Sprawozdanie dotyczące porównanie użycia metody FOL i sieci neuronowej w grze Saper

Maksim Huzino 187718
Kanstantsin Kalenik 187704
Filip Sołtys 185815
Mariia Kyrychenko 192425

29.05.2023

1. Wstęp

Implementacja tego projektu została wykonana w języku Python. Do sieci neuronowej wykorzystano bibliotekę TensorFlow, a część wizualna oparta była na bibliotece Tkinter. Główne zadanie polegało na implementacji algorytmów zdolnych do gry w klasycznego Sapera.

Gra Saper to popularna gra logiczna, w której celem gracza jest odkrycie wszystkich pól na planszy, które nie zawierają min. Gracz musi ostrożnie odkrywać pola, unikając min, a także korzystając z informacji o liczbie min w sąsiednich polach, aby dedukować, gdzie mogą znajdować się miny.

Na początku każdej rozgrywki tworzona jest plansza z zakrytymi polami, które można odkrywać poprzez kliknięcie w nie.
Istnieją trzy możliwości odkrycia pola po kliknięciu:

- ***Pole z cyfrą:*** Oznacza ono liczbę min sąsiadujących z tym polem (maksymalnie osiem możliwych). W takim przypadku pole zostaje otwarte i wiemy, że pod nim zajmuje się mina.

- ***Pole puste:*** W tym przypadku pola sąsiadujące z tym polem są odkrywane w sposób rekurencyjny, aż do momentu, gdy utworzą one zamknięty obszar ograniczony polami z cyframi. Oznacza to, że pod danym polem nie było miny.

- ***Pole z miną:*** Kliknięcie na to pole prowadzi do przegranej w rozgrywce, ponieważ pod polem była mina.

Jedyną możliwością zwycięstwa jest odkrycie wszystkich pól na planszy, które nie zawierają min. Agenci sztucznej inteligencji, wybierając odpowiednie pole, sprawdzają symbol odpowiadający danemu polu na rzeczywistej planszy i umieszczają go na aktualnie działającej planszy. Nasza implementacja rozważa pola o rozmiarze ustawionym przez użytkownika (w terminalu).

2.1. FOL - opis

First-Order Logic (FOL), znane również jako logika pierwszego rzędu, jest systemem formalnym, który wykorzystuje zmienne ilościowe nad obiektami nielogicznymi. Pozwala na formułowanie zdań, w których występują zmienne, na przykład "Istnieje takie x, że x to...". W FOL tworzy się zazwyczaj określoną dziedzinę dyskursu, która jest bezpośrednio zbiorem zdań w logice pierwszego rzędu.

W logice pierwszego rzędu język jest bardziej rozbudowany i umożliwia jawne reprezentowanie obiektów (stałych), relacji (predykatów) jako logiczne funkcje oraz zwykłych funkcji. W przeciwieństwie do innych logik, FOL nie używa symboli, lecz stosuje stałe, predykaty, funkcje, łączniki, zmienne i kwantyfikatory. Zmienne mogą być używane tylko w połączeniu z kwantyfikatorami.

Reprezentacja liczb całkowitych wygląda następująco: definiujemy stałą 0 i definiujemy funkcję następnika (successor()). Nie ma potrzeby jednoznacznej definicji stałych 1, 2 itd. Są one niejawnie zdefiniowane jako wartości zwracane przez funkcję następnika.

Podstawowe predykaty do działania agenta:

- Czy pole jest warte uwagi? (czy można stwierdzić z pewnością, czy na nim jest bomba czy nie):

  Predykat: *intresting(board, position)*

  Interesujące pole to takie, które spełnia dwa warunki jednocześnie:
  - Jest zakryte (nie wiadomo, co się pod nim znajduje)
  - Ma przynajmniej jednego sąsiada będącego liczbą (z dowolnej strony)

  W ten sposób eliminujemy pola, o których nie możemy nic powiedzieć, pozostawiając jednoznaczne i niejednoznaczne pola.

- Czy pole zawiera liczbę?

  Predykat: *contain_number(board, position)*

- Czy miny w okolicy mają oczywiste pozycje?

  Predykat: *can_flag_bombs around(board, position)*

  Liczba sąsiadów z niewidocznym elementem lub posiadających flagę równa się liczbie min (podanej na danym polu).
  W przypadku pól, które nie zawierają liczby, otrzymujemy fałsz.
  W ten sposób wiemy, że każdy z sąsiadów może być oznaczony flagą.

- Czy każdy nieoznaczony pusty sąsiad jest bombą?

  Predykat: *not_flagged_neighbours_contain_bombs(board, position)*

  Liczba bomb (pokazana na elemencie planszy) równa się liczbie oznaczonych pól sąsiadujących (tylko jeśli zawierają liczby).

- Czy zawiera bombę?

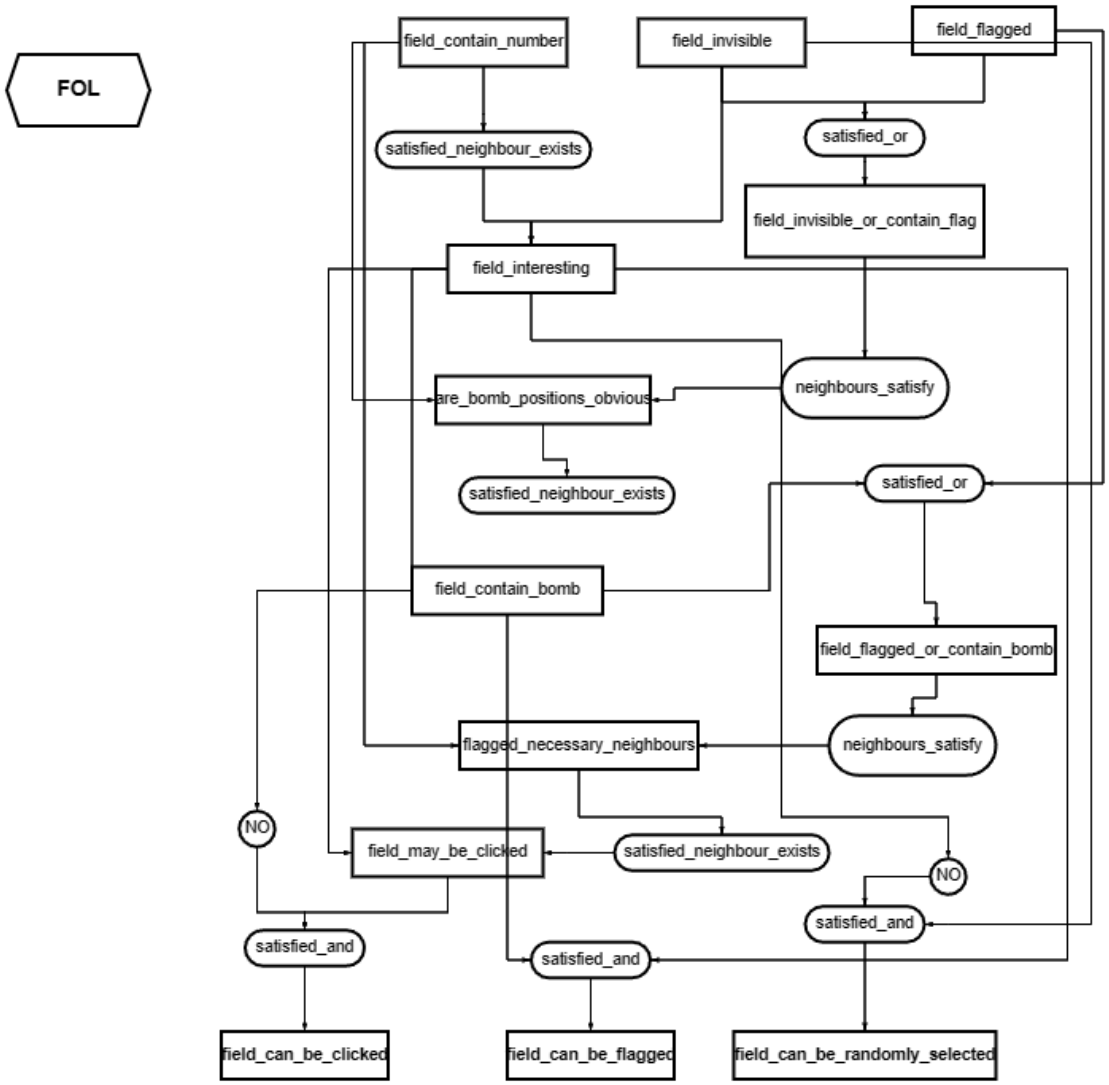
  Predykat: *contain_bomb(board, position), interesting(board, position) and (for each neighbour not_flagged_neighbour(board, neighbour)).*

  Pole zawiera bombę, jeśli jest interesujące oraz jeśli dowolny sąsiad spełnia predykat *not_flagged_neighbour(board, neighbour)*.

2.2. Implementacja FOL

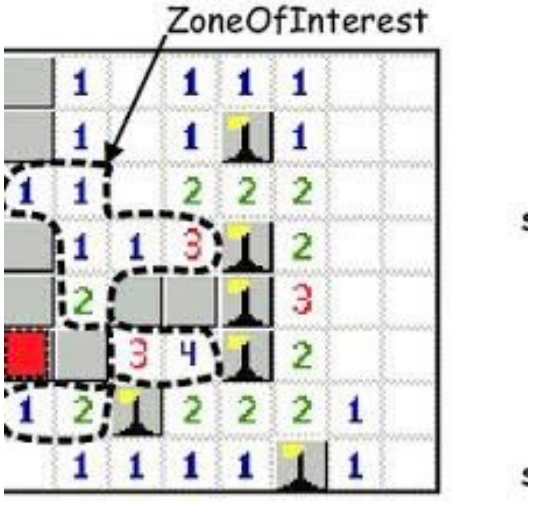
Użyliśmy tutaj dodatkowej biblioteki Numpy.

1. Przypisujemy wartości do zmiennych (mines, size, tests).
2. Tworzymy agenta FOL.
3. Tworzymy planszę o podanych parametrach (miny są rozmieszczone losowo przez generator).
4. W pętli while, dopóki gra się nie zakończy (po kliknięciu na minę lub wygraniu), sprawdzamy pola, które mogą być ewentualnie odkryte poprzez sprawdzanie warunków predykatów.



Rysunek 1 - zobrazowanie logiki FOL

5. Odkrywamy pola, które spełniają warunki, jeśli istnieją takie. Jeśli nie, odkrywamy pola, które mają ustawioną wartość „field_can_be_randomly_selected”. Ma to miejsce głównie na początku.



Rysunek 2 - Pole zainteresowania

3.1. Sieć neuronowa - opis

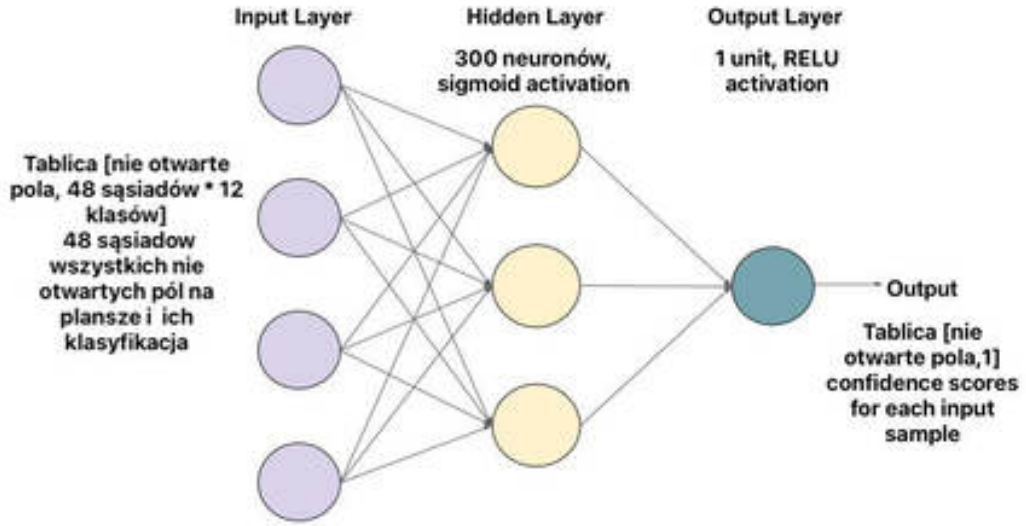
Sieć neuronowa to system przetwarzający informacje, który posiada zdolność do rozwiązywania praktycznych problemów bez konieczności wcześniejszej matematycznej formalizacji. Jest to jedna z jej wyróżniających cech. Kolejną zaletą sieci neuronowych jest brak potrzeby opierania się na teoretycznych założeniach dotyczących problemu, który jest rozwiązywany.

Najbardziej charakterystyczną cechą sieci neuronowych jest możliwość uczenia się na podstawie przykładów oraz automatyczne uogólnianie zdobytej wiedzy, co nazywane jest generalizacją. Oznacza to, że sieć neuronowa może analizować zbiór danych uczących, dopasowywać się do wzorców w tych danych i potem wykorzystywać nabyte umiejętności do rozpoznawania i przetwarzania nowych danych, które nie były wcześniej uwzględnione w zbiorze uczącym.

Sieć neuronowa składa się z połączonych ze sobą sztucznych neuronów, które naśladują działanie biologicznych neuronów w mózgu. Neurony te otrzymują dane wejściowe, przetwarzają je za pomocą funkcji aktywacji i przekazują wyniki do kolejnych neuronów w sieci. Dzięki temu sieć neuronowa może wykonywać skomplikowane obliczenia, identyfikować wzorce, rozpoznawać obrazy, tłumaczyć teksty i podejmować decyzje na podstawie dostarczonych danych.

3.2. Opis zaimplementowanej sieci neuronowej

Stworzyliśmy model sieci neuronowej typu feedforward zaimplementowany przy użyciu interfejsu API Keras w TensorFlow. Sieci typu feedforward są stosunkowo proste i łatwe do zrozumienia i wdrożenia. Mają wyraźny przepływ informacji od wejścia do wyjścia bez żadnych pętli lub połączeń zwrotnych. Jest to model sekwencyjny z dwiema gęstymi warstwami: jedną warstwą ukrytą z 300 jednostkami i sigmoidalną funkcją aktywacji oraz jedną warstwą wyjściową z 1 jednostką i funkcją aktywacji ReLU. Ogólnie, model sieci neuronowej przyjmuje tablice z closed polami i dla tych pól klasyfikujemy 48 sąsiadów jako dane wejściowe, przepuszcza ją przez dwie gęste warstwy z różnymi funkcjami aktywacji i generuje dane wyjściowe reprezentujące prawdopodobieństwo zamknięcia każdej komórki.



Rysunek 3 - Zobrazowanie sieci neuronowej

3.3. Implementacja sieci neuronowej

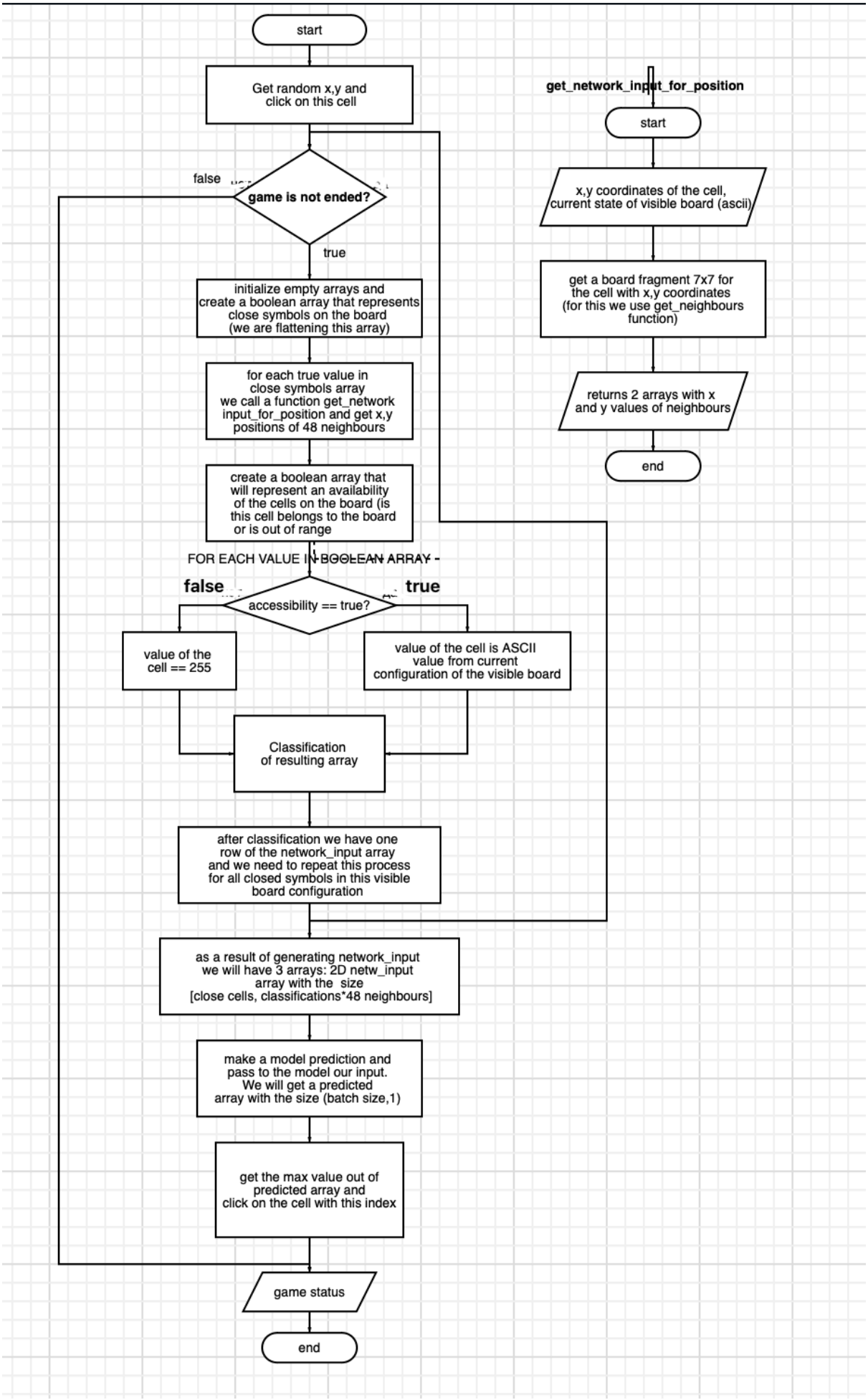
1. Przypisujemy wartości do zmiennych (mines, size, tests).
2. Tworzymy agenta sieci neuronowej.
3. Tworzymy planszę o podanych parametrach (miny są rozmieszczone losowo przez generator).
4. Pierwsze pole jest wybierane losowo.
5. W pętli while, dopóki gra nie jest zakończona (poprzez kliknięcie na minę lub wygraną), tworzymy listę wszystkich zamkniętych pól, jednocześnie je klasyfikując.
6. Następnie przewidujemy listę pól, które najbardziej pasują.
7. Odkrywamy pierwsze pole z tej listy.

3.4. Trening modelu sieci neuronowej

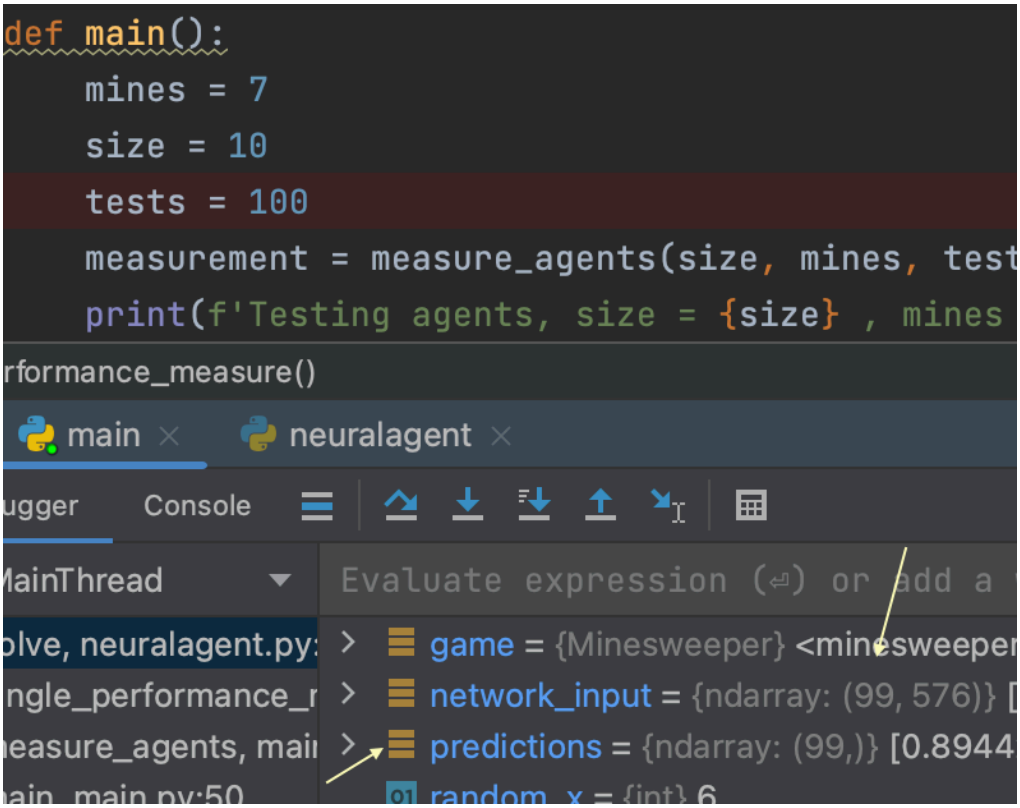
Model jest trenowany poprzez łączenie wielu zestawów danych wygenerowanych z różnych konfiguracji gier. Zbiory danych składają się z tablic wejściowych reprezentujących konfiguracje planszy i tablic docelowych reprezentujących powiązane wartości docelowe. Zbiory danych są tasowane, a model jest trenowany przy użyciu funkcji fit() z określoną liczbą epok(50). Wydajność modelu jest oceniana przy użyciu oddzielnego testowego zestawu danych, a wytrenowany model jest zapisywany do wykorzystania w przyszłości. Trenujemy model przy użyciu 1000 konfiguracji gier z rozmiarem planszy 20x20, 40 minami i 10 losowymi klikami.

3.5. Dane wejściowe i wyjściowe w modelu

Wynik przewidywania modelu jest zbiorem prawdopodobieństw, tablica będzie zawierać przewidywane wartości dla każdej próbki wejściowej, gdzie każdy element reprezentuje przewidywanie dla pojedynczej próbki. Metoda rozwiązywania agenta iteracyjnie wybiera najlepszy ruch na podstawie przewidywań z modelu i kontynuuje klikanie komórek aż do zakończenia gry. Celem agenta jest zmaksymalizowanie liczby zwycięstw poprzez wybranie przewidywanych ruchów o najwyższym prawdopodobieństwie wygranej.
Szczegółowy proces generowania danych wejściowych został zilustrowany na poniższym schemacie (Rysunek 4). Dodatkowo na rysunkach 5.1-5.3 pokazane są 3 przejścia przez pętle while.

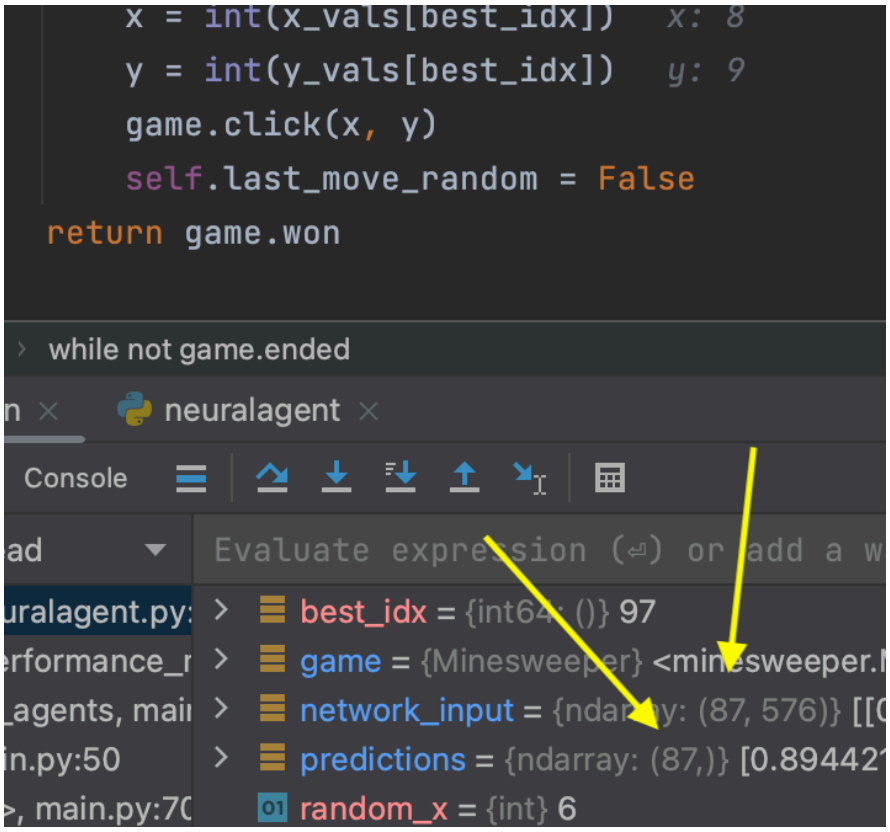


Rysunek 4 - Zobrazowanie procesu generowania danych

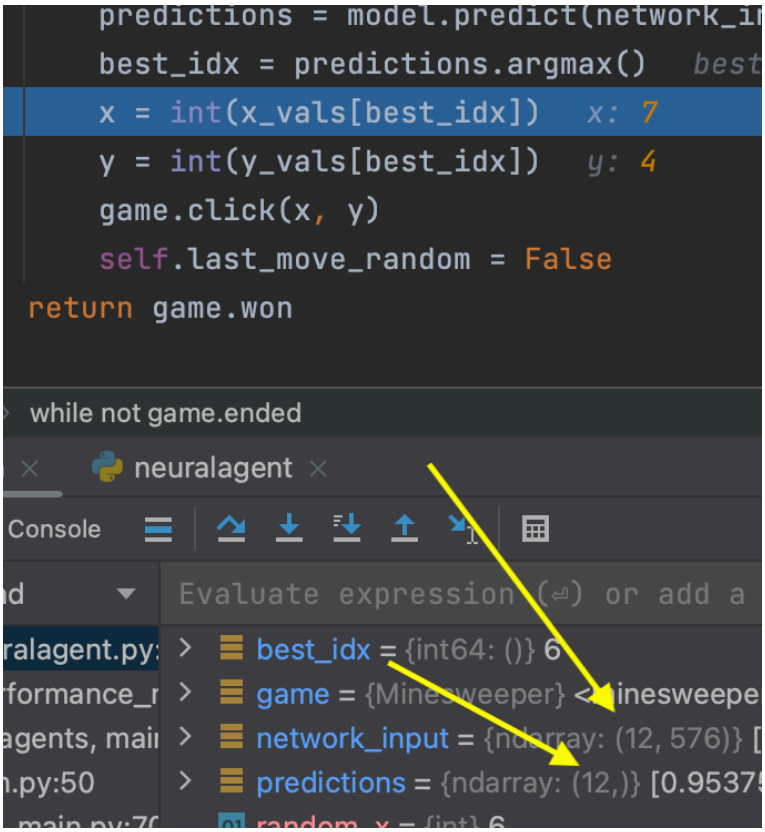


Rysunek 5.1 - Pierwsze przejście pętli while (Size=10)

Liczba klas (12) pomnożona przez liczbę sąsiadów (48) = 576 - jest to liczba pól w rzędzie, które charakteryzują closed pole.



Rysunek 5.2 - Drugie przejście pętli while (Size=10)



Rysunek 5.3 - Trzecie przejście pętli while (Size = 10)

3.6. Klasyfikacja

W przypadku gry Saper, One-Hot Encoding jest używany do reprezentowania różnych klas lub symboli obecnych na planszy. Każda komórka na planszy gry Saper może mieć różne symbole, takie jak odkryta komórka, zakryta komórka, komórka z liczbą wskazującą sąsiednie miny lub symbol miny. Przeprowadzamy klasyfikację One-Hot Encoding wartości ASCII dla 48 sąsiadów wszystkich zamkniętych komórek, które są spłaszczone w jednym rzędzie w oparciu o predefiniowany zestaw klas. Funkcja pobiera tablicę, którą opisaliśmy wcześniej i konwertuje ją na reprezentację One-Hot Encoding, w wyniku czego otrzymujemy tablicę [neighbours*classes, classes] danych.

Symbol	ASCII
#	35
' '	32
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
255	nie jest zakodowany, oznacza to, że ten symbol jest poza zakresem planszy

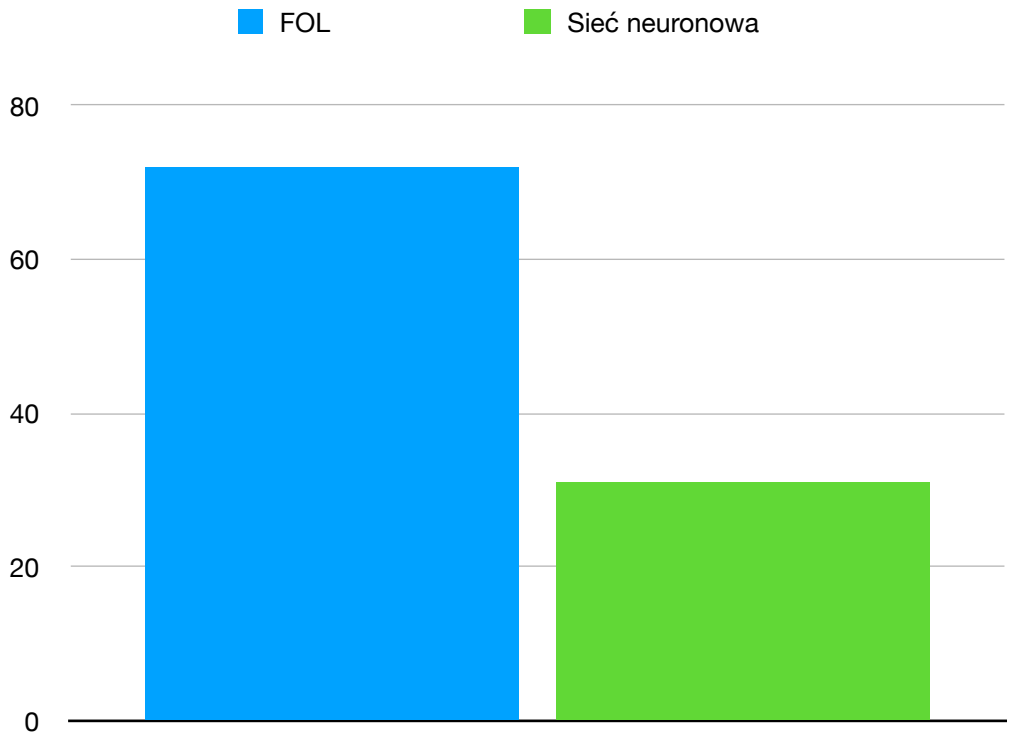
Pętla jest używana do iteracji po liście klas. Dla każdej klasy odpowiednia kolumna w wyniku jest ustawiana na 1,0, jeśli odpowiedni element w tablicy wprowadzonej do funkcji klasyfikacji pasuje do wartości klasy, i 0,0 w przeciwnym razie. Na koniec spłaszczamy tę dwuwymiarową tablicę i tworzymy dane wejściowe sieci dla pojedynczej zamkniętej komórki.

4. Wnioski

Graficznie rozwiązanie gry wygląda w następujący sposób:

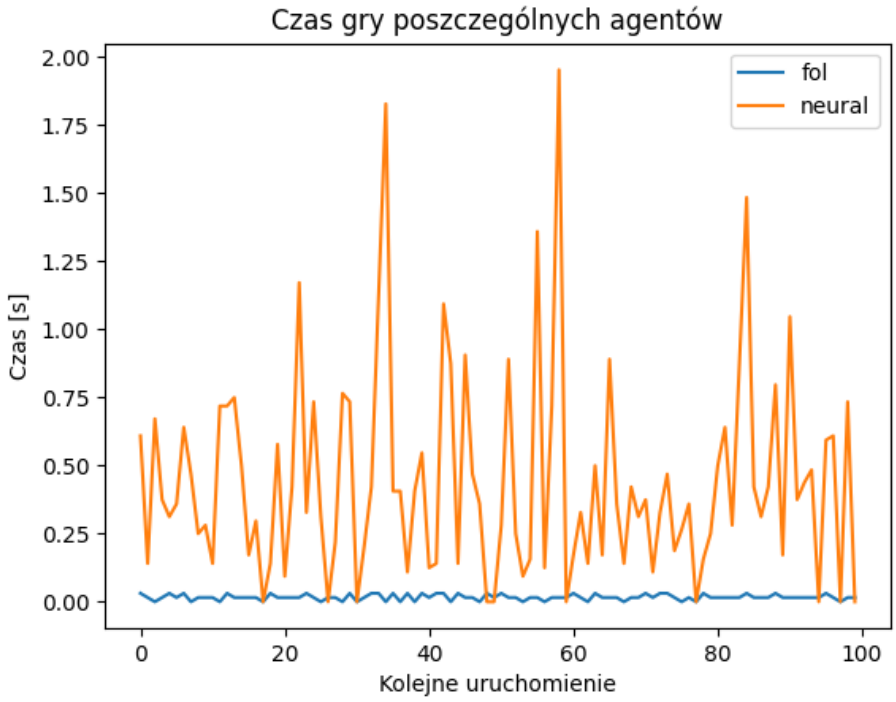
Rysunek 4.1 - wygląd planszy

Jeżeli chodzi o bezpośrednie porównanie skuteczności obydwu metod w przypadku wygranych gier, FOL rodzi sobie z zadaniem średnio ze skutecznością 72%, natomiast sieć neuronowa ma skuteczność średnio 31%, co można zobaczyć na poniższym wykresie (Wykres 4.1).



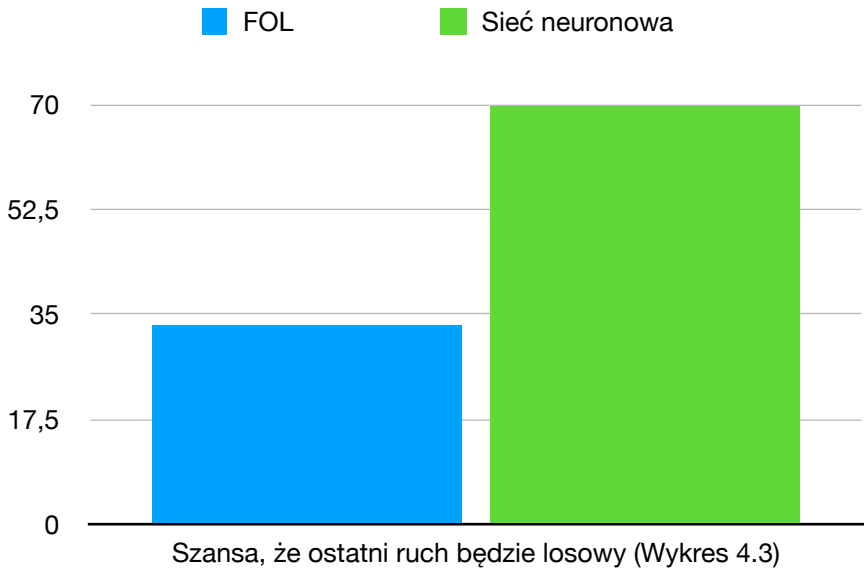
% wygranych gier (Wykres 4.1)

Jeśli mówimy o czasie gry w przypadku poszczególnych agentów, to agent FOL ma dość podobne wartości na każdym rozwiązaniu i mieszczą się one w przedziale od 0.01s do 0.04s, natomiast agent sieci neuronowej podczas kolejnych rozgrywek ma tendencję “skokowości” jeżeli chodzi o czas. Wartości te mieszczą się w przedziale od 0.01 s do 2 s. Zobrazowane jest to na poniższym wykresie (Wykres 4.2).

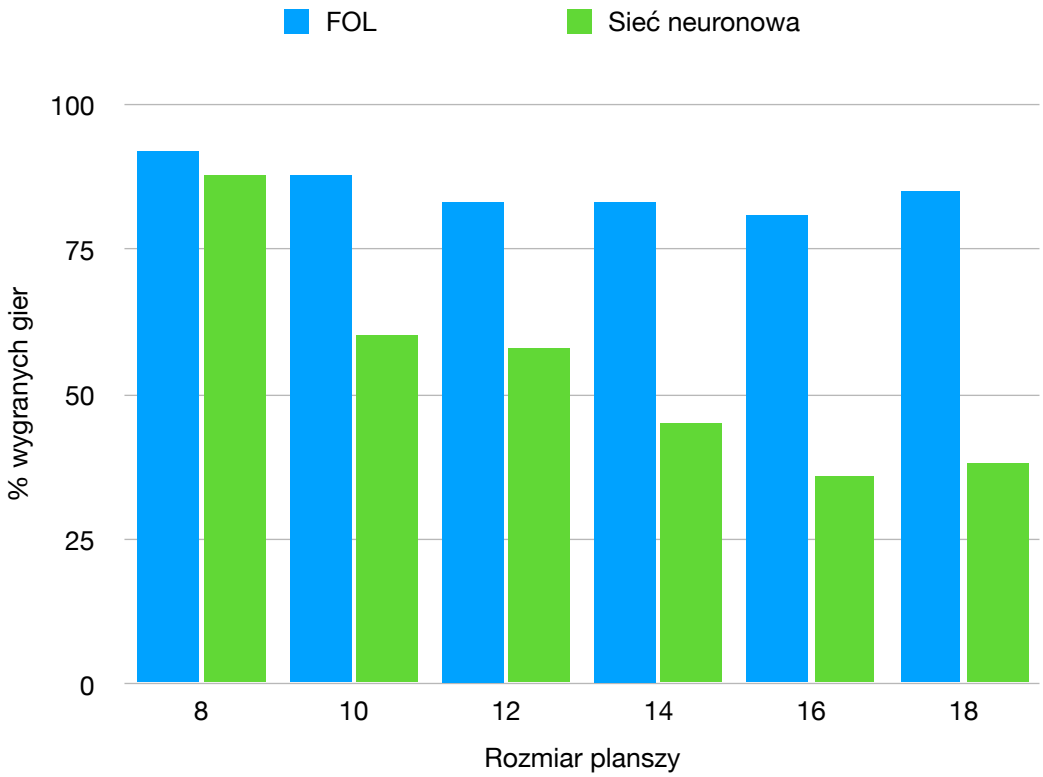


Wykres 4.2

Jeśli chodzi o wybór ostatniego ruchu w przypadku metody FOL, to w 33% przypadków odbywa się ono losowo pola. Wynika to z tego, że w niektórych przypadkach nie można jednoznacznie określić, gdzie będzie usytuowana mina, co wynika z charakterystyki gry. W przypadku metody sieci neuronowej, szansa ta wynosi około 70%. Zobrazowane jest to na wykresie poniżej (Wykres 4.3)



Szansa, że ostatni ruch będzie losowy (Wykres 4.3)



W przypadku porównania procentów wygranych gier w stosunku do rozmiaru zadanej planszy, w każdym rozmiarze wygrywa FOL, aczkolwiek przy rozmiarze planszy 8x8 metody są do siebie zbliżone. Wyniki są uśrednieniem wyników z rozegranych 100 gier przy każdym rozmiarze. Zajęcie planszy przez miny wynosi 7% w tym przypadku. Zobrazowane jest to na wykresie 4.4.

Procent wygranych gier w zależności od rozmiaru planszy (Wykres 4.4)

Podsumowując można wywnioskować ze realizacja metody FOL wyszła nam razy lepiej niż realizacja sieci neuronowej. Aby poprawić wydajność sieci neuronowej, można spróbować dostosować różne hiperparametry, takie jak liczba ukrytych warstw, liczba neuronów, szybkość uczenia się, funkcje aktywacji.