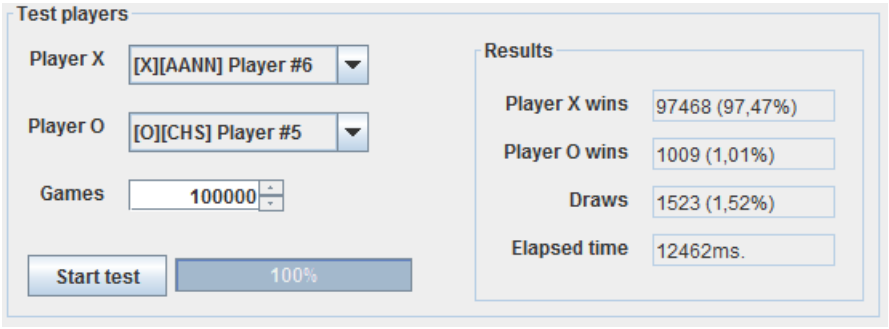
- **AppController** - główny kontroler aplikacji
- **GameController** - kontroler gry dwuosobowej (każda gra obsługiwana przez oddzielny wątek)
- klasy graczy (dziedziczące po **Player**) implementacje gracza, podejmują decyzję na podstawie różnych algorytmów
    - **AfterstatesANNPlayer** - wykonuje ruch na podstawie porównania wyjść sieci typu afterstates dla danych stanów następnych
    - **BeforestatesANNPlayer** - wykonuje ruch wybierając największą wartość wyjścia sieci typu beforestates

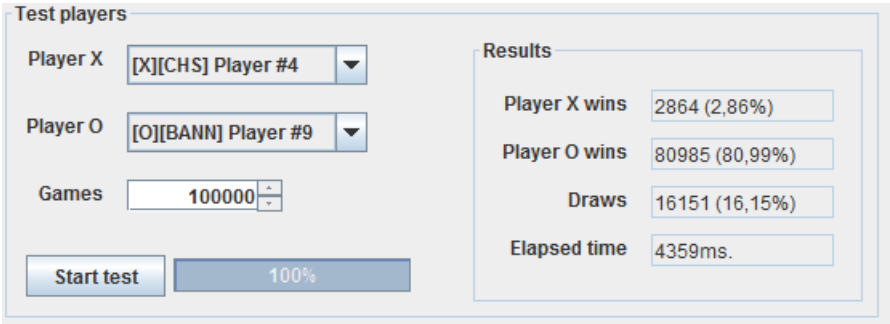
- **ChaoticNeutralPlayer** - gracz losowy
- **HumanPlayer** - oczekuje na wprowadzenie ruchu przez człowieka
- **LookupTablePlayer** - wykonuje ruch wybierając z tabeli stanów stan następny o najwyższej przewidywanej nagrodzie

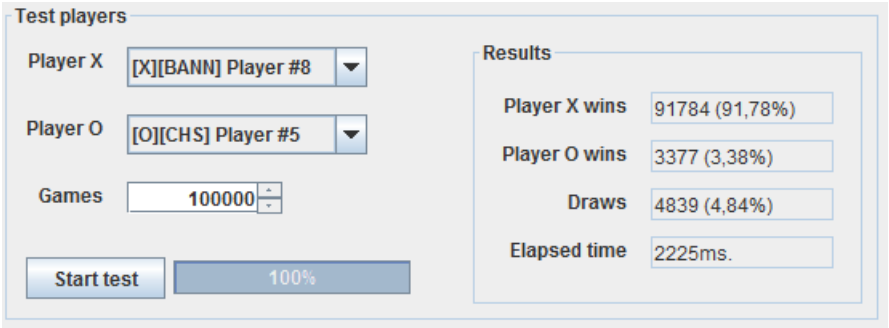
Wnioski

Wytrenowane sieci neuronowe zostały poddane procedurze testowej polegającej na rozegraniu 100 tys. gier przeciwko graczowi losowemu. Wyniki testów przedstawiono na poniższych ilustracjach.



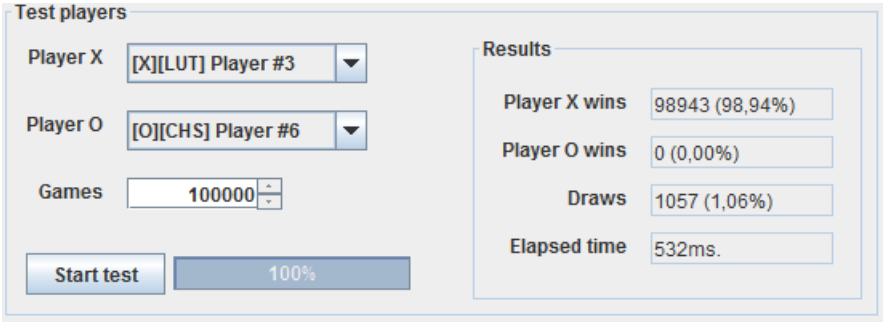


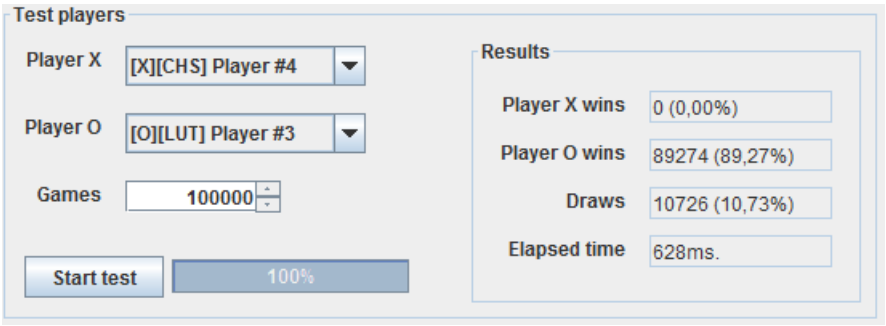




Jak widać dla sieci typu afterstates osiągnięto skuteczność na poziomie około 99%, zaś dla sieci beforestates około 97%.

Można pokazać, poddając tej samej procedurze testującej tabele LookupTable, że błąd klasyfikacji nie wynika z błędnych danych testowych. Tabele uzyskały skuteczność na poziomie 100% co widać na poniższej ilustracji.





Występujący błąd w klasyfikacji wynika z błędu aproksymacji odwzorowania tabeli przez sieć neuronową. Dla tak dużego wektora danych wejściowych mogą wystąpić miejscowe pojedyncze znaczące odchylenia wartości wyjścia sieci aproksymującej odwzorowanie z tabeli. Odchyłki takie nie powodują istotnego wzrostu błędu średniego odwzorowania, jednak mogą okazać się kluczowe dla podjęcia optymalnej decyzji.

Możliwymi metodami rozwiązania problemu są:

- zwiększanie rozmiarów sieci - dobór większej sieci w teorii pozwala na aproksymację funkcji odwzorowania z dowolną dokładnością. Niestety, dla wysokich wskaźników dopasowania, wzrost kosztu obliczeniowego treningu sieci oraz koszt obliczenia wyjść sieci, jest niewspółmierny do uzyskiwanej poprawy.
- zmniejszenie rozmiarów wektora danych wejściowych - dla mniejszej ilości danych łatwiej uzyskać dopasowanie, a sieć szybciej zbiega do optimum. Ponadto zmniejszenie kosztu obliczeniowego jednej rundy uczenia pozwala na wykonanie w tym samym czasie większej liczby rund. Dla rozpatrywanego problemu zmniejszenie rozmiaru danych wejściowych można uzyskać wykorzystując wysoką symetrię stanów planszy do gry (cztery obroty oraz ich odbicia lustrzane).
- zidentyfikowanie problematycznych stanów i zaimplementowanie dla nich odpowiednich heurystyk, mających priorytet nad siecią.