Rok akademicki 2011/2012

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Michał Soczewka

Silnik sztucznej inteligencji wykorzystujący sieci neuronowe na przykładzie bota w strzelance 2D

Opiekun pracy:
dr inż. Tomasz Martyn

Ocena: .....

.....
Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Specjalność:	Inżynieria Systemów Informatycznych
Data urodzenia:	24.09.1988
Data rozpoczęcia studiów:	21.02.2008

Życiorys

Urodziłem się 24 września 1988r. w Rzeszowie. W roku 2004 rozpocząłem naukę w Katolickim Liceum Ogólnokształcącym im. Jana Pawła II w Rzeszowie na profilu matematyczno–fizycznym. W lutym 2008 rozpocząłem studia I stopnia na wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej na kierunku Informatyka w trybie stacjonarnym. Na czwartym semestrze studiów wybrałem specjalność Inżynieria Systemów Informatycznych w Instytucie Informatyki.

.....

Podpis studenta

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu .....

z wynikiem .....

Ogólny wynik studiów: .....

Dodatkowe wnioski i uwagi Komisji: .....

.....

.....

Streszczenie

Celem projektu było stworzenie silnika sztucznej inteligencji na bazie sztucznych sieci neuronowych. Zadanie biblioteki to nauka wzorców zachowań, a główne założenia to uniwersalność i prostota w użyciu. Produkt jest przeznaczony głównie dla gier komputerowych.

Słowa kluczowe:

sztuczna inteligencja, sieci neuronowe, wzorce zachowań, strzelanki

---

Artificial intelligence engine based on neural networks demonstrated on a 2D shooter.

The goal of the project was to build an artificial intelligence engine library using artificial neural networks. The aim of the library is learning behaviour patterns, while being abstract and as easy to use as possible. The project's result is intended mailny for computer games.

Keywords:

artificial intelligence, neural networks, behaviour pattern recognition, shooters

Spis treści

1. Wstęp.

Celem niniejszej pracy inżynierskiej było stworzenie silnika sztucznej inteligencji opartego na sieciach neuronowych. Silnik przeznaczony jest dla sztucznego gracza[1], który uczy się wzorców zachowania swojego przeciwnika.

W ramach projektu działanie silnika zostało zaprezentowane na przykładzie prostej demonstracji, w której bot strzela tak, aby z jak największą skutecznością trafiać w swój cel, omijając przy tym przeszkody. W założeniach moduł silnika jest przenośny i abstrakcyjny. Wykorzystanie go w innej grze nie jest trudnym zadaniem, nawet dla niedoświadczonego programisty.

Powodem wyboru tematu był brak na rynku rozwiązań opartych o sieci neuronowe, mimo całkiem sporego asortymentu gotowych, komercyjnych silników. Kolejnym powodem były stale rozwijające się możliwości obliczeniowe komputerów osobistych, przez co takie rozwiązania daje się zastosować w praktyce. Ostatnim powodem była chęć poszerzenia własnej wiedzy z zakresu sztucznej inteligencji, rozwoju umiejętności programistycznych i zainteresowanie tematyką sztucznych sieci neuronowych.

[1] „Sztuczny gracz" nazywany jest w literaturze „agentem". Jednak, ponieważ ani pierwsza, ani druga nazwa nie zostały dobrze przyjęte przez społeczność Internetu, dlatego w dalszej części niniejszej pracy użyto potocznego określenia „bot".

2. Powszechnie stosowane metody sztucznej inteligencji.

2.1. Przykłady metod, algorytmów i struktur danych.

Metody sztucznej inteligencji wykorzystywane w grach dostępnych na dzisiejszym rynku można podzielić na dwie kategorie:

- rozwiązania skryptowe,
- uczenie się maszyn.

Oba te podejścia zostały poniżej krótko omówione i porównane.

2.1.1. Sztucznej inteligencja oparta na skryptach.

Większość gier wykorzystuje sztuczną inteligencję opartą na skryptach. Takie rozwiązania najczęściej realizuje się za pomocą języków skryptowych (najczęściej Lua lub Python).

Zaletą ich stosowania jest przede wszystkim łatwa implementacja. Skrypty pisze się o wiele szybciej niż programy lub biblioteki. Ponadto często zmieniane moduły gry wydzielane są na zewnątrz kodu kompilowanego, aby umożliwić szybką korektę błędów bez potrzeby ponownej kompilacji programu.

Wadą natomiast jest to, że wynik działania skryptu jest w pełni deterministyczny. Przy takich rozwiązaniach gracz bardzo często jest w stanie wypracować taktykę, przy pomocy której **zawsze** wygra.

2.1.2. Sztuczna inteligencja bez wykorzystania doświadczenia.

Do rozwiązania standardowych problemów istnieją już od bardzo dawna gotowe i dobrze sprawdzające się algorytmy i struktury danych. Przykładami są tu:

- przycinanie alfa-beta (gry deterministyczne, na przykład szachy),
- algorytm A* (znajdowanie ścieżki),
- metody Monte Carlo (aproksymacja prawdopodobieństw zdarzeń losowych),
- automaty stanów skończonych (maszyny stanów),
- logika rozmyta (przedstawianie danych dyskretnych w sposób ciągły) [2, 7, 8].

2.1.3. „Uczące się" algorytmy oparte na doświadczaniu.

„Uczenie się" maszyn nie jest często stosowaną metodą realizacji sztucznej inteligencji w grach komputerowych ze względu na wymagany duży nakład obliczeń i trudność realizacji. Przykładami używanych tu algorytmów i struktur danych są:

- maszyny wektorów nośnych, algorytmy k-NN (klasyfikacja obiektów),
- algorytmy genetyczne i ewolucyjne (optymalizacja),
- uczenie ze wzmacnianiem, Q-learning (opracowywanie taktyki),
- sieci neuronowe (aproksymacja funkcji) [1, 2, 3, 5].

2.2. Porównanie wymienionych rodzajów sztucznej inteligencji.

W odróżnieniu od skryptów uczenie się na doświadczaniu charakteryzuje się tym, że możliwe odpowiedzi cały czas się zmieniają, dążąc do jak najlepszego rezultatu. Przy ich zastosowaniu nie da się już łatwo przewidywać wyników. Niestety rozwiązania takie nie tylko są czaso–, pamięcio–, bądź obliczeniochłonne, ale również, ze względu na ich naturę, nie są uniwersalne. Nie istnieje jedno rozwiązanie pasujące idealnie do każdego problemu.

Często takie algorytmy i struktury danych zamyka się w oddzielnych modułach zwanych silnikami. Implementuje się je w językach kompilowanych. Najczęściej jest to C++ (wybierany jest on ze względu na szybkość działania kodu i możliwości efektywnego zarządzania pamięcią).

2.3. Przegląd dostępnych komercyjnych rozwiązań.

Tworząc grę komputerową nie trzeba implementować skomplikowanej sztucznej inteligencji od podstaw. Można wykorzystać gotowy, wspierany i rozwijany silnik. Przykładami najpopularniejszych komercyjnych rozwiązań są:

- AI.implant (firmy Pregasis)[2],
- SPIROPS A.I.[3],
- Artificial Contender (firmy Tru Soft)[4],

[2] `http://www.presagis.com/products_services/products/simulation/aiimplant/more/aiimplant_for_games`
[3] `http://www.spirops.com/products.php`
[4] `http://www.trusoft.com/ac_overview.html`

- Live AI[5],
- xaitMap i xaitControl (firmy xaitment)[6],
- EKI One (firmy MASA)[7],
- Kynapse (firmy Autodesk)[8],
- Unreal Engine (firmy Epic Games)[9].

Przykłady możliwości wyżej wymienionych silników to:

- tworzenie postaci niezależnych, modelowanie zachowań postaci niezależnych,
- automatyczne rozwijanie fabuły gry na podstawie działań gracza,
- nauka zachowań, adaptacja,
- efektywne poruszanie się w terenie, znajdowanie ścieżek,
- określanie percepcji postaci,
- automatyczna akwizycja danych.

[5] http://ailive.net/liveAI.html
[6] http://www.xaitment.com/english/product-overview.html
[7] http://www.ekione.com/
[8] http://gameware.autodesk.com/kynapse
[9] http://www.unrealengine.com/

3. Wprowadzenie teoretyczne do sieci neuronowych.

W projekcie została wykorzystana sieć neuronowa zwana *siecią jednokierunkową* z nauką algorytmem *wstecznej propagacji błędu*. Niżej zamieszczony opis teoretyczny skupia się na tym właśnie typie. Rodzajów sieci jest o wiele więcej, ale w kontekście tej pracy ich prezentacja nie jest potrzebna.

3.1. Czarnoskrzynkowy model sieci neuronowej.

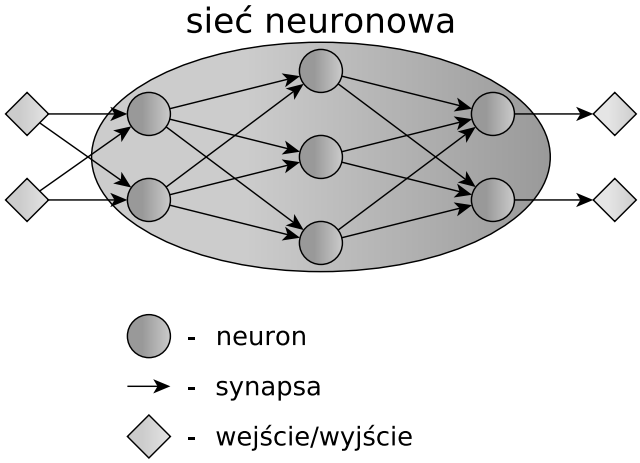
Sztuczna sieć neuronowa to opis matematyczny, programowa struktura danych, bądź sprzętowa implementacja modelu realizującego przetwarzanie danych przez rzędy elementów zwanych *neuronami* ułożonych w *warstwy*, połączonych między sobą tak zwanymi *synapsami*. Sztuczne neurony na podanych na ich wejścia danych, zwanych *pobudzeniami*, wykonują pewną operację zwaną *funkcją aktywacji*.

Rys. 1. Czarnoskrzynkowy model sieci neuronowej.

Historycznie inspiracją tego modelu były badania nad budową mózgu, pod koniec XIX wieku.

3.2. Opis budowy sieci neuronowej.

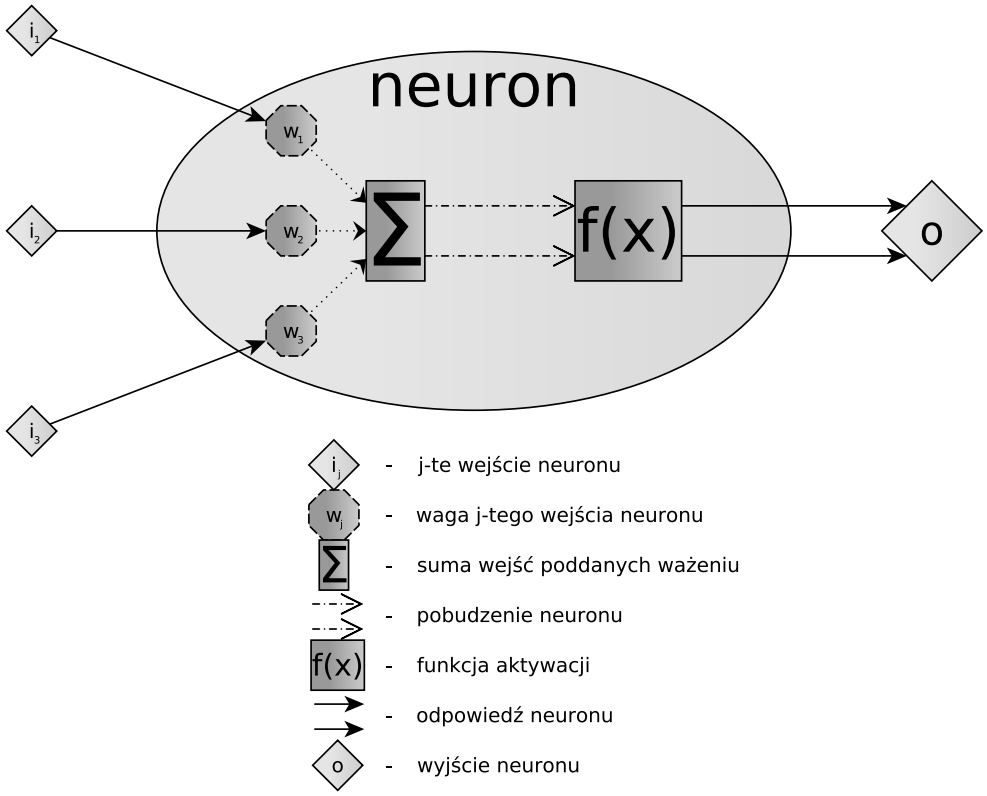
Poniższe rysunki przedstawiają ogólny schemat budowy sieci neuronowej i pojedynczego neuronu.



Rys. 2. Budowa sieci neuronowej.

Każdy neuron składa się z:

- wejść,
- wagi każdego z wejść,
- funkcji aktywacji,
- wyjścia.



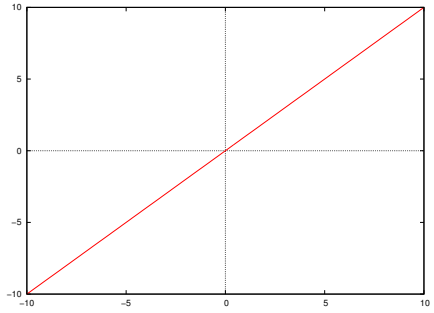
Rys. 3. Budowa pojedynczego neuronu.

3.2.1. Funkcja aktywacji.

Funkcja aktywacji jest częścią składową neuronu. Jej wartość dla danego pobudzenia jest odpowiedzią neuronu na to pobudzenie. W przypadku ogólnym każdy neuron może mieć dowolną funkcję aktywacji. Zazwyczaj przyjmuje się jednak te same funkcje, o jednakowych parametrach w obrębie całej warstwy, a często nawet i całej sieci. Poniżej zostały wymienione funkcje najczęściej stosowane w sztucznych sieciach neuronowych.

- funkcja liniowa:

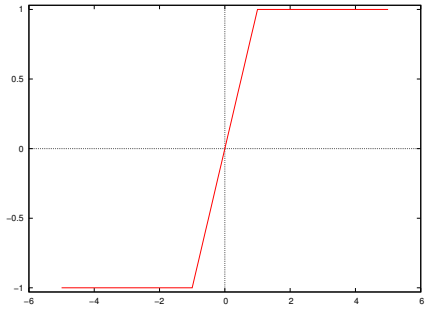
$$f(ax) = x$$



Rys. 4. Funkcja liniowa.

funkcja liniowa obcięta:

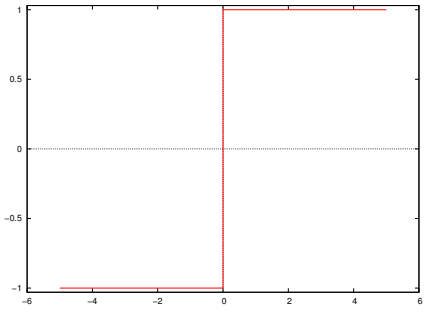
$$f(x) = \begin{cases} -1 & , \ x < -1 \\ ax & , \ \frac{-1}{a} \leqslant x \leqslant \frac{1}{a} \\ 1 & , \ x > 1 \end{cases}$$



Rys. 5. Funkcja obcięta.

- funkcja progowa bipolarna:

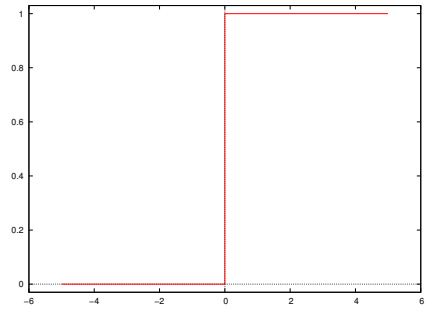
$$f(x) = \begin{cases} 0 & , \ x \leqslant 0 \\ 1 & , \ x > 0 \end{cases}$$



Rys. 6. F. progowa bipolarna.

funkcja progowa unipolarna:

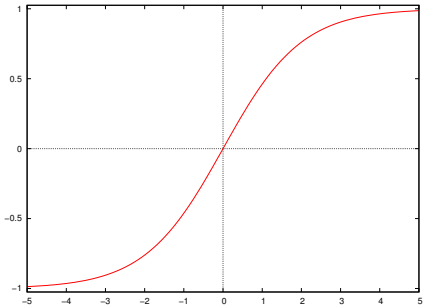
$$f(x) = \begin{cases} 0 & , \ x \leqslant 0 \\ 1 & , \ x > 0 \end{cases}$$



Rys. 7. F. progowa unipolarna.

- funkcja sigmoidalna bipolarna:

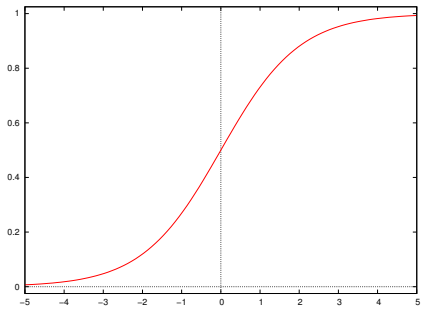
$$f(x) = \frac{2}{1+e^{-\beta x}} - 1 = \\ = \tanh(\beta x)$$



Rys. 8. F. sigmoidalna bipolarna.

funkcja sigmoidalna unipolarna:

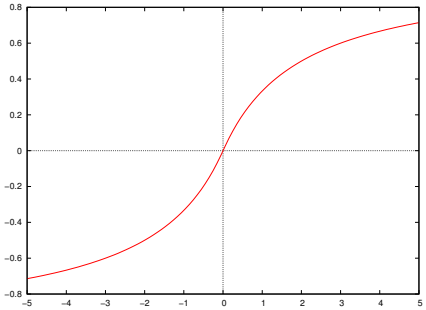
$$f(x) = \frac{1}{1+e^{-\beta x}} = \\ = \frac{\tanh(\beta x)+1}{2}$$



Rys. 9. F. sigmoidalna unipolarna.

- funkcja Elliott'a bipolarna [6]:

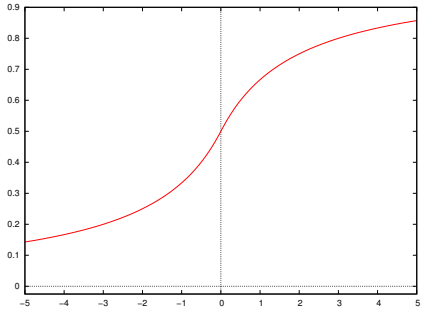
$$f(x) = \frac{xs}{1+|xs|}$$



Rys. 10. F. Elliott'a bipolarna.

funkcja Elliott'a unipolarna [6]:

$$f(x) = \frac{\frac{1}{2}xs}{1+|xs|} + \frac{1}{2}$$

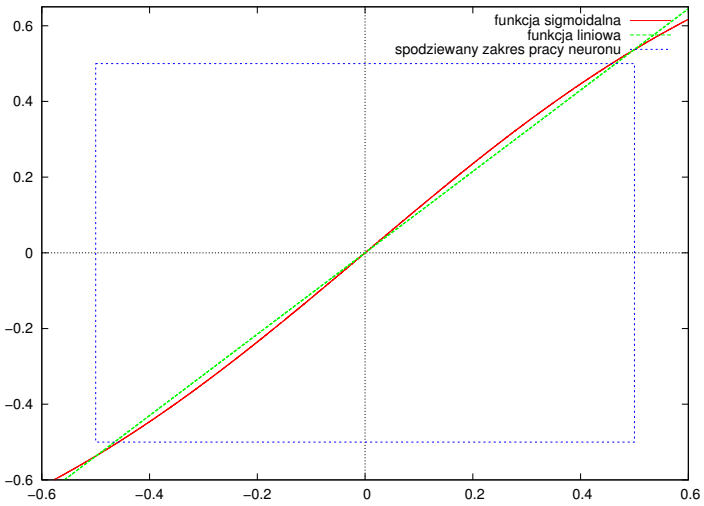


Rys. 11. F. Elliott'a unipolarna.

3.2.2. Funkcja sigmoidalna a inne funkcje.

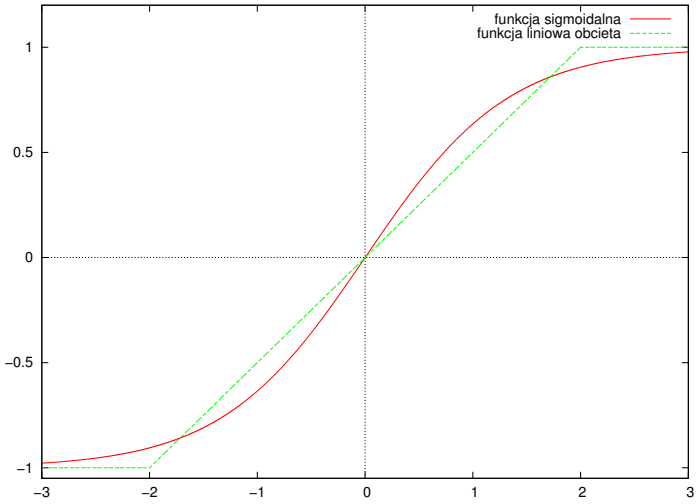
Uniwersalną funkcją aktywacji jest funkcja sigmoidalna. Jest tak, ponieważ przy odpowiednim doborze parametrów przybiera ona kształty podobne do innych funkcji.

- Przy $\beta \to 0$ funkcja sigmoidalna przybiera kształt podobny do funkcji liniowej.



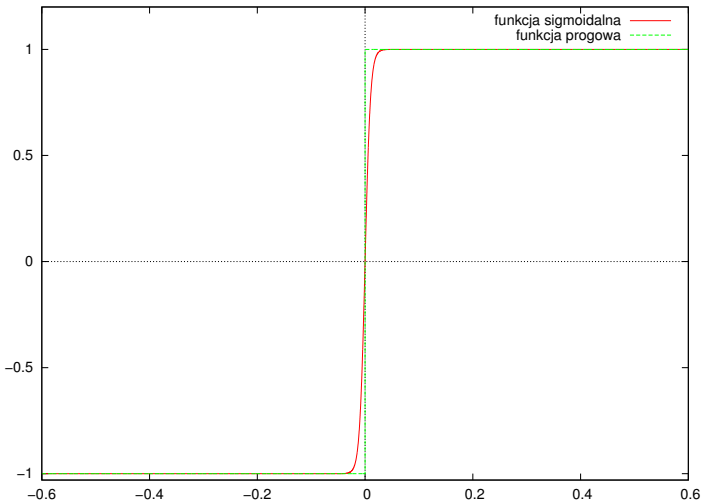
Rys. 12. Porównanie funkcji sigmoidalnej i liniowej.

- Przy $\beta \ll \infty$ funkcja sigmoidalna przybiera kształt podobny do funkcji liniowej obciętej.



Rys. 13. Porównanie funkcji sigmoidalnej i liniowej obciętej.

- Przy $\beta \to \infty$ funkcja sigmoidalna przybiera kształt podobny do funkcji progowej.



Rys. 14. Porównanie funkcji sigmoidalnej i progowej.

Z powodu swojej uniwersalności funkcja ta jest najczęściej wybieraną do prostych zastosowań. Z reguły przyjmuje się $\beta \in (0; 1]$, na przykład 0.75.

3.2.3. Zastosowanie funkcji liniowej i progowej.

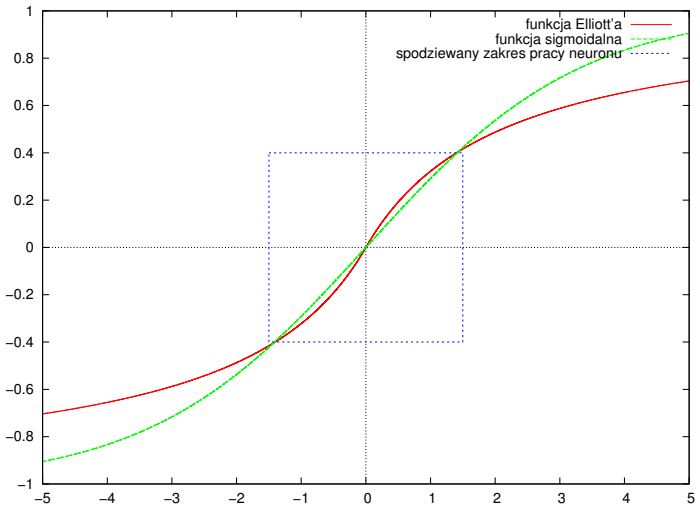
Patrząc z perspektywy wymaganego nakładu obliczeń funkcja sigmoidalna jest dość skomplikowana (funkcja tanh wymaga policzenia exp). Dlatego w niektórych sytuacjach upraszcza się obliczenia, stosując funkcję liniową lub progową. Rozwiązanie to nie jest ogólne. Działa tylko w szczególnych sytuacjach, na przykład:

- przeskalowanie wejść sieci neuronowej — do tego celu w warstwie neuronów wejściowych stosuje się funkcję liniową z odpowiednio dobranym parametrem skalującym,
- sieć neuronowa z wyjściem binarnym — w takim przypadku w warstwie neuronów wyjściowych stosuje się funkcję progową.

Jednak w większości przypadków w warstwie ukrytej umieszcza się zawsze neurony sigmoidalne.

3.2.4. Funkcja sigmoidalna a wydajność.

Dobrą alternatywą dla funkcji sigmoidalnej jest funkcja Elliott'a. Policzenie jej wartości jest o wiele szybsze. Wymaga jedynie prostej wartości bezwzględnej i ilorazu, zamiast czasochłonnej eksponenty. Kształty obu tych funkcji są bardzo zbliżone.



Rys. 15. Porównanie funkcji sigmoidalnej i Elliott'a.

W ramach pracy przeprowadzono test porównujący te dwie funkcje. Polegał on na zestawieniu różnic pomiędzy wartościami zegara systemowego[10], używając najpierw jednej funkcji, a następnie drugiej. Wynik wykazał, iż funkcja Elliott'a jest o niecałe 39% szybsza. Z tego powodu w projekcie wykorzystano właśnie tę funkcję.

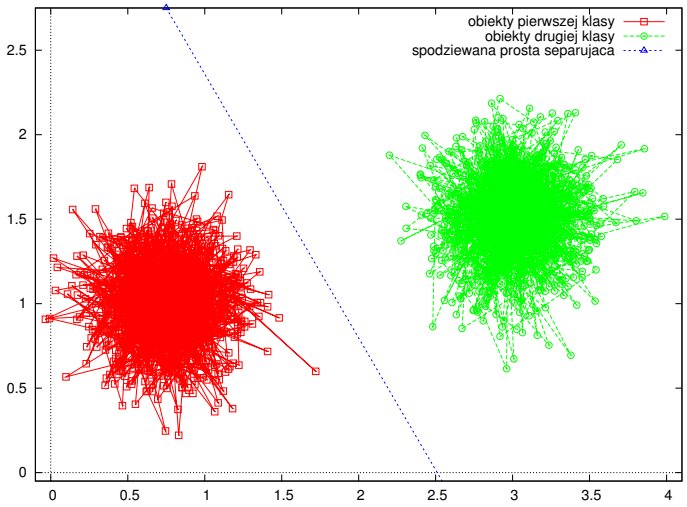
3.2.5. „Bias".

Sieć neuronowa opisana wyżej nie będzie działać w każdym przypadku. Brakuje jej pewnego elementu określanego po angielsku *„bias"* — przesunięcie, ukierunkowanie, stronniczość. Polega to na tym, że do zestawu wejść każdego neuronu dodaje się jedno, na które podawana jest zawsze stała wartość 1. Wejście to również podlega ważeniu.

Aby zobrazować ten problem przeanalizujmy przykład klasyfikatora liniowego. Załóżmy, że klasyfikację rozwiązujemy za pomocą sieci neuronowej. Na

[10] Dokładność takiego pomiaru jest rzędu nanosekund.

poniższym rysunku punkty czerwone i zielone reprezentują elementy należące do dwóch rozdzielnych klas obiektów. Niebieska linia to spodziewana prosta separująca klasy.



Rys. 16. Przykład klasyfikatora liniowego.

Jeśli sieć nie miałaby wejścia *bias* to realizowałaby funkcję liniową $f(x) = ax$. Widać zatem, że sieć taka nigdy nie rozdzieliłaby elementów na dwie klasy, ponieważ brakuje jej parametru b.

3.3. Działanie sieci neuronowej.

3.3.1. Algorytm obliczania odpowiedzi.

Dla pojedynczego neuronu jego odpowiedzią jest wartość funkcji aktywacji w punkcie pobudzenia. Z kolei pobudzenie to suma wejść z uwzględnieniem wag:

$$stimulation = \sum_{i=1}^{n} input_i \cdot weigh_i =$$
$$= INPUTS \circ WEIGHS,$$

gdzie:

$$INPUTS = [input_1, \ \dots \ , input_n],$$

a:

$$WEIGHS = \begin{bmatrix} weigh_1 \\ \vdots \\ weigh_n \end{bmatrix}.$$

Odpowiedzią neuronu jest w takim razie:

$$\begin{aligned} output &= f(stimulation) = \\ &= f(INPUTS \circ WEIGHS). \end{aligned}$$

Patrząc na budowę sieci nie trudno dojść do wniosku, że obliczenie jej odpowiedzi sprowadza się do obliczenia odpowiedzi warstw, podając wyjścia poprzedniej na wejścia kolejnej. Z kolei obliczenie odpowiedzi pojedynczej warstwy sprowadza się do obliczenia odpowiedzi każdego z należących do niej neuronów. Zapisując to na macierzach, dla jednej warstwy mamy:

$$LAYER\_OUTPUTS = f(LAYER\_INPUTS \cdot LAYER\_WEIGHS^T),$$

gdzie:

$$LAYER\_WEIGHS = \begin{bmatrix} NEURON\_WEIGHS_1 \\ \vdots \\ NEURON\_WEIGHS_n \end{bmatrix}.$$

Dalej można zapisać wzór na odpowiedź całej sieci:

$$OUTPUTS = f(\dots f(INPUTS \cdot WEIGHS_1) \cdot \dots WEIGHS_n),$$

gdzie:

- $OUTPUTS$ — odpowiedź sieci,
- $INPUTS$ — wejście sieci,
- $WEIGHS_i$ — macierz wag i-tej warstwy.

Z powyższego wzoru wynika, że policzenie odpowiedzi sieci sprowadza się do mnożenia macierzy i obliczania wartości funkcji aktywacji. Zatem złożoność obliczeniowa określenia odpowiedzi sieci trójwarstwowej jest równa sumie złożoności obliczania odpowiedzi każdej z warstw, czyli:

$$InSz^2 \cdot NumInp \ + \ NumInp^2 \cdot NumHid \ + \ NumHid^2 \cdot NumOut,$$

gdzie:

- $InSz$ — rozmiar wektora wejściowego (liczba parametrów),
- $NumInp$ — rozmiar warstwy wejściowej (stopień złożoności sieci),
- $NumHid$ — rozmiar warstwy ukrytej (stopień złożoności sieci),
- $NumOut$ — rozmiar warstwy wyjściowej (liczba wyjść).

Wszystkie te parametry są ze sobą w jakiś sposób skorelowane. Można przyjąć uproszczenie, że złożoność obliczeniowa wynosi

$$\theta(3 \cdot N^3) = \theta(N^3),$$

gdzie N jest pewną liczbą określającą stopień złożoności problemu i rozmiar używanej sieci neuronowej.

3.3.2. Nauka sieci neuronowej.

Uczenie algorytmem wstecznej propagacji jest bardzo podobne do liczenia odpowiedzi. Nazwa algorytmu pochodzi od sposobu jego wykonania. Działa on od końca sieci (od strony wyjść) oraz pracuje na wartości błędu (pomyłki) wyniku.

Algorytm polega na tym, że znając poprawny wynik dla pewnego zestawu danych wejściowych jesteśmy w stanie poprzestawiać wagi tak, aby kolejne obliczenie dla tego wejścia dało odpowiedź bliższą tej oczekiwanej.

Nauka sieci algorytmem wstecznej propagacji błędu:

1. Mamy zestaw danych wejściowych (zwany *zbiorem uczącym*), dla których znamy poprawne odpowiedzi.

2. Każdą z próbek podajemy na wejście sieci i uzyskujemy jej wersję odpowiedzi.

3. Odejmujemy od uzyskanych odpowiedzi odpowiedzi poprawne. Są to wartości błędu (pomyłki) sieci.

4. Średnia kwadratów wartości pomyłki dla każdej z próbek nazywamy *błędem średnim kwadratowym* (ang. *MSE — mean square error*). Ta wartość jest używana do opisu jakości przystosowania sieci do danego zbioru uczącego.

5. Dla każdego neuronu, dla każdego połączenia:

    - przekazujemy błąd do warstwy poprzedniej,

- do wagi dodajemy błąd przemnożony przez aktualną wagę, ostatnią wartość pobudzenia i tak zwany *współczynnik nauki*.

Ważnym jest, aby najpierw przekazać błąd do poprzedniej warstwy, a dopiero potem uaktualniać wagi. Zapominanie o tym fakcie jest często popełnianym błędem.

Współczynnik nauki (ang. *threshold*) jest dodatnią stałą rzeczywistą. Zwyczajowo przyjmuje się, że $threshold \in (0; 1)$. Służy on do dostrojenia czułości sieci na zmiany i szybkości uczenia się. Mniejszy współczynnik oznacza, że sieć wolniej reaguje. Z kolei jego przestrojenie może skutkować tym, że sieć nigdy nie dojdzie do oczekiwanego stanu, ponieważ zmiana wag będzie się odbywała o zbyt wysokie wartości.

3.4. Podsumowanie opisu sieci neuronowych.

Sztuczna sieć neuronowa jako całość realizuje pewną funkcję $\mathbb{R}^N \to \mathbb{R}^M$. Kształt funkcji zależy od parametrów sieci i może być bardziej lub mniej skomplikowany w zależności od złożoności sieci — liczby neuronów i funkcji aktywacji. Ponieważ właściwie jedynym ograniczeniem funkcji możliwych do uzyskania jest ich ciągłość, można uzyskać w zasadzie dowolną funkcję. Struktury te często stosuje się do aproksymacji funkcji o nieznanych przebiegach.

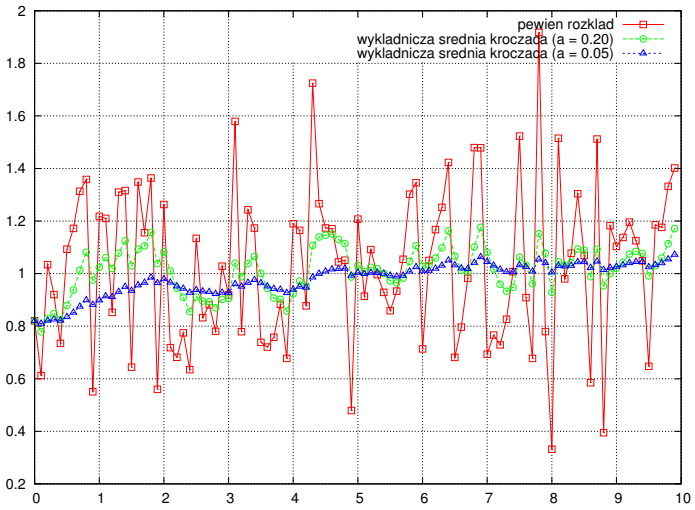
Wyżej wykazano, że obliczenie odpowiedzi sieci to pomnożenie macierzy i policzenie wartości funkcji. Wykazano również, że operacja taka ma złożoność algorytmiczną $\theta(n^3)$. Uwzględniając fakt, że zazwyczaj neuronów

w sieci jest stosunkowo mało, to taka złożoność nie jest dużym problemem dla dzisiejszych komputerów.

4. Wprowadzenie teoretyczne do średnich kroczących.

Wewnątrz projektu wykorzystano średnie kroczące do przewidywania czasu nauki i do określania kiedy możliwości danego zbioru uczącego zostały wyczerpane. W tym rozdziale zostało przedstawionych kilka rodzajów średnich kroczących i zastosowanie tych tworów w dzisiejszym świecie.

Średnia krocząca jest miarą statystyczną służącą do analizy zbioru danych poprzez ciągłe obliczanie pewnych umownie średnich wartości tego zbioru w kolejnych punktach. Wielkość ta wywodzi się z ekonomii i jest stosowana do uśrednienia w czasie wartości instrumentu na rynku. Znalazła ona również zastosowanie w informatyce. Często stosuje się ją do analizy wydajności systemów komputerowych i jakości usług (tak zwane *SLA*[11]).



Rys. 17. Przykład średniej kroczącej.

[11] **SLA** — ang. Service Level Agreement. Umowa pomiędzy usługodawcą a usługobiorcą dotycząca poziomu jakości usługi. W języku polskim nie istnieje unormowany odpowiednik tego terminu.

4.1. Prosta średnia krocząca.

Najprostszym przykładem średniej kroczącej jest średnia arytmetyczna z ostatnich n próbek, zwana *prostą średnią kroczącą*. Średnia taka dla funkcji f w punkcie czasu t wynosi

$$avg(f,t,n) = \begin{cases} \frac{\sum_{i=t-n-1}^{t-1} f(i)}{n} & , \ t \geqslant n \\ \frac{\sum_{i=0}^{t-1} f(i)}{t} & , \ t < n \end{cases}.$$

4.2. Wykładnicza średnia krocząca.

Prosta średnia krocząca ma tę wadę, że dla parametru n trzeba pamiętać n ostatnich próbek. W projekcie wykorzystano tak zwaną *wykładniczą średnią kroczącą*, która wymaga pamiętania jedynie ostatniej wartości średniej. Dla funkcji f i pewnego parametru α w punkcie t wyraża się ją rekurencyjnym wzorem

$$ema(f,t,\alpha) = \begin{cases} f(0) & , \ t = 0 \\ (1-\alpha) \cdot ema(t-1) + \alpha \cdot f(t) & , \ t > 0 \end{cases}.$$

Średnia ta porównywana jest z odpowiedzią kondensatora na impuls jednostkowy. Parametr α oznacza jak duży wpływ na wartość średniej ma pojedyncza próbka. Często przekształcany jest on na łatwiejszy do opanowania przez człowieka parametr N przy pomocy równania: $\alpha = \frac{2}{N+1}$. Ten z kolei porównywany jest do czasu połowicznego rozpadu pierwiastków chemicznych.

5. Projekt inżynierski — biblioteka silnika sztucznej inteligencji.

5.1. Realizacja projektu i użyte biblioteki.

Projekt został wykonany w możliwie przenośnej technologii. Moduł silnika sztucznej inteligencji wykorzystuje biblioteki standardowe. Obliczenia neuronowe początkowo były przeprowadzane na własnej implementacji sieci, jednak później zostało to zastąpione biblioteką FANN[12] [4]. Gra demonstracyjna została napisana z wykorzystaniem biblioteki Qt[13]. Nauka silnika została zrealizowana wielowątkowo, wykorzystując autorską bibliotekę opakowującą wątki POSIX i zmieniającą parametry szeregowania odpowiadających tworzonym wątkom procesów LWP w jądrze systemu operacyjnego.

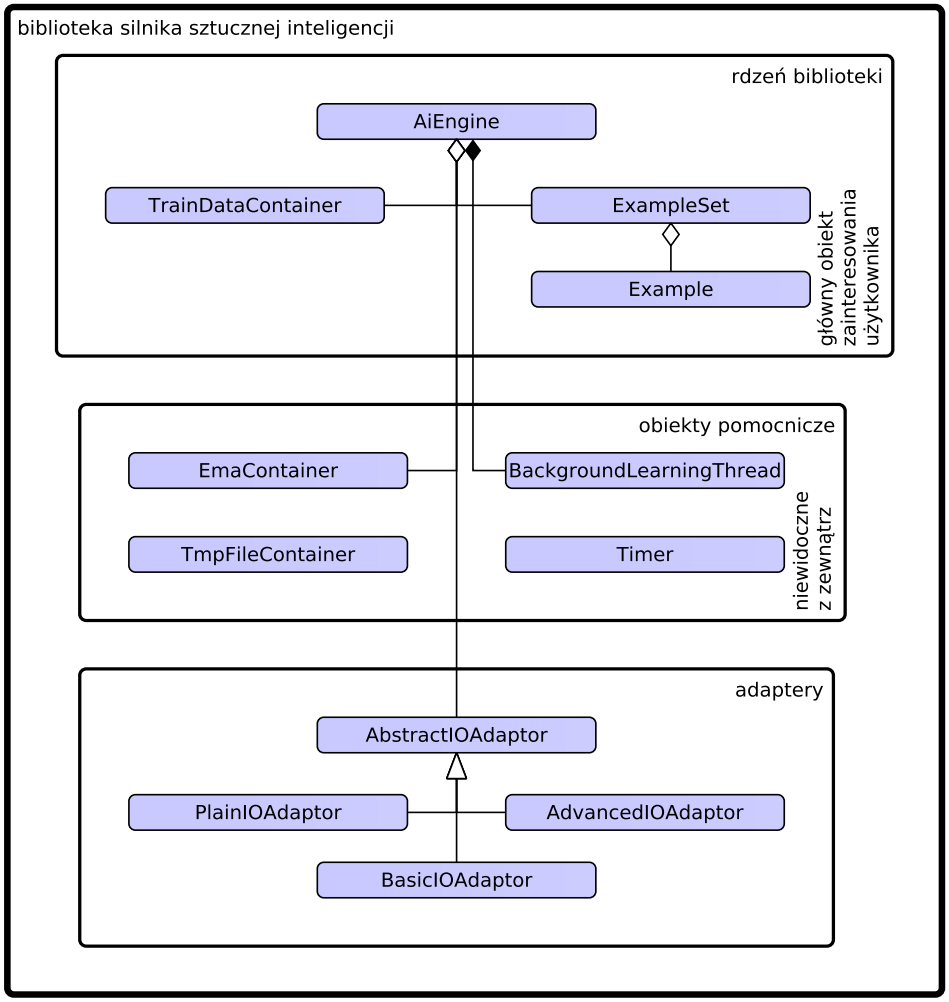
Projekt był tworzony i testowany na systemie Linux. Działanie na systemie Windows nie było sprawdzane, ale nie ma żadnych przesłanek, aby projekt na tym systemie się nie zbudował.

5.2. Moduł silnika sztucznej inteligencji.

Głównym założeniem tego modułu była możliwie największa ogólność. Z tego powodu silnik przyjmuje na wejście stan świata gry w postaci wektora liczb i odpowiada na wyjściu również wektorem. Od osoby tworzącej grę nie jest wymagana znajomość działania tego modułu, a jedynie dostarczenie stanu świata i odebranie odpowiedzi w ustalonym formacie.

[12] `http://sourceforge.net/projects/fann`
[13] `http://qt.nokia.com`

Na poniższym diagramie została przedstawiona architektura modułu silnika.



Rys. 18. Diagram architektury biblioteki silnika.

W celu użycia biblioteki nie jest wymagane roszerzanie jej kodu. Jest jednak zalecane, aby zgodnie ze wskazówkami autora stworzyć kilka klas–adapterów w celu optymalizacji działania silnika i poprawy elegancji kodu źródłowego tworzonej gry. Zagadnienie to zostało opisane w dalszej części pracy.

5.3. Przedstawienie możliwości silnika.

Niżej zostało przedstawione ogólne spojrzenie na możliwości i zalety biblioteki silnika sztucznej inteligencji.

1. Funkcyjna aproksymacja reakcji na wzorce przy pomocy sztucznych sieci neuronowych.
2. Całkowita enkapsulacja sieci neuronowej. Jest ona niewidoczna dla użytkownika biblioteki.
3. Automatyczne określanie struktury sieci neuronowej.
4. Nauka na przygotowanym wcześniej zbiorze danych.
5. Nauka na bieżąco[14] na danych zbieranych podczas działania programu.
6. Możliwość nauki w tle.
7. Liczenie statystyk na podstawie *wykładniczych średnich kroczących*.
8. Automatyczne ograniczanie rzeczywistego czasu nauki[15] na podstawie zebranych statystyk.
9. Automatyczne sugerowanie momentu zakończenia nauki na podstawie zebranych statystyk.
10. Wydajne działanie i możliwość jego zrównoleglania.

[14] W teorii sieci neuronowych do określenia nauki „na bieżąco” stosuje się termin „on-line”.

[15] Wewnątrz biblioteki czas jest rzeczywisty, pobierany z zegara systemowego, niezależny od żadnych zewnętrznych czynników.

5.4. Minimum wiedzy do użycia biblioteki we własnej grze.

Poniżej przedstawiono potrzebne minimum wiedzy i minimum wkładu własnego kodu, aby użyć biblioteki silnika w swojej grze. Dołączone zostały również krótkie opisy poszczególnych elementów bibliotecznych, w celu przedstawienia sposobu ich działania oraz przykłady kodu źródłowego.

5.4.1. Adaptery.

Silnik zwraca odpowiedzi w postaci wektora wartości w przedziale $[-1; +1]$. Jeśli istnieje potrzeba, aby wyście było w innej postaci, na przykład w szerszym zakresie, to trzeba je odpowiednio przeskalować. Silnik robi to automatycznie, pod warunkiem dostarczenia mu przy inicjacji obiektu *adaptera*.

Domyślnym adapterem jest `PlainIOAdaptor`, którego implementacja nie zmienia wartości (jest pusta). Standardowym i uniwersalnym adapterem jest `BasicIOAdaptor`, który przeskalowuje wartości do podanego zakresu (ten jest zalecany do najbardziej podstawowego użycia). Dostępne są jeszcze dwa bardziej zaawansowane adaptery:

- `AdvancedIOAdaptor` — przesuwa i przeskalowuje wartości na podstawie dostarczonych wielkości statystycznych, takich jak wartość średnia i odchylenie standardowe. Wartości te muszą być znane z góry.
- Istnieje jeszcze możliwość stworzenia własnego adaptera. Musi on rozszerzyć klasę bazową `AbstractIOAdaptor` i uzupełnić implementację

funkcji `adaptI`, `adaptO` oraz `clone`. Funkcje te są używane do przeskalowania wektorów wartości odpowiednio wchodzących lub wychodzących z silnika i do kopiowania obiektu. Przykład sposobu implementacji takiego obiektu można podejrzeć w kodzie `BasicIOAdapter`'a.

5.4.2. Inicjacja silnika.

Utworzenie obiektu silnika jest proste. Wystarczy:

- znać ilość wejść i wyjść,
- stworzyć adapter dla wyjścia (lub pamiętać, że zwracane wartości będą w przedziale $[-1; +1]$),
- wymyślić nazwę (dzięki temu zapisane pliki są łatwiejsze do rozpoznania, ale nie jest to obowiązkowe),
- wczytać silnik z pliku (przy pomocy funkcji `load`) lub wylosować jego początkowe parametry (przy pomocy funkcji `initialize`).

Poniżej znajduje się przykład kodu tworzącego obiekt silnika celującego w poruszający się obiekt.

```cpp
/** Tworzenie obiektu silnika. */

#include <ai-engine/ai-engine.hpp>

using namespace ai_engine;
using namespace ai_engine::adaptor;

EngineIO_t outAdMaxVal; // Obiekt tymczasowy przechowujący
                        //   maksymalne wartości kąta.
outAdMaxVal.push_back(M_PI);  // Kąt zawiera się
                              //   w przedziale [-M_PI; +M_PI).
AbstractIOAdaptor* angleAdaptor          // Adapter dla kąta
    = new BasicIOAdaptor(outAdMaxVal);   //   (wyjściowy).

AiEngine silnik_celujacy(
    6,                      // Sześć wejść: po dwie współrzędne
                            //   na położenie i prędkość celu
                            //   oraz dwie na położenie lufy.
    1,                      // Jedno wyjście - kąt nachylenia lufy.
    new PlainIOAdaptor(),   // Adapter wejściowy
                            //   (wystarczy domyślny).
    outputAdaptor,          // Wcześniej utworzony
                            //   adapter wyjściowy.
    "celowanie");           // Nazwa. Będzie ona doklejana
                            //   do nazw plików.

// Wczytanie stanu silnika z poprzedniego uruchomienia:
silnik_celujacy.load("plik_ze_stanem_silnika");

// Wczytanie zbioru uczącego, zebranego wcześniej
// (na przykład podczas poprzedniej gry):
silnik_celujacy.trainData().load("plik_ze_zbiorem_uczacym");
```

Listing 1. Tworzenie obiektu silnika.

5.4.3. Zbieranie przykładów.

Aby dodać nowy przykład uczący wystarczy wywołać funkcję `addExample`. Wartości zostaną automatycznie przeliczone, zgodnie z dostarczonymi wcześniej adapterami.

```
/** Dodawanie przykładu uczącego. */

#include <ai-engine/ai-engine.hpp>

using namespace ai_engine;

// Załóżmy, że obiekt Bullet ma publiczne funkcje:
//   getPosX, getPosY, getVelX, getVelY,
//   getSrcX, getSrcY, oraz getAngle,
//   zwracające współrzędne położenia, składowe prędkości,
//   współrzędne miejsca wystrzelenia
//   oraz kąt, pod jakim dany pocisk został wystrzelony.
// Załóżmy, że obiekt Target posiada referencję:
//   AiEngine& engine,
//   wskazującą na obiekt silnika.

// Załóżmy, że poniższa funkcja jest wywoływana
//   w momencie trafienia pocisku w cel.
void Target::onCollisionWithBullet(
    const Bullet& bullet) {

  // Wektor wartości wejściowych:
  EngineIO_t inputs;
  inputs.push_back(bullet.getPosX());
  inputs.push_back(bullet.getPosY());
  inputs.push_back(bullet.getVelX());
  inputs.push_back(bullet.getVelY());
  inputs.push_back(bullet.getSrcX());
  inputs.push_back(bullet.getSrcY());

  // Oczekiwane wyjście:
  EngineIO_t desiredOutput;
  desiredOutput.push_back(bullet.getAngle());

  // Dodanie przykładu uczącego:
  engine.addExample(
      Example(inputs, desiredOutput));
}
```

Listing 2. Dodawanie przykładu uczącego.

5.4.4. Nauka silnika.

W celu rozpoczęcia procesu uczącego silnik należy najpierw przygotować zbiór uczący. Można to zrobić na kilka sposobów:

- załadować zbiór z pliku — za pomocą funkcji `load`,
- przenieść przykłady uczące do zbioru uczącego — za pomocą funkcji `merge`,
- przenieść przykłady z istniejącego już w pamięci programu zbioru — za pomocą konstruktora kopiującego lub funkcji `merge`.

Zbiór przykładów zbieranych na bieżąco został oddzielony od zbioru uczącego. Zazwyczaj nie ma potrzeby od razu uczyć silnika na nowo uzyskanym przykładzie. Lepiej jest poczekać aż zbierze się większy zbiór i upewnić się, że warto przeliczać jeszcze raz parametry silnika.

Uwaga! W celach optymalizacyjnych funkcje takie jak `merge` czy konstruktor kopiujący są modyfikujące (mimo modyfikatora `const`)! Rzeczywiste kopie nie są tworzone, a jedynie przepisywane są odpowiednie adresy. Jeśli chce się pracować na kopiach należy wywołać funkcję `copy`, tworzącą głęboką kopię obiektów.
Mając już przygotowany zbiór uczący można przystąpić do nauki silnika. Istnieje tu kilka możliwości:

1. Nauka na pierwszym planie.

   Należy wywołać funkcję `trainOnce`, wywołującą jedną pętlę nauki[16]

[16] Jedno wywołanie pętli nauki jest najmniejszą operacją jaką można wykonać.

lub `foregroundTrain`, uczącą silnik aż do napotkania jednego z ograniczeń, podawanych jako parametry. Możliwe ograniczenia to wartość błędu lub czas spędzony na nauce[17,18]. Domyślne parametry oznaczają brak ograniczeń, czyli pętlę nieskończoną.

2. Nauka w tle.

   Tryb ten uruchamia się funkcją `backgroundTrain`, z takimi samymi parametrami jak `foregreoundTrain`. Funkcja (o ile to możliwe) uruchamia nowy wątek, w którym odbywa się nauka. Progres nauki można pobrać przy pomocy funkcji `getMSE`, a stan wątku funkcją `getFlags`. Silnik sugeruje również moment zakończenia nauki, jednak sam jej nie wyłączy przed upłynięciem jednego z warunków zakończenia. Aby wyłączyć naukę należy wywołać funkcję `stopBackgroundTraining` z opcjonalnym parametrem umożliwiającym asynchroniczne zatrzymanie wątku. Poniżej zostały wymienione możliwe flagi nauki w tle:

   (a) `NO_FLAG` — gdy nic się nie dzieje (uczenie w tle w ogóle nie zostało uruchomione),

   (b) `WORKING` — gdy nauka trwa,

   (c) `SUGGESTED_FINISH` — gdy z heurystyki sugestii wynika, że należy zakończyć naukę,

[17] Jak zostało wcześniej wspomniane, czas wewnątrz biblioteki jest rzeczywisty. Oznacza to, że duże obciążenie komputera będzie skutkowało wcześniejszym zakończeniem nauki. Dzięki temu obliczenia mogą być przerywane dostatecznie często, umożliwiając wykonanie na czas kodu odpowiedzialnego za grafikę i logikę aplikacji.

[18] Liczba przebiegów pętli nauki określana jest na podstawie statystyk poprzednich przebiegów. Liczba ta będzie określona niedokładnie przy pierwszych wywołaniach funkcji oraz przy zmianie obciążenia systemu.

(d) `WORK_FINISHED` — gdy wątek uczący został pomyślnie zakończony.

3. Nauka wsadowa.

   Jest to tryb przeznaczony do wstępnego przeliczenia parametrów silnika dla podstawowego zbioru uczącego. Tryb ten jest przewidziany jedynie do zastosowania w osobnym programie. Inne użycie jest stanowczo odradzane.

   Aby go uruchomić należy wywołać funkcję `batchTrain`, podając jako parametry nazwę pliku ze zbiorem uczącym, docelową wartość błędu i maksymalną ilość obiegów pętli.

```cpp
// Przykład ilustrujący naukę na pierwszym planie.

#include <ai-engine/ai-engine.hpp>

using namespace ai_engine;

// Klasa World reprezentuje świat gry.
// Załóżmy, że posiada ona następujące pola:
//
const unsigned World::UPDATE_INTERVAL_ = 25; // w milisekundach
// Co taki kwant czasu wywoływana jest funkcja uaktualniająca
//   świat gry (przeliczenie stanu świata, odrysowanie ekranu).
//   Funkcja ta MUSI się wykonywać regularnie, ale niekoniecznie
//   idealnie co taki kwant czasu
//   (użytkownik nie zauważy małego opóźnienia).
//
// Zmienna określająca czy nauka już się odbyła.
bool World::isTrained_ = false;
//
// Obiekt silnika.
AiEngine World::engine_;

// Załóżmy, że zbiór uczący mamy już gotowy.
// Załóżmy, że poniższa funkcja uaktualnia świat gry
//   i wywoływana jest regularnie co UPDATE_INTERVAL_.
void World::advance(void) {

  // ...
  // Kod odpowiedzialny za przeliczenie parametrów świata gry.
  // ...
  // Kod odpowiedzialny za odrysowanie okna gry.
  // ...

  static const double TARGET_MSE  // Docelowe MSE.
      = 25e-5;

  if (not isTrained_) {
    engine_.foregroundTrain(
        0.95 * UPDATE_INTERVAL_ * 1000, // Ograniczenie czasowe,
                                        //   lekko zaniżone.
        TARGET_MSE);
    if (engine_.getMSE() <= TARGET_MSE) // Zakończenie nauki,
      isTrained = true;                 //   Po osiągnięciu
                                        //   zadanego MSE.
  }
}
```

Listing 3. Nauka silnika na pierwszym planie.

```
// Przykład ilustrujący naukę w tle.

// Założenia i oznaczenia jak w poprzednim listingu.

// Wywoływane na początku:
void World::start(void) {
  engine_.backgroundTrain();   // Zdanie się wyłącznie
                               //   na heurystykę sugestii
                               //   momentu zakończenia.
}

void World::advance(void) {

  // ...
  // Kod odpowiedzialny za przeliczenie parametrów świata gry.
  // ...
  // Kod odpowiedzialny za odrysowanie okna gry.
  // ...

  if (not isTrained_) {
    AiEngine::Flags flags
        = engine_.getFlags();
    if (flags == AiEngine::SUGGESTED_FINISH) {
      // Zakończenie nauki.
      engine_.stopBackgroundTraining();
      isTrained_ = true;
    }
  }
}
```

Listing 4. Nauka silnika w tle.

5.4.5. Pobranie odpowiedzi silnika.

Pobranie wartości odpowiedzi silnika odbywa się przy użyciu funkcji `run`. Funkcja ta przyjmuje jako parametr wektor stanu świata.

```cpp
// Przykład ilustrujący pobranie wartości odpowiedzi silnika.

#include <ai-engine/ai-engine.hpp>

using namespace ai_engine;

// Klasa Enemy reprezentuje przeciwnika, który w nas strzela.
//
// Załóżmy, że ma ona zapisaną wewnątrz siebie referencje:
// AiEngine& engine_ - na silnik,
// Target& target_ - na cel, w który strzela.
//
// Załóżmy, że klasa ta ma funkcje getPosX i getPosY,
// zwracające odpowiednio współrzędne położenia lufy.

// Załóżmy, że poniższa funkcja wywoływana jest za każdym
// razem, gdy przeciwnik strzela.
void Enemy::shoot(void) {

  // Stworzenie wektora stanu świata:
  EngineIO_t inputs;
  inputs.push_back(target_.getPosX());
  inputs.push_back(target_.getPosY());
  inputs.push_back(target_.getVelX());
  inputs.push_back(target_.getVelY());
  inputs.push_back(getPosX());
  inputs.push_back(getPosY());

  // Odebranie odpowiedzi:
  const EngineIO_t output = engine_.run(inputs);
  const double angle = output.at(0);

  // ...
  // Kod odpowiedzialny za właściwy strzał.
  // ...
}
```

Listing 5. Pobranie wartości odpowiedzi silnika.

5.4.6. Zapisanie uzyskanych efektów.

Zarówno podczas wyłączania aplikacji, jak i w trakcie jej działania istnieje możliwość zapisania stanu silnika. Każdy z obiektów (silnik, zbiór przykładów i uczący) zapisywany jest osobnym wywołaniem i w osobnym pliku.

Plikom nadawana jest domyślna nazwa, która informuje jaki obiekt i kiedy został w nim zapisany. Zachowanie to można oczywiście zmienić, nadając własną nazwę.

5.5. Wskazówki autora w celu lepszego wykorzystania biblioteki.

5.5.1. Uwaga dotycząca adapterów.

Silnik nie nakłada żadnych ograniczeń ani na zakres wartości podawanych na wejście, ani na ich rozkład. Jedyne ograniczenie nałożone jest na wyjście, gdzie wartości (przy stosowaniu zwykłych adapterów) nie mogą wykroczyć poza przedział $[-1; +1]$. Sieci neuronowe pracują jednak znacznie lepiej na liczbach skupionych wokół zera. Można również zaobserwować poprawę działania, jeśli wartości te będą w rozkładzie normalnym. Dodatkowo dobrze jest, gdy rozrzut liczb nie jest za duży ani za mały, czyli gdy wariancja wynosi w przybliżeniu 1. Wynika to z budowy funkcji aktywacji — u wszystkich funkcji w okolicach zera wartości zmieniają się znacząco, natomiast w nieskończoności są prawie płaskie. Dlatego też w ostatecznych wersjach tworzonych gier warto jest napisać dla każdego z silników dedykowane adaptery.

5.5.2. Opakowanie klasy silnika.

Ze względu na elegancję ostatecznego kodu wynikowego warto jest opakować stworzone silniki w klasy–adaptery. Ich zadaniem jest oddzielenie kodu odpowiedzialnego za komunikację z modułem sztucznej inteligencji od logiki

gry. Poniżej został przedstawiony przykłady takiego przekształcenia.

Rozważmy następujące fragmenty kodu:

```
/** Kod przedstawiający odpytywanie
  *   silnika wymieszane zlogiką gry.
  */

#include <ai-engine/ai-engine.hpp>

using namespace ai_engine;

// Założenia analogincze jak w poprzednich listingach.
void Bot::fire(void) {

  EngineIO_t input;
  // ...
  // Tutaj kod odpowiedzialny za uzupełnienie
  //   wektora stanu świata.
  // ...

  const EngineIO_t output
    = engine_.run(input);
  const angle = output.at(0);

  // ...
  // Tutaj kod odpowiedzialny za wykorzystanie
  //   wyliczonego przez silnik kąta.
  // ...
}
```

Listing 6. Odpytywanie silnika wymieszane z logiką gry.

```
/** Kod przedstawiający tworzenie zbioru
  *   uczącego wymieszane zlogiką gry.
  */

#include <ai-engine/ai-engine.hpp>

using namespace ai_engine;

// Założenia analogincze jak w poprzednich listingach.
void Bullet::onCollisionWith(const Target&) {
  EngineIO_t input;
  // ...
  // Tutaj kod odpowiedzialny za uzupełnienie
  //   wektora stanu świata.
  // ...

  EngineIO_t desiredOutput;
  // ...
  // Tutaj kod odpowiedzialny za uzupełnienie
  //   wektora oczekiwanej odpowiedzi.
  // ...

  engine_.addExample(
    Example(input, desiredOutput));
}
```

Listing 7. Tworzenie zbioru uczącego wymieszane z logiką gry.

Niedużym nakładem pracy można je przekształcić do o wiele lepszej postaci:

```
/** Kod przedstawiający klasę opakowującą
  *   odwołania do silnika.
  */

#include <ai-engine/ai-engine.hpp>

// Założenia analogincze jak w poprzednich listingach.

class AimingEngine : public ai_engine::AiEngine {
public:
  AimingEngine(void);
  double calculateAngle(const World& world) const;
  void addExample(const Bullet& bullet);

private:
  ai_engine::adaptor::BasicIOAdaptor*
  createOutputAdaptor_(void) const;
};

AimingEngine::AimingEngine(void)
  : AiEngine(6, 1, "aiming",
  new ai_engine::adaptor::PlainIOAdaptor(),
  createOutputAdaptor_()) { }

ai_engine::adaptor::BasicIOAdaptor*
  AimingEngine::createOutputAdaptor_(void) const {

  using namespace ai_engine::adaptor;

  EngineIO_t maxValues;
  maxValues.push_back(2 * M_PI);

  return new BasicIOAdaptor(maxValues);
}
```

Listing 8. Opakowanie odwołań do silnika w osobną klasę (część 1).

```
/** Kod przedstawiający klasę opakowującą
  *   odwołania do silnika.
  * (Druga część fragmentu kodu.)
  */

double AimingEngine::calculateAngle(
  const World& world) const {

  using namespace ai_engine::adaptor;

  EngineIO_t input;
  input.push_back(world.getTarget().getPosX());
  input.push_back(world.getTarget().getPosY());
  input.push_back(world.getTarget().getVelX());
  input.push_back(world.getTarget().getVelY());
  input.push_back(world.getCannon().getPosX());
  input.push_back(world.getCannon().getPosY());

  const EngineIO_t output
    = AiEngine::run(input);

  return output.at(0);
}

void AimingEngine::addExample(const Bullet& bullet) {

  using namespace ai_engine::adaptor;

  EngineIO_t input;
  input.push_back(bullet.getPosX());
  input.push_back(bullet.getPosY());
  input.push_back(bullet.getVelX());
  input.push_back(bullet.getVelY());
  input.push_back(bullet.getSrcX());
  input.push_back(bullet.getSrcY());

  EngineIO_t desiredOutput;
  desiredOutput.push_back(bullet.getAngle();

  AiEngine::addExample(
    Example(input, desiredOutput));
}
```

Listing 9. Opakowanie odwołań do silnika w osobną klasę (część 2).

Dzięki powyższej klasie odpytanie silnika o odpowiedź będzie wyglądać następująco:

```
/** Kod przedstawiający odpytywanie silnika
  *   wykorzystujący klasę opakowującą.
  */

// Założenia analogincze jak w poprzednich listingach.
void Bot::fire(void) {

  const double angle
      = engine_.calculateAngle(world_);

  // ...
  // Tutaj kod odpowiedzialny za wykorzystanie
  //   wyliczonego przez silnik kąta.
  // ...
}
```

Listing 10. Wykorzystanie klasy opakowującej silnik do pobrania wartości.

Utworzenie przykładu uczącego będzie wyglądać następująco:

```
/** Kod przedstawiający tworzenie zbioru uczącego
  *   wykorzystujący klasę opakowującą silnik.
  */

// Założenia analogincze jak w poprzednich listingach.
void Bullet::onCollisionWith(const Target&) {

  engine_.addExample(*this);
}
```

Listing 11. Wykorzystanie klasy opakowującej do tworzenia zbioru uczącego.

5.6. Szczegóły implementacji.

5.6.1. Określanie rozmiaru sieci neuronowej.

Ciekawym zagadnieniem może okazać się problem określenia struktury sieci neuronowej — liczby neuronów w poszczególnych warstwach. Jeśli chodzi o ostatnią warstwę to odpowiedź jest oczywista. Jest w niej tyle neuronów ile wyjść ma sieć. Struktura pierwszej warstwy również jest znana. Liczba wejść do sieci określona jest przez programistę. Natomiast określenie struktury warstwy ukrytej nie jest już tak oczywiste.

Teoria mówi, że neuronów w warstwie ukrytej powinno być „wystarczająco dużo". Ustawienie zbyt małej liczby spowoduje, że sieć nie będzie działać poprawnie. Z kolei ustawienie za dużej nie poprawi jakości jej działania i niepotrzebnie skomplikuje obliczenia. Dobór właściwej wartości jest sztuką, której nauka wymaga doświadczenia i wyczucia.

Nie można arbitralnie określić tej liczby z góry. Metodą empiryczną stwierdzono, że wartość optymalna znajduje się w przedziale $(min(I, O);\ I \cdot O]$, gdzie I — liczba wejść sieci, a O — liczba wyjść sieci. W projekcie liczba neuronów w warstwie ukrytej określana jest następującym wzorem:

$$max(2;\ \tfrac{1}{4}(log_2 I + log_2 O)(I\ +\ O)) =$$
$$= max(2;\ \tfrac{1}{4} log_2 IO \cdot (I + O)).$$

W powyższym wzorze funkcja max jest zabezpieczeniem przed wygenerowaniem się sieci ze zbyt małą liczbą neuronów w warstwie ukrytej. Wzór

ten został opracowany na podstawie rozmiarów sieci proponowanych w artykułach wymienionych w bibliografii [1, 2, 3, 4, 5] oraz próbami manipulacji liczbą neuronów i badania zachowania się sztucznej inteligencji w grze demonstracyjnej.

Dodatkowo została przewidziana możliwość edycji tej liczby przez programistę rozumiejącego zagadnienie sztucznych sieci neuronowych. Wprowadzono dodatkowy parametr, który domyślnie przyjmuje wartość 1. Liczba neuronów w warstwie ukrytej jest mnożona przez jego wartość.

5.6.2. Predykcja czasu nauki.

W aplikacjach z interfejsem graficznym istotne jest kontrolowanie czasu obliczeń, szczególnie jeśli obliczenia te wykonywane są na pierwszym planie, pomiędzy odwołaniami do funkcji wejścia–wyjścia. Niestety czasu wykonania danego fragmentu kodu nie da się z góry określić. Jest to nieprzewidywalne nawet przy zapewnieniu programowi zwiększonego dostępu do zasobów komputera. Co gorsza obciążenie systemu jest zmienne w czasie, przez co dany kod będzie się wykonywał z inną długością czasu w każdym obiegu pętli.

Z pomocą przychodzi tu aproksymacja i predykcja. W projekcie zostało to rozwiązane za pomocą wykładniczej średniej kroczącej liczonej dla czasu przebiegu jednej pętli nauki silnika. Parametrowi $N(\alpha)$ została przypisana na stałe wartość 10. Jest ona na tyle wysoka, aby wyeliminować chwilowe fluktuacje pomiaru (okresowo wykonywane funkcje systemu operacyjnego i usług systemowych), ale wystarczająco niska, aby nie ignorować nagłego zwiększenia obciążenia systemu (na przykład uruchomienia programu wykonującego

ciężkie obliczenia). Można to zobrazować tak, że pomiar sprzed 20 iteracji pętli nauki jest pomijalnie nieistotny.

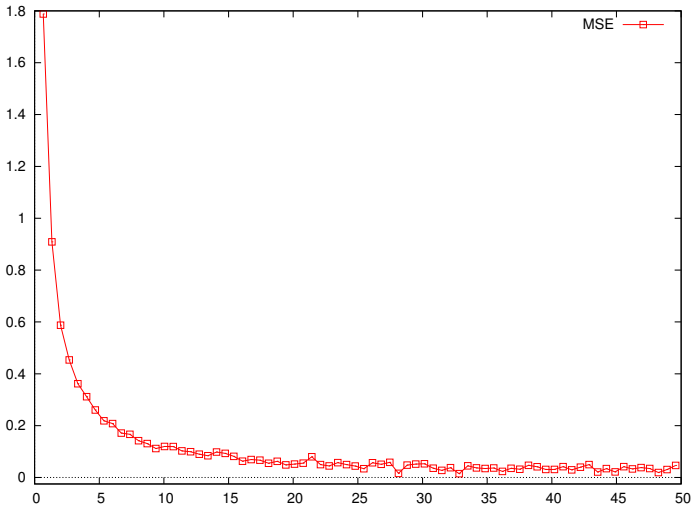
5.6.3. Heurystyka sugestii zakończenia nauki.

Często stosowanym warunkiem zakończenia nauki jest określona na stałe wartość błędu średniego kwadratowego. Z jednej strony ta metoda jest skuteczna, ponieważ jest prosta i zazwyczaj człowiek jest w stanie w przybliżeniu określić jaka dokładność mu wystarczy. Jednak z drugiej strony takie rozwiązanie posiada kilka wad:

- nie zawsze wiadomo jaka dokładność wystarczy,
- czasem potrzeba uzyskać możliwie największą dokładność,
- przede wszystkim dokładność, jaką można uzyskać na danych zbiorach uczących jest ograniczona jakością tych zbiorów.

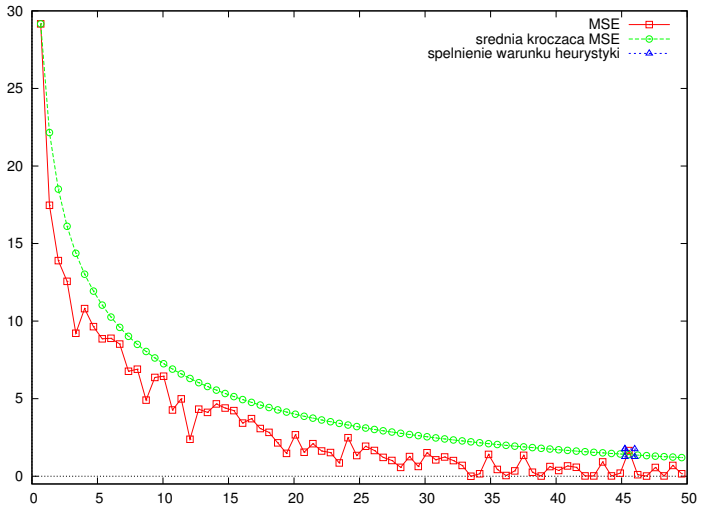
Głównie ze względu na ten ostatni powód do biblioteki została dodana heurystyka sugerowania momentu zakończenia nauki. Spełnienie jednego z jej wewnętrznych warunków skutkuje jedynie pojawieniem się propozycji zatrzymania nauki. Heurystyka ta jest jedynie dodatkiem do dwóch podstawowych warunków: wartości błędu średniego kwadratowego i ograniczenia czasowego.

Poniższy wykres przedstawia przykładowe zachowanie się wartości błędu średniego kwadratowego w kolejnych iteracjach pętli nauki.



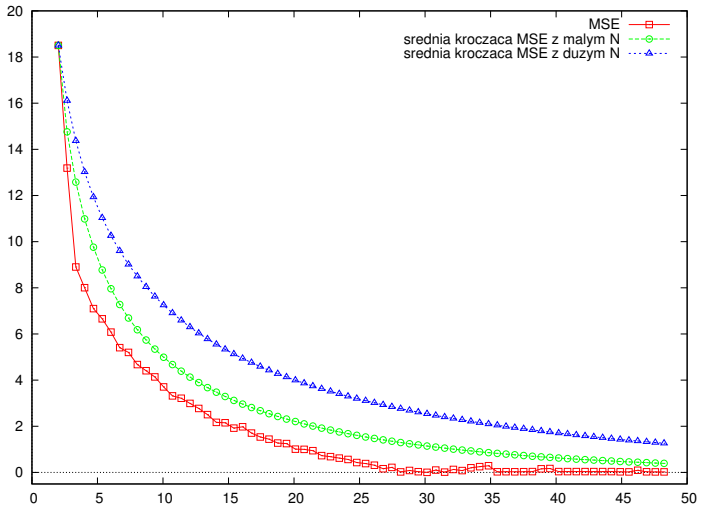
Rys. 19. Przykładowy wykres zmienności MSE w czasie.

Heurystyka sugestii składa się z dwóch podwarunków. Pierwszym jest wykładnicza średnia krocząca dla błędu średniego kwadratowego, o bardzo wysokiej wartości parametru N (domyślnie 1000). Po każdej iteracji pętli nauki aktualizowana jest wartość tej średniej. Na powyższym rysunku można zaobserwować, że w nieskończoności wykres błędu średniego kwadratowego jest w przybliżeniu płaski z drobnymi fluktuacjami. Jeśli w pewnym momencie wartość błędu przetnie się z wartością tej średniej, to występuje podejrzenie wyczerpania się zbioru uczącego oraz przeuczenia silnika i sugerowane jest zakończenie nauki. Na poniższym wykresie przedstawiono na czym polega uaktywnienie się tego warunku:



Rys. 20. Pierwszy warunek heurystyki sugestii końca nauki.

Drugi warunek tworzą dwie wykładnicze średnie kroczące dla wartości błędu średniego kwadratowego, znacznie różniące się wartością parametru N między sobą i o znacznie niższej wartości tego parametru niż w poprzednim warunku. Domyśle wartości to 10 i 100. Na poniższych przykładowych wykresach zobrazowano jak mogą się układać wartości dla tego warunku i na czym polega jego działanie.



Rys. 21. Drugi warunek heurystyki sugestii końca nauki.

Jeśli wartości tych średnich zbliżą się do siebie na pewną ustaloną odległość, to tak jak w poprzednim warunku sugerowane jest zakończenie procesu nauki. Progowi temu domyślnie nadawana jest bardzo niska wartość (10^{-9}).

5.7. Zaawansowane sterowanie silnikiem.

W przykładach nie zostały ujawnione niektóre parametry przewidziane do zaawansowanego zarządzania działaniem silnika. Poniżej zostały one krótko przedstawione.

5.7.1. Kontrola rozmiaru sieci neuronowej.

Istnieje możliwość podania do funkcji inicjującej silnik dodatkowego parametru, określającego rozmiar sieci. Jest to przewidziane do jawnego zaznaczenia, że problem, nad którym będzie pracował silnik jest prostszy lub trudniejszy niż byłoby przyjęte domyślnie. Przez podaną wartość mnożona jest liczba neuronów w warstwie ukrytej sieci neuronowej.

5.7.2. Kontrola parametrów średnich kroczących.

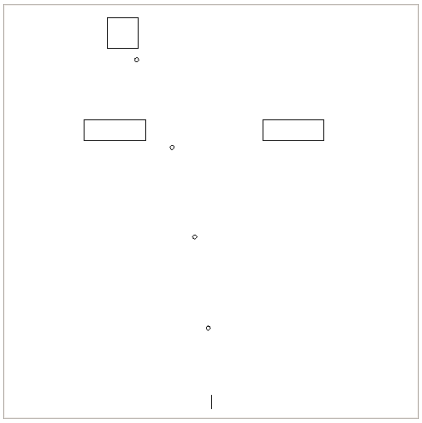
Konstruktor silnika przyjmuje trzy dodatkowe parametry. Przekazywane są one do kontenerów średnich kroczących używanych do algorytmu sugerowania zakończenia nauki. Nie jest zalecane, aby zmieniać domyślne wartości.

5.7.3. Kontrola progu aktywacji sugestii zakończenia nauki.

Funkcje uczące przyjmują dodatkowy parametr, będący progiem aktywacji heurystyki zakończenia nauki. Można skorzystać z możliwości podania tu wartości innej niż domyślna w celu wcześniejszego spełnienia warunku algorytmu.

6. Demonstracja.

Moduł demonstracyjny został zaprojektowany w oparciu o stary typ zręcznościowych gier jednoosobowych: *„shoot–'em–up"*. Bardzo znanymi przykładami takich pozycji są: *„Raptor — Call of the Shadows"*, *„Tyrian"* i *„Stargunner"*, popularne w połowie lat '90 poprzedniego wieku. W tym przypadku grafika została uproszczona do możliwego minimum. Moduł ten został stworzony za pomocą biblioteki Qt.



Rys. 22. Zrzut ekranu z demonstracji.

6.1. Zasady gry.

Świat gry ograniczony jest kwadratem. W grze uczestniczą działko i ruchomy cel, które nie mogą wyjść poza granicę świata. Zadaniem działka jest strzelanie w cel. Utrudniają to umieszczone na drodze przeszkody. Pociski są niszczone, jeśli wylecą poza granicę świata lub wpadną w cel lub przeszkodę. Zadaniem sztucznej inteligencji jest określenie czy opłaca się strzelić i jeśli tak to pod jakim kątem, aby jak najwięcej pocisków trafiło w cel i jak najmniej zostało zmarnowanych na przeszkodach.

6.2. Rola człowieka.

Moduł demonstracji nie do końca można nazwać grą. Udział człowieka jest znikomy. Można jedynie przestawić działko w inną pozycję. Wszystko dzieje się automatycznie. Po obu stronach grają boty. Gra została tak stworzona, aby było łatwiej przedstawić działanie silnika. Działanie automatyczne jest bardziej przewidywalne.

W grze została przewidziana możliwość przełączenia działka w tryb sterowania ręcznego. W tym trybie grający człowiek może sterować wszystkim: położeniem działka, kątem jego nachylenia i decydować kiedy powinien zostać oddany strzał. Ponieważ podczas testów wykorzystywany był jedynie generator przykładów tryb ręczny został kompletnie wyłączony.

6.3. Sztuczna inteligencja.

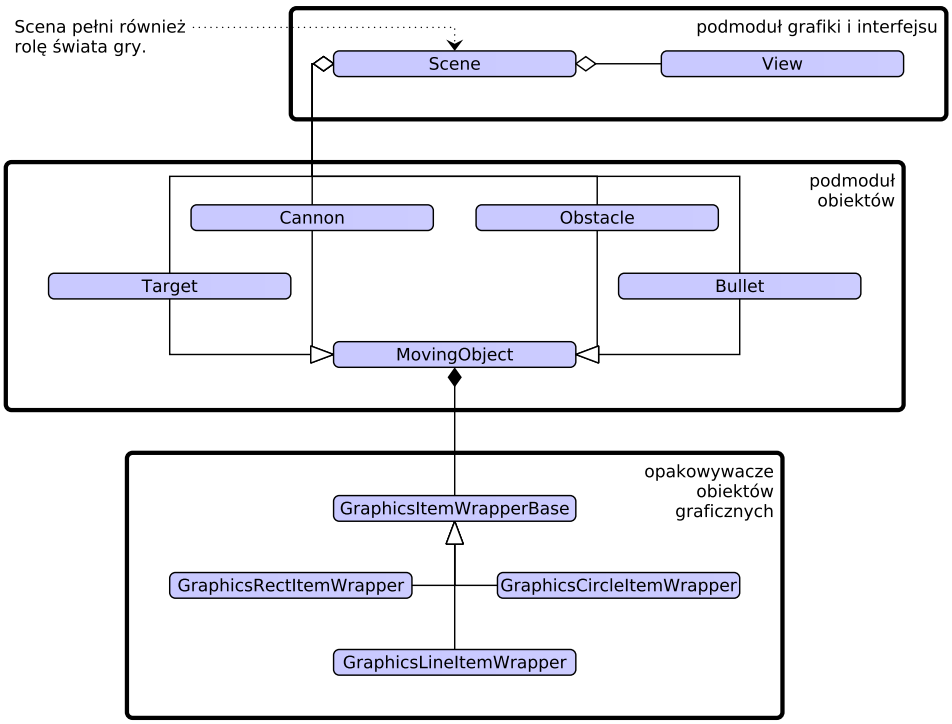
W grze zastosowano dwa rodzaje sztucznej inteligencji.

1. Frakcja uciekająca (cel) ma statycznie zaprogramowane zachowanie. Działanie jest w pełni przewidywalne. Polega ono na poruszaniu się ze stałą prędkością po pewnej łamanej.

2. Frakcja strzelająca wykorzystuje bibliotekę silnika. Działa on na podstawie parametrów obiektów w świecie gry takich jak położenia i prędkości.

6.4. Architektura modułu gry.

Z gry można wydzielić dwa podmoduły: obiekty (warstwa logiki gry) i warstwa prezentacji (okno gry). Do tego warto wymienić świat gry, który agreguje wspomniane podmoduły. Na poniższym schemacie została przedstawiona architektura modułu gry.



Rys. 23. Architektura modułu gry.

6.4.1. Warstwa logiki.

Warstwa ta składa się ze świata i należących do niego przedmiotów.

1. `Scene` — zawiera świat gry i obiekty silnika sztucznej inteligencji.
2. Obiekty świata gry:
   (a) `MovingObject` — Klasa bazowa każdego obiektu świata gry.
   (b) `Cannon` — Klasa działka. Zawiera logikę strzelania pociskami.
   (c) `Target` — Klasa celu. Zawiera logikę „unikania”.
   (d) `Bullet` — Klasa pocisku. Zawiera logikę zderzeń i tworzenia przykładów uczących.
   (e) `Obstacle` — Klasa przeszkody.

6.4.2. Warstwa prezentacji.

Warstwa ta właściwie sprowadza się jedynie do adapterów opakowujących klasy biblioteki Qt.

1. `Scene` — opakowuje klasę `QGraphicsScene` z biblioteki Qt. Odpowiada wyświetlanie obiektów i agreguje ich obiekty opakowujące.
2. `View` — opakowuje klasę `QGraphicsView` z biblioteki Qt. Odpowiada za komunikację z człowiekiem.
3. Adaptery obiektów graficznych:
   - `GraphicsItemWrapperBase`,
   - `GraphicsRectItemWrapper`,

- `GraphicsLineItemWrapper`,
- `GraphicsCircleItemWrapper`

— opakowują proste obiekty sceny z biblioteki Qt. Stworzone, aby łatwiej było dodawać nowe rodzaje obiektów prostych lub obiekty złożone.

6.5. Ciekawsze szczegóły implementacji.

Sztuczna inteligencja została rozdzielona pomiędzy dwa obiekty silnika. Zakładając, że wygenerowane zostały losowe silniki, bez wcześniejszej nauki:

1. Najpierw w jak najkrótszym czasie jeden silnik doprowadzany jest do stanu „w miarę dobrego" (został przyjęty próg $MSE \leqslant 10^{-3}$).
2. Następnie silnik ten jest kopiowany. Istnieją teraz dwie kopie: podstawowa i drugorzędna.
3. Na podstawie kopii podstawowej strzela działko. Silnik nie jest jeszcze dobrze nauczony, więc pociski nie są zawsze celne, ale lecą w odpowiednim kierunku.
4. W międzyczasie silnik drugorzędny uczony jest do możliwego maksimum możliwości.
5. Po zakończeniu nauki silnik podstawowy jest zastępowany przez silnik drugorzędny.

W celu zmniejszenia potrzebnych rozmiarów sieci neuronowych, bardziej przejrzystej implementacji i możliwości lepszego zrównoleglenia sztuczna inteligencja odpowiedzialna za strzelanie została rozdzielona na dwie kategorie:

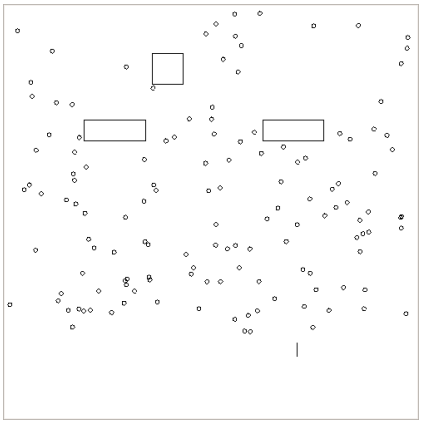
1. silnik określający kąt strzału
2. i silnik określający czy warto wystrzelić pocisk.

7. Eksperymenty.

Po zakończeniu implementacji biblioteki silnika i demonstracyjnej gry zostało przeprowadzonych kilka eksperymentów, kolejno rosnących stopniem skomplikowania. Aby przyspieszyć proces testowania, do programu ładowane były uprzednio przygotowane zbiory uczące i sztuczna inteligencja operowała jedynie na tych przykładach. Ponieważ przeciwnik (ruchomy cel) nie zmieniał taktyki uników podczas działania, brak przetwarzania danych zbieranych na bieżąco nie mógł mieć żadnego wpływu na wyniki.

Przygotowywane zbiory uczące tworzone były za pomocą generatora. Do jego utworzenia posłużył istniejący kod gry poddany lekkim modyfikacjom. Zastosowane zmiany to:

- kompletne wyłączenie sterowania przez człowieka,
- strzelanie wielu pocisków jednocześnie (25 – 50),
- losowanie kąta dla każdego z pocisków,
- losowanie nowej pozycji po każdym strzale.



Rys. 24. Podgląd świata gry podczas generowania zbiorów uczących.

Podczas takiego działania zbierane były „dobre" przykłady. Następnie zbiory zapisywane były do plików. Wygenerowane pliki wczytywane były do niezmodyfikowanej wersji gry i przeprowadzana była na nich nauka

Poniżej przedstawiono kolejne przeprowadzane eksperymenty. Wszystkie przeprowadzane były na (dość starym w momencie tworzenia projektu) komputerze z procesorem Intel E8400[19] (2×3.00 GHz, brak HT, 6M Cache, 1333 MHz FSB). Podczas przeprowadzania eksperymentów obciążenie komputera związane z innymi programami i systemem operacyjnym było pomijalnie małe.

7.1. Eksperyment pierwszy — statyczne działko i jednowymiarowy ruch celu.

W tym eksperymencie działko znajdowało się zawsze w jednym miejscu. Cel poruszał się horyzontalnie, odbijając się od granicy świata gry. Zatem problem był dwuwymiarowy. Parametrami wejściowymi były: jedna składowa położenia celu i jedna składowa prędkości celu.

Problem był bardzo prosty do rozwiązania. Silnik rozwiązywał go w zaledwie kilka sekund, osiągając niemal stuprocentową skuteczność. Wygenerowanie zbiorów uczących zajmowało mało czasu.

Pomyłki zdarzały się jedynie przy ekstremalnych kątach nachylenia lufy, ponieważ generator bardzo rzadko tworzył przykłady uczące na granicy dziedziny. Pomyłki te zostałyby wyeliminowane po dodatkowym ręcznym zebraniu tych rzadkich przykładów.

[19] `http://ark.intel.com/products/33910`

Tab. 1. Zestawienie wyników pierwszego eksperymentu.

czas generowania zbioru uczącego:	$30s$
liczba równoległych generatorów:	2
rozmiar zbioru uczącego:	500
czas trwania nauki:	$1500ms$
liczba strzałów:	72
liczba strzałów celnych:	68
liczba strzałów chybionych:	4
celność:	95%

(a) Po zakończeniu nauki. (b) Po zakończeniu nauki.

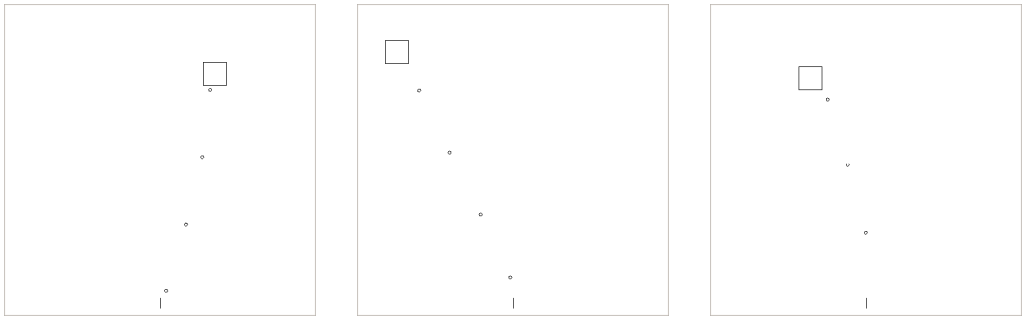
Rys. 25. Zrzuty ekranu z pierwszego eksperymentu.

7.2. Eksperyment drugi — dodany ruch wertykalny celu.

W tym eksperymencie celowi została dodana druga składowa prędkości, czyli liczba wymiarów problemu wzrosła o 2. Problem był praktycznie równie prosty jak poprzedni, a wyniki podobne.

Tab. 2. Zestawienie wyników drugiego eksperymentu.

czas generowania zbioru uczącego:	$80s$
liczba równoległych generatorów:	2
rozmiar zbioru uczącego:	1000
czas trwania nauki:	$2500ms$
liczba strzałów:	67
liczba strzałów celnych:	62
liczba strzałów chybionych:	6
celność:	93%



(a) Po zakończeniu nauki. (b) Po zakończeniu nauki. (c) Po zakończeniu nauki.

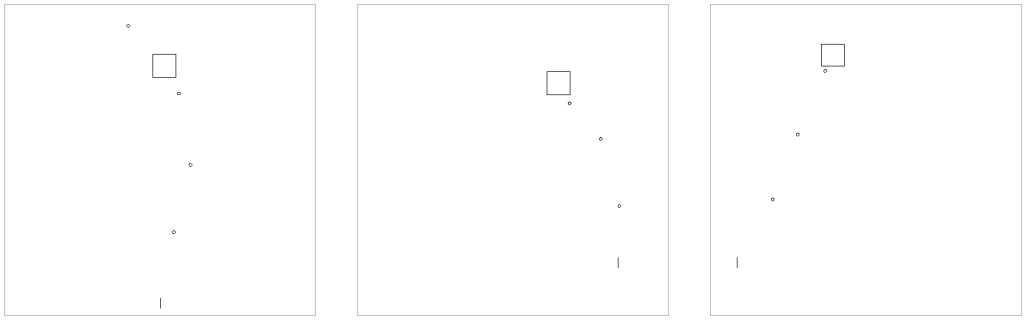
Rys. 26. Zrzuty ekranu z drugiego eksperymentu.

7.3. Eksperyment trzeci — dodany ruch działka.

Eksperyment ten różnił się od poprzedniego tym, że kąty były obliczane dodatkowo w zależności od położenia działka. Zatem w stosunku do poprzedniego eksperymentu liczba wymiarów wektora wejściowego wzrosła o 2. Podobnie jak poprzednio silnik szybko prawie bezbłędnie rozwiązał problem, myląc się jedynie przy ekstremalnych kątach nachylenia lufy. Zajęło to kilka–kilkanaście sekund. W tym eksperymencie wygenerowanie zbiorów zajęło wyraźnie więcej czasu, ale nadal nie sprawiało to większego problemu.

Tab. 3. Zestawienie wyników trzeciego eksperymentu.

czas generowania zbioru uczącego:	$3min$
liczba równoległych generatorów:	2
rozmiar zbioru uczącego:	2500
czas trwania nauki:	$10s$
liczba strzałów:	63
liczba strzałów celnych:	57
liczba strzałów chybionych:	6
celność:	91%



(a) W trakcie nauki. (b) Po zakończeniu nauki. (c) Po zakończeniu nauki.

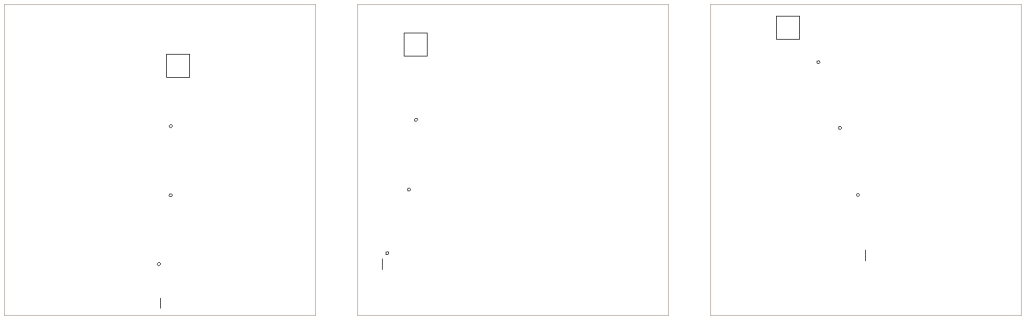
Rys. 27. Zrzuty ekranu z trzeciego eksperymentu.

7.4. Eksperyment czwarty — skomplikowanie ruchu celu.

W kolejnym eksperymencie algorytm ruchu celu został zmieniony. Cel zaczął się poruszać po dowolnie określonej łamanej, a nie z góry znanym „zygzakiem”. Działanie silnika się nie zmieniło. Wyniki tego eksperymentu były analogiczne do poprzedniego.

Tab. 4. Zestawienie wyników czwartego eksperymentu.

czas generowania zbioru uczącego:	$3.5min$
liczba równoległych generatorów:	4
rozmiar zbioru uczącego:	5500
czas trwania nauki:	$15s$
liczba strzałów:	70
liczba strzałów celnych:	63
liczba strzałów chybionych:	7
celność:	90%



(a) W trakcie nauki. (b) Po zakończeniu nauki. (c) Po zakończeniu nauki.

Rys. 28. Zrzuty ekranu z czwartego eksperymentu.

7.5. Eksperyment piąty, ostatni — dodanie przeszkód na drodze.

W tym eksperymencie do świata gry zostały dodane przeszkody osłaniające część planszy przed pociskami. Celem eksperymentu było wykazanie, że silnik jest dodatkowo w stanie stwierdzić celowość strzału i nie marnować pocisków na przeszkodach.

Parametry wejściowe nie zmieniły się. Nadal były to: obie składowe położenia działka, obie składowe położenia i prędkości celu. Natomiast na wyjściu pojawił się dodatkowy nowy parametr określający celowość strzału

w danych warunkach.

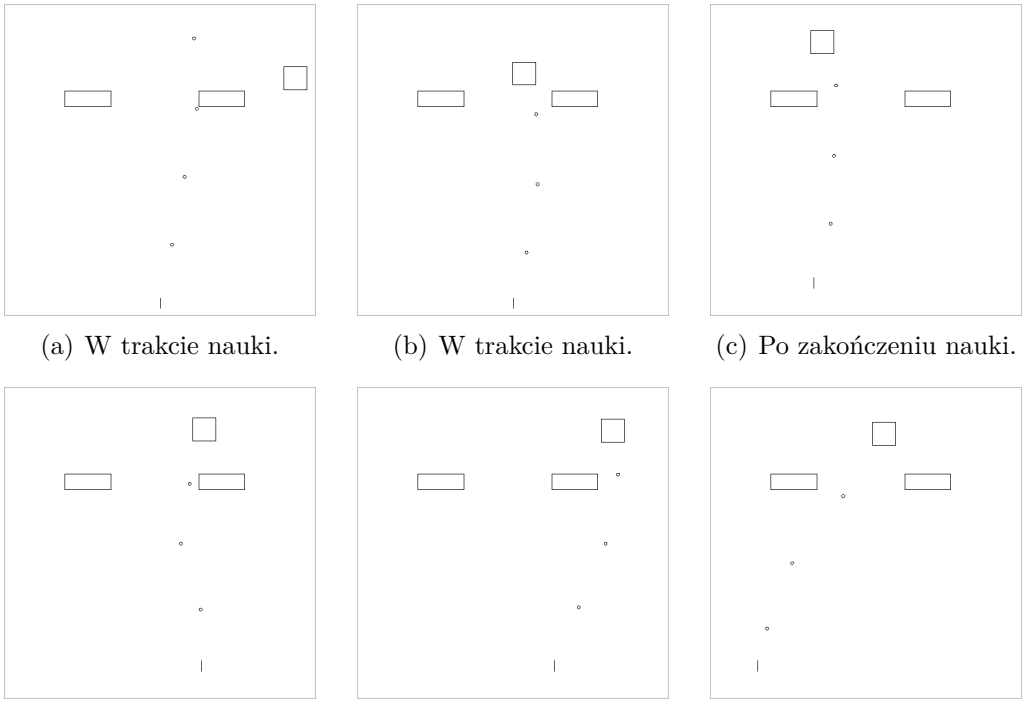
W późniejszej wersji eksperymentu silnik został rozdzielony na dwa mniejsze podsilniki:

- Pierwszy określał kąt lufy,
- drugi decydował o fakcie strzału.

Drugi podsilnik nie potrzebował tak wielu parametrów wejściowych. Wystarczyły mu położenie startowe pocisku i kąt strzału.

Tab. 5. Zestawienie wyników piątego eksperymentu.

		silnik celujący	silnik strzelający
czas generowania zbiorów uczących:	$20min$		
liczba równoległych generatorów:	4		
rozmiar zbiorów uczących:		9000	125000
czas trwania nauki:		$25s$	$1.5min$
liczba strzałów:	108		
liczba strzałów celnych:	66		
liczba strzałów chybionych:	14		
liczba strzałów zmarnowanych:	14		
celność:	83%		
omijanie przeszkód:	83%		



(a) W trakcie nauki. (b) W trakcie nauki. (c) Po zakończeniu nauki.

(d) Po zakończeniu nauki. (e) Po zakończeniu nauki. (f) Po zakończeniu nauki.

Rys. 29. Zrzuty ekranu z piątego eksperymentu.

Wyniki tego eksperymentu znacząco odstawały od poprzednich. Czas nauki strzelania był porównywalny z tymi w poprzednich eksperymentach — nie przekraczał trzydziestu sekund, natomiast czas nauki silnika określającego celowość strzału zachował się odwrotnie do oczekiwań. Struktura sieci neuronowej była znacznie mniejsza, ale stopień skomplikowania funkcji znacznie wyższy. Z tego powodu silnik, aby dojść do sensownych wyników, potrzebował przetworzyć o wiele więcej przykładów. Całość nauki trwała ponad minutę, ale nie przekraczała pięciu minut.

Podczas tego eksperymentu wystąpił jeszcze dodatkowy problem. Mianowicie komputer, na którym program był uruchomiony, posiadał jedynie dwa rdzenie, bez technologii HT, przez co uruchomienie silnika na pełnej wy-

dajności powodowało niską responsywność modułu graficznego. Z pomocą przyszło tu zwiększenie wartości nice i zmiana algorytmu kolejkowania procesów LWP w jądrze systemu operacyjnego.

7.6. Pomysły na skomplikowanie demonstracji.

Na etapie piątego eksperymentu zakończono dalszy ich rozwój. Zostało wykazane, że moduł silnika działa i daje dobre rezultaty, pod warunkiem dostarczenia odpowiednich zbiorów uczących.

Eksperymenty można było oczywiście nadal rozwijać. Przykładowe pomysły na bardziej skomplikowane problemy i propozycje ich rozwiązań to:

- ruchome przeszkody (do parametrów dochodzą położenia i prędkości przeszkód),
- więcej celów (powstaje nowy silnik, określający w który cel należy strzelać, a do parametrów dochodzą położenia i prędkości wszystkich celów),
- cel uciekający od pocisków (do parametrów silnika dochodzą położenia i kąty pewnej liczby ostatnio wystrzelonych pocisków).

8. Podsumowanie.

W niniejszej pracy opisano projekt bota grającego w demonstracyjną wersję strzelanki 2D i uczącego się wzorców zachowań. Dzięki wykorzystaniu sztucznej sieci neuronowej zachowania bota nie da się łatwo przewidywać, w przeciwieństwie do sztucznej inteligencji opartej na skryptach, które są najczęściej wykorzystywane w grach dostępnych na rynku.

Na przykładzie napisanego dema gry można zauważyć, że przeniesienie silnika do dowolnej innej gry, niekoniecznie strzelanki, jest łatwe. Wadą rozwiązania stosującego wykorzystaną metodę aproksymacji zachowania jest fakt, że nauka nie może się odbyć bez dostarczania poprawnych odpowiedzi. Wymaga to tworzenia zbiorów uczących. Proces ten jest wybitnie żmudny, nawet jeśli korzysta się z generatora.

Na podstawie przeprowadzonych eksperymentów okazało się, że rozwiązanie wykorzystane w projekcie nie jest złe i daje się zastosować w praktyce. Problemem może okazać się złożoność analizowanych zagadnień, znacznie komplikująca obliczenia przeprowadzane wewnątrz silnika.

Rezultat projektu jest zadowalający. Wszystkie cele zarówno postawione na początku, jak i te powstałe w trakcie tworzenia zostały zrealizowane.

A. Bibliografia.

[1] materiały do wykładów z przedmiotu „Rozpoznawanie obrazów”,

[2] materiały do wykładów z przedmiotu „Podstawy sztucznej inteligencji”,

[3] „Perełki programowania gier” (ang. „Game programming gems”), tom 2, rozdział 3, artykuł 14: J. Manslow — „Using a Neural Network in a Game: A Concrete Example”,

[4] S. Nissen — „Neural Networks Made Simple”[20],

[5] M. Gorywoda — „Sieci neuronowe w grach” (Software Developer’s Journal)[21],

[6] wiki projektu Encog[22],

[7] J. Pawlewicz — „Techniki sztucznej inteligencji w programach grających”,

[8] U. R. Ertürk — „Short term decision making with fuzzy logic and long term decision making with neural networks in real-time strategy games”[23].

[20] `http://leenissen.dk`
[21] `http://sdjournal.pl/magazine/56-sztuczna-inteligencja`
[22] `http://www.heatonresearch.com`
[23] `http://www.hevi.info/tag/artificial-intelligence-in-real-time-strategy-games`

B. Spis rysunków.

C. Spis listingów.

D. Spis tablel.

E. Załączniki.

Płyta CD z:

- niniejszą pracą (w formacie PDF),
- kodem źródłowym biblioteki silnika,
- kodem źródłowym demonstracji,
- plikami z zapisanym stanem silnika i przykładowymi zbiorami uczącymi.